Université de Neuchâtel
Faculté des sciences économiques

SCALING ADAPTIVE SUBSTRATE NEUROEVOLUTION

PhD thesis submitted to the Faculty of Economics and Business

Information Management Institute

University of Neuchâtel

For the PhD degree in Computer Science

by

Romain CLARET

Approved by the dissertation committee:

Prof. Kilian STOFFEL, University of Neuchâtel, thesis co-director

Adjunct Prof. Paul COTOFREI, University of Neuchâtel, thesis co-director

Prof. Adrian HOLZER, University of Neuchâtel

Prof. Christos DIMITRAKAKIS, University of Neuchâtel

Assistant Prof. Mark CONNOR, University College Dublin, Ireland

Defended on 11 August 2026



unine
Université de Neuchâtel
Faculté des sciences économiques

IMPRIMATUR POUR LA THÈSE

Scaling Adaptive Substrate Neuroevolution

Romain CLARET

UNIVERSITÉ DE NEUCHÂTEL
FACULTÉ DES SCIENCES ÉCONOMIQUES

La Faculté des sciences économiques,
sur le rapport des membres du jury

Prof. Kilian STOFFEL, Université de Neuchâtel, co-directeur de thèse
Prof. titulaire Paul COTOFREI, Université de Neuchâtel, co-directeur de thèse
Prof. Adrian Holzer, Université de Neuchâtel
Prof. Christos DIMITRAKAKIS, Faculté des sciences, Université de Neuchâtel
Prof. assistant Mark CONNOR, University College Dublin, Ireland

autorise l'impression de la présente thèse.

Neuchâtel, le 8 septembre 2026

La doyenne
Claudia Jonczyk Sédès

Faculté des sciences économiques
Avenue du 1er-Mars 26
CH-2000 Neuchâtel
Email: secretariat.seco@unine.ch
Tel: +41 32 718 15 00
www.unine.ch/seco

Abstract

Most artificial neural networks are designed by hand: a researcher picks the layers, the connections, and the computations. *Adaptive substrate neuroevolution* does the opposite: a small evolved recipe—DNA-like—generates a large network's substrate (its neuron placement and connections), scaling far beyond direct designs. The promise is real; the practical utility is not. This thesis targets ES-HyperNEAT, the canonical algorithm.

Three structural barriers stand in the way. Consider MNIST: a trivial classification task for gradient-trained networks, but stubborn for neuroevolution. Navigating the algorithm's vast configuration space takes months of Bayesian optimization; that campaign, once done, shows the ceiling is structural, not a search failure. Opening the substrate reveals why: evolved networks connect to a central cluster of input pixels—blind to most of the image. The bias is built into the encoding: the building blocks generating connections all peak at the image center, so every evolved substrate collapses into a narrow basin before the search explores elsewhere. Partitioning the input across region-specialized experts more than doubles accuracy, but each expert is evolved one at a time. Scaling further requires parallel hardware, but the quadtree produces a different graph per genome—a shape it cannot batch.

Breaking this wall requires reformulating substrate discovery itself. We introduce EMR-HyperNEAT, which inverts the order: instead of growing the substrate one decision at a time, it evaluates a fixed grid of candidate positions in parallel and filters by variance to keep the informative ones. Substrate discovery becomes batched matrix arithmetic on any JAX-supported hardware, reaching depths infeasible under quadtree subdivision. The deeper gain is representational: because the substrate is now a tensor, adding biological mechanisms (recurrence, per-node functions, neuromodulation) becomes appending columns to a matrix rather than redesigning the algorithm. The same shift enables external composition: substrates can be broken into parts, frozen, and recombined through evolved mappers. This leads to the GEENNS algorithm: a compositional architecture of small specialist networks, inspired by the cortex's repeated columnar units, aiming at emergent modularity, developmental growth, and lifelong learning.

Our prototype validates GEENNS's central premise: specialist grids evolved on primitive tasks, frozen and recombined by evolved mappers, solve harder compound tasks they were never trained on. Composed networks succeed where monolithic baselines fail, and the same frozen grids support new compound tasks without retraining. Whether mappers learn general compositional reasoning rather than task-specific routing remains open; the infrastructure to answer it is now in place.

The contribution is diagnostic, not competitive: indirect encoding will not outperform gradient-trained systems on standard benchmarks yet, but adaptive substrate neuroevolution can now run at the scale where its broader question becomes tractable—whether evolution, working within encodings and selection pressures that humans still choose, can shape networks that are not bred to imitate any human and whose structure no human

hand directly specifies. That is a challenge for artificial life.

Keywords: neuroevolution; indirect encoding; ES-HyperNEAT; EMR-HyperNEAT; adaptive substrate discovery; CPPN; GPU acceleration; hyperparameter optimization; mixture of experts; compositional architectures; GEENNS; artificial life.

Résumé

La plupart des réseaux de neurones artificiels sont conçus de manière explicite : le chercheur définit leur architecture, notamment les couches, les connexions et les opérations de calcul. La *neuroévolution à substrat adaptatif* adopte une approche radicalement différente : une représentation compacte, évoluée et analogue à un « programme génétique », génère automatiquement le substrat d'un réseau de grande taille, c'est-à-dire l'organisation spatiale de ses neurones et de leurs connexions. Cette approche offre, en principe, des possibilités de passage à l'échelle bien supérieures à celles des méthodes de conception directe. Toutefois, si son potentiel conceptuel est établi, son utilité pratique demeure limitée. Cette thèse s'intéresse à ES-HyperNEAT, l'algorithme de référence de cette famille, et analyse les principaux facteurs structurels qui entravent son développement.

Trois limitations majeures sont mises en évidence. Le problème apparaît clairement sur MNIST, une tâche de classification relativement simple pour les réseaux entraînés par descente de gradient, mais qui demeure difficile pour les approches neuroévolutionnaires. L'exploration de l'espace considérable des configurations d'ES-HyperNEAT nécessite plusieurs mois d'optimisation bayésienne des hyperparamètres. Les résultats de cette recherche approfondie indiquent toutefois que les performances obtenues ne sont pas limitées par une exploration insuffisante, mais par des contraintes intrinsèques à la structure même de l'algorithme. L'analyse des substrats évolués en fournit l'explication : les connexions se concentrent sur une région centrale restreinte de l'espace d'entrée, ce qui conduit les réseaux à négliger la majeure partie de l'image. Ce comportement résulte directement du biais introduit par l'encodage : les primitives responsables de la génération des connexions présentent systématiquement leur activation maximale au centre de l'image, entraînant ainsi les substrats évolués vers un espace de solutions fortement contraint avant que l'évolution ne puisse explorer des configurations alternatives. Une partition de l'entrée entre plusieurs experts spécialisés sur différentes régions permet de plus que doubler la précision. Cette stratégie reste néanmoins difficilement extensible, puisque chaque expert doit être évolué indépendamment. Une généralisation à plus grande échelle requiert donc une exécution parallèle, alors que la subdivision par *quadtree* génère une topologie différente pour chaque génome, empêchant une vectorisation efficace des calculs.

Pour dépasser cette limitation, cette thèse propose une reformulation du processus de découverte du substrat. Nous introduisons EMR-HyperNEAT, une approche qui inverse la logique traditionnelle d'ES-HyperNEAT. Plutôt que de construire progressivement le substrat au moyen d'une succession de décisions adaptatives, EMR-HyperNEAT évalue simultanément une grille fixe de positions candidates, puis sélectionne les positions pertinentes à l'aide d'un critère fondé sur la variance. La découverte du substrat peut dès lors être formulée comme un ensemble d'opérations matricielles exécutées par lots, compatibles avec tout matériel pris en charge par JAX. Cette reformulation permet d'explorer des substrats d'une profondeur hors de portée de la subdivision récursive en *quadtree*.

Au-delà du gain computationnel, cette transformation apporte un avantage représentationnel majeur. Le substrat étant désormais représenté sous forme tensorielle, l'intégration de mécanismes supplémentaires d'inspiration biologique — tels que les connexions récurrentes, les fonctions spécifiques aux nœuds ou la neuromodulation — peut être réalisée par simple ajout de colonnes à la représentation matricielle, plutôt que par une refonte de l'algorithme. Cette représentation favorise également la composition modulaire : des sous-réseaux peuvent être isolés, figés, puis recombinés au moyen de mécanismes de mise en correspondance, les mappeurs, eux-mêmes soumis à évolution. Cette propriété conduit à la formulation de GEENNS, une architecture compositionnelle fondée sur un ensemble de petits réseaux spécialisés. Inspirée de l'organisation du cortex en unités colonnaires répétées, cette architecture vise à favoriser l'émergence de la modularité, la croissance développementale et, à terme, l'apprentissage continu.

Le prototype développé dans cette thèse fournit une première validation du principe central de GEENNS. Des grilles spécialisées, initialement évoluées sur des tâches élémentaires, peuvent être figées puis recombinées par des mappeurs évolués afin de résoudre des tâches composées plus complexes sur lesquelles elles n'ont jamais été entraînées. Les architectures composées parviennent ainsi à résoudre des problèmes sur lesquels les architectures monolithiques de référence échouent. En outre, les mêmes grilles figées peuvent être réutilisées pour de nouvelles tâches composées sans nécessiter de nouvel entraînement. Il reste néanmoins à déterminer si les mappeurs acquièrent une capacité de composition véritablement générale ou s'ils apprennent principalement des stratégies de routage spécifiques aux tâches considérées. L'infrastructure expérimentale mise en place permet désormais d'aborder directement cette question.

La contribution de cette thèse doit ainsi être comprise principalement comme une contribution méthodologique et diagnostique, plutôt que comme une démonstration de supériorité en termes de performances. À ce stade, les méthodes d'encodage indirect ne rivalisent pas avec les réseaux entraînés par gradient sur les bancs d'essai standard. En revanche, les travaux présentés rendent désormais possible l'étude de la neuroévolution à substrat adaptatif à une échelle suffisante pour examiner une question plus fondamentale : dans quelle mesure l'évolution, agissant dans le cadre d'encodages et de pressions de sélection qui restent définis par l'être humain, peut-elle faire émerger des réseaux dont l'architecture n'est pas conçue manuellement et dont les comportements ne sont pas explicitement sélectionnés pour imiter l'humain ? Cette question constitue un défi pour la recherche en vie artificielle.

Mots-clés : neuroévolution ; encodage indirect ; ES-HyperNEAT ; EMR-HyperNEAT ; découverte de substrat adaptatif ; CPPN ; accélération GPU ; optimisation d'hyperparamètres ; mélange d'experts ; architectures compositionnelles ; GEENNS ; vie artificielle.

I wouldn't bother with that thing.
My guess is that touching it will just make your life even worse somehow.

— GLaDOS

Acknowledgments

My first research proposal, written in 2020, examined four ways to build a reasoning system. The fourth was a network of specialists that collaborate, each one narrow, none of them sufficient alone. I ruled it out: the coordination problem was, at the time, beyond reach. Six years later, this thesis is the machinery I would need to go back and take that option seriously. It took me most of those years to notice that I had been living inside the answer the whole time.

The years alternated between stable eras and chaotic ones; the work only ever knew one kind. The greater part of them went into an algorithm that had to fail first. This thesis asks whether ES-HyperNEAT can be made practical and answers no, not by optimizing it. That sentence was expensive: hyperparameter sweeps that ran for months to establish a ceiling, a champion substrate that turned out to be wired to forty-eight pixels out of seven hundred eighty-four, a bottleneck that no amount of engineering could remove because it lived in the shape of the algorithm rather than in the code. The broader vision I had proposed was set aside until the substrate was ready. Learning what does not work is a slow and unglamorous way to spend a doctorate, and there was no version of this that skipped the testing. I did not get through it alone. Work like this gathers a crew one person at a time, and what follows is mine.

Kilian opened the door, the kind that only opens from the other side. He gave me the chance to be in academia at all, and then the room to prove I belonged there, which is a harder thing to give than it sounds and is given far less often than it should be.

Paul believed in the project before there was much of it to believe in, and gave me the one thing more useful than belief: time. Years of it, which is what an overambitious proposal needs to mature into something a thesis can actually carry. Supervisors are rarely thanked for that particular patience, because from the outside it looks like nothing is happening. Something was. Thank you also for the science fiction, traded back and forth across all those years; some of our best conversations about intelligence were nominally about a desert planet.

Catalin helped me when I needed it most, and more than he knows.

A Doc.Mobility scholarship from the University of Neuchâtel and swissuniversities was the gate: six months at University College Dublin and, after the dry years, a green place. Mike, who hosted me there, made those six months count. His support was immeasurable, and it came with a steady good humor that let me see value in work I had stopped being able to judge for myself. His group, the Natural Computing Research and Applications Group, showed me what researchers in my field look like from the inside, day after ordinary day, and that turned out to be exactly what I needed in order to want to stay. The group also opened its machines and UCD's Sonic cluster to this work. The methodological standards I picked up in Dublin are visible on every page here: they are why the results rest on thirty seeds rather than on however many would have fit the deadline.

Arthur, the brilliant master's student whose thesis I had the chance to supervise, ran

experimental campaigns that would have cost me another year alone, and ran them carefully, while I worked day and night on the quadtree optimizations and EMR-HyperNEAT.

Pascal gave me the machines. Every long descent needs a vessel: the cluster at his institute, the Institut d'informatique (IIUN), carried this work from the very beginning, and he made a researcher from another faculty feel like part of his lab.

Miki gave me far more than teaching me statistics and accounting: he listened to the complaints, and knew when to distract me from them.

This work also rests on software written by people who will never know my name, and the wider open-source community's tools are the only reason any of these experiments were possible. One team deserves its name here: TensorNEAT, whose library inspired what became EMR-HyperNEAT. Everything here is built from bricks that other hands molded first.

More than three hundred dense pages is a lot to ask of anyone: my thesis committee read them all, then listened to me defend them. Thank you for the discussion that followed, and for what it left me with: the feeling that some people understand what I am doing.

Adrian let me be myself at the institute: a ghost in my corner, doing my own things, and he never once disturbed the haunting.

Institutes run on the people who get thanked last and should be thanked first.

Eugenia's kind words made the institute feel like a place where someone was looking out for me; there is no more accurate way to put it. Florence handled years of schedules and made the conversations around them genuinely good ones. Natacha listened when listening was the thing that was needed, and quietly made the examination periods easier than they had any right to be. Vlad talked me back down whenever the institution became too much, which was more often than I would like to admit.

James's positive attitude never once ran out, and his smiles tended to arrive on precisely the days I felt most alone.

Luana held things up when they went from bad to worse, which is when support actually counts and when it is least often offered.

Christel helped me stay with what was happening when my mind would rather have been anywhere else.

Wai gave me energy I did not have, and taught me more about myself than I had managed to work out on my own.

Fiona gave days. Thousands of them, years of her own life spent alongside a project that was never hers, and on every stretch where I could not find the bright side myself, she kept it in view. Some books say the greatest gift one person can give another is water. It is days.

And my parents gave the ordinary, unspectacular, load-bearing kind of support: the stability that let me spend the first of these years on something ambitious and uncertain without once checking whether the ground would hold. The last years rearranged our roles, and it became my turn to hold steady for them. I am grateful for both: for the time they carried me, and for the discovery that, when it mattered, they had taught me how to carry.

Everything in this thesis argues one point in different words: the things worth having are not designed into a system; they are given the conditions in which they can arrive. So was I.

Still alive.

Neuchâtel, August 2026 Romain Claret

Contents

Abstract **5**

Résumé **7**

Acknowledgments **11**

List of Publications **31**

Notation and Symbols **33**

I Foundations 37

1 Introduction **39**
1.1 The Case for Evolved Neural Architectures 40
1.2 ES-HyperNEAT: Powerful but Impractical 43
1.2.1 Barrier Cluster 1: The Configuration Expense 43
1.2.2 Barrier Cluster 2: The Representational Collapse 44
1.2.3 Barrier Cluster 3: The Computational Bottleneck 44
1.2.4 The Three Barriers as a Research Program 45
1.3 Research Questions 45
1.3.1 TRQ1: How can the hyperparameter space of adaptive substrate neuroevolution be navigated efficiently? 46
1.3.2 TRQ2: Can ES-HyperNEAT's fundamental limitations be diagnosed empirically? 46
1.3.3 TRQ3: Can adaptive substrate discovery be redesigned for massively parallel execution without sacrificing adaptivity? 46
1.3.4 TRQ4: Can input decomposition extend adaptive substrates to high-dimensional problems? 47
1.4 Experimental Domains 47
1.5 Summary of Contributions 47
1.6 The Motivating Vision: GEENNS 50
1.7 Thesis Structure 51

2 Background and State of the Art **55**
2.1 From Direct Encoding to Adaptive Substrates 55
2.1.1 Evolutionary Algorithms: Foundations 55
2.1.2 NEAT: NeuroEvolution of Augmenting Topologies 57
2.1.3 CPPNs: Compositional Pattern-Producing Networks 61

2.1.4 HyperNEAT: Indirect Encoding via CPPNs . . . 63
2.1.5 Multi-Output CPPNs . . . 65
2.1.6 ES-HyperNEAT: Adaptive Substrate Discovery . . . 65
2.1.7 DES-HyperNEAT: Deep Evolvable Substrates . . . 70
2.1.8 GEENNS: Toward Compositional Substrates . . . 72
2.1.9 The Central Bias: A Structural Consequence of CPPN and Quadtree Geometry . . . 73
2.1.10 Indirect Encoding: Mathematical Formulation . . . 74
2.2 Hyperparameter Optimization . . . 75
2.2.1 The Hyperparameter Problem in Neuroevolution . . . 76
2.2.2 Tree-structured Parzen Estimators . . . 76
2.2.3 Multi-Fidelity Methods: Hyperband and Early Stopping . . . 78
2.3 Mixture of Experts and Modular Architectures . . . 79
2.3.1 MoE and Modular Neuroevolution . . . 79
2.4 GPU Acceleration for Evolutionary Algorithms . . . 80
2.4.1 GPU Programming Model . . . 80
2.4.2 Memory Hierarchy and Coalescing . . . 81
2.4.3 Why Uniform Computation Matters . . . 81
2.4.4 JAX: Composable Transformations for Scientific Computing . . . 81
2.4.5 GPU-Accelerated Evolutionary Computation . . . 81
2.4.6 The Core Challenge: Irregular Algorithms on Regular Hardware . . . 82
2.5 Problem Definitions and Evaluation Methodology . . . 83
2.5.1 Boolean Logic Problems . . . 83
2.5.2 Parity-N Problems . . . 84
2.5.3 MNIST . . . 84
2.5.4 Convergence Criterion . . . 85
2.5.5 Statistical Methodology . . . 85
2.5.6 Notation Summary . . . 85
2.6 Relationship to Neural Architecture Search . . . 85
2.7 Chapter Summary . . . 89

II Scaling ES-HyperNEAT 91

3 Hyperparameter Optimization and Early-Stopping 93
3.1 The Hyperparameter Search Problem . . . 94
3.2 TPE Optimization of ES-HyperNEAT . . . 95
3.2.1 Experimental Setup . . . 95
3.2.2 Results . . . 96
3.2.3 Statistical Significance . . . 97
3.2.4 Optimal Configurations . . . 98
3.2.5 Validation and the 29% Result . . . 99
3.2.6 Extended Generation Analysis . . . 100
3.2.7 Computational Cost of Extended Evolution . . . 103
3.2.8 Multi-Configuration Convergence . . . 106
3.3 Hyperparameter Sensitivity Analysis . . . 107
3.3.1 Correlation Structure . . . 107
3.3.2 Constant Parameters: The Convergent Recipe . . . 108
3.3.3 Sensitive Parameters: Variance Threshold and CPPN Complexity . 108

3.3.4 Summary of Sensitivity Hierarchy . . . 109
3.4 Transferability of Optimized Configurations . . . 110
3.4.1 Logic Operations . . . 110
3.4.2 Fashion-MNIST . . . 112
3.4.3 Interpretation: When and Why Transfer Works . . . 113
3.5 An Early-Stopping Pruner for ES-HyperNEAT . . . 114
3.5.1 Phase 1: Deriving the Pruning Rule . . . 116
3.5.2 Phase 2: Validation . . . 120
3.6 Generalization and Boundary Conditions . . . 126
3.6.1 Computational Cost Reduction . . . 126
3.6.2 Comparison with Hyperband . . . 127
3.6.3 Limitation: Converged Search Spaces . . . 130
3.7 Bridge: Why Two Methods Are Needed . . . 133
3.8 Combined Optimization Pipeline . . . 133
3.8.1 Unified Framework . . . 134
3.8.2 Deployment Guidance . . . 135
3.9 Synthesis: What Optimization Reveals About Architecture . . . 135
3.10 Chapter Summary . . . 137

4 Breaking the Central Bias 141
4.1 The Central Bias Problem . . . 142
4.1.1 Root Cause: CPPN Geometry and Quadtree Interaction . . . 142
4.1.2 Hypothesis . . . 142
4.2 Partitioned-Experts Architecture (MoE-inspired) . . . 142
4.2.1 Input Partitioning Strategy . . . 143
4.2.2 Independent variant . . . 143
4.2.3 Shared variant . . . 144
4.3 Aggregation Strategies . . . 144
4.3.1 Simple Heuristics . . . 144
4.3.2 Mask-Based Heuristics . . . 144
4.3.3 Performance-Weighted Methods . . . 145
4.4 Experimental Design . . . 145
4.4.1 Control Factors . . . 145
4.4.2 Hyperparameter Configuration . . . 146
4.4.3 Evaluation Protocol . . . 146
4.4.4 Receptive Field Analysis Methodology . . . 146
4.5 Results . . . 147
4.5.1 Performance Comparison . . . 147
4.5.2 Expert Granularity . . . 149
4.5.3 Statistical Significance . . . 150
4.5.4 Aggregation Comparison . . . 151
4.5.5 Per-Class Prediction Analysis . . . 155
4.5.6 Control Factor Effects . . . 157
4.6 Mechanistic Analysis: Receptive Field Evolution . . . 158
4.6.1 Coverage Expansion . . . 158
4.6.2 Evolution Dynamics Within a Single Trial . . . 158
4.6.3 Receptive Field Evolution Across Expert Counts . . . 160
4.6.4 Receptive Field Density . . . 160

4.6.5 The Mechanism: Why Partitioning Works . . . 161
4.7 Synthesis: From Diagnosis to Decomposition . . . 164
4.7.1 Central Bias as an Encoding-Geometry Failure Mode . . . 164
4.7.2 Why $N_e = 13$ Is Optimal . . . 165
4.7.3 Comparison to Classical Mixture-of-Experts . . . 166
4.7.4 Connection to the GEENNS Vision . . . 167
4.7.5 From Fixed to Evolved Decomposition . . . 167
4.7.6 Scope Limitations . . . 168
4.7.7 Cost–Benefit Analysis . . . 169
4.8 Chapter Summary . . . 170

III Beyond the Quadtree 173

5 ES-HyperNEAT's Quadtree Problem 175
5.1 The Quadtree Bottleneck Hypothesis . . . 176
5.2 Implementation Architecture . . . 177
5.2.1 System Overview . . . 177
5.2.2 Three-Phase Substrate Discovery . . . 177
5.2.3 Division Initialization (Quadtree Subdivision) . . . 178
5.2.4 Pruning Extraction (Band Detection) . . . 180
5.2.5 Network Cleaning . . . 181
5.3 Optimization Strategies . . . 181
5.4 The HSHG Alternative . . . 185
5.4.1 Spatial Hash Function . . . 185
5.4.2 Fixed-Shape State for JIT Compatibility . . . 185
5.4.3 O(1) Radius Query . . . 185
5.4.4 Dense Grid Pre-Generation . . . 186
5.4.5 Why HSHG Fails: The Discovery Approach Mismatch . . . 187
5.4.6 HSHG vs. Quadtree Comparison . . . 187
5.4.7 Evolutionary Confirmation . . . 189
5.5 Experimental Results . . . 190
5.5.1 Benchmark Configuration . . . 190
5.5.2 Implementation Comparison Framework . . . 190
5.5.3 Solve Rates by Depth . . . 191
5.5.4 Convergence Speed by Population . . . 192
5.5.5 Generation Time Scaling . . . 192
5.5.6 JIT Compilation Overhead . . . 193
5.5.7 Statistical Validation . . . 197
5.5.8 Multi-Benchmark Validation . . . 199
5.6 Analysis: The Impossibility Result . . . 201
5.6.1 The Fundamental Limitation . . . 201
5.6.2 JIT Compilation Plateau . . . 201
5.6.3 Post-JIT Performance Characteristics . . . 202
5.6.4 Implementation Comparison Caveats . . . 204
5.6.5 Additional Implementation Optimizations . . . 204
5.6.6 Architectural Limitation: Two-Layer Substrates . . . 205
5.6.7 Implications for the Thesis Roadmap . . . 205
5.7 Synthesis: From SIMT Incompatibility to Architectural Replacement . . . 206

5.7.1 The Incompatibility as a Design Constraint . . . 206
5.7.2 Why HSHG Over-Discovers . . . 207
5.7.3 Connecting Chapters 4 and 5: WHERE and WHY . . . 208
5.7.4 Connection to the GEENNS Vision . . . 208
5.8 Chapter Summary . . . 208

6 EMR-HyperNEAT: Eager Multi-Resolution Substrate Discovery 211
6.1 The Eager Evaluation Hypothesis . . . 212
6.2 The EMR Algorithm . . . 213
6.2.1 Hierarchical Grid Definition . . . 214
6.2.2 Variance-Based Filtering . . . 216
6.2.3 Connection Discovery . . . 221
6.3 Connection Type Taxonomy . . . 221
6.4 Implementation and Execution Strategies . . . 223
6.4.1 Two-Level Caching . . . 225
6.5 Experimental Results . . . 227
6.5.1 Experimental Setup . . . 228
6.5.2 Benchmark Problems . . . 229
6.5.3 Scaling and Speedup . . . 230
6.5.4 Positioning Among Neuroevolution Frameworks . . . 230
6.5.5 Statistical Validation . . . 235
6.5.6 Solution Quality . . . 236
6.5.7 Population Scaling . . . 239
6.5.8 Extreme-Depth Scaling . . . 240
6.5.9 GPU Scaling and Real-World Validation . . . 242
6.5.10 Cross-Domain Validation . . . 243
6.5.11 Connection Type Comparison . . . 245
6.6 Analysis . . . 250
6.7 Synthesis: From Sequential Traversal to Tensor-Based Substrate Discovery 255
6.7.1 Resolving the Chapter 5 Incompatibility . . . 255
6.7.2 Connection Type Taxonomy as Architectural Freedom . . . 255
6.7.3 What EMR Enables: A Quantitative Before-and-After . . . 256
6.7.4 Implications for GEENNS . . . 257
6.7.5 Connection Types as GEENNS Building Blocks . . . 257
6.7.6 Generalizability Constraints . . . 258
6.8 Chapter Summary . . . 258

IV The GEENNS Vision 261

7 The GEENNS Vision: Proof of Concept and Future Directions 263
7.1 Motivating Hypotheses and Scope . . . 263
7.1.1 The Seven Hypotheses . . . 264
7.1.2 Research Scope . . . 264
7.1.3 From Breadth to Depth . . . 265
7.2 From Substrate Engineering to Composition . . . 265
7.3 Biological Inspiration . . . 265
7.3.1 Columnar Organization: The Extracted Principle . . . 265
7.3.2 Emergent Modularity Under Evolutionary Pressure . . . 266

7.3.3 From Mimicry to Emergence . . . 267
7.4 GEENNS Algorithm Design . . . 268
7.4.1 Core Architectural Principles . . . 268
7.4.2 The Columnar Substrate . . . 269
7.4.3 Mappers as Compositional Coordinators . . . 270
7.5 Why the Substrate Problem Must Be Solved First . . . 271
7.5.1 Barrier 1: Computational. ES-HyperNEAT Cannot Scale . . . 272
7.5.2 Barrier 2: Expressiveness. Substrates Lack Function Diversity . . . 273
7.5.3 Barrier 3: Multi-Task. Substrates Cannot Handle Multiple Tasks . 273
7.5.4 The Three Barriers as Thesis Structure . . . 274
7.6 Prototype Evidence . . . 275
7.6.1 Experimental Design . . . 275
7.6.2 What Works . . . 279
7.6.3 Scaling Behavior: Composition vs. Baselines . . . 284
7.6.4 The Confounds . . . 286
7.6.5 Beyond Boolean: Continuous Gate Composition . . . 287
7.6.6 Synthesis: What the Prototype Answers . . . 289
7.6.7 Limitations and Open Questions . . . 289
7.7 Chapter Summary . . . 291

V Conclusion 293

8 Conclusion and Future Work 295
8.1 Contributions Summary . . . 295
8.2 Research Questions Revisited . . . 298
8.3 Limitations and Lessons Learned . . . 300
8.3.1 Thesis-Level Limitations . . . 300
8.3.2 Failed Approaches and Negative Results . . . 302
8.3.3 Lessons for Neuroevolution Researchers . . . 303
8.3.4 Threats to Validity . . . 304
8.3.5 Reproducibility and Computational Resources . . . 304
8.4 Future Work: What EMR Enables . . . 305
8.4.1 Research Directions Enabled by EMR . . . 305
8.4.2 The GEENNS Vision . . . 307
8.4.3 Scaling to Complex Problems . . . 308
8.5 Closing Reflection . . . 309

A Research Evolution and Philosophical Framework 319
A.1 Biological Foundations of the Original Proposal . . . 319
A.2 Research Evolution . . . 320
A.2.1 Grounded Symbolic Reasoning . . . 320
A.2.2 The Pivot to Neuroevolution . . . 321
A.2.3 Doc.Mobility and the 48-Pixel Discovery . . . 321
A.2.4 New Research Directions . . . 322
A.2.5 The EMR Year . . . 323
A.3 Connection to the Original Proposal . . . 324
A.3.1 Reflecting on the Journey . . . 325
A.4 Historical Note: Six Stages in the Evolution of Collaboration . . . 325

B GEENNS Formal Specification **327**
B.1 Substrate Evolution Landscape and CoDeepNEAT Comparison 327
B.2 Why Evolution Determines Architecture 330
B.3 Architectural Detail: From Network to Node 330
B.4 Multi-Level CPPN Connectivity 334
B.5 The Developmental Process 337
B.6 Plasticity Mechanisms 338
B.7 GEENNS among Continual Learning Approaches 339
B.8 Formal Mathematical Specification 341
B.8.1 Grid Formalism 342
B.8.2 State Dynamics 342
B.8.3 Mapper Formalism 343
B.8.4 CPPN Connectivity Generation 343
B.8.5 Development Function 344
B.8.6 Multi-Task Fitness 344

C Supplementary Background Material **345**
C.1 Parity Truth Tables and Scaling 345
C.2 GPU SIMT Architecture 345
C.3 GPU Memory Hierarchy 346
C.4 Hyperparameter Search Methods 347
C.4.1 Random Search 347
C.4.2 Bayesian Optimization 347
C.5 ES-HyperNEAT Hyperparameter Space 347
C.6 ES-HyperNEAT Division and Pruning Walkthrough 349
C.7 Speciation and CPPN Complexification Dynamics 349
C.8 MoE Historical Development 349
C.9 Neuroevolution and the NAS Taxonomy 352
C.10 Extended Benchmark Definitions 352
C.10.1 Visual Discrimination 353
C.10.2 Retina Problem 353
C.10.3 Orientation Selectivity 353
C.10.4 Two Spirals 353
C.10.5 Concentric Circles 354
C.10.6 Symmetry Detection 354
C.10.7 Turing Patterns 354
C.10.8 Tower of Hanoi 354
C.10.9 Reinforcement Learning Control 355
C.10.10CCM Financial Data 355
C.10.11Earnings-Price Regression 355
C.10.12California Housing 356
C.11 Statistical Methodology 356
C.11.1 Convergence Criterion Justification 356
C.11.2 Experimental Design 356
C.11.3 Confidence Intervals 357
C.11.4 Hypothesis Testing 357
C.11.5 Tests for Proportions 357
C.11.6 Why Non-Parametric Tests 357

C.11.7 Worked Example . . . 357
C.11.8 Effect Sizes . . . 358
C.11.9 Multiple Comparisons . . . 358
C.11.10ANOVA . . . 358
C.11.11Tukey HSD . . . 358
C.11.12Reporting Format . . . 359
C.12 JAX API Transformations . . . 359
C.13 JAX-ESHN Implementation Algorithms . . . 359

D Supplementary Experimental Material **361**
D.1 Expert Density versus Exploitation (Chapter 4) . . . 361
D.2 Cross-Framework Positioning (Chapter 6) . . . 361
D.3 Prototype Mechanism Detail (Chapter 7) . . . 362
D.3.1 Gated Selective Routing: Mechanism . . . 362
D.3.2 Continuous-Gate Protocol and Operator Analysis . . . 363
D.3.3 Warm-Start Transfer and the Latent-Solver Census . . . 364

List of Figures

1.1 Thesis structure showing parts, chapters, barriers, and papers 53

2.1 Generational evolutionary algorithm lifecycle 57
2.2 Sparse neural network topology evolved by NEAT 57
2.3 NEAT crossover alignment by innovation number 59
2.4 One-dimensional profiles of five standard CPPN activation functions 62
2.5 HyperNEAT overview with CPPN-generated substrate connectivity 64
2.6 ES-HyperNEAT overview with quadtree-based topology discovery 68
2.7 ES-HyperNEAT substrate construction from CPPN via quadtree decomposition 69
2.8 ES-HyperNEAT evolutionary loop with per-individual substrate construction 70
2.9 DES-HyperNEAT overview with multi-layer substrate discovery 71
2.10 Algorithmic lineage from NEAT to GEENNS 72
2.11 TPE Parzen estimation density contours for good and bad configurations . 77
2.12 TPE candidate selection via density ratio maximization 78

3.1 TPE optimization trajectory over 2,013 MNIST trials 97
3.2 Per-seed learning curves over 100 generations 100
3.3 Per-digit accuracy and output collapse visualization 102
3.4 Per-generation wall-clock time scaling over 100 generations 104
3.5 Population genetic distance divergence over 100 generations 105
3.6 Accuracy versus cumulative compute over 100 generations 105
3.7 Multi-configuration convergence to the same accuracy band 106
3.8 Hyperparameter pairwise correlation heatmap 107
3.9 Sample images from MNIST and Fashion-MNIST datasets 112
3.10 Fitness trajectory divergence between successful and unsuccessful trials . . 119
3.11 Pruning threshold optimization at generation 3 120
3.12 ROC curve for the pruning rule with validation operating points 122
3.13 Decision boundary scatter for the derivation set 123
3.14 Fitness trajectories of the eight false-negative slow-starter trials 124
3.15 Bimodal distribution of final MNIST accuracy across TPE trials 128
3.16 Simulated pruner effect on late-stage TPE search progress 131
3.17 Late-stage false-negative trajectories in converged search space 132
3.18 Combined TPE and early-stopping optimization pipeline 135

4.1 Input partitioning scheme for four-expert partitioned-experts on MNIST . 143
4.2 Mean test accuracy versus expert count 150
4.3 Per-expert fitness distributions across expert counts 152
4.4 Per-expert fitness convergence curves across selected expert counts 153

4.5 Aggregation strategy comparison across experimental conditions 154
4.6 Prediction distribution by digit class for monolithic and partitioned experts 155
4.7 Prediction class diversity over generations for monolithic and partitioned experts 156
4.8 Per-expert prediction specialization heatmap at thirteen experts 157
4.9 Receptive field comparison: monolithic, replication, and partitioned-experts ensemble 158
4.10 Per-generation receptive field expansion for a three-expert trial 159
4.11 Composite receptive field expansion with increasing expert count 163

5.1 ES-HyperNEAT system architecture with sequential substrate discovery bottleneck 178
5.2 Quadtree subdivision process for hidden node discovery 179
5.3 HSHG spatial hash grid with constant-time neighbor queries 186
5.4 HSHG over-discovery mechanism compared to quadtree convergence 188
5.5 Solve rate heatmaps comparing Baseline and JAX-ESHN by depth 192
5.6 JAX-ESHN convergence speed heatmap with gen-1 solve region 193
5.7 Total runtime comparison between Baseline CPU and JAX-ESHN GPU . . 196
5.8 JIT compilation time scaling with depth and population 197
5.9 JIT amortization ratio across depths and populations 198
5.10 Post-JIT per-generation time heatmap across depth and population 203
5.11 Per-generation speedup ratio of Baseline over JAX-ESHN 204

6.1 Lazy versus eager substrate discovery flowcharts 213
6.2 Isometric visualization of lazy versus eager substrate discovery 214
6.3 Hierarchical multi-resolution grid structure 215
6.4 Hierarchical variance thresholding with bottom-up and top-down passes . . 217
6.5 Variance thresholding pipeline with isometric grid illustration 218
6.6 EMR connection type primitives and classification criteria 222
6.7 EMR memory management architecture with GPU-CPU offloading 225
6.8 Hidden-to-hidden cache impact on generation time 227
6.9 Cross-framework positioning map 231
6.10 Cross-framework runtime versus depth 235
6.11 Population–depth solve rate heatmap comparing ES-HN and EMR 237
6.12 Speedup ratio heatmap by population and depth 239
6.13 Runtime over 30 generations across depths and backends 242
6.14 Connection type–depth solve rate heatmap for XOR 247
6.15 Memory strategy boundary by depth and population 252
6.16 GPU VRAM versus CPU RAM scaling by depth 253

7.1 Columnar principle in neocortical and GEENNS organization 267
7.2 Monolithic versus compositional substrate design 268
7.3 End-to-end compositional routing through frozen specialist grids 271
7.4 Substrate coordinate layout for monolithic and compositional conditions . . 277
7.5 Success rates across all prototype conditions 281
7.6 Convergence curves for composition and baselines across grid scales 285

B.1 Hand-designed versus evolutionary architecture search 331
B.2 Full GEENNS architecture with mappers and columnar network 332

B.3 GEENNS intra-column architecture with four CPPN levels 333
B.4 Grid computation walkthrough showing injection, propagation, and extraction 334
B.5 GEENNS minicolumn architecture with layered substrates 335
B.6 Five-stage genome-to-grid developmental pipeline 338
B.7 GEENNS positioned among continual learning methods by isolation and flexibility 341

C.1 ES-HyperNEAT division and pruning on a banded weight landscape 350
C.2 Speciation dynamics and CPPN complexification over 100 generations . . . 351

D.1 Gated selective routing: evolved per-column sigmoid gates 363
D.2 Continuous gate composition: the operator defines the target 364
D.3 Continuous gates are the smooth extensions of Boolean truth tables 365
D.4 Warm-start transfer: the mechanic 366
D.5 Warm-start transfer at 3-grid scale: latent solvers are the converged elite . 367

List of Tables

1.1 Quantified thesis contributions organized by research question 49
1.2 Experimental scale of this thesis by chapter 52

2.1 CPPN activation function properties and central bias behavior 62
2.2 Truth tables for six Boolean logic benchmark tasks 83
2.3 Comparison of cell-level and full-topology architecture search approaches . 88

3.1 ES-HyperNEAT sixteen-parameter search space 94
3.2 MNIST accuracy for random search versus TPE optimization 96
3.3 Comparison with published ES-HyperNEAT and HyperNEAT MNIST results 96
3.4 Welch's t-tests for MNIST accuracy comparisons 98
3.5 Varying hyperparameters across the five optimal configurations 98
3.6 Accuracy metric disambiguation for the best configuration 99
3.7 Diminishing accuracy returns by generation window 101
3.8 Per-seed convergence velocity statistics 101
3.9 Runtime and cost-efficiency ratio by generation window 104
3.10 Converged hyperparameters constant across optimal configurations 108
3.11 Logic operations accuracy with transferred MNIST hyperparameters 111
3.12 Welch's t-tests for logic operation transfer significance 111
3.13 Fashion-MNIST accuracy with transferred MNIST hyperparameters 113
3.14 Extended hyperparameter search space for pruner derivation 117
3.15 Sensitivity analysis for success-threshold percentile selection 117
3.16 Paired t-tests comparing median best and last best fitness metrics 118
3.17 Pruner confusion matrix on the validation set 121
3.18 Pruning rule classification performance on the validation set 121
3.19 Pruning generation robustness at adjacent checkpoints 125
3.20 Threshold robustness analysis across perturbation range 126
3.21 Domain-specific pruner versus Hyperband and unpruned baseline 127
3.22 Tukey HSD post hoc comparisons of mean final fitness 129
3.23 TOST equivalence test between pruner and unpruned baseline 129
3.24 Counterfactual pruner simulation on late-stage converged trials 130
3.25 Summary of Chapter 3 key results 139

4.1 Experimental conditions and control factors 146
4.2 Mean accuracy by architecture with Performance-Weighted aggregation . . 147
4.3 Detailed performance statistics per condition at optimal expert count . . . 148
4.4 Performance with naive average aggregation across conditions 149
4.5 Key pairwise statistical tests with effect sizes 150
4.6 Summary of Chapter 4 key results 170

List of Tables

5.1 Cumulative optimization speedups for quadtree substrate discovery 182
5.2 HSHG versus Quadtree theoretical comparison 188
5.3 HSHG evolutionary pipeline solve rates vs. Quadtree Baseline 189
5.4 Baseline total run time by depth and population 194
5.5 JIT compilation time by depth and population 194
5.6 JAX-ESHN total run time projected to a 30-generation budget by depth and population 195
5.7 Multi-benchmark validation of the quadtree scaling exponent 200
5.8 Steady-state per-generation comparison of Baseline and JAX-ESHN at D4 201
5.9 Summary of Chapter 5 key results 209

6.1 Hierarchical grid position counts for depths 1–13 215
6.2 Computational complexity comparison of ES-HyperNEAT and EMR 217
6.3 Six substrate connection configurations 222
6.4 Hidden-to-hidden cache impact on steady-state performance 227
6.5 Benchmark problems used in EMR evaluation 229
6.6 Average time per generation for XOR at population 1000 230
6.7 Cross-framework positioning under a single measurement protocol 233
6.8 Cross-framework positioning on GPU: the two tensor arms 234
6.9 Statistical comparison of EMR and ES-HyperNEAT generation times . . . 236
6.10 Population–depth performance map for XOR 238
6.11 EMR timing at extreme depths with memory streaming 241
6.12 Per-generation time on CCM financial dataset 243
6.13 Cross-domain validation across 15 problems in four domains 244
6.14 Connection type comparison on XOR with weight threshold effects 247
6.15 Cross-problem solve rates by connection type 249
6.16 JIT amortization breakeven generations by population and depth 251
6.17 Capabilities before and after EMR reformulation 256
6.18 Summary of Chapter 6 key results 260

7.1 Research tasks and thesis disposition 266
7.2 Mapping from GEENNS barriers to thesis contributions 274
7.3 Shared EMR-HyperNEAT substrate parameters for prototype conditions . 276
7.4 Population and generation budgets by experimental scale 278
7.5 Input dimensionality and truth-table size by grid scale 278
7.6 GEENNS prototype experiment summary by research question 280
7.7 Success rates and convergence speed across 1–5 grid scales 284

8.1 At-a-glance resolution of the three barrier clusters 296
8.2 Quantified thesis contributions with statistical evidence 297
8.3 Summary of thesis limitations by chapter 300
8.4 Computational resources by chapter 305
8.5 Research directions enabled by EMR-HyperNEAT 307

A.1 Mapping between original proposal modules and thesis realization 324

B.1 Substrate evolution approaches compared against GEENNS requirements . 329
B.2 Multi-level CPPN hierarchy for GEENNS connectivity 336
B.3 Speculative wall-clock envelope for a GEENNS hyperparameter sweep . . . 337
B.4 GEENNS compared with continual learning and modular approaches . . . 340

C.1 Parity-3 truth table 345
C.2 Parity-N input pattern scaling and output alternation 346
C.3 ES-HyperNEAT hyperparameters optimized via TPE 348

List of Publications

The work presented in this thesis is based on the following publications and preprint:

Paper 1 (Published — Chapter 3)
R. Claret, M. O'Neill, P. Cotofrei, and K. Stoffel. "Investigating Hyperparameter Optimization and Transferability for ES-HyperNEAT: A TPE Approach." In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '24 Companion)*, July 14–18, 2024, Melbourne, VIC, Australia. ACM, 2024, 9 pages. `https://doi.org/10.1145/3638530.3664144`. Open access: `https://claret.tech/pdf/claret2024tpe`

Paper 2 (Published — Chapter 3)
R. Claret, A. Gygax, M. O'Neill, P. Cotofrei, and P. Felber. "Early-Stopping Thresholds for ES-HyperNEAT: A Data-Driven Approach from Fitness Dynamics." In *Proceedings of the 2026 IEEE Congress on Evolutionary Computation (CEC), part of IEEE WCCI 2026*, June 21–26, 2026, Maastricht, Netherlands. IEEE, 2026, 8 pages. Open access: `https://claret.tech/pdf/claret2026pruner`

Paper 3 (Published — Chapter 4)
R. Claret, A. Gygax, M. O'Neill, P. Cotofrei, M. Palma Mendes, and P. Felber. "Breaking the Central Bias: Spatially Partitioned Experts for Coordinate-Based Neuroevolution." In *Pattern Recognition. ICPR 2026 International Workshops and Challenges (BIOMAP)*. Springer, 2026, 15 pages. Open access: `https://claret.tech/pdf/claret2026partitioned`

Paper 4 (Preprint — Chapter 5)
R. Claret, M. O'Neill, P. Cotofrei, and K. Stoffel. "On Scaling Coordinate-Based Neuroevolution: The Quadtree Bottleneck in ES-HyperNEAT." Preprint, arXiv:2608.24480 [cs.NE], 2026, 27 pages. `https://doi.org/10.48550/arXiv.2608.24480`. Open access: `https://claret.tech/pdf/claret2026quadtree`

Paper 5 (Published — Chapter 6)
R. Claret, M. O'Neill, P. Cotofrei, and K. Stoffel. "Tensor-Accelerated Eager Multi-Resolution Grids for Evolving Large-Scale Substrates." In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '26 Companion)*, July 13–17, 2026, San Jose, Costa Rica. ACM, 2026, 4 pages. `https://doi.org/10.1145/3795101.3805361`. Open access: `https://claret.tech/pdf/claret2026emr`

Author contributions. Romain Claret is the first author on all five papers. He conceived all research questions, designed all algorithms and experimental protocols, implemented the codebase used across all studies, performed or verified the statistical analyses, and wrote all manuscripts. Arthur Gygax contributed to Papers 2 and 3; he implemented the experiments, executed the experimental campaigns, collected the experimental data, performed statistical analyses, and contributed to the initial drafts under Claret's direction, using Claret's codebase, pre-written experiment scaffolding, and figure and table drafts. Papers 2 and 3 carry an equal-contribution notice for R. Claret and A. Gygax. Michael Palma Mendes contributed to Paper 3 by assisting with statistical review. Michael O'Neill supervised the Doc.Mobility period at University College Dublin and provided guidance and review on all five papers. Paul Cotofrei co-supervised the thesis and provided guidance and review on all five papers. Kilian Stoffel co-supervised the thesis. All co-authors reviewed and approved the final versions of the manuscripts.

Notation and Symbols

Acronyms

Acronym	Definition
ANOVA	Analysis of Variance
CCM	CRSP/Compustat Merged (financial dataset)
CI	Confidence Interval
CPPN	Compositional Pattern-Producing Network
DES-HyperNEAT	Deep Evolvable-Substrate HyperNEAT
EMR	Eager Multi-Resolution (HyperNEAT variant)
ES-HyperNEAT	Evolvable-Substrate HyperNEAT
GEENNS	Grid-based Emergent Evolution of Neocortical Network Substrates
GPU	Graphics Processing Unit
HSD	Honestly Significant Difference (Tukey)
HSHG	Hierarchical Spatial Hash Grid
HyperNEAT	Hypercube-based NEAT
JAX	Google's Autograd + XLA numerical computing library
JIT	Just-In-Time (compilation)
MNIST	Modified National Institute of Standards and Technology (dataset)
MoE	Mixture of Experts
NAS	Neural Architecture Search
NEAT	NeuroEvolution of Augmenting Topologies
ROC	Receiver Operating Characteristic
SIMT	Single Instruction, Multiple Threads
TOST	Two One-Sided Tests (equivalence testing)
TPE	Tree-structured Parzen Estimator
TRQ	Thesis Research Question
VRAM	Video Random-Access Memory (GPU memory)
XLA	Accelerated Linear Algebra (compiler)

Mathematical Symbols

Symbol	Definition
θ	Variance threshold governing quadtree subdivision and EMR filtering (written $\theta_{\text{div}}/\theta_{\text{var}}$ where the two phases are distinguished; θ_{band} is the band-membership threshold; further subscripts are defined at point of use); in the algorithm listings of Chapter 5 and Appendix C (and their QueryCPPN convention), the CPPN genome passed to each routine
D	Depth of quadtree or EMR hierarchical grid
ϕ	Fitness value
ϕ^*	Convergence threshold (default 0.95)
γ	TPE quantile threshold (Chapter 2); in Chapter 6, locally, the multi-hop weight-decay factor
$w_{\min}$	Minimum connection weight threshold
N	Population size in the evolutionary algorithm
N_e	Number of experts in the partitioned-experts architecture (Ch. 4)
G	Generation number
G^*	Pruning generation (generation at which early stopping is evaluated)
T^*	Pruning fitness threshold
F_1	Harmonic mean of precision and recall
d	Cohen's d effect size (standardized mean difference)
r	Rank-biserial correlation (non-parametric effect size)
η^2	Eta-squared (proportion of variance explained)
δ	Cliff's δ (non-parametric effect size for ordinal data); in Chapter 2, NEAT's compatibility distance (threshold δ_t) and, locally, Gaussian mutation noise
$\mathcal{C}$	CPPN (pattern-generating network)
$f(\cdot)$	Activation function
f'_i	Adjusted fitness (fitness sharing: $f_i/\lvert S_j\rvert$)
S_j	Species j
n_j	Offspring allocation for species j
δ_t	Compatibility threshold (NEAT speciation)
τ	Survival threshold (truncation fraction); in Chapter 3, locally, the candidate pruning threshold swept to obtain T^*; in Appendix B, the task index
P	Number of GPU cores / parallel units; in Chapter 2, locally, the population set
$\mathbf{x}$	Vector (bold lowercase)
$\mathbf{W}$	Matrix (bold uppercase); substrate weight matrix when subscripted
$\mathcal{S}$	Set (calligraphic)
$\mathbb{1}[\cdot]$	Indicator function (1 if condition is true, 0 otherwise)

Conventions

- Statistical significance is reported at the $\alpha = 0.05$ level unless otherwise stated.
- Confidence intervals are 95% bootstrap intervals with 10,000 resamples unless otherwise noted.
- Effect sizes follow Cohen's conventions: small ($d = 0.2$), medium ($d = 0.5$), large ($d = 0.8$).
- Experimental results use the median as the measure of central tendency unless otherwise stated (robust to outliers in evolutionary fitness distributions).
- Success rates are reported as a percentage with the underlying sample size; significance is assessed with non-parametric tests by default (Mann-Whitney U for two-group comparisons, Kruskal-Wallis for multi-group designs), while the campaigns of Chapters 3–5 also report parametric tests (Welch's t, one- and two-way ANOVA with Tukey HSD) where their protocols used them.

Part I

Foundations

Chapter 1

Introduction

> The best prophets lead you up to the curtain and let you peer through for yourself.
>
> Frank Herbert

Deep learning has transformed artificial intelligence. In the span of a decade, gradient-trained networks have achieved superhuman performance on image classification [52], protein folding [46], machine translation [97], and language generation [8]. Yet all of these systems share a structural assumption so pervasive it is rarely questioned: a human designs the architecture, and the machine learns only the weights. The topology (how many layers, which connections exist, what activation functions to use) remains frozen from the moment an engineer decides on it. This division of labor has proven productive, but it also imposes a ceiling. The architecture encodes human intuition about what computation should look like, and human intuition is not always right.

Neuroevolution offers an alternative. Rather than fixing the topology and optimizing weights through gradient descent, evolutionary algorithms co-discover topology *and* weights simultaneously. The resulting artificial neural networks, which we call substrates, are optimized for the task, not for human readability. This idea dates back to the early 1990s [105] and has been repeatedly demonstrated on problems where hand-designed architectures struggle: reinforcement learning with sparse rewards, multi-objective optimization, and open-ended problem solving [91]. The most influential lineage—NeuroEvolution of Augmenting Topologies (NEAT) [88], HyperNEAT [90], and ES-HyperNEAT [78]—progressively introduced indirect encoding through Compositional Pattern-Producing Networks (CPPNs), enabling compact genomes to generate large-scale substrates with geometric regularities.

The promise of neuroevolution is real: a network whose topology is discovered by evolution rather than imposed by a designer. The practical utility is not: even the best adaptive variant falls far short of gradient-trained networks on standard benchmarks.

Verbancsics and Harguess [98] achieved 23.9% accuracy on MNIST using HyperNEAT with a generic seven-layer fully-connected substrate, or 27.7% with a hand-designed LeNet-5 CNN substrate [57], in both cases using a population of 256 and 2,500 generations. Our ES-HyperNEAT work, which discovers its own substrate rather than relying on a task-specific hand design, achieves 29.0% on MNIST using a population of 100 and 20 generations, but this comes at a cost. ES-HyperNEAT exposes roughly 40 tunable

hyperparameters in the implementation used in this work: about 30 control NEAT's core mutation dynamics (weight, bias, and response rates, compatibility, speciation, reproduction) and 6 are specific to ES-HyperNEAT (initial and maximum quadtree depth, division and variance thresholds, band cutoff, iteration level). Even a restricted 16-parameter search subspace, with the ranges we adopt, exceeds three billion configurations, over 90% of which produce substrates that stagnate at random guessing, with little reason to expect the unexplored remainder to fare meaningfully better. Its substrates cover only approximately 28 of 784 input pixels, ignoring over 96% of the available information. And its quadtree-based substrate discovery is fundamentally sequential: a single experiment requires hours to days on CPU, and the algorithm's irregular output cannot be accelerated on massively parallel hardware.

These are not minor engineering inconveniences; they are structural barriers that prevent adaptive substrate neuroevolution from being used as a practical tool for scientific investigation or real-world application. This thesis systematically diagnoses, characterizes, and addresses them.

1.1 The Case for Evolved Neural Architectures

Three approaches exist for constructing neural architectures, and they differ in what is determined by evolution or learning versus what is fixed by human decision.

Hand-designed architectures reflect the engineer's intuition about the problem. For example, convolutional neural networks encode the assumption that spatial locality matters [57], transformers encode the assumption that attention over sequences matters [97], and recurrent architectures encode the assumption that temporal dynamics matter [39]. In each case, the topology is frozen before training begins, and the optimizer adjusts only the weights within that frozen structure. The success of these designs is undeniable, but it comes with a cost: every architectural choice is a human bias imposed on the solution space. In effect, the engineer brute-forces data into a predetermined topology, trusting that the structure is flexible enough to accommodate the problem's computational requirements. When the bias is correct, the architecture converges quickly. When the bias is wrong, or when the problem does not match any standard template, the architecture fails in ways that no amount of weight optimization can repair.

The extent of this bias is rarely acknowledged. A ResNet [37] imposes skip connections every two layers, but why every two layers? A Vision Transformer [23] divides images into 16×16 patches, but why that granularity? These choices are not derived from the problem structure; they are drawn from human engineering experience and validated empirically. They work well on ImageNet, but there is no guarantee they are optimal, and no mechanism to discover whether a fundamentally different topology would perform better.

Neural Architecture Search (NAS), a core technique within Automated Machine Learning[1] (AutoML), partially addresses this limitation by automating the selection among a predefined set of architectural components [27, 108]. NAS treats architecture design as an optimization problem and searches over cell structures, connection patterns, and hyperparameters, producing architectures that match or exceed hand-designed ones on standard benchmarks. However, the search space itself remains hand-designed: the

[1] AutoML is the broader effort to automate the end-to-end machine learning pipeline from data preparation to model deployment.

building blocks (convolutions, pooling, attention), the connectivity constraints (sequential layers, residual connections), and the evaluation protocols are all human-designed. NAS optimizes *within* a human-defined space instead of discovering fundamentally new computational structures. Section 2.6 provides a detailed comparison.

Evolved architectures operate differently. NEAT, introduced by Stanley and Miikkulainen [88], co-evolves both the topology and the weights of a neural substrate through mutation and crossover, guided by speciation to protect structural innovations. The algorithm makes no assumption about the number of layers, the presence or absence of recurrence, the connectivity pattern, or the activation functions. These properties emerge from evolutionary pressure. The resulting substrates often look nothing like human-designed networks (they exhibit irregular connectivity, unexpected recurrent loops, and non-standard layer organization), yet they solve the task. NEAT has been successfully applied to game playing [89], evolutionary robotics [31], and a range of benchmark problems where the optimal topology is unknown in advance.

These evolved architectures point to something deeper than automated engineering. When evolution discovers computational structures unlikely to occur to a human designer, and those structures solve the task, the result is a form of emergent problem-solving: the solution arises from evolutionary dynamics, not explicit programming. The benchmarks across this thesis (Boolean logic, image classification, regression, and control) are proxies for a more fundamental capability: the capacity for evolved substrates to produce behaviors that were not designed into them, only selected for. Each contribution in this thesis, among them the tensor reformulation of substrate discovery, full input coverage, and cross-domain generalization, brings neuroevolution closer to the computational scale and representational flexibility at which such emergent behaviors become possible to investigate systematically.

The key limitation of NEAT is that it uses *direct encoding*: every connection in the substrate is explicitly represented in the genome. This creates a linear relationship between genome size and substrate complexity, making large-scale substrates impractical. A substrate with 10,000 connections requires a genome with 10,000 connection genes, and the evolutionary search space grows combinatorially with genome size.

Why indirect encoding is meaningful. The direct-encoding approach writes every detail of an individual into its genome. Biology does not. A human genome contains roughly three billion base pairs; a human adult contains on the order of 10^{14} synapses. The genome does not list them. It encodes a developmental program that, interpreted by cellular machinery, unfolds into the phenotype. DNA is not a blueprint of the body; it is a compact generative rule from which the body is expressed. This asymmetry—orders of magnitude between description and expression—is the feature of life that makes scale tractable, and it is the feature that direct encoding throws away. Indirect encoding in neuroevolution reconstructs this asymmetry computationally: a small generative rule produces a large structured substrate, with the regularities of the substrate emerging from the rule rather than being written into it one element at a time. The motivation for focusing on indirect encoding in this thesis is not a preference between two software designs. It is the hypothesis that scalable, adaptive computation requires the same compression asymmetry that biology discovered, and that the engineering difficulties documented in the chapters that follow (hyperparameter expense, representational collapse, computational bottleneck) are the cost of learning to work in that regime rather than arguments against it.

The indirect encoding lineage in neuroevolution resolved this limitation. HyperNEAT [90] uses a CPPN (a small neural network with geometric activation functions such as sine, Gaussian, and linear) to generate connection weights as a function of the spatial coordinates of source and target neurons. Unlike NEAT, HyperNEAT requires the user to predefine the substrate topology (the positions and arrangement of neurons) and evolves only the CPPN. A compact CPPN genome of perhaps 50 genes can thereby specify millions of connections with spatial regularity, exploiting the same geometric patterns (symmetry, repetition, gradient) that characterize biological neural connectivity. The separation between genotype (CPPN) and phenotype (substrate) enables a form of developmental encoding: the genome does not specify the substrate directly but rather specifies a *rule* for constructing it.

Risi and Stanley [78] extended HyperNEAT with adaptive substrate discovery (ES-HyperNEAT). Where HyperNEAT requires the user to predefine the neuron positions (typically on a fixed grid), ES-HyperNEAT determines neuron placement automatically by querying the CPPN and placing neurons where the output variance is highest. A quadtree algorithm recursively subdivides the coordinate space, concentrating resolution where the CPPN encodes complex patterns and leaving sparse regions where the pattern is simple. The result is substrates whose topology adapts to the problem structure, a capability that distinguished ES-HyperNEAT from earlier CPPN-based methods.

Tenstad and Haddow [94] added evolved depth to this framework (DES-HyperNEAT) by extending ES-HyperNEAT to multi-substrate architectures: multiple ES-HyperNEAT substrates stacked and evolved jointly, enabling deep indirect-encoded networks whose layering is itself a product of evolution. Each step in this lineage brought neuroevolution closer to the ideal of fully automatic architecture discovery: topology, weights, neuron placement, and network depth all determined by evolution rather than engineering. The ideal is asymptotic, not achieved: the human designer never leaves the loop, choosing the encoding's primitives, the fitness function, and the search's hyperparameters; what changes along the lineage is how far downstream of those choices the architecture itself is decided.

Why not gradient descent? A natural question is whether the problems addressed in this thesis (substrate discovery, function selection, multi-resolution evaluation) could be solved with gradient-based methods instead of evolution. Four factors motivate the evolutionary approach. First, ES-HyperNEAT's substrate discovery is inherently discrete: the quadtree decides where to place nodes based on CPPN variance, a non-differentiable operation. Second, the search space includes topology (number of nodes, connectivity pattern) alongside weights, which gradient descent cannot explore without differentiable relaxations that introduce their own limitations [61]. Third, the indirect encoding through CPPNs exploits geometric regularities that gradient methods applied directly to weight matrices cannot capture. Fourth, the choice space extends beyond structure to function, where the same argument applies with more force: which activation a node computes is a discrete assignment with no derivative through the choice, and companion work on per-node function evolution (Section 8.4) finds the assignment can be the binding constraint. In evolutionary search over sparse feedforward substrates, every uniform monotonic activation tested (tanh, sigmoid, ReLU) yields 0% solve rates at parity-4 (0.95 solve criterion), uniform tanh remains at 0% through parity-8, and oscillatory functions reach 100% throughout. When the deciding variable is which function to compute rather than which weights to assign, gradient methods have nothing to differentiate; evolution is the search

mechanism that remains. Section 8.4.1 outlines a hybrid direction (evolving topology via Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT), then training weights via gradient descent), suggesting the approaches are complementary rather than competing.

The gap between this ideal and current practice is the subject of this thesis. ES-HyperNEAT's adaptive substrate discovery is the most principled mechanism available for reducing human bias in evolved architectures (though not eliminating it: Chapter 4 traces a residual bias to the encoding itself), but three clusters of barriers prevent it from scaling beyond toy problems. We characterize these barriers next.

1.2 ES-HyperNEAT: Powerful but Impractical

ES-HyperNEAT's key innovation is *adaptive substrate discovery*: the algorithm determines not only the connection weights but also *where* to place neurons, based on the information density encoded in the CPPN. This distinguished it from earlier CPPN-based methods: HyperNEAT requires a fixed substrate layout, but ES-HyperNEAT discovers the layout automatically. The result is substrates whose topology matches the problem structure, concentrating neurons where the CPPN encodes complex patterns and leaving sparse coverage where the pattern is simple.

This adaptivity is ES-HyperNEAT's greatest strength and, simultaneously, the source of its three most severe limitations. We organize these limitations into three barrier clusters (groups of interrelated limitations sharing a common structural root cause), each requiring a fundamentally different type of solution.

1.2.1 Barrier Cluster 1: The Configuration Expense

ES-HyperNEAT's behavior depends on a large set of interacting hyperparameters. The CPPN encoding requires architectural choices: the number of hidden nodes and the available activation functions. NEAT, the evolutionary algorithm that evolves the CPPN, contributes population size, species compatibility threshold, crossover rates, and the various mutation rates governing how the CPPN's topology and weights change over generations. The quadtree requires an initial resolution, a maximum subdivision depth, a variance threshold θ that determines when to subdivide, and a band-pruning threshold that filters connections. The substrate parameters include connection weight range and bias handling. The combinatorial space formed by interacting hyperparameters—67 configurable keys in total (59 in `neat-python`, eight substrate-side), of which roughly 40 are genuinely tunable—makes it prohibitively difficult to search or even quantify the full set of possible configurations.

Without systematic optimization, a practitioner selecting hyperparameters by hand or by grid search has almost no chance of finding a working configuration. The problem is compounded by evaluation cost: each hyperparameter trial requires a full evolutionary run (20 generations in the MNIST campaign of Chapter 3), which on CPU takes minutes to hours depending on problem size. A 2,013-trial optimization sweep on MNIST required months of continuous computation.

This barrier is *practical*, not fundamental. It can be addressed through principled hyperparameter optimization and cost-reduction strategies, but the cost of addressing it is itself substantial, requiring thousands of optimization trials before a single productive experiment can begin.

1.2.2 Barrier Cluster 2: The Representational Collapse

Even with optimal hyperparameters, ES-HyperNEAT substrates collapse to a small fraction of the available input space, a phenomenon we term the *central bias*. The mechanism is geometric and inherent to the algorithm's design. The quadtree must start its recursive search from some coordinate: by default the origin, though a practitioner could shift it or sample one at random. Because the optimal starting point is unknown in advance (the whole purpose of the search is to discover where the task's salient features lie), the default anchors the CPPN's evolutionary search. Most of the CPPN's standard activation functions (Gaussian, sigmoid, tanh) concentrate their output variance near that anchor, and the quadtree allocates resolution proportionally. The result is that neurons cluster near the starting point regardless of the problem, leaving peripheral regions unrepresented. What appears to be a learning failure is actually an information failure: the substrate does not have access to the data it would need to succeed.

This representational collapse is not a hyperparameter problem. No amount of Tree-structured Parzen Estimator (TPE) optimization, no choice of variance threshold, and no adjustment of quadtree depth resolves it. The collapse is structural: a monolithic ES-HyperNEAT substrate on a high-dimensional input will always concentrate its resolution on a small fraction of the input space.

The consequence is a hard performance ceiling. Our best monolithic ES-HyperNEAT result on MNIST (29.0% accuracy after 2,013 optimization trials and 30-run validation) reflects the information content of approximately 28 pixel positions, not the information content of the full 784-pixel image.

1.2.3 Barrier Cluster 3: The Computational Bottleneck

ES-HyperNEAT's substrate discovery relies on a quadtree algorithm that recursively subdivides the input-output coordinate space. At each level of the tree, the algorithm queries the CPPN at the center of each cell, measures the variance of the output across the four child cells, and subdivides only those cells where variance exceeds the threshold θ. This process is inherently sequential in two ways: first, each subdivision decision depends on the result of the previous query (the tree cannot be expanded at level $k+1$ without knowing which cells survived at level k); and second, each individual in the population generates a different quadtree structure, because different CPPNs produce different variance patterns.

The per-individual, per-level dependency chain makes population-level vectorization impossible. Modern GPU architectures rely on the Single Instruction, Multiple Threads (SIMT) execution model, which achieves high throughput by executing the same operation across hundreds or thousands of data elements simultaneously. This model requires *uniform work*: all threads in a warp (typically 32 threads) must execute the same instruction at the same time. When each individual in the population produces a different quadtree (different numbers of active cells, different branching patterns, different memory access patterns), the GPU cannot execute them in parallel. Some threads would be subdividing while others have already terminated; some would access memory region A while others access region B. The result is severe warp divergence and memory access inefficiency, negating the GPU's parallelism advantage entirely.

This is not an implementation limitation that can be resolved through better engineering. Controlled experiments (Chapter 5) confirm that quadtree depth dominates runtime scaling and that no implementation optimization overcomes the algorithmic bottleneck. The quadtree is fundamentally incompatible with massively parallel hardware. GPUs

can accelerate the CPPN evaluation step (which is uniform across individuals), but the quadtree traversal itself (the sequential, conditional, per-individual subdivision) must run serially.

The practical consequence: every extension of ES-HyperNEAT (TPE optimization, spatial decomposition, or future variants) inherits this bottleneck. Substrates that take minutes to construct on CPU would take seconds on GPU, but the quadtree prevents this acceleration entirely. For MNIST-scale problems, single experiments require hours to days. For higher-dimensional problems, the computation becomes infeasible on any current hardware.

1.2.4 The Three Barriers as a Research Program

These three barrier clusters are not independent, but they require different types of solutions:

- **Cluster 1** (configuration expense) is a *search problem*: the viable region exists but is hard to find. It is addressed through principled hyperparameter optimization and early-stopping strategies that reduce search cost.
- **Cluster 2** (representational collapse) is an *architectural problem*: no configuration produces a substrate with adequate input coverage. It is addressed through structured input decomposition that forces the substrate to attend to the full input space.
- **Cluster 3** (computational bottleneck) is an *algorithmic problem*: the quadtree produces a uniquely-shaped graph for every individual, so the population cannot be batched on parallel hardware. It is addressed through a fundamental tensor reformulation that replaces the sequential quadtree with eager evaluation on fixed multi-resolution grids followed by variance filtering, producing uniform matrices that vectorize on both CPU and GPU.

Cluster 1 establishes the performance ceiling: what ES-HyperNEAT can achieve with optimal configuration. Cluster 2 proves that the ceiling is structural, not parametric[2]: even with optimal hyperparameters, the representational collapse limits performance. Cluster 3 reveals that even if the representational problem were solved, the computational bottleneck would prevent the system from scaling to practical problem sizes. Together, these three clusters define the set of barriers this thesis addresses to make adaptive substrate neuroevolution a practical tool; §7.5 adds two further prerequisites specific to compositional architectures.

The research questions that follow define precisely what must be solved.

1.3 Research Questions

We organize the thesis around four research questions, derived bottom-up from the five papers that constitute the thesis. Each question corresponds to a specific aspect of the barrier structure described above, and each is addressed by one or more chapters. The

[2]We use *structural* to mean a limitation that persists across all configurations in the viable hyperparameter region, and *parametric* to mean one that a different configuration within that region can resolve.

questions are numbered TRQ1–TRQ4 (Thesis Research Questions) to distinguish them from the paper-level research questions discussed within individual chapters.

1.3.1 TRQ1: How can the hyperparameter space of adaptive substrate neuroevolution be navigated efficiently?

Context. The 16-parameter subspace exceeds three billion configurations (§1.2.1), over 90% of them stagnating at random guessing. Naive search (grid search, random search, or manual tuning) is prohibitively expensive and overwhelmingly likely to miss the narrow region of viable configurations. Prior work on ES-HyperNEAT used hand-selected hyperparameters without systematic optimization, making it impossible to determine whether reported results represented the algorithm's capability or merely a fortunate parameter choice.
Narrative role. TRQ1 establishes what ES-HyperNEAT can achieve *as-is*, with optimal hyperparameters and reduced search cost. The answer is informative but limiting: even with exhaustive optimization, a monolithic ES-HyperNEAT substrate plateaus far below what convolutional networks achieve on the same tasks. This ceiling motivates TRQ2: *why* does the ceiling exist, and *is it parametric or structural?*

1.3.2 TRQ2: Can ES-HyperNEAT's fundamental limitations be diagnosed empirically?

Context. Even with optimal hyperparameters (TRQ1), ES-HyperNEAT's performance plateaus far below what the task permits. The question is whether this plateau reflects insufficient optimization (solvable with more trials or better algorithms) or a structural limitation that no amount of optimization can address.
Narrative role. TRQ2 is the diagnostic core of the thesis. It proves that ES-HyperNEAT's limitations are structural, not parametric: no amount of optimization fixes them. This diagnosis is what transforms the thesis from "we made ES-HyperNEAT faster" into "we identified fundamental problems and built principled solutions." Without TRQ2, the contributions in TRQ3 and TRQ4 would be engineering improvements; with it, they are necessary responses to proven impossibilities.

1.3.3 TRQ3: Can adaptive substrate discovery be redesigned for massively parallel execution without sacrificing adaptivity?

Context. TRQ2 establishes that the quadtree must be *replaced*, not optimized. But the quadtree provides the adaptivity that makes ES-HyperNEAT valuable: substrates whose topology matches the problem structure, with neurons placed only where the CPPN encodes meaningful information. Replacing the quadtree with a fixed grid would eliminate the computational bottleneck but also eliminate the adaptivity, reducing ES-HyperNEAT back to standard HyperNEAT. The question is whether a redesign can preserve the adaptivity while removing the sequential bottleneck.
Narrative role. TRQ3 is the thesis's principal algorithmic contribution: the solution that resolves the computational bottleneck which has constrained indirect encoding neuroevolution for over a decade. EMR-HyperNEAT makes it practical to run the thousands of evolutionary runs that systematic investigation requires, and enables scaling adaptive substrates to problem sizes that were previously infeasible.

1.3.4 TRQ4: Can input decomposition extend adaptive substrates to high-dimensional problems?

Context. Even with fast substrate discovery (TRQ3), a monolithic substrate encoding a high-dimensional input will still suffer from representational collapse (TRQ2). The computational bottleneck and the representational bottleneck are independent problems requiring independent solutions. A fast algorithm that produces a centrally biased substrate is still a centrally biased substrate; it merely arrives at its inadequate solution faster.
Narrative role. TRQ4 complements TRQ3. Where TRQ3 provides the computational solution (fast substrate discovery via EMR-HyperNEAT), TRQ4 provides the representational solution (full input coverage via spatial decomposition). Together, they transform adaptive substrate neuroevolution from a CPU-bound, center-biased method into a vectorizable, full-coverage system whose tensor representation opens new research directions beyond the speedup itself. Neither solution alone is sufficient—both are necessary.

1.4 Experimental Domains

The experiments in this thesis span four domains:

- **Boolean logic tasks** (XOR, parity-k, compound Boolean expressions): canonical neuroevolution benchmarks that test whether evolution can discover the correct topology for a known function. Used across Chapters 3 and 5–7 for controlled evaluation of algorithmic changes and compositional prototypes.

- **Image classification** (MNIST handwritten digits, visual pattern recognition): high-dimensional classification benchmarks that test substrate scalability. Used in Chapters 3 and 4 for hyperparameter optimization, central bias diagnosis, and spatial decomposition; extended to further visual tasks in Chapter 6's cross-domain validation.

- **Regression** (financial earnings-price; synthetic continuous gates): a real-world regression task provides external validity beyond synthetic benchmarks (Chapter 6's CCM financial data), and synthetic continuous gate composition (XOR_c, AND_c, OR_c on $[0, 1]^2$) probes whether the compositional scaling story generalizes beyond Boolean structure (Chapter 7).

- **Reinforcement learning control** (cart-pole and related): tests temporal dynamics and continuous control. Used in Chapter 6 for cross-domain validation.

Full descriptions of all benchmarks appear in Section 2.5 of Chapter 2.

1.5 Summary of Contributions

This thesis makes five contributions across the four research questions, supported by approximately 11,300 experimental runs across five papers. We describe each contribution below; Table 1.1 provides a quantified summary at the end of the section.

- **C1: TPE hyperparameter optimization** (Chapter 3). The first systematic optimization study for ES-HyperNEAT: 2,013 TPE trials yielding a validated 29.0% MNIST peak, surpassing the prior 23.9% [98]. Variance threshold and CPPN complexity dominate sensitivity; the ceiling is structural, not a search failure.
- **C2: The early-stopping pruner** (Chapter 3). Predicts at generation 3 which runs will fail ($F_1 = 0.872$), eliminating 41.6% of cost while maintaining equivalent fitness quality (TOST $p < .001$). The pruner requires 64% fewer generations than Hyperband [60] (a multi-fidelity hyperparameter-optimization baseline that progressively early-stops underperforming configurations; $d = 1.884$), a cost reduction available to the search campaigns of Chapters 4–6. Beyond the computational savings, the pruner study's analysis of the 2,013-trial fitness distribution provides the first empirical evidence that ES-HyperNEAT's configuration space is bimodal (the final-accuracy distribution across the 2,013 TPE-sampled trials separates sharply: 16.6% stagnate at the 10% chance floor, 74.9% form a broad productive band, and 8.4% sit between; the generation-3 signal predicts the 80th-percentile success label (defined on the derivation set) with $F_1 = 0.872$ on the held-out 180-trial validation set): a fitness-landscape observation whose implications for search-space design are discussed in Chapter 3's synthesis.
- **C3: The central bias diagnosis and Spatially Partitioned Experts** (Chapter 4). ES-HyperNEAT substrates on MNIST cover only 3.6% of pixels, a structural collapse caused by CPPN geometry interacting with the quadtree's variance-driven subdivision. A 13-expert spatial partitioning forces evolution across the full input, more than doubling accuracy (Table 1.1) and providing the first empirical support within adaptive indirect encoding for specialist composition.
- **C4: The quadtree–GPU incompatibility** (Chapter 5). Each CPPN produces a unique topology, preventing the uniform tensors that GPU vectorization requires. Depth dominates runtime (ANOVA $F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$; regression fit $R^2 = 0.95$) and four optimization strategies exhaust at 1.65×; the quadtree must therefore be replaced, not optimized.
- **C5: EMR-HyperNEAT** (Chapter 6). A lazy-to-eager tensor reformulation that recasts adaptive substrate discovery as batch matrix operations on fixed multi-resolution grids followed by variance filtering. The reformulation produces uniform tensors that vectorize on any hardware backend (CPU or GPU), and—more importantly—opens the compositional matrix arithmetic that companion work exploits for per-node function evolution, neuromodulation, and aggregation selection. Validated across 15 problems in 4 domains (Table 1.1).

While the primary contributions are engineering solutions (optimization pipelines, architectural redesigns, and a tensor reformulation of adaptive substrate discovery), two findings go beyond engineering to reveal previously unknown properties of the algorithm. The central bias diagnosis (C3, Chapter 4) goes beyond naming a phenomenon the literature already recognized: it quantifies the collapse, characterizes its three compounding mechanisms (coordinate symmetry, symmetric CPPN primitives, and the quadtree's variance-driven subdivision), and proves no tuning resolves it. The quadtree–GPU incompatibility (C4, Chapter 5) demonstrates that no implementation strategy yields more than a marginal speedup on SIMT hardware; the bottleneck is therefore structural, not an implementation artifact.

Table 1.1: Quantified contributions of this thesis, organized by the research question each contribution addresses. All statistical claims are backed by experimental evidence reported in the corresponding chapters.

Contribution	**TRQ**	**Key Evidence**	**Ch.**
TPE hyperparameter optimization	TRQ1	29.0% MNIST via 2,013 TPE trials	3
Early-stopping pruner	TRQ1	41.6% cost reduction; $F_1 = 0.872$; 64% fewer gens than Hyperband ($d = 1.884$); reveals bimodal fitness landscape	3
Central bias diagnosis + Spatially Partitioned Experts	TRQ2, TRQ4	43% mean MNIST accuracy (105% improvement over 21% baseline); 85% peak pixel coverage (73% at $N_e = 13$) vs. 3.6% baseline	4
Quadtree–GPU incompatibility	TRQ2, TRQ3 (diag.)	ANOVA $F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$ ($R^2 = 0.95$ regression fit); depth dominates runtime	5
EMR-HyperNEAT	TRQ3	Lazy→eager tensor reformulation; 12–34× per-generation speedup at depths 5–7 and ∼33× cumulative over 30 generations (XOR, GPU, pop 1000; the 34× endpoint and the ∼33× cumulative rest on the depth-7 streaming estimate); 5.5× on financial data (depth 6, pop 100); CPU speedup also demonstrated; validated across 15 problems in 4 domains	6
Aggregate		∼11,300 unique runs across 5 papers	All

1.6 The Motivating Vision: GEENNS

The substrate-level work in this thesis is motivated by a longer-term goal: GEENNS (Grid-based Emergent Evolution of Neocortical Network Substrates), a compositional neuroevolutionary architecture designed to enable lifelong learning without catastrophic forgetting. The following subsections describe what GEENNS proposes, what it requires to function, and the current status of its prototype; subsequent chapters return to this vision in their synthesis paragraphs, linking each individual finding back to the compositional goal.

What GEENNS proposes. GEENNS decomposes problems into sub-problems and delegates each to a *specialist grid*, a computational unit with intrinsic plasticity that performs step-by-step computation through internal signal modulation and propagation. Evolved *mapper networks* connect task inputs and outputs to the appropriate grids and compose their responses into a solution. In the current prototype, grids are frozen after specialist training; the full architecture envisions grids whose synaptic plasticity continues during operation while the structural scaffold remains fixed. New tasks require only new mappers; existing grids are reused without modification. The biological inspiration is the neocortical column: a repeated computational template that, wired differently, supports the full diversity of cortical function [69]. The computational goal is a system where knowledge accumulates as reusable components rather than overwritable weight patterns, eliminating the catastrophic forgetting that plagues conventional continual learning.

The deeper motivation is emergence. GEENNS does not prescribe what each grid computes; it evolves grids under task pressure and observes what emerges. The frozen-grid architecture creates conditions where compositional behavior arises without being explicitly programmed: the mapper discovers how to route inputs and compose outputs through evolutionary search, not through human instruction. The proof-of-concept prototype (Section 7.6) provides the first instance: evolution discovers that routing through frozen specialists solves compound tasks, without any explicit directive to compose. This is not a claim of general emergent intelligence (the prototype's compositional strategies remain task-specific, see Current Status below), but it demonstrates that the architectural conditions for emergence can be created.

What GEENNS requires. Three infrastructure prerequisites must be in place before any multi-grid compositional system can function; each thesis research question addresses one of these prerequisites:

1. *Efficient hyperparameter navigation* for the substrate engine, because GEENNS evolves several CPPNs in parallel: four base intra-grid connectivity CPPNs, one per grid pair for inter-grid coupling, and two mappers per task. Appendix B gives the total as $4 + k(k-1)/2 + 2T$ for a k-grid, T-task system: 13 CPPNs at the prototype's (k=3, T=3) scale, 34 at a modest (k=5, T=10) deployment, 89 at a full (k=10, T=20) architecture. Each CPPN demands its own hyperparameter optimization (TRQ1).
2. *Representationally adequate substrates* that exploit the full input space, because specialist grids that collapse to a fraction of their assigned inputs cannot serve as reliable components (TRQ2, TRQ4).
3. *Tensor-accelerated substrate generation*, because the quadtree-based discovery mechanism makes multi-grid evolution infeasible: on the order of years of CPU compu-

tation for a multi-dozen-CPPN pipeline (TRQ3).

Without the infrastructure contributions of this thesis, multi-grid neuroevolution at the scale GEENNS requires would demand months or years of CPU time. With them, it becomes a days-to-weeks GPU job—a reduction from infeasible to practical that is the thesis's primary engineering impact.

Current status. A proof-of-concept prototype, validated across approximately 3,100 experimental runs (Chapter 7), demonstrates that frozen specialist grids composed through evolved mappers solve compound Boolean tasks that monolithic baselines cannot, with an evident gap at 5-grid scale. Frozen grids support novel task compositions without retraining, validating modular reuse. However, mappers learn task-specific routing, not general compositional strategies; compositional generalization remains the open problem (full analysis in Section 7.6). Warm-start transfer qualifies the negative: when the channel assignment is re-wired to match the recomposed target, source-evolved populations already contain zero-shot solvers of the unseen task, typically the converged champion itself; what remains open is transfer across recompositions that change what the channels mean.

Role in this thesis. This thesis does not implement GEENNS fully. Instead, each chapter diagnoses and resolves one infrastructure barrier that would otherwise block the compositional vision. Synthesis paragraphs at the end of Chapters 3–6 connect individual findings back to that vision, building the feasibility case incrementally. The full synthesis, including the prototype evidence, the computational feasibility arithmetic, and an assessment of what remains open, appears in Chapter 8.

Appendix A traces how this thesis evolved from its original 2022 proposal, including a module-by-module mapping from the proposed framework to the realized contributions.

1.7 Thesis Structure

This thesis is organized into five Parts comprising eight chapters. The structure follows the progressive arc from understanding the problem space, through diagnosing and resolving barriers, to the GEENNS vision and synthesis. Figure 1.1 provides a visual overview.

Table 8.1 provides a one-page synthesis mapping each barrier to its diagnosis, solution, evidence, and remaining limitations. Table 1.2 summarizes the experimental scope of the thesis across all five papers.

Part I: Foundations (Chapters 1–2). Chapter 1 (this chapter) frames the thesis and states the four research questions. Chapter 2 provides the technical background on the NEAT-to-ES-HyperNEAT lineage, hyperparameter optimization, partitioned experts, and GPU acceleration.

Part II: Scaling ES-HyperNEAT (Chapters 3–4). Chapter 3 addresses TRQ1 through hyperparameter optimization and early-stopping (Papers 1 and 2). Chapter 4 addresses TRQ2 and TRQ4 through the central bias diagnosis and spatially partitioned experts (Paper 3).

Part III: Beyond the Quadtree (Chapters 5–6). Chapter 5 addresses TRQ2 and the diagnostic half of TRQ3 by proving the quadtree–GPU incompatibility (Paper 4). Chapter 6 addresses TRQ3 through the EMR-HyperNEAT reformulation (Paper 5).

Part IV: The GEENNS Vision (Chapter 7). Chapter 7 traces the research evolution, presents the GEENNS algorithm design, and reports proof-of-concept prototype evidence.

Part V: Conclusion (Chapter 8). Chapter 8 revisits the four research questions, assesses contributions against open problems, and charts the path forward.

Notation and statistical reporting conventions are defined in the Notation and Symbols section preceding Chapter 1.

The rest of this thesis provides the evidence.

Table 1.2: Experimental scale of this thesis by chapter. Each chapter contributes independent experimental evidence; the aggregate reflects the breadth of investigation required to diagnose and resolve the three barrier clusters.

Chapter	**Focus**	**Runs**	**Paper**
Ch 3	TPE hyperparameter optimization	∼2,013	Paper 1
Ch 3	Early-stopping pruner	∼270	Paper 2
Ch 4	Partitioned experts + central bias	3,570	Paper 3
Ch 5	Quadtree–GPU incompatibility	456	Paper 4
Ch 6	EMR-HyperNEAT	5,002	Paper 5
Total (Ch 3–6)		11,311	5 papers

Note: Counts are per-paper headline totals; per-experiment breakdowns and supplementary trials appear in each chapter.

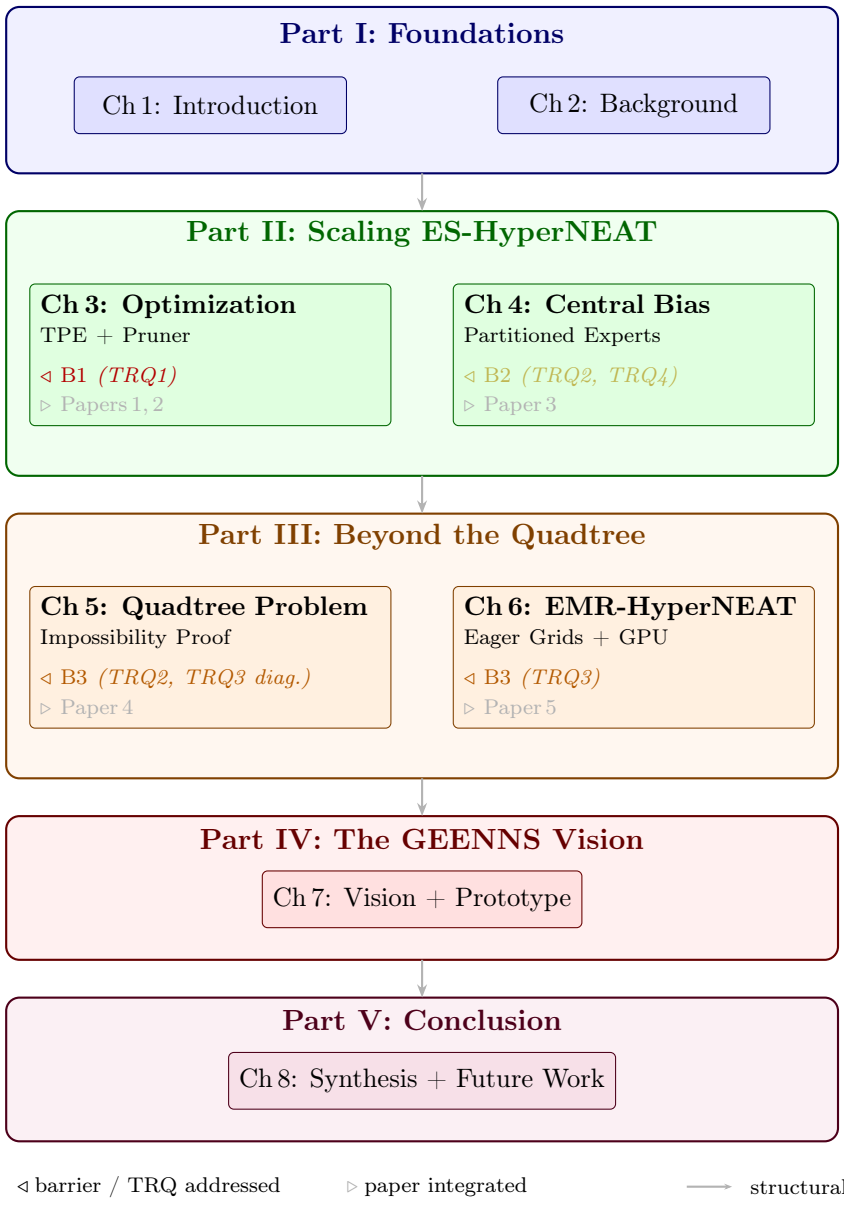


Figure 1.1: Thesis structure as nested containment. Each Part (outer box) contains its chapters. Chapters 3–6 annotate which barriers (B1–B3) and thesis research questions (TRQ1–TRQ4) they address (◁) and which papers they integrate (▷). Four arrows mark the structural flow between Parts.

Chapter 2

Background and State of the Art

> If I can't picture it, I can't understand it.
>
> Albert Einstein

This chapter provides the foundational material on which all subsequent chapters build. We cover the neuroevolutionary algorithms that form the computational substrate of this thesis (NEAT, CPPNs, HyperNEAT, ES-HyperNEAT, and GEENNS), the hyperparameter optimization methods used to search the configuration space, the Mixture-of-Experts approach that motivates our spatial decomposition work, the GPU programming model that constrains high-performance implementations, the benchmark problems and statistical methodology used throughout the experimental chapters, and the position of this lineage within the broader neural architecture search landscape.

We deliberately write each topic in full here so that later chapters can reference specific subsections without re-explanation. Readers familiar with particular topics may skip to the sections most relevant to their interest; the section-level structure is designed for selective reading.

2.1 From Direct Encoding to Adaptive Substrates

Neuroevolution, the application of evolutionary algorithms to the design and optimization of neural networks, has a history spanning several decades [31, 34, 106]. Where gradient-based training [79] adjusts weights in a fixed topology, neuroevolution can coevolve network structure alongside weights. This section traces the key developments from NEAT through adaptive substrate methods, establishing the algorithmic lineage that this thesis extends.

We begin with a brief review of evolutionary computation fundamentals to establish the terminology and mechanisms that underlie the entire neuroevolutionary stack.

2.1.1 Evolutionary Algorithms: Foundations

Evolutionary algorithms (EAs) are population-based optimization methods inspired by Darwinian natural selection [26, 32, 81, 107]. A population of candidate solutions undergoes iterative cycles of evaluation, selection, and variation until a termination criterion

is met. Algorithm 1 gives the generic generational template, shown schematically in Figure 2.1; every evolutionary system in this thesis is an instantiation of this loop.

Algorithm 1 Generational evolutionary algorithm template

Require: population size N, fitness function f, termination criterion
Ensure: best individual found
1: Initialize population $P = \{p_1, \ldots, p_N\}$ randomly
2: **while** termination criterion not met **do**
3: Evaluate $f(p)$ for each $p \in P$
4: Select parents from P based on fitness
5: Generate offspring via crossover and mutation
6: Form next generation P' (optionally preserve elite)
7: $P \leftarrow P'$
8: **end while**
9: **return** $\arg\max_{p \in P} f(p)$

Key concepts. A *population* P consists of N *individuals*, each encoding a candidate solution in a representation-specific *genotype*. A *generation* is one iteration of the evaluate–select–vary loop. The *fitness function* $f : \mathcal{S} \to \mathbb{R}$ maps the solution space $\mathcal{S}$ to a scalar quality measure; in neuroevolution, $\mathcal{S}$ is the space of neural network topologies and weights, and f is the task performance metric.

Selection. Selection determines which individuals contribute offspring to the next generation. Three common mechanisms are:

- *Proportional selection*: individual p_i is selected with probability $P(\text{select } p_i) = f(p_i) / \sum_j f(p_j)$.
- *Tournament selection*: sample k individuals uniformly at random and return the fittest. Tournament size k controls selection pressure: larger k favors exploitation; smaller k favors exploration.
- *Truncation selection*: retain the top-τ fraction of the population, sample parents uniformly from this set.

NEAT uses a speciation-based variant (described in §2.1.2) in which truncation selection operates *within each species*, and offspring allocation across species is proportional to adjusted fitness.

Variation operators. *Crossover* (recombination) combines genetic material from two parents to produce offspring, enabling the evolutionary process to combine beneficial partial solutions. *Mutation* perturbs a single individual, introducing novel variation. The balance between crossover (exploitation of existing solutions) and mutation (exploration of new regions) is a fundamental EA design parameter.

Replacement and elitism. In *generational* replacement, the offspring population entirely replaces the parents. In *steady-state* replacement, offspring replace individual members of the current population. *Elitism* (copying the best individual unchanged into the next generation) guarantees monotonically non-decreasing best fitness. All systems in this thesis use generational replacement with species-level elitism (each species preserves its champion).

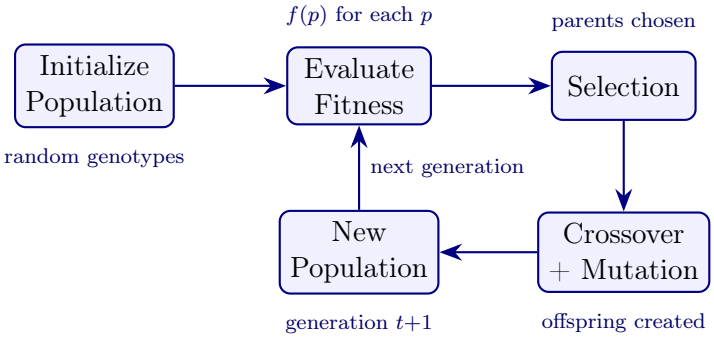


Figure 2.1: Generational EA lifecycle. A population is initialized, evaluated, and iteratively refined through selection and variation. The loop repeats until a termination criterion is met. Every neuroevolutionary algorithm in this thesis, from NEAT to GEENNS, is an instantiation of this template.

2.1.2 NEAT: NeuroEvolution of Augmenting Topologies

Stanley and Miikkulainen [88] introduced NEAT (NeuroEvolution of Augmenting Topologies), which provides three mechanisms that together make topology evolution tractable: historical markings via innovation numbers, speciation through compatibility distance, and incremental complexification from minimal initial structures (Figure 2.2).

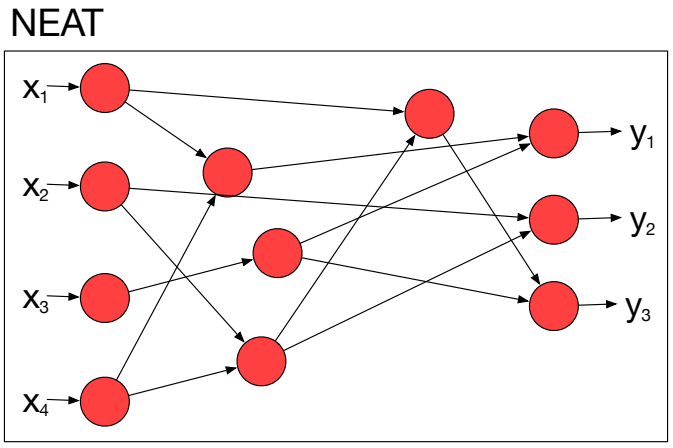


Figure 2.2: A sparse neural network topology evolved by NEAT. Four inputs (x_1–x_4) connect through hidden nodes to three outputs (y_1–y_3). The topology is irregular: not all possible connections exist, and hidden nodes appear at varying depths. This structure was not designed but discovered by NEAT's mutation operators (add node, add connection) starting from a minimal seed network.

Genome Encoding

A NEAT genome consists of two lists: a list of node genes and a list of connection genes. Each node gene specifies a node identifier and a node type (input, hidden, or output). Each connection gene specifies a source node, a destination node, a weight value $w \in \mathbb{R}$, an enabled/disabled flag, and an innovation number.

The NEAT genome does not include an activation function field per node. All nodes in the original implementation use a single fixed activation function: a steepened sigmoid $\varphi(x) = 1/(1 + e^{-4.9x})$ [88]. This function is hardcoded and applies identically to every hidden and output node. The only evolvable parameters are topology (which nodes and connections exist) and connection weights: there is no mutation operator that changes a node's activation function. Function choice is a fixed experimental parameter, not part of the evolutionary search.

Historical Markings: Innovation Numbers

The innovation number is the central mechanism for tracking genetic history: the historical marker that makes topology crossover tractable. Each new structural mutation (adding a node or adding a connection) receives a globally unique innovation number. When the same structural mutation arises independently in different individuals during the same generation, it receives the same innovation number, because the same structural change to the same network position represents the same innovation regardless of which individual discovers it. This global counter ensures that homologous genes (genes encoding the same structural element) can be identified across any two genomes in the population by matching their innovation numbers.

Figure 2.3 illustrates innovation numbers in action during crossover, aligning two parents with partially overlapping topologies (Parent A more fit than Parent B); the inheritance rules follow in the next subsection. Without innovation numbers, the same structural element could appear at different positions in different genomes, making alignment ambiguous: the *competing conventions* problem that blocked earlier topology-evolving methods [74].

Crossover and Mutation

Crossover in NEAT aligns two parent genomes by their innovation numbers. Matching genes (those with identical innovation numbers in both parents) are inherited randomly from either parent. Disjoint genes (present in one parent but within the innovation number range of the other) and excess genes (beyond the range of the shorter genome) are inherited from the more fit parent. This alignment-based crossover resolves the competing-conventions problem: two networks that compute the same function but encode it differently can still produce meaningful offspring.

Mutation operates through three primary operators:

1. **Weight perturbation**: Each connection weight is perturbed by adding Gaussian noise $\delta \sim \mathcal{N}(0, \sigma^2)$ with probability $p_{\text{mutate_weight}}$, or replaced entirely with a new random value with probability $p_{\text{replace_weight}}$.

2. **Add connection**: A new connection gene is created between two previously unconnected nodes, with a random initial weight and a new innovation number.

3. **Add node**: An existing connection is disabled, and two new connections are created through a new hidden node. The connection from the source to the new node receives weight 1.0, and the connection from the new node to the destination receives the original connection's weight. This preserves the network's behavior immediately after mutation, allowing incremental refinement instead of catastrophic disruption.

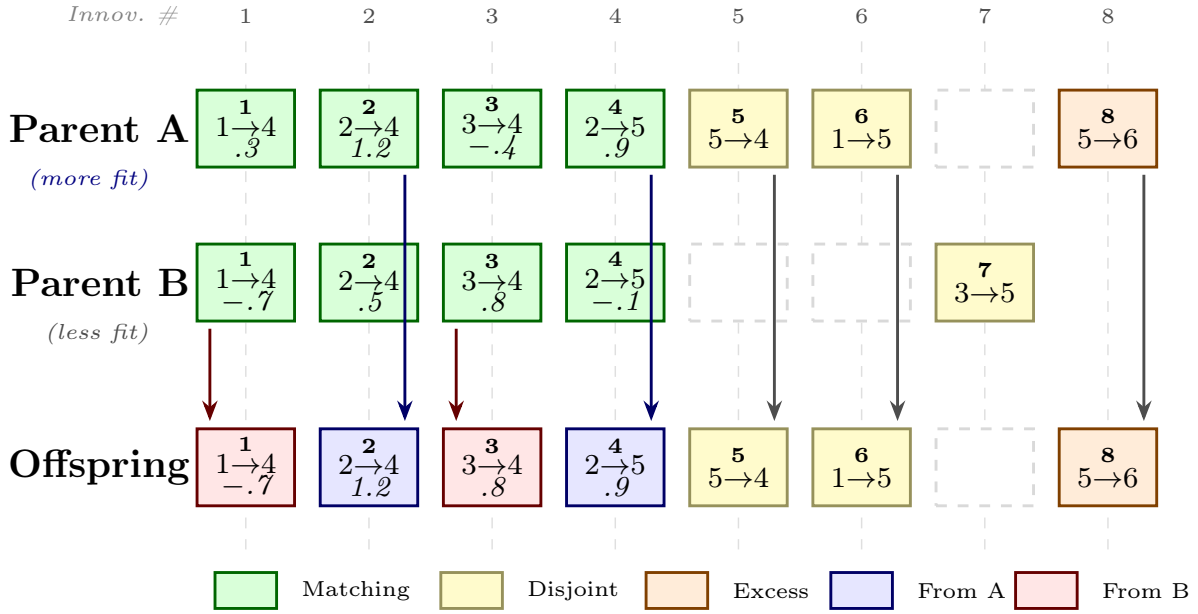


Figure 2.3: NEAT crossover aligns parent genomes by innovation number. Matching genes (1–4) show connection and weight; each is inherited randomly from either parent, with the offspring color indicating the source (blue = Parent A, red = Parent B). Disjoint genes (5, 6 from A; 7 from B) and excess genes (8) are inherited only from the more fit parent. Gene 7 is not inherited because it belongs to the less fit parent.

Speciation

NEAT protects structural innovations through speciation. The compatibility distance δ between two genomes g_1 and g_2 measures their structural and parametric similarity:

$$\delta(g_1, g_2) = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W} \tag{2.1}$$

where E is the number of excess genes, D the number of disjoint genes, N the number of genes in the larger genome (or 1 if both genomes are small), $\bar{W}$ is the average weight difference of matching genes, and c_1, c_2, c_3 are coefficients controlling the relative importance of each term.

Individuals are assigned to species based on a compatibility threshold δ_t: each individual is compared against a representative of each existing species, and assigned to the first species whose representative it falls within δ_t of. If no compatible species exists, a new species is created. Fitness sharing within species prevents any single species from dominating the population, giving novel topologies time to optimize their weights before competing against established structures.

Fitness Sharing and Offspring Allocation

Speciation alone does not protect innovation; the *fitness sharing* mechanism makes it work. Each individual's raw fitness f_i is divided by the size of its species $|S_j|$. The result is an adjusted fitness:

$$f'_i = \frac{f_i}{|S_j|} \tag{2.2}$$

This adjusted fitness penalizes membership in large species: an individual in a species of 10 receives one-tenth the credit of an equally fit individual in a species of 1. The result is implicit fitness pressure toward novel niches: species that are small (because their innovations are new) receive proportionally more reproductive opportunities.

Offspring are allocated to each species in proportion to its total adjusted fitness. For K species, species S_j receives:

$$n_j = N \cdot \frac{\sum_{i \in S_j} f'_i}{\sum_{k=1}^{K} \sum_{i \in S_k} f'_i} \tag{2.3}$$

where N is the total population size. Within each species, parents are selected by truncation: only the top τ fraction (typically $\tau = 0.2$) of the species is eligible for reproduction. This within-species truncation is the selection mechanism that drives NEAT's evolutionary search.

NEAT Generation Lifecycle

Algorithm 2 summarizes the complete NEAT generation as an instantiation of the EA template (Algorithm 1). This lifecycle underlies every evolutionary run in the thesis: Chapters 3–6 all rely on NEAT's speciation and fitness sharing for maintaining population diversity during hyperparameter search, multi-task learning, and substrate discovery.

Algorithm 2 NEAT generation lifecycle

Require: population P, fitness function f, threshold δ_t, survival fraction τ
1: Evaluate $f(p)$ for each $p \in P$
2: Assign each p to species S_j by compatibility: $\delta(p, \text{rep}_j) < \delta_t$
3: Compute adjusted fitness: $f'_i \leftarrow f_i / |S_j|$ for each individual i in species j
4: Allocate offspring: $n_j \propto \sum_{i \in S_j} f'_i$ for each species S_j
5: **for** each species S_j **do**
6: Select parents: top τ fraction of S_j by raw fitness
7: Produce n_j offspring via crossover + mutation
8: Preserve species champion (elitism) if $|S_j| > 5$[1]
9: **end for**
10: $P \leftarrow$ union of all offspring and preserved champions
11: Remove species stagnant for >15 generations (no fitness improvement)

[1]The size condition follows the original NEAT paper, which copies the champion of each species with more than five members into the next generation unchanged [88]. The frameworks used in this thesis expose species-size control through a different knob (`min_species_size`, defaulting to 2 in `neat-python` [66] and 1 in TensorNEAT), which sets a floor on each surviving species' offspring allocation rather than governing champion elitism; champion preservation itself is configured separately, and every configuration

Incremental Complexification

NEAT begins with minimal networks (direct connections from inputs to outputs, with no hidden nodes) and adds structure only when mutations that increase complexity prove beneficial. This bias toward simplicity has two consequences. First, evolution searches the space of simple solutions before exploring complex ones, finding minimal-complexity solutions when they exist. Second, each new structural element is introduced into a context where the existing structure has already been optimized, avoiding the premature complexity that plagues methods starting from random large networks.

The combination of these mechanisms (innovation numbers enabling alignment-based crossover, speciation for diversity protection, and complexification for efficient search) made NEAT the first neuroevolution algorithm to reliably evolve both topology and weights simultaneously [88]. NEAT remains the foundation on which all subsequent work in this thesis builds, either directly (as the evolutionary engine) or indirectly (through HyperNEAT and its extensions).

2.1.3 CPPNs: Compositional Pattern-Producing Networks

Stanley [87] introduced Compositional Pattern-Producing Networks (CPPNs), a class of neural networks that produce geometric patterns through the composition of activation functions with distinct mathematical properties. Unlike NEAT's substrate networks, which apply a single hardcoded activation function uniformly across all nodes (§2.1.2), CPPNs use a diverse palette of functions (each chosen for its geometric properties) and compose them across multiple layers to generate complex spatial patterns from coordinate inputs. This dual nature—structurally a neural network evolved by NEAT's operators, yet functionally a pattern-generating program that composes mathematical primitives—is what makes CPPNs effective as indirect encodings. The entire HyperNEAT lineage rests on this bridge between program synthesis and network evolution.

A CPPN is formally a function $\mathcal{C} : \mathbb{R}^n \to \mathbb{R}^m$ that maps spatial coordinates to output values through a directed acyclic (or recurrent) graph of nodes, where each node applies one of several activation functions:

- **Sine** ($\sin(x)$): Produces periodic, repeating patterns. When applied to spatial coordinates, sine generates regular stripes, grids, or wave-like motifs. Its periodicity enables a single compact CPPN to encode structures with arbitrary repetition.

- **Gaussian** (e^{-x^2}): Produces localized, radial patterns centered at the origin. Gaussian functions create spots, bumps, or gradients, encoding locality: the principle that nearby positions tend to have similar properties.

- **Sigmoid** ($\sigma(x) = 1/(1+e^{-x})$): Produces smooth step transitions. Sigmoid functions create boundaries and decision surfaces, dividing space into distinct regions with gradual transitions.

- **Linear** (x): Produces gradients. Linear functions create smooth ramps across space, encoding directional trends.

in this thesis enables it (per-species `elitism` in `neat-python`, TensorNEAT's `genome_elitism`), copying each species' top genomes unchanged (with `elitism` of 15 in the PUREPLES campaigns, the effective allocation floor rises to 15).

- **Tanh** ($\tanh(x)$): Similar to sigmoid but centered at zero and ranging from -1 to 1. Produces symmetric step transitions useful for encoding bilateral symmetry.

When NEAT's add-node mutation creates a new hidden node in a CPPN, that node receives a random activation function from the canonical set [90]. The function is then fixed for the lifetime of that node; no operator changes it after creation. Function diversity in the CPPN therefore comes entirely from topological growth: each new node brings a randomly chosen geometric primitive, and the evolved topology determines how these primitives compose. Gaussian produces symmetry, sine produces repetition, and sigmoid produces boundaries; their composition through the network graph, not mutation of individual node functions, generates complex patterns.

The progression from NEAT's uniform fixed activation, through CPPN's random-at-creation per-node assignment, to mutable per-node evolution (as explored in companion work, §8.4) adds successive degrees of freedom to how activation functions participate in evolutionary search.

Table 2.1 summarizes the mathematical properties of these five functions. The final column indicates whether the function produces high variance across the full $[-1, 1]^2$ coordinate space (relevant to ES-HyperNEAT's variance-driven node placement, formalized in §2.1.9): only periodic functions and the linear function escape the central concentration that saturating functions produce.

Table 2.1: CPPN activation function properties. "Escapes bias" indicates whether the function produces variance uniformly across the coordinate space (§2.1.9).

Function	Formula	Range	Geometric property	Escapes bias
Sine	$\sin(x)$	$[-1, 1]$	Periodic	Yes
Gaussian	e^{-x^2}	$(0, 1]$	Localized	No
Sigmoid	$1/(1 + e^{-x})$	$(0, 1)$	Saturating	No
Tanh	$\tanh(x)$	$(-1, 1)$	Sym. saturating	No
Linear	x	$\mathbb{R}$	Uniform gradient	Yes

Figure 2.4 plots the one-dimensional profile of each function, and makes the geometric distinctions concrete: sine oscillates uniformly, Gaussian concentrates at the origin, sigmoid and tanh saturate in the periphery, and the linear function ramps uniformly.

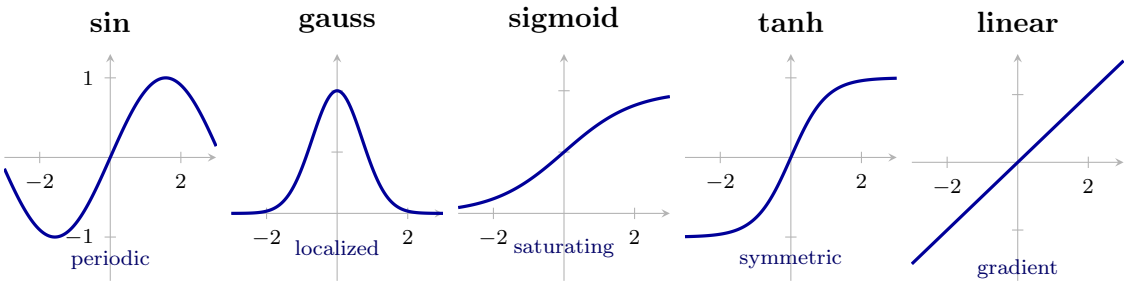


Figure 2.4: One-dimensional profiles of the five standard CPPN activation functions. Sine produces uniform oscillation across the domain; Gaussian concentrates activity at the origin; sigmoid and tanh saturate at large $|x|$ (producing the central bias of §2.1.9); the linear function has constant gradient everywhere.

The principle behind CPPNs is that the composition of these functions through a network generates patterns far more complex than any single function. A Gaussian composed with a sine, for example, produces a pattern that is periodic but with localized amplitude, resembling the receptive fields found in biological visual systems. The CPPN framework thus provides a vocabulary of geometric primitives (symmetry, repetition, locality, gradients, boundaries) that can be composed to produce arbitrary spatial patterns.

CPPNs differ from traditional direct encoding in a fundamental way. In direct encoding, each parameter of the target structure (e.g., each connection weight in a neural network) corresponds to a separate gene. In the CPPN approach, the genes encode the structure of the pattern-generating function, not the pattern itself. A CPPN with 50 parameters can generate a connectivity pattern with thousands of weights, because the pattern arises from evaluating the CPPN at many different coordinate positions. This compression is not arbitrary: it specifically encodes geometric regularities. A symmetric pattern requires explicit duplication under direct encoding but emerges naturally from a Gaussian CPPN node. A repeating pattern requires explicit repetition under direct encoding but emerges from a sine node.

This indirect encoding provides the foundation for scalable neuroevolution, as we describe in the next section.

2.1.4 HyperNEAT: Indirect Encoding via CPPNs

Stanley et al. [90] introduced HyperNEAT (Hypercube-based NeuroEvolution of Augmenting Topologies), which combines NEAT's evolutionary engine with CPPN-based indirect encoding to evolve large-scale neural network substrates. The central idea is to evolve not the connectivity directly, but a CPPN that generates the substrate's connectivity pattern from spatial coordinates (Figure 2.5).

The Substrate Concept

A substrate in HyperNEAT is a neural network defined by a set of nodes at fixed spatial positions and connections between them. Nodes are arranged in a geometric layout (typically a grid or layered structure) where each node has coordinates (x, y) in a normalized space (usually $[-1, 1]^2$). The spatial arrangement encodes task-relevant structure: for visual tasks, input nodes may be arranged in a 2D grid matching the pixel layout; for control tasks, inputs and outputs may be arranged along a line reflecting the body's bilateral symmetry.

The Query Process

To determine the connection weight between two substrate nodes at positions (x_1, y_1) and (x_2, y_2), HyperNEAT queries the evolved CPPN:

$$w = \mathcal{C}(x_1, y_1, x_2, y_2) \tag{2.4}$$

The CPPN takes the four-dimensional input representing the source and target positions and produces a weight value. If the absolute weight exceeds a threshold $w_{\min}$, the connection is included in the substrate; otherwise, it is pruned. This process is repeated for all pairs of nodes between all connected layer pairs in the substrate layout, producing the complete substrate connectivity. The substrate nodes themselves apply a fixed activation function (in the original HyperNEAT, the same steepened sigmoid used by NEAT,

§2.1.2); the CPPN's diverse palette serves the *encoding function*, not the target network's internal computation.

The scheme's principal advantage is that it is geometry-aware: because the CPPN receives spatial coordinates as input, it naturally produces connectivity patterns that exploit the geometric structure of the substrate. A CPPN that has learned to produce strong connections between nearby nodes (via Gaussian activation) will generalize this local connectivity to any substrate size. A CPPN that produces periodic weight patterns (via sine activation) will create regular repeating motifs regardless of the number of nodes.

Scalability Through Indirect Encoding

HyperNEAT's indirect encoding provides a qualitative scalability advantage over direct encoding. Under direct encoding, the genome size scales with the number of connections in the substrate: a fully connected network with n nodes in one layer and m nodes in the next requires $n \times m$ genome parameters. Under HyperNEAT's indirect encoding, the genome size is determined by the CPPN's complexity, which is independent of the substrate size. The same CPPN can generate a 10×10 substrate or a 100×100 substrate simply by querying it at more positions.

This decoupling of genome complexity from substrate complexity enables HyperNEAT to evolve large substrates that would be infeasible under direct encoding. Gauci et al. [35] demonstrated the practical advantage of this geometry-aware encoding by showing that HyperNEAT could exploit the regular geometry of a checkers board to learn strategies that direct-encoding methods could not match, and D'Ambrosio and Stanley [24] extended HyperNEAT to multiagent systems where the same CPPN generated policies for teams of varying size. However, HyperNEAT comes with a limitation: the substrate topology must be specified in advance. The researcher must decide how many nodes to place and where to put them before evolution begins. This constraint motivates the adaptive substrate methods described in the following sections.

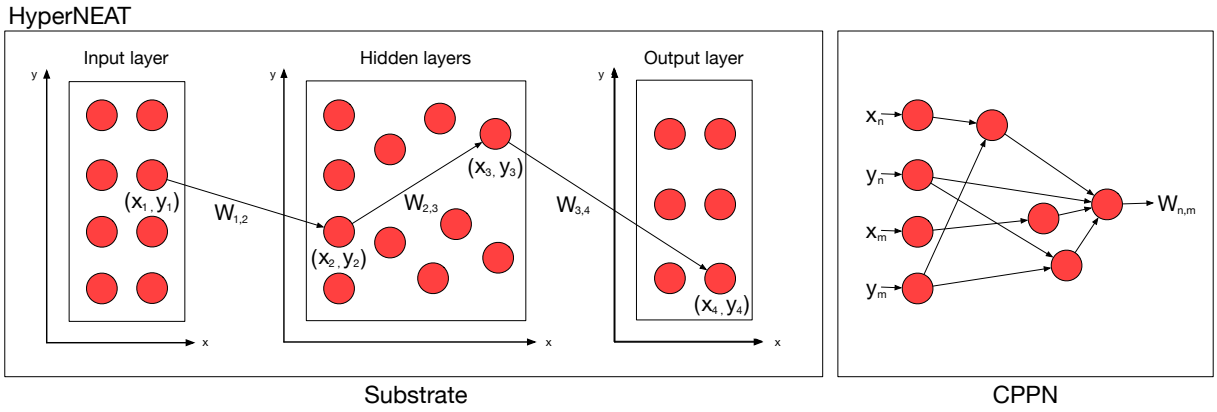


Figure 2.5: Overview of HyperNEAT. A CPPN (right) is evolved by NEAT and queried with the coordinates of every pair of substrate nodes to produce connection weights. The substrate layout (left) is fixed by the experimenter; the CPPN generates its connectivity pattern through indirect encoding.

2.1.5 Multi-Output CPPNs

The basic HyperNEAT query produces a single output: the connection weight. Multi-output CPPNs extend this to produce additional substrate properties from a single query. Common outputs include:

- **Weight**: The connection strength $w \in \mathbb{R}$.
- **Bias**: A per-node bias value b, queried at each node's coordinates.
- **Connection existence**: A dedicated output that explicitly gates whether a connection should exist, decoupling the existence decision from weight magnitude. Without this decoupling, the CPPN must use a single weight output to satisfy two conflicting pressures at once: produce a small magnitude (below w_{min}) to suppress a connection, and produce a precise magnitude to parameterize an expressed one. Verbancsics and Stanley [99] introduced *Link Expression Output* (LEO) to let evolution optimize each decision independently.
- **Per-node function indices**: Outputs that assign activation or aggregation functions to individual substrate nodes, enabling heterogeneous substrates. The Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT) architecture's batch evaluation is structurally compatible with this extension via additional CPPN output dimensions; per-node function evolution is validated in companion work (§8.4, Table 8.5).

The general multi-output CPPN query is:

$$(w, b, \ldots) = \mathcal{C}(x_1, y_1, x_2, y_2) \tag{2.5}$$

where additional outputs are appended as needed by extending the CPPN's output layer. The substrate includes only those connections whose weight magnitude exceeds a threshold w_{min} (or whose dedicated existence output exceeds a separate threshold, when present).

2.1.6 ES-HyperNEAT: Adaptive Substrate Discovery

Risi and Stanley [78] introduced ES-HyperNEAT (Evolvable-Substrate HyperNEAT), which addresses HyperNEAT's principal limitation (the requirement for a predefined substrate layout) by evolving not just the connectivity pattern but the substrate topology itself (Figure 2.6). Nodes are placed automatically where the CPPN's output exhibits high variance, on the principle that regions of high information content in the CPPN correspond to regions where the substrate needs fine-grained resolution.

Quadtree Decomposition

ES-HyperNEAT discovers substrate topology through recursive quadtree decomposition of the coordinate space, executed *once per source neuron*. For a fixed source at (a, b), the CPPN defines a scalar field over the plane, $w(x, y) = \mathcal{C}(a, b, x, y)$: the source's *weight landscape*. The algorithm's task is only to *detect* whatever structure the evolved CPPN put there and express it as substrate topology. For each source neuron at position (a, b), it proceeds in two phases:

1. **Division/Initialization**: The coordinate space is recursively subdivided into quadrants. At each level, four child quadrants are created and the CPPN is queried at each child's *center* using a 4D query $\mathcal{C}(a, b, x_c, y_c)$, where (a, b) is the anchoring source neuron and (x_c, y_c) is the child center. If the variance of the four resulting weights exceeds a division threshold θ_{div}, or the depth has not yet reached the initial depth d_0, the quadrant is subdivided further, up to maximum depth D. The result is a tree that is fine where the weight pattern changes and coarse where it is uniform; every queried center, together with its stored weight, is a *candidate* position for the second phase.

2. **Pruning/Extraction**: The completed quadtree is traversed, and two pruning mechanisms act in sequence. First, the traversal descends into a child only while the weight variance of that child's subtree exceeds a variance threshold θ_{var}; descent stops at the shallowest uniform node, so each uniform region contributes a single candidate at the coarsest scale the region admits, and finer candidates beneath it are never visited. Second, each stopping candidate must pass a *band membership test*, which asks whether it lies inside a *band* of the weight landscape (a thin feature that differs from its surroundings) rather than in a flat region or on a one-sided edge. The test queries the CPPN at four neighboring positions, displaced by the candidate cell's full side (twice its half-width), and computes:

$$\beta = \max\big(\min(d_T, d_B),\ \min(d_L, d_R)\big) > \theta_{\text{band}} \tag{2.6}$$

   where d_T, d_B, d_L, d_R are absolute weight differences to the top, bottom, left, and right neighbors. The inner min requires change on *both* sides along at least one axis: the candidate sits *between* different weight regions, inside a band narrower than twice the probe spacing. The outer max accepts a band along either axis, and a smooth one-sided gradient fails because one neighbor always remains similar. Each candidate passing the test is extracted as a connection carrying its stored weight, and the connection's non-source endpoint thereby becomes a newly discovered hidden node: node placement is a byproduct of connection extraction, not a separate procedure.

The two phases divide the labor: division *proposes* candidate positions at a resolution adapted to the local complexity of the weight landscape, and pruning *disposes* of every candidate whose expression would carry no information, extracting connections only where the landscape forms bands. Appendix C.6 develops the weight-landscape view in full, with a worked example on a banded landscape (Figure C.1).

This per-source procedure runs in three stages: (i) outgoing from each input neuron to discover hidden nodes, (ii) outgoing from each discovered hidden node (iterated for a configurable number of levels) to discover further hidden nodes, and (iii) incoming to each output neuron to establish connections from hidden to output. A final network-cleaning step removes nodes and connections not on any input-to-output path.

Algorithm 3 formalizes the per-source procedure; the full three-phase substrate discovery and its computational bottleneck are analyzed in Chapter 5 (Algorithm 7).

The division threshold θ_{div} controls the trade-off between substrate resolution and computational cost: low values produce fine-grained substrates with many nodes, high values produce coarse substrates with few nodes. The variance threshold θ_{var} plays the corresponding role during extraction, halting descent in uniform subtrees so that each uniform region is represented once. The band threshold θ_{band} independently controls

Algorithm 3 ES-HyperNEAT quadtree discovery (per source neuron)

Require: CPPN $\mathcal{C}$, source coordinate (a, b), direction flag d
Require: Initial depth d_0, max depth D, division threshold θ_{div}, variance threshold θ_{var}, band threshold θ_{band}
Ensure: Set of connections $\mathcal{E}$

{Outgoing case shown; incoming reverses query and connection direction}

{**Phase 1: Division/Initialization**}
root $\leftarrow$ QuadPoint(center=$(0, 0)$, width=1, level=1) {`width` is the half-width (center-to-edge), so a cell's side is $2 \times$ width}
$Q \leftarrow$ {root}
while $Q \neq \emptyset$ **do**
  $p \leftarrow Q$.dequeue()
  Create four children at centers ($p.x \pm p.\text{width}/2$, $p.y \pm p.\text{width}/2$), each with width = $p.\text{width}/2$
  for each child c **do**
    $c.w \leftarrow \mathcal{C}(a, b, c.x, c.y)$ {4D query anchored to source}
  end for
  if $p.\text{level} < d_0$ **or** ($p.\text{level} < D$ **and** $\text{Var}(\{c_i.w\}) > \theta_{\text{div}}$) **then**
    $p.\text{children} \leftarrow \{c_1, c_2, c_3, c_4\}$
    $Q \leftarrow Q \cup \text{children}$
  end if
end while

{**Phase 2: Pruning/Extraction**}
$\mathcal{E} \leftarrow \emptyset$
for each node ℓ where descent stops (subtree variance $\leq \theta_{\text{var}}$; leaves always stop) **do**
  $w_p \leftarrow \ell.\text{width} \times 2$ {Parent width for neighbor spacing}
  $d_L \leftarrow |\ell.w - \mathcal{C}(a, b, \ell.x - w_p, \ell.y)|$
  $d_R \leftarrow |\ell.w - \mathcal{C}(a, b, \ell.x + w_p, \ell.y)|$
  $d_T \leftarrow |\ell.w - \mathcal{C}(a, b, \ell.x, \ell.y + w_p)|$
  $d_B \leftarrow |\ell.w - \mathcal{C}(a, b, \ell.x, \ell.y - w_p)|$
  if $\max(\min(d_T, d_B),\ \min(d_L, d_R)) > \theta_{\text{band}}$ **and** $\ell.w \neq 0$ **and** $b < \ell.y$ **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(a, b) \rightarrow (\ell.x, \ell.y)$ with weight $\ell.w\}$ {Feedforward}
  end if
end for
return $\mathcal{E}$

connection density by requiring that the CPPN output form a genuine band before expressing a connection. The maximum depth D limits the finest possible resolution: at depth D, the coordinate space is divided into 4^D potential positions (e.g., 1,024 at depth 5, 16,384 at depth 7). Algorithm 3 follows the reference implementation in indexing levels from 1 at the root, whereas these position counts index depth from 0 (root = depth 0), so level ℓ corresponds to depth $\ell - 1$, and the algorithm's bound $p.\text{level} < D$ reaches depth $D - 1$; the position counts above quote the depth directly. For incoming connections (stage iii), both the CPPN query and the connection direction are reversed: the query becomes $\mathcal{C}(\ell.x, \ell.y, a, b)$ and connections run from $(\ell.x, \ell.y) \to (a, b)$, and the feedforward constraint reverses to $\ell.y < b$.

Information-Theoretic Justification

The variance-based criterion has an information-theoretic interpretation. Regions where the CPPN output varies rapidly across nearby positions encode detailed connectivity patterns that require multiple substrate nodes to represent. Regions where the CPPN output is nearly constant encode uniform connectivity that can be captured by a single node or no node at all. By placing nodes where information density is high, ES-HyperNEAT allocates representational resources where they are most needed.

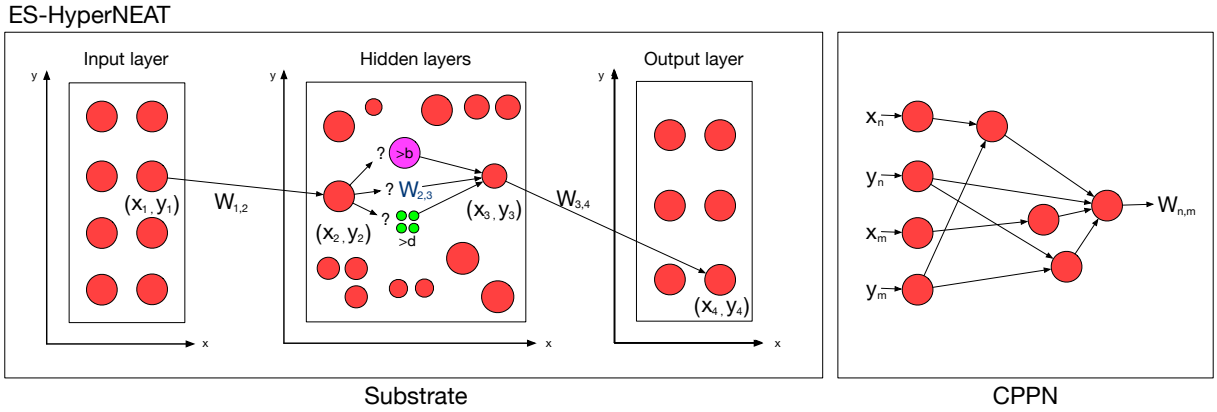


Figure 2.6: Overview of ES-HyperNEAT. Starting from an existing source node at (x_2, y_2), the CPPN is queried at candidate positions between the source and a potential target at (x_3, y_3). At each candidate position ("?" in the diagram), the CPPN's output resolves to one of two conditional outcomes: **(a)** *express a connection* carrying the CPPN-produced weight $W_{2,3}$ when the band indicator exceeds the band threshold θ_{band}, which places the connection's non-source endpoint as a new hidden node (the figure draws this one event as its two markers, the carried weight $W_{2,3}$ and the resulting purple node $> b$); or **(b)** *subdivide the quadrant* into four sub-quadrants (green markers, $> d$) when the local variance exceeds the division threshold θ_{div}, driving finer-resolution exploration. Unlike HyperNEAT, both the connectivity *and* the substrate topology emerge from this recursive quadtree decomposition rather than being specified in advance.

Figure 2.7 details the three-stage substrate construction process: a NEAT population of CPPNs is queried across the coordinate space, the quadtree identifies regions of high output variance for node placement, and the resulting sparse substrate is assembled with CPPN-determined weights. Figure 2.8 shows the complete evolutionary loop that

wraps this construction step, highlighting the per-individual bottleneck that motivates Chapter 5.

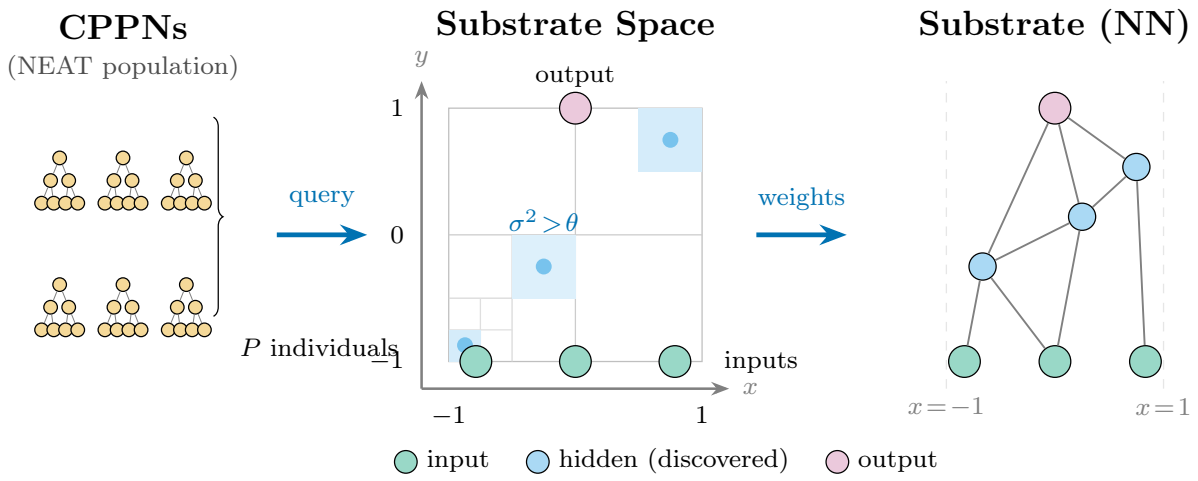


Figure 2.7: ES-HyperNEAT substrate construction. **Left**: A NEAT population of CPPNs, each with 4-dimensional input (x_1, y_1, x_2, y_2). **Center**: The CPPN is queried across the $[-1, 1]^2$ coordinate space; a quadtree recursively subdivides regions of high output variance ($\sigma^2 > \theta$), placing hidden nodes (blue dots) where information density is greatest. Shaded cells indicate regions that triggered further subdivision. **Right**: The resulting substrate neural network, with sparse connectivity determined by the CPPN's weight outputs.

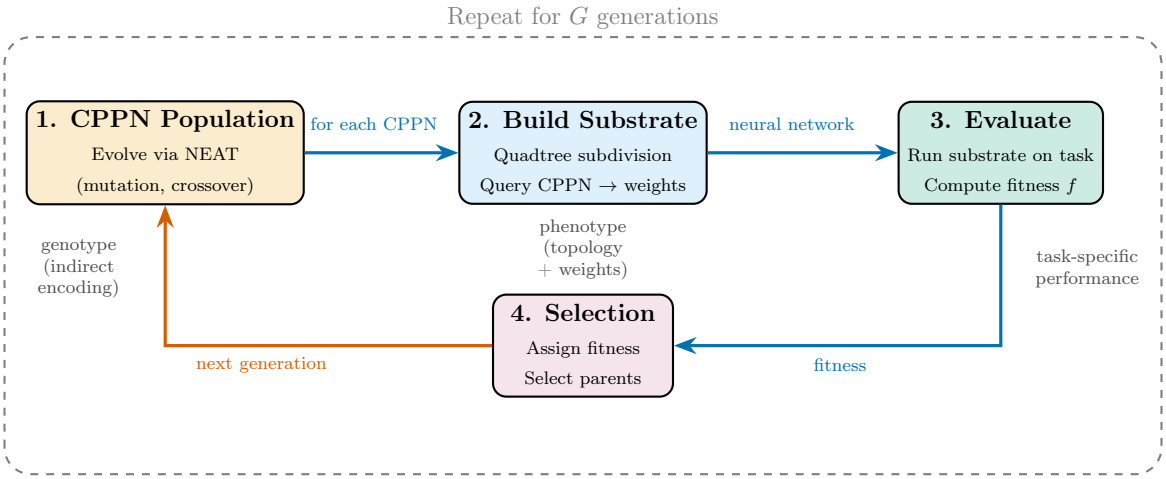


Figure 2.8: ES-HyperNEAT evolutionary loop. Each generation: (1) a NEAT population of CPPNs is evolved through mutation and crossover; (2) each CPPN is queried via quadtree subdivision to construct a substrate neural network; (3) each substrate is evaluated on the target task; (4) fitness-based selection determines which CPPNs reproduce. The critical bottleneck is Step 2: each CPPN produces a *unique* quadtree subdivision, yielding a different substrate topology per individual. This per-individual variability prevents population-level batching on GPU hardware (Chapter 5).

The Quadtree Bottleneck

Despite its conceptual elegance, ES-HyperNEAT's quadtree algorithm has a computational limitation: it is inherently sequential. Each CPPN in the population produces a *unique* set of quadtree subdivisions, resulting in a different substrate topology per individual. This per-CPPN variability prevents population-level vectorization: one cannot batch-process the substrate discovery step across the population because each individual follows a different subdivision path. On modern GPU hardware, where parallelism is the primary source of speedup, this sequential bottleneck limits scalability. We characterize this limitation formally in Chapter 5 and present a solution that reformulates adaptive substrate discovery for parallel execution in Chapter 6.

2.1.7 DES-HyperNEAT: Deep Evolvable Substrates

Tenstad and Haddow [94] further extended ES-HyperNEAT to Deep Evolvable-Substrate HyperNEAT (DES-HyperNEAT), supporting deep substrates with multiple hidden layers. The same quadtree-based decision mechanism is applied at two levels, within each hidden layer and between adjacent layers, so that both the topology within each layer and the number of hidden layers are discovered by evolution rather than specified (Figure 2.9). Each hidden layer and each inter-layer path carries its own evolved CPPN; each layer pair undergoes its own variance analysis, so node placements are tailored to the information flow at that depth. DES-HyperNEAT inherits both the adaptive discovery capabilities and the quadtree bottleneck that Chapter 5 analyzes.

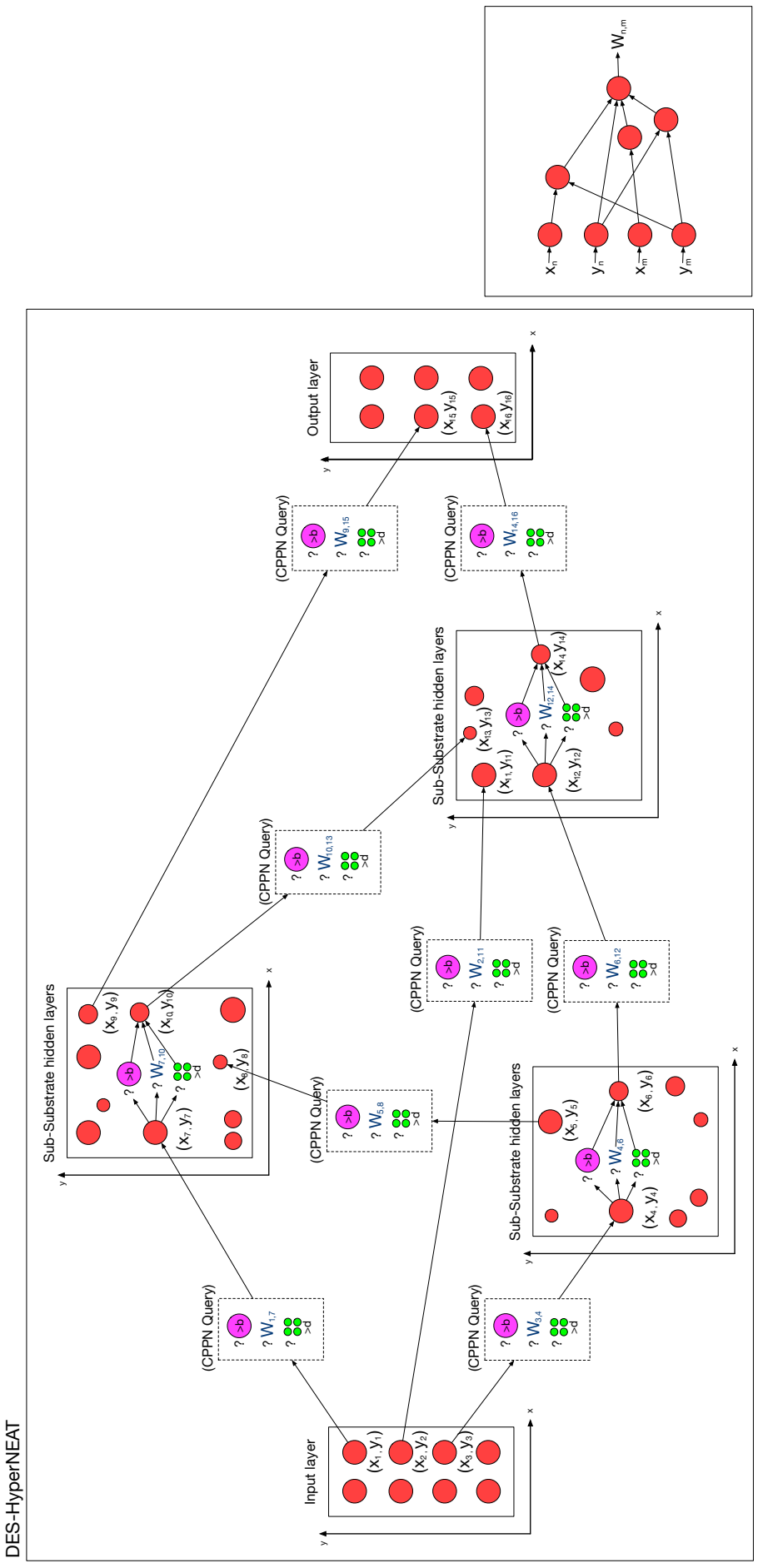


Figure 2.9: Overview of DES-HyperNEAT. DES-HyperNEAT extends ES-HyperNEAT's decision mechanism (Figure 2.6) to a *deep* substrate with multiple hidden layers. The same decision applies at two levels: **within** each sub-substrate hidden layer, the CPPN is queried at candidate positions to either express a connection with weight $W_{i,j}$, whose endpoint becomes a new hidden node (drawn as two markers, the weight label and the purple node $> b$), or subdivide the quadrant further ($> d$, green markers); **between** adjacent layers, the dashed *CPPN Query* boxes apply the same decision to determine inter-layer connectivity (e.g., $W_{1,7}$ from input to the first hidden layer). The CPPN box (right) shows the query interface, normalized source/target coordinates (x_n, y_n, x_m, y_m) in and weight $W_{n,m}$ out; in DES-HyperNEAT each layer and each path evolves its own CPPN behind this interface. As a result, both the number of hidden layers and the topology within each layer are discovered by evolution rather than specified in advance.

2.1.8 GEENNS: Toward Compositional Substrates

The evolutionary lineage from NEAT through DES-HyperNEAT produces increasingly sophisticated substrates: from direct-encoded networks (NEAT) to indirectly-encoded fixed substrates (HyperNEAT) to adaptive single-layer substrates (ES-HyperNEAT) to adaptive deep substrates (DES-HyperNEAT). Each algorithm still produces a single monolithic substrate for a single task. GEENNS (Grid-based Emergent Evolution of Neocortical Network Substrates) aims to extend this lineage to *compositional* substrates: multiple specialist substrates that can be independently evolved, frozen, and composed through evolved routing.

The GEENNS architecture, including its biological inspiration from neocortical columnar organization, the grid and mapper design, the multi-level CPPN hierarchy, the formal mathematical specification, and the prototype validation, is presented in Chapter 7. This section provides only the lineage context (Figure 2.10) needed to understand how the thesis contributions serve as prerequisites for compositional substrate neuroevolution.

This thesis addresses the substrate-level prerequisites: computational efficiency (Chapter 6), function-level expressiveness (§8.4), and multi-task capability (Chapter 4 and §8.4). These must be solved before the full GEENNS architecture can be realized.

Algorithmic Lineage

Figure 2.10 summarizes the evolutionary lineage from NEAT to GEENNS. Each successor algorithm adds a capability that addresses a limitation of its predecessor: HyperNEAT adds indirect encoding to overcome NEAT's direct-encoding scalability ceiling; ES-HyperNEAT adds adaptive substrate discovery to remove HyperNEAT's fixed-topology requirement; DES-HyperNEAT extends adaptive discovery to deep multi-layer substrates; and GEENNS aims to add compositional architecture through multi-grid organization and evolved mappers. This thesis contributes four substrate-level results that cluster around ES-HyperNEAT: hyperparameter optimization to make its configuration space navigable (Chapter 3), diagnosis of its central bias with a spatial-decomposition remedy (Chapter 4), a formal analysis of its quadtree bottleneck (Chapter 5), and the EMR-HyperNEAT reformulation that replaces the quadtree with tensor-accelerated adaptive substrates (Chapter 6). A proof-of-concept prototype (Chapter 7) then validates the core compositional principle that motivates GEENNS.

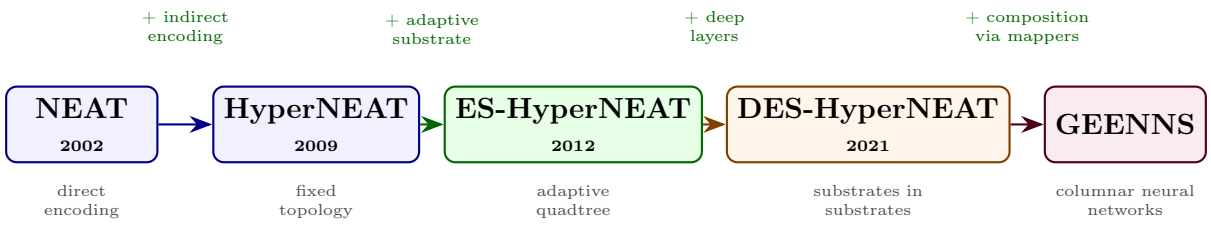


Figure 2.10: Algorithmic lineage from NEAT to GEENNS. Labels above arrows indicate the capability each successor adds; labels below boxes identify each algorithm's defining characteristic. Node colors follow the thesis part structure: blue (Part I, Foundations), green (Part II, Scaling ES-HyperNEAT), orange (Part III, Beyond the Quadtree), purple (Part IV, The GEENNS Vision).

2.1.9 The Central Bias: A Structural Consequence of CPPN and Quadtree Geometry

ES-HyperNEAT's variance-driven quadtree discovers substrate nodes where the CPPN output changes rapidly. While the principle is sound in theory, the activation functions that CPPNs use introduce a systematic spatial bias toward the origin of the coordinate space, which the quadtree's origin-anchored refinement then amplifies.

Why variance concentrates near the origin. Most CPPN activations concentrate their output near the origin, albeit through two distinct mechanisms. Sigmoid and tanh exhibit their highest *gradient* at $x = 0$: they transition most rapidly through their midpoint and saturate at large $|x|$, so the output varies most where inputs are small. Gaussian exhibits its peak *value* at $x = 0$: e^{-x^2} is maximal at the origin and decays toward zero at the boundaries, so the strongest activations occur there. In a composed multi-layer CPPN, both behaviors reinforce: the network's output is systematically most responsive and most variable near the origin, and least so near the periphery.

Only periodic functions (sine) and the linear function escape this pattern: sine varies uniformly across the entire input range, and the linear function has constant gradient everywhere. However, a CPPN composed of multiple layers inevitably passes outputs through non-periodic functions at some depth, re-introducing the central concentration.

The quadtree's entry point. A second, algorithmic property compounds the CPPN-side geometry. The quadtree requires a choice of root coordinate: ES-HyperNEAT's canonical formulation places the root QuadPoint at the center of the coordinate space, covering the full $[-1, 1]^2$ domain (Algorithm 3, Phase 1), so every subsequent subdivision is symmetric about the origin. A practitioner could in principle shift this anchor or scale the coordinate space, but the algorithm as published commits to the origin before any task-specific evidence is gathered, and the optimal anchor is not known in advance, because the purpose of the search is to discover where task-relevant features lie. Combined with the CPPN-side variance concentration, the two properties reinforce: the algorithm's geometric center and the CPPN's variance peak both sit at $(0, 0)$, so variance-driven refinement deepens precisely where the quadtree's structure was already centered. Tuning θ_{div} or θ_{band} modulates how much structure is refined and expressed, but cannot displace this coincidence.

How the quadtree amplifies the bias. Because the quadtree subdivides only where variance exceeds the threshold (Equation 2.11), the concentration of CPPN variance near the origin translates directly into a concentration of substrate nodes near the origin. The central quadrants subdivide to maximum depth while peripheral quadrants terminate early. The result is substrates where the majority of nodes cluster in a small spatial region around $(0, 0)$.

For tasks where relevant information is distributed across the full input space (such as MNIST, where digit strokes appear at all locations in the 28×28 image), this central clustering causes the substrate to attend to only a fraction of the input. Peripheral pixels are either unrepresented or represented at coarse resolution, limiting the information available to the evolved network.

This central bias is not a tunable parameter or an implementation artifact; it is a structural consequence of the mathematical properties of standard CPPN activation functions

interacting with the variance-based node placement criterion and the quadtree's origin-anchored refinement. Chapter 4 presents a spatial decomposition approach that addresses this limitation.

2.1.10 Indirect Encoding: Mathematical Formulation

This subsection provides the formal mathematical framework for CPPN-based indirect encoding and establishes notation used throughout the thesis.

Formal CPPN Definition

A CPPN is a directed graph $G = (V, E)$ where V is a set of nodes and E is a set of directed edges. Each node $v_i \in V$ has an associated activation function f_i drawn from a palette $\mathcal{F} = \{f^{(1)}, f^{(2)}, \ldots, f^{(k)}\}$. Each edge $(v_i, v_j) \in E$ has an associated weight $w_{ij} \in \mathbb{R}$.

The nodes are partitioned into three sets: input nodes V_{in}, hidden nodes V_{hid}, and output nodes V_{out}, with $V = V_{\text{in}} \cup V_{\text{hid}} \cup V_{\text{out}}$. The CPPN computes a function $\mathcal{C} : \mathbb{R}^{|V_{\text{in}}|} \to \mathbb{R}^{|V_{\text{out}}|}$ defined by the composition of node activation functions through the network graph.

For a feedforward CPPN with nodes ordered topologically as $v_1, v_2, \ldots, v_{|V|}$, the output of node v_j is:

$$a_j = f_j\left(\sum_{(v_i,v_j)\in E} w_{ij} \cdot a_i\right) \tag{2.7}$$

where a_i is the output of node v_i (equal to the corresponding input value for input nodes).

HyperNEAT Query Formalism

In the HyperNEAT framework, the CPPN maps source-target coordinate pairs to substrate properties. For a standard 2D substrate, the input space is $\mathbb{R}^4$:

$$\mathcal{C} : (x_s, y_s, x_t, y_t) \mapsto (w, b, \ldots) \tag{2.8}$$

where (x_s, y_s) are the source node coordinates, (x_t, y_t) are the target node coordinates, w is the connection weight, and b is the target node bias. Additional outputs can be appended as needed by extending the CPPN's output layer.

The substrate construction process evaluates the CPPN at all relevant coordinate pairs. For a substrate with n_s source nodes and n_t target nodes, this requires $n_s \times n_t$ CPPN queries. Each query produces a candidate connection; the substrate includes only those connections where the weight magnitude exceeds a minimum threshold $w_{\min}$.

Formally, the substrate weight matrix $\mathbf{W} \in \mathbb{R}^{n_t \times n_s}$ is:

$$W_{ji} = \begin{cases} \mathcal{C}_w(x_i, y_i, x_j, y_j) & \text{if } |\mathcal{C}_w(x_i, y_i, x_j, y_j)| > w_{\min} \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

where $\mathcal{C}_w$ denotes the weight output channel of the CPPN.

Geometric Properties of CPPN Composition

The geometric properties of CPPN-generated patterns arise from the mathematical properties of the activation functions composed through the network. We identify three fundamental geometric properties exploited by HyperNEAT and its extensions:

Symmetry. A Gaussian function $g(x) = e^{-x^2}$ is symmetric about $x = 0$: $g(x) = g(-x)$. When a Gaussian node receives the difference of two coordinates $(x_s - x_t)$ as input, it produces connectivity patterns where the weight depends on the distance between source and target regardless of direction. This bilateral symmetry is exploited in tasks with geometric symmetry (e.g., bipedal locomotion).

Periodicity. A sine function $\sin(\omega x)$ produces repeating patterns with period $2\pi/\omega$. When applied to spatial coordinates, sine nodes generate regular, repeating connectivity motifs: stripes of strong and weak connections that tile the substrate. This is useful for tasks requiring regular spatial sampling (e.g., visual feature detection).

Locality. Gaussian functions also encode locality: $e^{-(x_s - x_t)^2}$ produces strong connections between nearby nodes and weak connections between distant ones. This local connectivity prior matches many natural tasks where spatial proximity implies functional relevance.

These properties compose through the CPPN layers. A sine node feeding into a Gaussian one produces a pattern that is both periodic and locally concentrated: periodic bumps rather than smooth waves. The evolutionary process discovers which compositions match the task requirements.

ES-HyperNEAT Variance Criterion

The variance criterion that drives ES-HyperNEAT's adaptive resolution can be expressed formally. For a quadrant Q, the CPPN is evaluated at four probe positions $\{p_1, p_2, p_3, p_4\}$ located at the centers of the four sub-quadrants of Q; the CPPN outputs at these positions are $\{o_1, o_2, o_3, o_4\}$ where $o_i = \mathcal{C}(p_i)$. The variance is:

$$\text{Var}(Q) = \frac{1}{4}\sum_{i=1}^{4}(o_i - \bar{o})^2, \quad \bar{o} = \frac{1}{4}\sum_{i=1}^{4} o_i \tag{2.10}$$

The subdivision decision is:

$$\text{subdivide}(Q) = \begin{cases} \text{true} & \text{if Var}(Q) > \theta \text{ and depth}(Q) < D \\ \text{false} & \text{otherwise} \end{cases} \tag{2.11}$$

The total number of positions discovered is bounded by $\sum_{d=0}^{D} 4^d = (4^{D+1} - 1)/3$. In practice, the division threshold θ (written θ_{div} in §2.1.6; the extraction-side threshold θ_{var} is distinct) causes most branches to terminate early, so substrates contain far fewer nodes than the maximum. In this thesis's own XOR measurements, the quadtree yields 1–2 hidden nodes per substrate (Chapter 5, §5.4).

2.2 Hyperparameter Optimization

Evolutionary algorithms expose many hyperparameters: population size, mutation rates, crossover probabilities, selection pressures, and (for indirect encoding methods) substrate-level parameters such as the variance threshold θ and maximum depth D. ES-HyperNEAT is particularly hyperparameter-sensitive because it introduces parameters at multiple levels (the NEAT evolutionary engine, the CPPN encoding, and the quadtree substrate

discovery), creating a combinatorial configuration space. This section reviews the optimization methods used to search this space.

2.2.1 The Hyperparameter Problem in Neuroevolution

The hyperparameter problem in neuroevolution differs from its counterpart in gradient-based deep learning in several respects. First, the evaluation of a single hyperparameter configuration requires running an entire evolutionary process (typically hundreds of generations), not just a training run, making each evaluation expensive. Second, the fitness landscape is stochastic: the same configuration can produce different results across random seeds, requiring multiple seeds per configuration for reliable assessment. Third, many hyperparameters interact non-linearly: the optimal mutation rate depends on the population size, which depends on the problem complexity, which depends on the substrate depth.

For ES-HyperNEAT, the configuration space is particularly large. The CPU-based library used in this work (neat-python 0.92) exposes 59 configurable keys (population dynamics, genome encoding, species management, stagnation detection); with the eight substrate-side parameters (the six enumerated below plus the connection-weight cap and the substrate activation function) the surface reaches 67 keys, of which roughly 40 are genuinely tunable. About 30 of those control NEAT's core mutation dynamics: rates and magnitudes for weight, bias, and response; plus compatibility, speciation, and reproduction. Risi and Stanley's 2012 ES-HyperNEAT formulation adds 6 more: initial and maximum quadtree depth, division and variance thresholds, band cutoff, and iteration level [78]. Even a representative 16-parameter subspace, with modest value ranges per parameter, spans billions of configurations [11], making exhaustive search infeasible. Table C.3 (Appendix C) enumerates the principal hyperparameters selected for the TPE study at each level, showing the ranges that define this combinatorial space.

Random search, as Bergstra and Bengio [3] demonstrated, and Bayesian optimization [83, 86] provide the two ends of the hyperparameter search spectrum: the former samples configurations uniformly at random (serving as the baseline that any principled method must beat), while the latter uses a probabilistic surrogate model to select promising configurations via an acquisition function such as Expected Improvement. Further details on both methods are provided in Appendix C.4.

2.2.2 Tree-structured Parzen Estimators

The Tree-structured Parzen Estimator (TPE), introduced by Bergstra et al. [4], is a Bayesian optimization method that inverts the standard surrogate modeling approach. Instead of modeling $p(y \mid \mathbf{x})$ (performance given configuration), TPE models $p(\mathbf{x} \mid y)$ (configuration given performance) by splitting the evaluated configurations into two groups:

$$p(\mathbf{x} \mid y) = \begin{cases} \ell(\mathbf{x}) & \text{if } y < y^* \\ g(\mathbf{x}) & \text{if } y \geq y^* \end{cases} \tag{2.12}$$

where y^* is the γ-quantile of observed objective values (with $\gamma = 0.25$ by default, so $y < y^*$ selects the best-performing 25% of configurations), $\ell(\mathbf{x})$ is a density estimate over the "good" configurations (those with $y < y^*$), and $g(\mathbf{x})$ is a density estimate over the remaining configurations. In the standard TPE formulation, y is the objective to be

minimized; $y < y^*$ therefore selects configurations with the lowest (best) objective values. Both densities are estimated using kernel density estimation with Parzen windows.

Bergstra et al. [4] show that under this model, Expected Improvement is proportional to:

$$\mathrm{EI}(\mathbf{x}) \propto \left(\gamma + \frac{g(\mathbf{x})}{\ell(\mathbf{x})}(1-\gamma)\right)^{-1} \tag{2.13}$$

where $\gamma = p(y < y^*)$ is the quantile threshold. Since this expression is monotonically increasing in $\ell(\mathbf{x})/g(\mathbf{x})$, maximizing Expected Improvement reduces to maximizing the density ratio. TPE exploits this by sampling candidates from $\ell(\mathbf{x})$ and selecting those with the highest $\ell(\mathbf{x})/g(\mathbf{x})$.

Figures 2.11 and 2.12 illustrate this process on a synthetic two-dimensional example. The density $\ell(\mathbf{x})$ (Figure 2.11, left) concentrates around configurations that produced high performance, while $g(\mathbf{x})$ (Figure 2.11, right) covers the poorly performing regions. Figure 2.12 then shows how TPE combines these densities for candidate selection: new candidates (green dots) are sampled from $\ell(\mathbf{x})$ and evaluated against the $g(\mathbf{x})$ landscape, selecting configurations where the ratio $\ell(\mathbf{x})/g(\mathbf{x})$ is highest.

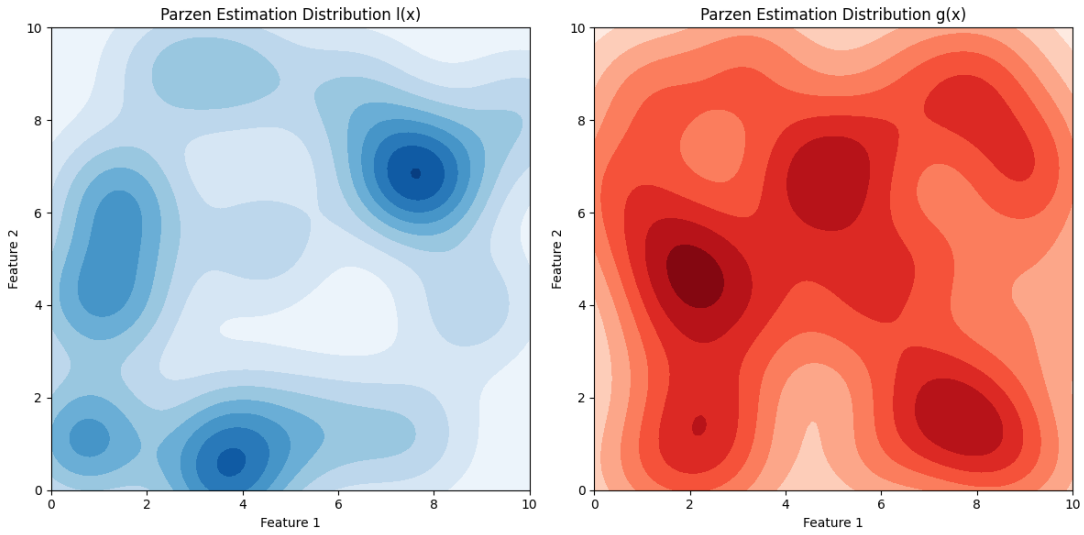


Figure 2.11: Parzen estimation density contours for a two-dimensional TPE example. Left: $\ell(\mathbf{x})$, the density over "good" configurations (top 25% by performance). Right: $g(\mathbf{x})$, the density over remaining configurations. TPE selects configurations that maximize $\ell(\mathbf{x})/g(\mathbf{x})$: regions that are dense in good configurations and sparse in bad ones.

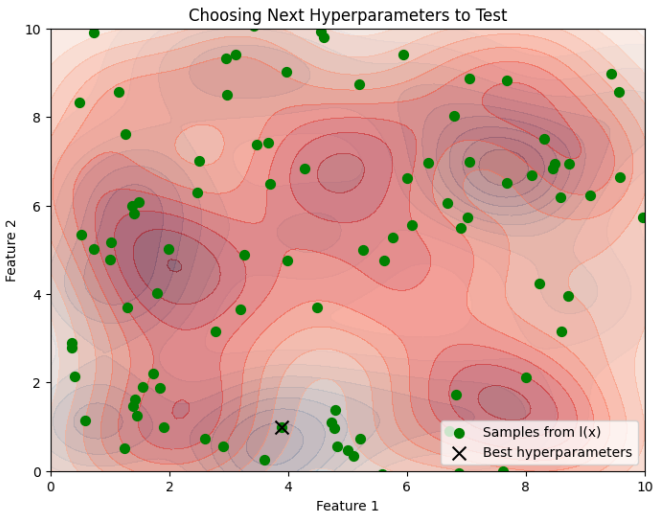


Figure 2.12: TPE candidate selection in the same two-dimensional space. The background contours show $g(\mathbf{x})$ (the density over poorly performing configurations). Green dots are candidates sampled from $\ell(\mathbf{x})$; the cross marks the current best. TPE evaluates these candidates by their $\ell(\mathbf{x})/g(\mathbf{x})$ ratio, preferring regions where good configurations are dense and bad configurations are sparse.

TPE has several properties that make it well-suited for neuroevolution hyperparameter optimization:

- **Native support for discrete and conditional parameters**: Unlike Gaussian Process-based methods that operate in continuous spaces, TPE naturally handles integer parameters (population size), categorical parameters (activation function palette), and conditional parameters (parameters that exist only when another parameter takes a specific value).
- **Scalability**: TPE uses kernel density estimation rather than Gaussian Process regression, avoiding the $O(N^3)$ covariance-matrix inversion that limits GP-based methods to a few hundred evaluations and making TPE practical for settings with thousands. The "tree" in the name refers to the handling of conditional dependencies, not to a space-partitioning scheme.

2.2.3 Multi-Fidelity Methods: Hyperband and Early Stopping

Standard Bayesian optimization treats each evaluation as atomic: a configuration is either fully evaluated or not evaluated at all. Multi-fidelity methods exploit the observation that bad configurations can often be identified early, before the full evaluation completes.

Successive Halving

Successive halving [44] starts with a large set of n configurations, each allocated a small budget (e.g., a few generations of evolution). After this initial evaluation, the bottom half is discarded and the surviving configurations receive double the budget. This halving continues until a single configuration remains, evaluated at full fidelity. The total cost is $O(B \log n)$ rather than $O(n \cdot B)$ where B is the budget needed to evaluate a single

configuration at full fidelity: summing the $\log_2 n$ rounds (each with the same total work $n \cdot B/n = B$) yields $B \log_2 n$, a speedup of $n/\log n$ over evaluating all n configurations at full budget.

Hyperband

Li et al. [60] introduced Hyperband, which addresses the key limitation of successive halving: the trade-off between the number of configurations n and the per-configuration budget B/n. Starting with many cheap evaluations identifies the best configuration quickly but risks discarding configurations that start slowly. Starting with few expensive evaluations gives each configuration a fair chance but explores fewer options.

Hyperband resolves this by running successive halving with multiple trade-off settings in parallel, called "brackets." Each bracket uses a different ratio of configurations to per-configuration budget. The most aggressive bracket evaluates many configurations cheaply; the most conservative bracket evaluates few configurations thoroughly. By running all brackets, Hyperband hedges against the unknown relationship between early performance and final performance.

Early-Stopping Predictors

An alternative to successive halving is to predict, from partial evaluation results, whether a configuration will ultimately succeed or fail. This prediction-based approach requires a model that maps early-stage metrics (e.g., fitness at generation 3) to final outcomes (converged or not by generation 100). If the model is accurate, bad configurations can be terminated as soon as the predictor is confident, without waiting for a fixed halving schedule.

The effectiveness of early-stopping prediction depends on two factors: the correlation between early and final performance (higher correlation enables earlier stopping), and the cost of false positives (terminating a configuration that would have succeeded). We develop a specific early-stopping predictor for ES-HyperNEAT on the MNIST benchmark in Chapter 3, where we demonstrate that generation-3 fitness is a reliable predictor of convergence outcome ($F_1 = 0.872$).

2.3 Mixture of Experts and Modular Architectures

The Mixture of Experts (MoE) framework decomposes complex problems by routing different inputs to specialized sub-models (experts) and combining their outputs. This section reviews the MoE framework, its modern incarnations, and its connection to the modular architectures that motivate our spatial decomposition work.

The historical development of MoE architectures, from Jacobs et al. [43] through Shazeer et al. [84], is detailed in Appendix C.8. The core idea is that multiple expert networks compete to handle different input regions, coordinated by a gating network that routes inputs to the most appropriate expert(s). Modern sparsely-gated MoE [84] activates only 1–2 experts per input, enabling conditional computation at scale.

2.3.1 MoE and Modular Neuroevolution

The MoE principle has a natural connection to neuroevolution through modular network architectures. Reisinger et al. [77] demonstrated that evolving networks with distinct,

reusable modules improves performance on compositional tasks. Modularity need not be designed in: Kashtan and Alon [49] showed that modularly varying environments spontaneously evolve modular network structure, even without explicit modularity objectives. Subsequent work established that connection-cost objectives also produce modular topologies: Clune et al. [20] demonstrated that performance-plus-connection-cost selection produces modular networks across a range of problems.

In this thesis, we take inspiration from MoE for spatial decomposition: instead of a single monolithic ES-HyperNEAT substrate processing the entire input, we use multiple specialist substrates that each process a partition of the input space. This MoE-inspired partitioning is developed in Chapter 4, where we demonstrate that structured input decomposition via independent experts achieves 105% improvement over the monolithic ES-HyperNEAT baseline on MNIST digit classification. The pattern is not a general feature of HyperNEAT; it is our reframing of the classical MoE idea as a remedy for the central bias diagnosed there.

Modern sparse MoE at scale. Switch Transformers [29] and GShard [59] use learned gating functions to route tokens to fixed-architecture expert networks. The partitioned-experts architecture in Chapter 4 differs in one respect: the expert architectures themselves are evolved, not fixed. Each expert substrate discovers its own topology through neuroevolution, while the gating mechanism (spatially partitioned input assignment with F1-weighted aggregation) coordinates their outputs. The GEENNS vision (Chapter 7) takes this further: evolved expert topologies composed through evolved mappers, where both the computational primitives and the composition strategy are discovered by evolution rather than specified by engineers, whose role recedes to choosing the encodings and the selection pressures.

2.4 GPU Acceleration for Evolutionary Algorithms

Modern GPU hardware offers orders-of-magnitude speedup for computations that map well to its execution model. Evolutionary algorithms, however, are inherently irregular: individuals with variable topologies, conditional branching during fitness evaluation, and population diversity all create challenges for GPU execution. This section reviews the GPU programming model, the specific constraints it imposes, and the frameworks that have addressed these constraints for evolutionary computation.

2.4.1 GPU Programming Model

Modern GPUs use the Single Instruction, Multiple Threads (SIMT) architecture, where thousands of lightweight threads execute the same program on different data. The execution hierarchy (warps of 32 threads executing in lockstep, thread blocks communicating via shared memory, and grids spanning the full computation) is detailed in Appendix C.2. The critical property for this thesis is *warp divergence*: when threads in a warp take different branches of a conditional, both branches must be executed serially, which is the primary source of GPU inefficiency for irregular algorithms such as the quadtree.

2.4.2 Memory Hierarchy and Coalescing

GPU memory spans four levels from fast per-thread registers to slow device-wide global memory (Appendix C.3). The key constraint is *memory coalescing*: consecutive threads accessing consecutive memory addresses combine into efficient transactions, while scattered accesses (as produced by tree structures and pointer-based navigation) sharply reduce bandwidth.

2.4.3 Why Uniform Computation Matters

The SIMT architecture imposes a fundamental constraint: GPU efficiency requires *uniform computation*, with all threads executing the same operations on data of the same shape. This constraint has three aspects:

1. **No branching**: Conditional branches cause warp divergence. Algorithms that make different decisions for different data elements (e.g., "subdivide this quadrant if its variance exceeds θ") force some threads to wait while others execute.

2. **Regular data layout**: Data must be organized in dense, contiguous arrays for coalesced memory access. Tree structures (quadtrees, octrees) with pointer-based navigation produce scattered memory accesses.

3. **Fixed work per element**: All threads should perform the same amount of work. Adaptive algorithms where some elements require more processing than others (e.g., deeper quadtree branches for some CPPNs) create load imbalance.

These constraints directly conflict with ES-HyperNEAT's quadtree, which is branch-heavy, topologically heterogeneous, and adaptive per individual. We formalize this conflict in Chapter 5.

2.4.4 JAX: Composable Transformations for Scientific Computing

JAX [6] is a Python library for high-performance numerical computing that provides four key transformations (`jit`, `vmap`, `pmap`, and `grad`) detailed in Appendix C.12.

JAX's compilation model imposes a key constraint: all array shapes must be known at compile time. Functions that produce variable-size outputs (e.g., ES-HyperNEAT's quadtree, which produces a different number of nodes per individual) cannot be compiled directly. This constraint must be addressed either by padding to a maximum size (wasting memory) or by redesigning the algorithm to produce fixed-size outputs (Chapter 6).

2.4.5 GPU-Accelerated Evolutionary Computation

Several frameworks have demonstrated the viability of running evolutionary algorithms on GPU hardware:

EvoJAX. Tang et al. [93] demonstrated GPU-accelerated Evolution Strategies[2] in JAX, reporting 10–20× speedup over CPU implementations. EvoJAX exploits the fact that Evolution Strategies, unlike NEAT, use fixed-topology networks, enabling straightforward vectorization across the population: when all individuals have the same network shape, network evaluation reduces to a single batched matrix multiplication.

TensorNEAT. Wang et al. [101] introduced TensorNEAT, which brings NEAT to JAX by representing genomes as fixed-size tensors with masking for inactive genes. Each genome is padded to a maximum number of nodes and connections; inactive genes are masked during computation. This tensor representation enables `vmap`-based population evaluation but introduces a fundamental tension: NEAT's strength is variable-topology genomes, but JAX's strength is fixed-shape tensors. TensorNEAT resolves this by setting a maximum genome size and accepting the memory overhead of padding.

TensorNEAT implements standard NEAT, HyperNEAT, and provides the infrastructure for ES-HyperNEAT. It serves as the implementation foundation for Chapters 5–7; the earlier chapters (Chapters 3–4) use the CPU-based `neat-python` [66] library.

2.4.6 The Core Challenge: Irregular Algorithms on Regular Hardware

Evolutionary algorithms are inherently irregular in ways that conflict with GPU execution requirements:

- **Variable topology**: Each individual in a NEAT population has a different network structure. Fitness evaluation must handle networks of different sizes and shapes.
- **Speciation**: Different species undergo different selection pressures, creating branching in the evolutionary logic.
- **Adaptive substrates**: ES-HyperNEAT produces different substrate topologies for different individuals, preventing uniform substrate evaluation.
- **Stochastic operations**: Mutation and crossover involve random decisions that differ per individual, creating divergent execution paths.

TensorNEAT addresses the first three through tensor padding and masking: represent all genomes as tensors of the same shape, mask inactive elements. This works for NEAT's genome representation and for standard HyperNEAT (where the substrate shape is predefined). It fails for ES-HyperNEAT's quadtree, where the per-individual adaptation is the algorithm's core feature, not an incidental irregularity.

This tension between adaptive substrate discovery and massive parallelism is the central computational challenge addressed in this thesis. Chapter 5 proves that the quadtree approach is incompatible with massively parallel hardware, and Chapter 6 presents a reformulation that preserves adaptivity while enabling full parallel execution.

[2]"Evolution Strategies" here refers to the algorithm family (e.g., CMA-ES, OpenAI-ES) that samples from a parameter distribution and updates that distribution from observed returns. It is *unrelated* to the "ES" in *ES-HyperNEAT*, which abbreviates *Evolvable-Substrate*. The two families share only a two-letter acronym.

2.5 Problem Definitions and Evaluation Methodology

This section defines the benchmark problems, convergence criteria, and statistical methodology used throughout the experimental chapters of this thesis. Subsequent chapters build on these definitions and state only their deviations from them; Appendix C.11 lists every deviation.

2.5.1 Boolean Logic Problems

We use six two-input Boolean logic tasks as our primary benchmark for evaluating substrate quality on simple problems. These tasks are deliberately simple (each has only four input patterns), allowing precise analysis of failure modes without confounding factors such as dataset size, generalization gaps, or stochastic sampling.

Table 2.2 defines the truth tables for all six tasks.

Table 2.2: Truth tables for the six Boolean logic benchmark tasks. Inputs $x_1, x_2 \in \{0, 1\}$. XOR and XNOR are the non-linearly separable tasks.

x_1	x_2	**XOR**	**AND**	**OR**	**NAND**	**NOR**	**XNOR**
0	0	0	0	0	1	1	1
0	1	1	0	1	1	0	0
1	0	1	0	1	1	0	0
1	1	0	1	1	0	0	1

The six tasks partition into two computational classes:

- **Linearly separable**: AND, OR, NAND, NOR. These can be computed by a single neuron with a threshold: $y = \mathbb{1}[\mathbf{w}^\top \mathbf{x} + b > 0]$. For example, AND is computed by $w_1 = w_2 = 1, b = -1.5$.
- **Non-linearly separable**: XOR and XNOR. Neither can be computed by a single threshold neuron. For XOR, no linear boundary separates the positive examples $(0, 1), (1, 0)$ from the negative examples $(0, 0), (1, 1)$. XNOR is the complement of XOR: its positive examples $\{(0, 0), (1, 1)\}$ are equally inseparable from $\{(0, 1), (1, 0)\}$. Both require at least one hidden layer.

Fitness evaluation. For all Boolean tasks, fitness is computed as one minus the mean squared error across all four input patterns:

$$\phi = \frac{1}{4} \sum_{i=1}^{4} \left(1 - (y_i - \hat{y}_i)^2\right) \tag{2.14}$$

where y_i is the target output and $\hat{y}_i$ is the network's output for input pattern i. For multi-task evaluation, the product fitness across all K tasks enforces simultaneous convergence:

$$\phi_{\text{multi}} = \prod_{k=1}^{K} \phi_k \tag{2.15}$$

2.5.2 Parity-N Problems

The Parity-N problem generalizes XOR to N inputs: the output is 1 if and only if an odd number of inputs are 1. XOR is Parity-2. The truth table has 2^N entries, growing exponentially with N. Complete truth tables and scaling properties for the parity family are provided in Appendix C.1.

$$\text{Parity-}N(x_1, x_2, \ldots, x_N) = \bigoplus_{i=1}^{N} x_i = \left(\sum_{i=1}^{N} x_i \right) \bmod 2 \tag{2.16}$$

Parity is a canonical hard problem for neural networks for two reasons:

1. **No shortcut features**: Every input bit matters. Unlike classification tasks where a few features may be sufficient, parity requires integrating information from all inputs. Ignoring any single input guarantees at most 50% accuracy (on the patterns where that input determines the output).

2. **Exponential alternation**: The output alternates with every unit change in the Hamming weight of the input. A function that increases monotonically with the input sum (as threshold functions do) must misclassify a constant fraction of the input patterns, a fraction that approaches one half as N grows.

Fitness evaluation for Parity-N follows the same MSE formula as Boolean tasks, evaluated across all 2^N input patterns. We test Parity-N for $N \in \{2, 3, 4, 5, 6, 7, 8\}$ across the experimental chapters, with Parity-3 (8 patterns) serving as the primary benchmark.

Extended benchmark definitions, including visual discrimination, retina problem, orientation selectivity, Two Spirals, concentric circles, symmetry detection, Turing patterns, Tower of Hanoi, reinforcement learning control tasks, and financial regression tasks, are provided in Appendix C.10.

2.5.3 MNIST

The MNIST handwritten digit dataset [56] is a standard machine learning benchmark consisting of 70,000 grayscale images of handwritten digits (0–9). Each image is 28×28 pixels, yielding 784 input dimensions with pixel values normalized to $[0, 1]$. The dataset is partitioned into 60,000 training images and 10,000 test images. Classification requires mapping each 784-dimensional input to one of 10 output classes (digits 0 through 9).

Fitness evaluation. For neuroevolution experiments, fitness is defined as classification accuracy: the fraction of correctly classified images. Because evaluating all 60,000 training images per individual per generation is computationally prohibitive for population-based methods, we subsample 200 images per class (2,000 images total) per fitness evaluation, following the protocol of Claret et al. [11]. Final reported accuracy is measured on the full 10,000-image test set.

We use MNIST as the primary benchmark for hyperparameter optimization (Chapter 3) and spatial decomposition experiments (Chapter 4), where it tests the ability of evolved substrates to process high-dimensional input. MNIST is deliberately chosen not for its difficulty (modern gradient-based methods achieve >99% accuracy [58]) but for its diagnostic value: the spatial structure of digit images enables analysis of which regions the evolved substrate actually uses (receptive field analysis), and the well-understood nature

of the task enables precise attribution of performance differences to architectural choices rather than task-specific confounds.

Prior work by Verbancsics and Harguess [98] achieved 23.9% accuracy on MNIST using HyperNEAT with a seven-layer fully-connected substrate, or 27.7% with a hand-designed LeNet-5 CNN substrate [57]. Our TPE-optimized ES-HyperNEAT configuration reaches a 30-run mean of 20.95%, below the generic-substrate baseline's 23.9%, while evolving a far smaller, self-discovered substrate on a fraction of the search budget (population 100 for 20 generations, versus 256 for 2,500), with a validated peak of 29.0% established through 2,013 systematic trials and subsequent replication (Chapter 3). Our spatially partitioned experts approach achieves 43% mean accuracy (49% best run for that variant; 50% across control variants) by addressing the central bias problem (Chapter 4). The modest absolute accuracies reflect the challenge of scaling indirect encoding to 784-dimensional inputs, not a fundamental limitation of neuroevolution.

2.5.4 Convergence Criterion

Throughout this thesis, unless a chapter states otherwise (Appendix C.11), a run is said to **converge** when the best individual in the population achieves fitness $\phi \geq \phi^* = 0.95$:

$$\text{converged} \iff \phi \geq \phi^* \tag{2.17}$$

The fraction of runs meeting this criterion is reported as the **solve rate** (equivalently, success rate). The justification for this threshold, along with the definitions of convergence generation and success rate, is provided in Appendix C.11.

2.5.5 Statistical Methodology

The statistical methodology, including specific test procedures (Mann-Whitney U, Kruskal-Wallis, Fisher's exact), bootstrap confidence interval implementations, effect size measures, and the standardized reporting format, is detailed in Appendix C.11.

Differences between two percentage values are reported in percentage points (pp) to distinguish absolute from relative change: an improvement from 23.9% to 29.0% accuracy is a 5.1 pp absolute gain, not a 5.1% relative gain (which would mean $23.9\% \times 1.051 = 25.1\%$).

For multi-group comparisons, we report the ANOVA F-statistic as $F(\mathrm{df}_1, \mathrm{df}_2) = \text{value}$ followed by the p-value and effect size η^2, with post hoc pairwise differences via Tukey HSD where applicable. Notation and interpretation conventions are detailed in Appendix C.11.

2.5.6 Notation Summary

The mathematical notation used throughout this thesis is collected in the Notation and Symbols section (p. 33); symbols local to this chapter (δ for compatibility distance and, briefly, mutation noise; P for the population set) are defined at first use.

2.6 Relationship to Neural Architecture Search

Neural Architecture Search (NAS) automates the design of neural network architectures, sharing neuroevolution's goal of replacing hand-designed topologies with discovered ones.

Table 2.3 contrasts the dominant NAS approaches with the neuroevolutionary lineage developed in this thesis.

NAS methods discover optimal configurations *within* a hand-designed search space; neuroevolution discovers the search space itself. This distinction is consequential for compositional reasoning, where the required topology is unknown a priori. The penalty is search cost: gradient-based NAS (DARTS) completes in GPU-hours to a few GPU-days on CIFAR-scale tasks, while population-based neuroevolution requires days to weeks: a gap that the EMR-HyperNEAT acceleration (Chapter 6) partially closes.

Is neuroevolution NAS? In the broad sense (automating the design of network architectures) NEAT has performed NAS since 2002; under the narrow modern decomposition (a search strategy proposing architectures from a predefined space, with weights fitted separately by gradient descent [27]) it is not: evolution here carries the topology and the weights together and returns a functioning network rather than a blueprint awaiting training. ES-HyperNEAT sits one level above the modern formulation still: the object under search is a CPPN, an architecture *generator*, which is precisely what keeps its search space open where cell-based spaces are fixed. The full definitional analysis is in Appendix C.9.

Parallel adaptive substrate discovery. The "Adaptive substrate" row in Table 2.3 highlights a capability unique to the ES-HyperNEAT lineage: discovering the network's node placement, connectivity density, and topology automatically rather than requiring a predefined substrate layout. EMR-HyperNEAT (Chapter 6) is, to our knowledge, the first method that enables parallel adaptive substrate discovery, scaling with the parallel cores available on the target hardware (CPU, GPU, or TPU). No other entry in Table 2.3 combines adaptive substrate discovery with parallel-hardware execution, and the combination narrows the execution-time gap with cell-based methods.

What neuroevolution discovers that NAS cannot. NAS selects among predefined building blocks (convolutions, pooling, attention heads) and arranges them within a fixed macro-architecture. The building blocks themselves are hand-designed. Neuroevolution operates at a lower level of abstraction: it discovers the operations themselves. A substrate that solves 6-bit parity (at the cross-domain sweep's relaxed 0.80 fitness criterion) from an adaptively discovered, irregular node placement (Chapter 6) is a fundamentally different kind of topology from a 3×3 separable convolution placed at cell position $(2, 3)$ by DARTS. The former was discovered by evolution; the latter was selected from a human-designed menu. This matters for compositional reasoning, where the required computational primitives are unknown in advance: there is no menu to select from.

Diagnostic findings that generalize beyond ES-HyperNEAT. Several contributions in this thesis apply to any system that uses CPPNs for substrate generation, not only to ES-HyperNEAT. The central bias (Chapter 4) is a CPPN geometry phenomenon: any substrate generator that places nodes by CPPN output variance will concentrate neurons where the coordinate system places the origin, regardless of the evolutionary framework. The quadtree–GPU incompatibility (Chapter 5) applies to any recursive-subdivision-based substrate discovery mechanism on SIMT hardware, including DES-HyperNEAT and future extensions. The EMR reformulation (Chapter 6) applies to any CPPN-based architecture that requires parallel evaluation of spatially indexed candidate

positions. These findings are therefore relevant to the broader NAS and Automated Machine Learning (AutoML) communities whenever indirect encoding through geometric pattern generators is considered.

Table 2.3: Comparison of architecture search approaches. The abbreviation "HN" denotes HyperNEAT: **ES-HN** = Evolvable-Substrate HyperNEAT [78]; **EMR-HN** = Eager Multi-Resolution HyperNEAT (this thesis, Chapter 6). Cell-level search optimizes within a hand-designed macro-architecture; full-topology search discovers the architecture itself. The "Adaptive substrate" row distinguishes methods that automatically discover node placement and connectivity from those that require a predefined substrate layout.

	DARTS [61]	**ENAS** [72]	**NAS** [108]	**NEAT** [88]	**ES-HN** [78]	**EMR-HN** (this thesis)
Topology search	Cell	Cell	Cell	Full	Full	Full
Weight optim.	Gradient	Gradient	RL+Grad	Evolution	Evolution	Evolution
GPU-parallel	Yes	Yes	Yes	Partial	No	Yes
Search space	Fixed	Fixed	Fixed	Open	Open	Open
Indirect encoding	No	No	No	No	Yes	Yes
Adaptive substrate	No	No	No	No	Yes	Yes

2.7 Chapter Summary

This chapter has established the foundational material for the thesis. We now summarize the key points and identify the gaps that motivate the subsequent chapters.

Neuroevolution (§2.1.1–§2.1.10): The evolutionary algorithm template (§2.1.1) provides the population-based search framework that all subsequent methods instantiate. NEAT co-evolves topology and weights through speciation-protected search. CPPNs enable indirect encoding through geometric pattern generation. HyperNEAT uses CPPNs to generate substrate connectivity, providing scalability through decoupled genome and substrate complexity. ES-HyperNEAT adds adaptive substrate discovery through quadtree variance detection, and DES-HyperNEAT extends this to deep multi-layer substrates. GEENNS proposes a compositional extension—specialist substrates (grids) evolved independently, frozen, and recombined through evolved mappers—inspired by the neocortical column hypothesis [69]; the architecture is detailed in Chapter 7. The quadtree that underlies ES-HyperNEAT and DES-HyperNEAT is inherently sequential, creating a computational bottleneck for parallel-hardware acceleration (addressed in Chapter 5, with the solution in Chapter 6).

Hyperparameter optimization (§2.2): ES-HyperNEAT exposes a high-dimensional configuration space; the 16-parameter subspace with the ranges adopted in Chapter 3 exceeds 10^9 combinations. TPE provides an efficient Bayesian approach to navigating this space, with native support for the discrete and conditional parameters common in neuroevolution. Multi-fidelity methods (Hyperband, early stopping) reduce evaluation cost by terminating unpromising configurations early. These methods are evaluated in Chapter 3: TPE against random search on the full space, and a domain-specific early-stopping pruner that reduces intra-trial cost by 41.6% while maintaining statistically equivalent search quality.

Mixture of Experts (§2.3): Classical MoE routes inputs to specialized experts via a learned gating network. Taking inspiration from this framework, we develop a spatially partitioned-experts architecture in Chapter 4, where independent experts processing input subregions achieve 105% improvement over monolithic ES-HyperNEAT.

GPU acceleration (§2.4): The SIMT architecture requires uniform computation (no branching, regular data layout, fixed work per element). JAX and TensorNEAT provide the infrastructure for GPU-accelerated neuroevolution, but ES-HyperNEAT's quadtree violates every GPU requirement: it branches, produces non-uniform topology data per individual, and produces variable work per individual. This conflict is formalized in Chapter 5 and resolved in Chapter 6.

Benchmarks and methodology (§2.5): Boolean tasks, Parity-N, and MNIST provide a graduated benchmark suite spanning trivial to high-dimensional problems. The default convergence criterion ($\phi \geq \phi^* = 0.95$; Appendix C.11 lists the campaigns that tighten or relax it), statistical methodology (bootstrap CI, Mann-Whitney U, Kruskal-Wallis, Fisher's exact, ANOVA with Tukey HSD), and reporting format are standardized for all experimental chapters.

Relationship to NAS (§2.6): NAS methods search within hand-designed spaces; the neuroevolutionary lineage searches open topology spaces and, in ES-HyperNEAT's case, evolves the architecture generator itself. EMR-HyperNEAT (Chapter 6) narrows the execution-speed gap that separated adaptive substrate discovery from cell-based NAS, and the diagnostic findings (central bias, quadtree–SIMT incompatibility) apply to any CPPN-based substrate generator.

The gap that emerges from this background is threefold:

1. **Configuration**: ES-HyperNEAT has a vast configuration space, and each evaluation is expensive. Without principled hyperparameter search and early stopping, practical deployment is infeasible (Chapter 3).

2. **Representational**: ES-HyperNEAT's quadtree-based substrate discovery exhibits a central bias that originates in the CPPN's output geometry, whose generating components peak where the coordinate system places the origin, and is expressed by the quadtree's variance-driven node placement: substrates typically converge to a small central fraction of the input space, leaving peripheral regions weakly covered. This spatial collapse limits performance on high-dimensional tasks (Chapter 4).

3. **Computational**: ES-HyperNEAT's quadtree is incompatible with massively parallel hardware, limiting the scale of adaptive substrates to what can be processed sequentially on CPU (Chapter 5, solved in Chapter 6).

These three barriers (configuration, representational, and computational) form the organizing structure of the thesis. Part II addresses the configuration and representational barriers through hyperparameter optimization and spatial decomposition. Part III addresses the computational barrier through formal analysis of the quadtree limitation and the EMR-HyperNEAT solution. Part IV tests whether the unblocked platform supports compositional reasoning (Chapter 7), and Part V draws the conclusions.

Part II

Scaling ES-HyperNEAT

Chapter 3

Hyperparameter Optimization and Early-Stopping

> Premature optimization is the root of all evil.
>
> Donald Knuth

This chapter addresses the first thesis research question:

> **TRQ1.** How can the hyperparameter space of adaptive substrate neuroevolution be navigated efficiently?

In the implementation used in this work, ES-HyperNEAT exposes roughly 40 genuinely tunable hyperparameters: about 30 from NEAT-Python's mutation and reproduction dynamics, and 6 added by Risi and Stanley's ES-HyperNEAT formulation for quadtree substrate discovery [78]. For this study, we selected 16 of these parameters (9 governing Compositional Pattern-Producing Network (CPPN) evolution and 7 governing substrate formation) whose Cartesian product over the value ranges we explored spans more than 3×10^9 configurations, the vast majority of which produce substrates incapable of learning. Navigating this space is a prerequisite for every subsequent investigation in this thesis: without reliable hyperparameter settings, it is impossible to distinguish algorithmic limitations from configuration failures. The challenge is particularly acute for the GEENNS compositional architecture (Chapter 7), which requires evolving several dozen CPPNs simultaneously (34 for a modest 5-grid, 10-task configuration per the $4 + k(k-1)/2 + 2T$ formula in Appendix B), making efficient per-CPPN optimization essential rather than merely convenient. We approach the problem from two complementary angles. First, we apply Tree-structured Parzen Estimator (TPE) to concentrate the inter-trial search on productive regions, achieving a validated peak of 29.0% MNIST accuracy across 2,013 trials and subsequent replication (§3.1–3.4). Second, we develop a domain-specific early-stopping pruner that terminates unpromising trials at generation 3, reducing intra-trial computational cost by 41.6% while maintaining statistically equivalent search quality (§3.5–3.6). Section 3.8 unifies the two methods into a combined pipeline, and §3.10 interprets the 29.0% ceiling as evidence that the performance barrier is structural rather than parametric, motivating the architectural innovations explored in Chapters 4–6.

Sections 3.1–3.4 are based on Paper 1 [11]; Sections 3.5–3.6 are based on Paper 2 [12]. Author contributions for all papers are detailed in the List of Publications (p. 31).

3.1 The Hyperparameter Search Problem

The first step toward understanding ES-HyperNEAT's performance limits is establishing what the algorithm can achieve under optimal conditions. Before investigating architectural modifications (Chapters 4–6), we must rule out the simpler explanation that poor performance stems from poor configuration. This requires a systematic exploration of ES-HyperNEAT's hyperparameter space, a task that, as this section demonstrates, is far from trivial.

ES-HyperNEAT's hyperparameter space jointly governs CPPN evolution and substrate formation (§2.1.6). The implementation used in this work exposes roughly 40 genuinely tunable hyperparameters (about 30 from NEAT-Python's core mutation dynamics and 6 added by ES-HyperNEAT for quadtree substrate discovery), out of 67 configurable keys total (§2.2.1). For this study, sixteen parameters and their value ranges were chosen to cover both NEAT and substrate components (Table 3.1). Their Cartesian product yields more than 3×10^9 distinct configurations.

Table 3.1: ES-HyperNEAT hyperparameter search space: sixteen parameters (nine NEAT, seven substrate-side) and the value ranges below span over 3×10^9 configurations.

Hyperparameter	Range of Options
NEAT Parameters	
`num_hidden`	0, 1, 2, 3, 4, 5
`pop_size`	10, 20, 40, 60, 80, 100
`conn_add_prob`	0.3, 0.5, 0.7
`conn_delete_prob`	0.3, 0.5, 0.7
`node_add_prob`	0.2, 0.4, 0.6, 0.8
`node_delete_prob`	0.2, 0.4, 0.6, 0.8
`initial_connection`	full_nodirect, full_direct, unconnected, fs_neat_hidden, fs_neat_nohidden
`connection_fraction`	0.2, 0.4, 0.6, 0.8
`activation_default`	sigmoid, gauss, clamped, relu, tanh, softplus
Substrate Parameters	
`initial_depth`	1, 2
`max_depth`	3, 4, 5
`variance_threshold`	0.01, 0.02, 0.03, 0.04
`division_threshold`	0.3, 0.5, 0.7
`max_weight`	3.0, 5.0, 7.0
`iteration_level`	0, 1, 2, 3
`activation`	sigmoid, gauss, clamped, relu, tanh, softplus

Exhaustive evaluation of this space is infeasible. At 20 generations per configuration with populations sampled from $\{10, 20, 40, 60, 80, 100\}$ on the MNIST classification task, a single trial consumes minutes to hours depending on the population size and the substrate topology that emerges; enumerating all 3×10^9 configurations exceeds any realistic budget by orders of magnitude.

A defining characteristic of this search space is that *most configurations fail outright.* The random search baseline (292 trials) produces a median accuracy of 10.00%, exactly the random-guessing rate for a 10-class problem. The mean of 11.41% (SD = 2.76%)

confirms that only a minority of configurations exceed chance performance. This is an important observation for the thesis as a whole: the predominance of degenerate configurations means that any investigation of ES-HyperNEAT's representational capacity must first ensure that the configurations under study actually produce functioning substrates. Without the optimization methodology developed here, distinguishing genuine algorithmic limitations from configuration failures is impossible.

Two families of hyperparameters dominate sensitivity:

Variance threshold (`variance_threshold`). This parameter controls the granularity of quadtree subdivision in ES-HyperNEAT's adaptive substrate formation (§2.1.10). Small values (0.01–0.02) permit finer subdivision, producing denser substrates capable of representing higher-resolution spatial structure. Larger values prune the quadtree aggressively, yielding sparser topologies. The importance of this parameter foreshadows a central theme of the thesis: substrate *topology*, not weight tuning, is the primary determinant of ES-HyperNEAT's performance.

CPPN complexity. Maximum quadtree depth (`max_depth`) sets how many levels the CPPN is probed at, controlling substrate resolution. CPPN and substrate activation functions jointly constrain which topologies are expressible, a dimension of the configuration space that per-node function evolution could exploit (§8.4).

This combination of vast scale, sparse rewards, and concentrated sensitivity motivates a directed search strategy. Random search is provably inefficient in this regime: with these 3×10^9 configurations and over 90% producing baseline performance, a method that concentrates evaluations in the narrow productive regions is essential. The Tree-structured Parzen Estimator (§2.2.2) provides exactly this capability, as the following section demonstrates.

3.2 TPE Optimization of ES-HyperNEAT

3.2.1 Experimental Setup

The purpose of this campaign was diagnostic: to determine whether ES-HyperNEAT's poor MNIST performance reflects bad hyperparameter choices or a deeper architectural limitation. We evaluated 2,013 hyperparameter configurations on MNIST [56] using Optuna [1] with a TPE surrogate (§2.2.2), as reported in [11]. Each trial evolved a single configuration for 20 generations with a population of 100 individuals. Fitness was defined as classification accuracy on a stratified batch of 200 images (20 per digit class), where the predicted class corresponds to the output node with maximum activation: the standard ES-HyperNEAT evaluation protocol (§2.5.3). The campaign ran on 13 heterogeneous CPU machines over 78 days of wall-clock time using the PUREPLES framework [103]. Parameters outside the 16-dimensional search subspace (Table 3.1) retained their NEAT-Python defaults.

Two random search baselines provided context: RandomSearch-30 (30 trials) and RandomSearch-292 (292 trials, matching the trial count at which TPE first surpassed the previous HyperNEAT MNIST baseline of 23.9% [98]). The two did not differ significantly ($t = 0.11$, $p = 0.912$, $d = 0.02$): random search converges quickly to its expected value because most of these 3×10^9 configurations fall in dead zones where substrates never learn. This binary outcome (learn or not) is a recurring theme: it appears here in the

distribution of random search trials, again in the binary fitness dynamics (Figure 3.15), and again in the per-seed learning curves (Figure 3.2). All subsequent comparisons use RandomSearch-292.

3.2.2 Results

Table 3.2 summarizes the three experimental conditions: a 292-trial random baseline, the full 2,013-trial TPE campaign, and a focused 30-run evaluation of the top-performing configuration.

Table 3.2: MNIST classification accuracy: random search versus TPE optimization. TPE-Best is the single best configuration evaluated over 30 independent runs.

Method	Mean (%)	Median (%)	SD (%)	Best (%)	Worst (%)
RandomSearch-292	11.41	10.00	2.76	20.50	10.00
TPE-Search	17.41	19.00	4.26	27.50	10.00
TPE-Best	23.40	23.25	2.07	28.00	20.00

To contextualize these results, Table 3.3 compares our TPE-optimized accuracy against published ES-HyperNEAT and HyperNEAT results on MNIST.

Table 3.3: Comparison with published ES-HyperNEAT/HyperNEAT MNIST results. Our TPE-optimized configuration achieves the highest reported accuracy for any indirect-encoding method on MNIST, using a smaller population and far fewer generations than prior work.

Study	Method	Pop	Gens	Acc. (%)
Verbancsics and Harguess [98]	HyperNEAT (7-layer FC)	256	2,500	23.9
Verbancsics and Harguess [98]	HyperNEAT (LeNet-5)	256	2,500	27.7
Risi and Stanley [78]	ES-HyperNEAT	—	—	—
This thesis (TPE-Search)	ES-HyperNEAT	100	20	27.5
This thesis (Validated)	ES-HyperNEAT	100	20	29.0

Verbancsics and Harguess [98]: hand-designed substrates (seven-layer fully-connected and LeNet-5 CNN variants). Risi and Stanley [78]: no MNIST results reported. TPE-Search: best of 2,013 trials. Validated: post-fix peak (§3.2.5).

The gap between the TPE-optimized result and the previous best generic-substrate result of 23.9% [98] is notable: a 5.1 percentage points (pp) improvement using 2.56× fewer individuals and 125× fewer generations. Systematic hyperparameter optimization compensates for reduced computational investment, but simultaneously sharpens the structural ceiling argument, because even the best-optimized configuration remains far below the 99%+ accuracy routinely achieved by gradient-trained networks on this dataset.

The TPE-Search condition produced a mean accuracy of 17.41%, representing a 52.6% relative improvement over RandomSearch-292's 11.41%. The median rose from 10.00% (random guessing) to 19.00%, and the best single trial reached 27.5%, discovered at trial 816 after an extended plateau. When this configuration was evaluated over 30 independent runs (TPE-Best), it produced a mean of 23.40% with substantially reduced variance

(SD = 2.07% versus 4.26% for the full search). This reduced variance confirms that the optimized configuration is robust rather than a lucky outlier.

Figure 3.1 shows the optimization trajectory. The surrogate model discovered progressively better configurations during the first ∼800 trials, after which gains plateau despite continued evaluation. The weak linear trend ($R^2 = 0.010$) reflects a binary landscape: trials either discover a productive substrate topology early or stagnate at baseline, with almost no intermediate outcomes. In the thesis context, this plateau is the first indication that the performance barrier is structural. If the ceiling were parametric (that is, if better parameter values existed but had not yet been sampled), TPE's surrogate model should continue to improve, since its search coverage grows monotonically. The plateau at 27.5% despite 2,013 trials suggests that no parameter configuration can overcome the underlying architectural limitation.

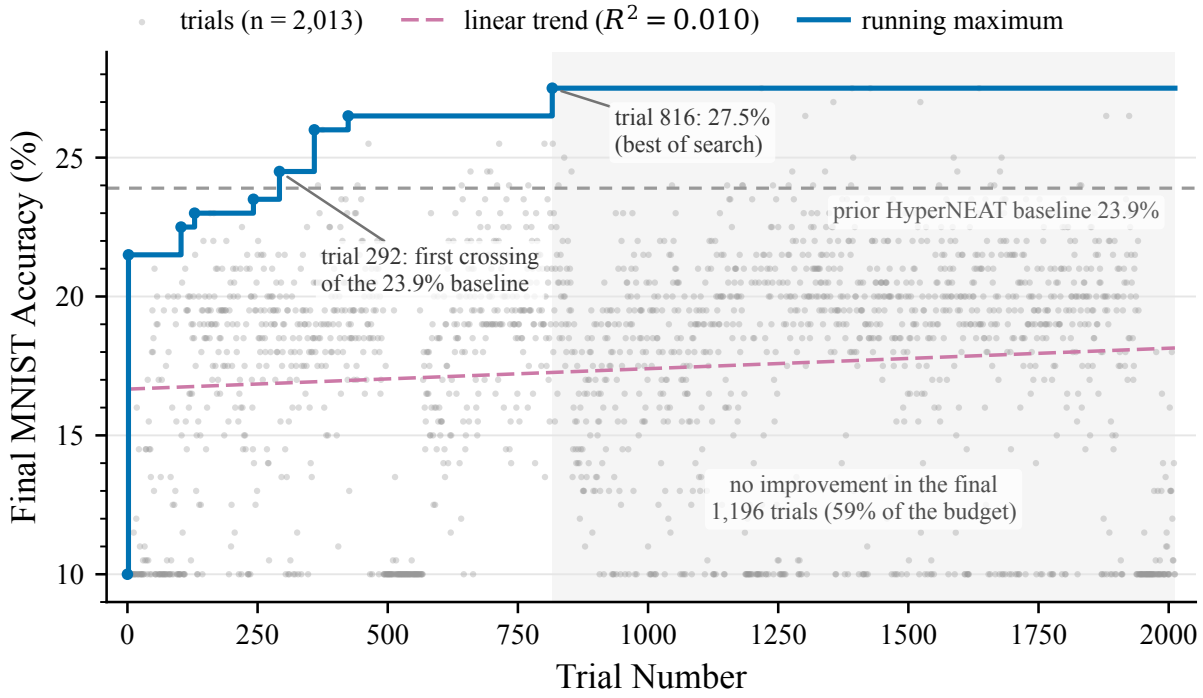


Figure 3.1: TPE optimization trajectory on MNIST (2,013 trials). Each point is one trial's best fitness in the final generation, and the step line marks new maxima. The running maximum first crosses the previous HyperNEAT baseline of 23.9% at trial 292, reaches 27.5% at trial 816, and then gains nothing across the remaining 1,196 trials (59% of the budget), despite the surrogate continuing to sample. The weak linear trend ($R^2 = 0.010$) reflects the binary landscape these trials occupy; the corresponding distribution of trial outcomes is shown in Figure 3.15.

3.2.3 Statistical Significance

Table 3.4 reports Welch's t-tests [102] comparing the three conditions. All three comparisons are highly significant ($p < .001$). The TPE-Best vs. RandomSearch-292 effect size ($d = 4.34$) is large; the optimized configuration occupies a qualitatively different performance regime. The full TPE search distribution also exceeds random search by $d = 1.47$,

a large effect, demonstrating that the surrogate concentrates evaluations in productive regions from early on, not merely as a consequence of finding a single good configuration.

Table 3.4: Welch's t-tests for MNIST accuracy comparisons. All TPE conditions significantly outperform random search with large effect sizes.

Comparison	t	p	Cohen's d
TPE-Search vs. RandomSearch-292	31.47	< 0.001	1.47
TPE-Best vs. RandomSearch-292	29.91	< 0.001	4.34
TPE-Best vs. TPE-Search	15.88	< 0.001	1.42

3.2.4 Optimal Configurations

Five configurations tied at the peak fitness of 27.5% during the TPE search. Table 3.5 lists only the parameters that vary across these configurations; the remaining six are constant: `num_hidden` = 0, `pop_size` = 100, `initial_depth` = 2, `conn_delete_prob` = 0.5, `division_threshold` = 0.5, and `iteration_level` = 0.

Table 3.5: Varying hyperparameters across the five optimal configurations (27.5% MNIST accuracy). Abbreviations: CAP = `conn_add_prob`, NAP = `node_add_prob`, NDP = `node_delete_prob`, IC = `initial_connection` (FN = full_nodirect, FD = full_direct), CF = `connection_fraction`, AD = `activation_default`, MD = `max_depth`, VT = `variance_threshold`, MW = `max_weight`, Act = substrate `activation`.

NEAT				**Substrate**					
CAP	NAP	NDP	IC	CF	AD	MD	VT	MW	Act
0.5	0.2	0.8	FN	0.8	softplus	5	0.01	3	clamped
0.5	0.2	0.2	FD	0.4	clamped	4	0.03	7	softplus
0.5	0.2	0.2	FD	0.4	tanh	3	0.03	7	clamped
0.7	0.2	0.2	FD	0.8	softplus	5	0.03	3	softplus
0.5	0.8	0.2	FD	0.8	tanh	5	0.01	3	tanh

The constant parameters converge to a consistent recipe that reveals several properties of ES-HyperNEAT's evolutionary dynamics. Zero initial hidden nodes (`num_hidden` = 0) confirms that the algorithm benefits from starting with a minimal CPPN and allowing topology to emerge through augmenting mutations, consistent with the original NEAT design philosophy of complexification from minimal structure [88]. The maximum available population size (100) is selected, though this is smaller than the 256 used by Verbancsics and Harguess [98], suggesting that systematic hyperparameter optimization compensates for reduced population diversity. Iteration level converging to 0 indicates that a single quadtree pass produces substrates of sufficient resolution for MNIST, possibly because the 28×28 pixel grid already constrains the useful spatial resolution.

Among the varying parameters, variance threshold clusters tightly at 0.01–0.03, favoring fine-grained substrate topology. Maximum depth spans 3–5, reflecting a trade-off between spatial resolution and computational cost. Activation function choices vary freely (softplus, clamped, and tanh all appear), suggesting that once substrate topology is ap-

propriately resolved, the specific nonlinearity is secondary.[1] Whether function diversity matters more than individual function choice is a question that per-node function evolution, enabled by the EMR architecture (Chapter 6) but beyond the scope of this thesis, could address.

3.2.5 Validation and the 29% Result

A parallel execution bug discovered mid-campaign, which intermittently swapped hyperparameter configurations between concurrently running trials, prompted a re-evaluation of 885 configurations under corrected code. Of the 199 configurations fully validated within the available time, 72 (36.2%) exceeded the corrected random search ceiling of 22.0%, confirming that TPE's advantage is genuine rather than an artifact of the implementation error.

The eight best configurations from validation were further tested over 30 trials each. The single best validated configuration (TPE-Val-Best) achieved a mean accuracy of 20.95% (SD = 2.41%) with a peak of 29.0%. This surpassed the previous HyperNEAT-family records reported by Verbancsics and Harguess [98]: 23.9% with a hand-designed seven-layer fully-connected substrate and 27.7% with a hand-designed LeNet-5 CNN substrate [57], both with a population of 256 and 2,500 generations. TPE-Val-Best also significantly outperformed the corrected random search baseline ($t = 20.93$, $p < .001$, $d = 3.72$).

The distinction between TPE-Best and TPE-Val-Best warrants clarification, as the two labels refer to the same configuration evaluated at different stages of the research pipeline. Table 3.6 summarizes all accuracy metrics reported for this configuration.

Table 3.6: Disambiguation of accuracy metrics for the best configuration (tanh/tanh). All evaluations use the same hyperparameter configuration; differences reflect evaluation context and stochastic variance.

Label	Value	Context	Usage in Thesis
TPE-Search peak	27.5%	Best single trial (#816)	Ceiling reference
TPE-Best mean	23.40%	30-run mean, original code	Configuration robustness
TPE-Best peak	28.0%	Best of 30 runs, original code	Upper bound (pre-fix)
TPE-Val-Best mean	20.95%	30-run mean, post-bug-fix	Conservative estimate
TPE-Val-Best peak	29.0%	Best of 30 post-fix runs	Absolute ceiling

Two conclusions follow from the validation. First, TPE genuinely concentrates evaluations in productive regions of the 16-parameter search subspace, not merely through luck. Second, the 29.0% peak demonstrates that ES-HyperNEAT can exceed previously published baselines when hyperparameters are optimized systematically, despite using a

[1] This observation holds for the MNIST task studied here, where substrate topology dominates performance. It does not generalize to all domains: companion work on per-node activation function evolution (§8.4) demonstrates that function type is foundational for parity tasks of order four and above under evolutionary search in sparse feedforward substrates. At parity-4 every uniform monotonic function tested (tanh, sigmoid, ReLU) produces 0% solve rate (0.95 solve criterion), uniform tanh stays at 0% through parity-8, and oscillatory functions (sin) achieve 100% throughout. The distinction is between tasks where topology dominates (MNIST) and tasks where function class is the binding constraint (higher-order parity).

smaller population (100 vs. 256) and fewer generations (20 vs. 2,500). However, 29% accuracy on MNIST, a dataset routinely solved above 99% by gradient-trained networks, also establishes a decisive performance ceiling. Whether this ceiling reflects a fundamental representational limit of ES-HyperNEAT's substrate formation or merely a search limitation is the question that drives the remainder of this chapter and motivates the architectural investigations of Chapters 4–6.

3.2.6 Extended Generation Analysis

The TPE campaign used a 20-generation budget per trial, and the pruner study used 17. A reasonable objection is that the 27.5% ceiling might simply reflect premature termination: the evolutionary process needs more time to escape local optima. We tested this directly by evaluating the top-scoring search configuration (softplus/clamped, Table 3.5, row 1) for 100 generations across 30 independent runs, a 5× extension of the standard budget. The results rule out premature termination as the cause and provide the empirical foundation for the structural ceiling argument developed in §3.10.

Figure 3.2 plots the individual seed trajectories alongside the mean accuracy with 95% confidence bands. The fan of individual trajectories reveals that seeds vary in their final accuracy (from ∼15% to ∼32%) but share a common temporal pattern: all plateau within the first 10–30 generations. Table 3.7 quantifies the marginal improvement by generation window, and Table 3.8 reports the per-seed convergence distribution.

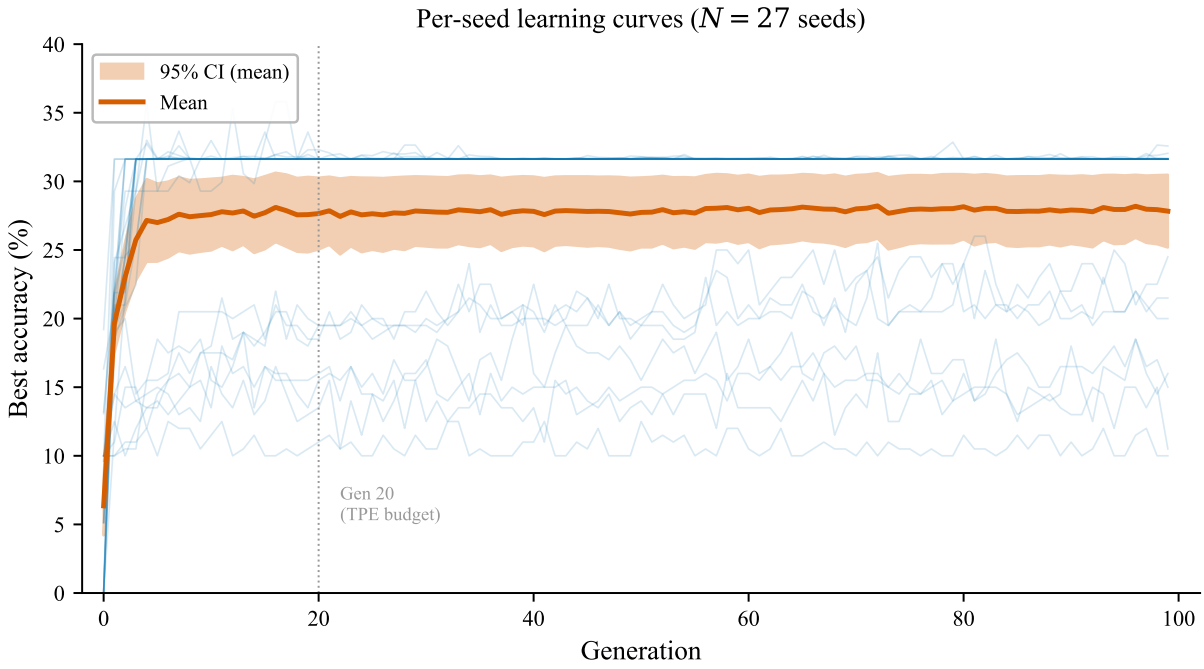


Figure 3.2: Per-seed learning curves for the softplus/clamped configuration (Table 3.5, row 1) over 100 generations. Thin blue lines show individual seeds ($N = 30$); red line and shaded band show the mean ± 95% confidence interval (CI). The vertical dashed line marks the 20-generation TPE budget. All seeds plateau early; the spread in final accuracy reflects per-seed substrate quality, not ongoing learning.

Even with 5× the evolutionary budget, accuracy improves by only 2.8pp beyond generation 20: from 20.8% to 23.6%. The plateau is not a consequence of premature ter-

Table 3.7: Diminishing returns in the 100-generation extended evaluation ($N = 30$). Each row shows the marginal accuracy gain within a 20-generation window. The first window captures 79% of the total improvement; the last three windows combined add 1.5pp.

Generation Window	Mean Accuracy (%)	Marginal Gain (pp)	Cumulative Gain (pp)
Gen 1	10.5	—	—
Gen 1–20	20.8	10.3	10.3
Gen 20–40	22.1	1.3	11.6
Gen 40–60	22.7	0.6	12.2
Gen 60–80	23.1	0.4	12.6
Gen 80–100	23.6	0.5	13.1

mination at 17–20 generations: it is a fundamental ceiling of the substrate architecture. The marginal gains follow a power-law decay: 10.3pp in the first window, then 1.3, 0.6, 0.4, and 0.5pp in subsequent windows. Generations 60–100 add less than 1pp combined: computational effort that yields no meaningful return.

Per-seed convergence velocity. Table 3.8 reports the generation at which each seed reaches 95% of its final accuracy. The median plateau generation is 24, with an interquartile range (IQR) of [16, 30]. Only 40% of seeds have converged by generation 20, and 7% by generation 10: slower than the marginal-gain analysis of Table 3.7 might suggest. The apparent discrepancy arises because accuracy is evaluated on 200-image validation batches, introducing noise that delays the $\geq$95% threshold even when the underlying substrate has stabilized. The latest-converging seed (gen 65) reaches only 28%, still well within the performance ceiling. This result directly addresses the "insufficient search" objection.

The 20-generation budget captures 79% of all achievable improvement; extending evolution 5-fold adds only 2.8 pp. The ceiling is architectural, not temporal.

Table 3.8: Convergence velocity: generation at which each seed reaches 95% of its final accuracy ($N = 30$, 200-image validation batches).

Statistic	Value
Median plateau generation	24
Mean $\pm$ SD	26.5 ± 13.3
IQR	[16, 30]
Range	[8, 65]
% converged by gen 10	6.7
% converged by gen 20	40.0

Per-digit output analysis. To understand *how* the substrate fails, we examined the end-of-run predictions from 21 runs that recorded per-digit output vectors. Figure 3.3 visualizes the mean cosine distance between expected and actual output vectors for each digit class (0–9), alongside the fraction of runs in which each digit was correctly classified (argmax of output). The bar chart makes the class-0 dominance immediately apparent: only digit 0 itself (61.9%) and digit 1 (33.3%) exceed the 10% chance level; the remaining eight digits are at or below 4.8%. The cosine distances (orange diamonds) are nearly

uniform across all digits (0.69–0.76, where ∼0.68 corresponds to a uniform output vector). The substrate therefore produces functionally identical responses regardless of input class.

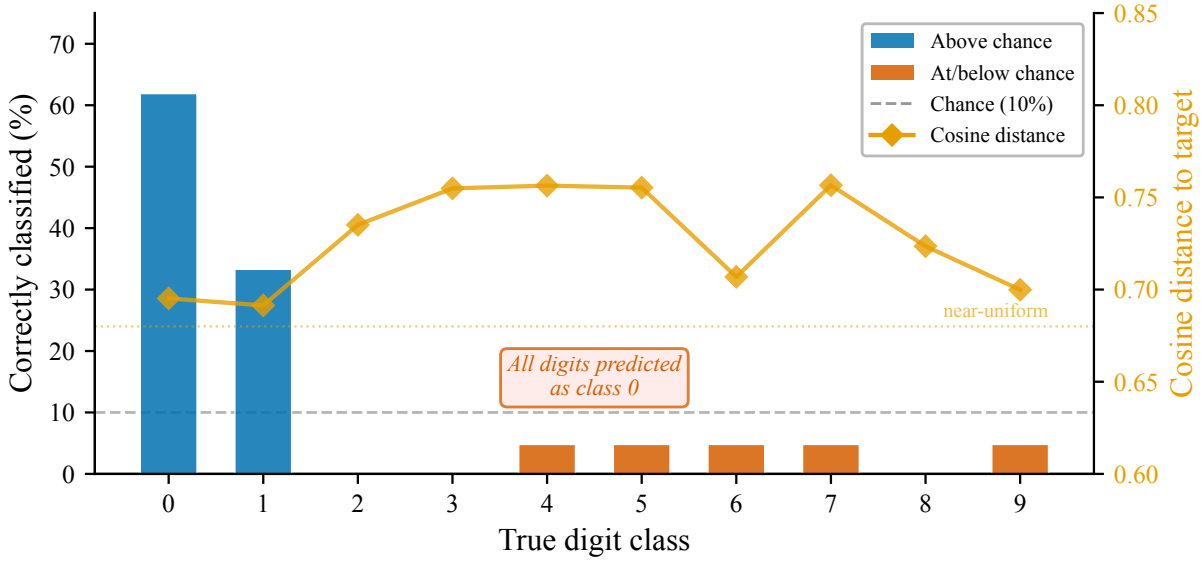


Figure 3.3: Per-digit classification accuracy and cosine distance from the 100-generation extended evaluation ($N = 21$ runs). Blue bars: digits exceeding chance; red bars: digits at or below chance. Orange diamonds: cosine distance between output vector and one-hot target (right axis; values near 0.68 indicate near-uniform outputs). All digits are predominantly predicted as class 0, confirming that the substrate never learns to differentiate output activations.

The failure is *uniform*: all 10 digits produce nearly identical cosine distances (0.69–0.76), and all digits are predominantly classified as class 0. The substrate's output vectors are near-uniform across the 10 output nodes; the evolved CPPN-substrate never learns to activate output units differentially. This is consistent with the central bias hypothesis of §3.10: a substrate that attends to only a small fraction of input pixels cannot distinguish digits that differ primarily in their peripheral structure. The failure is global (resolution) rather than class-specific (feature). This confirms that the ceiling is a structural limitation of the substrate formation mechanism, not a consequence of particular digit shapes being intrinsically harder.

Network topology of converged versus failed runs. The 100-generation logs also reveal a topology signature that distinguishes successful from failed substrates. Runs that converged above 30% accuracy evolved CPPNs with a median of 3 nodes and 6 connections, producing substrates with sufficient spatial resolution to discriminate at least some digit classes. Runs that stagnated below 20% evolved smaller CPPNs (median 2 nodes, 4 connections) whose substrates lacked the spatial complexity to activate output units differentially. Among the activation functions discovered by NEAT in successful runs, `sin` and `gauss` appeared alongside `tanh`, suggesting that oscillatory and radially symmetric nonlinearities provide representational capabilities beyond those of monotonic functions. This observation is consistent with the companion evidence on per-node function evolution, where function class can be the binding constraint on higher-order parity tasks; the role of activation-function selection in substrate performance is a direction for future work (§8.4).

3.2.7 Computational Cost of Extended Evolution

The diminishing accuracy returns of Table 3.7 are compounded by increasing computational cost. To quantify this, we parsed the per-generation wall-clock times from 27 complete 100-generation runs across 14 heterogeneous machines.[2] Each machine's times were normalized to a common reference using the generation-0 baseline (the simplest generation with a minimal substrate), preserving the *scaling pattern* while removing hardware variance.

Figure 3.4 plots the normalized per-generation time on a logarithmic scale. Each generation grows progressively more expensive as NEAT adds nodes and connections to the CPPN, which in turn produces denser substrates requiring more quadtree subdivision and more individual-level evaluations. From generation 0 to generation 89, the median per-generation cost increases approximately 383-fold. The figure annotates two operationally important checkpoints: generation 3, where the early-stopping pruner of §3.5 terminates unpromising trials before costs escalate; and generation 20, the TPE evaluation budget from §3.2.

Genetic distance trajectory. Figure 3.5 plots the mean genetic distance (pairwise genome dissimilarity) across the population over 100 generations. Unlike fitness, which plateaus by generation 20, genetic distance shows a net upward trend, from a median of 1.39 at generation 0 to 2.43 at generation 99. This reveals a sharp decoupling: the population grows more structurally diverse while making no functional progress. NEAT's complexification pressure continues to add nodes and connections. The resulting genomes differ from each other in topology but produce substrates of identical quality. The algorithm explores a widening region of genotype space without discovering phenotypes that improve task performance—structural divergence without functional gain.

Table 3.9 pairs the runtime data with the accuracy gains from the $N = 30$ validation runs. The result is a double penalty: later generations cost more *and* deliver less. Generations 51–100 consume 74% of the total wall-clock budget while contributing only 1.4 pp of accuracy improvement. The final column quantifies this asymmetry as a cost-efficiency ratio: the percentage of total computation consumed per percentage point of accuracy gained. The cost-efficiency collapse is pronounced. The first window (gen 0–3) costs less than 0.1% of total compute per percentage point gained; the last window (gen 51–100) costs 53% per percentage point, a ∼1,300-fold efficiency collapse. This is not a gradual decline but an exponential divergence between computational investment and scientific return.

Figure 3.6 visualizes this asymmetry on a dual-axis plot. Generation 20 captures 88% of the final accuracy while consuming only 4% of the computational budget. Extending evolution from 20 to 100 generations adds only 2.8 pp of accuracy while consuming 23 times the wall-clock time of the first 20 generations.

This cost explosion is not a quirk of hardware or implementation. It is an intrinsic property of ES-HyperNEAT's substrate formation: as CPPNs complexify through NEAT's mutation operators, the quadtree must subdivide more cells, producing more nodes, requiring more forward-pass evaluations per individual per generation. Chapter 5 formalizes this as a structural incompatibility with GPU parallelization; the runtime data presented here provides the empirical foundation for that analysis, and Appendix C.7

[2]Thirty seeds were executed (matching the $N = 30$ accuracy analysis), but timing logs survived for only 27; three additional partial logs (3–18 generations) were excluded.

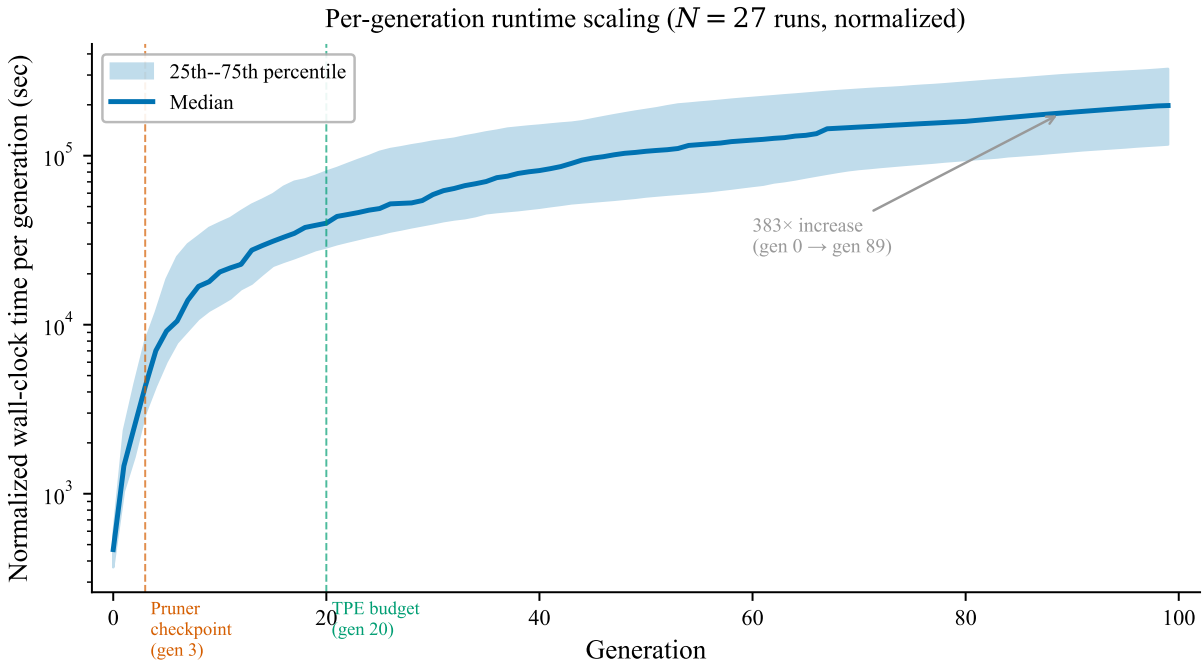


Figure 3.4: Per-generation wall-clock time (normalized across 14 machines, $N = 27$ runs) on a logarithmic scale. Shaded region: 25th–75th percentile. Vertical dashed lines mark the pruner checkpoint (gen 3) and TPE budget (gen 20). Cost increases approximately 383-fold from generation 0 to generation 89.

documents the underlying CPPN complexification (nodes $1 \to 3$, connections $5 \to 6$ over 100 generations) and species-stabilization trajectory that drive the cost growth.

Table 3.9: Runtime and cost efficiency by generation window. Timing data from $N = 27$ log runs (normalized across 14 machines); accuracy gains from $N = 30$ validation runs (200-image batches). The cost/gain ratio expresses the percentage of total compute consumed per percentage point of accuracy gained.

Window	Time/Gen (s)	Cumul. (s)	% Total	Gain (pp)	Cost/Gain
Gen 0–3	1987	8746	0.1	2.4	<0.1
Gen 4–20	22 761	412 550	4.1	7.8	0.5
Gen 21–50	72 229	2 186 943	21.6	1.5	14.4
Gen 51–100	153 629	7 494 613	74.2	1.4	53.0
Gen 0–17 (pruner)	16 948	305 067	3.0	9.7	0.3
Gen 0–20 (TPE)	20 062	421 295	4.2	10.3	0.4

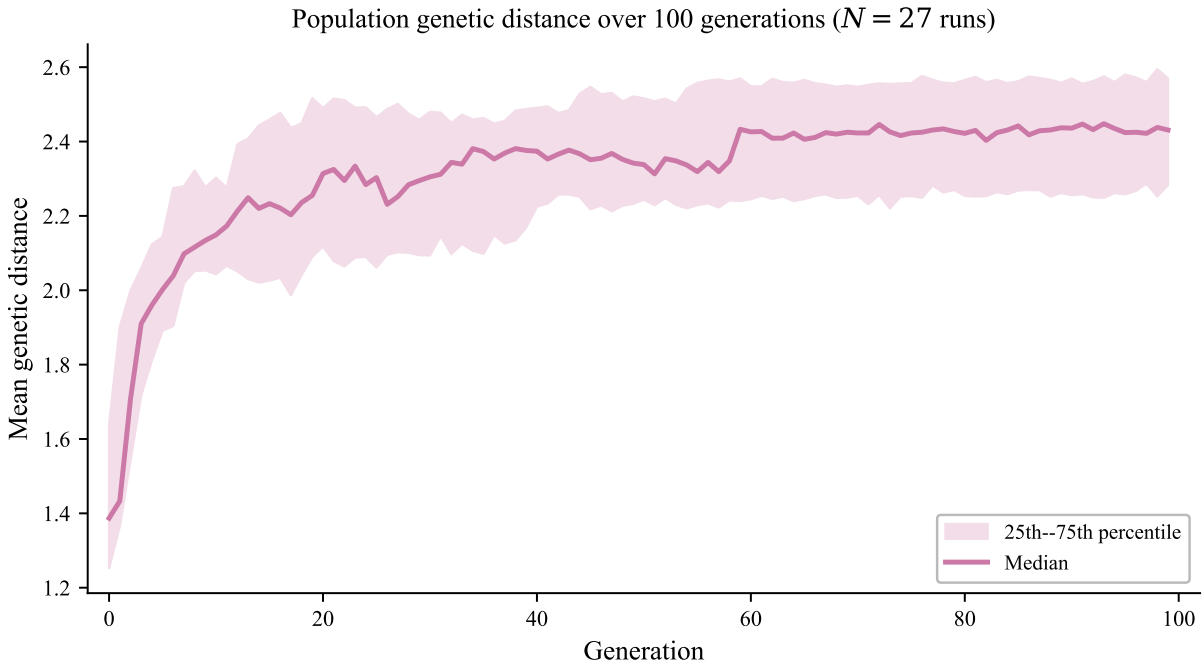


Figure 3.5: Population genetic distance over 100 generations ($N = 27$ runs). Shaded region: 25th–75th percentile. Genetic distance shows a net upward trend, indicating structural divergence even as fitness plateaus. The population explores ever more genotype space without discovering improved substrates.

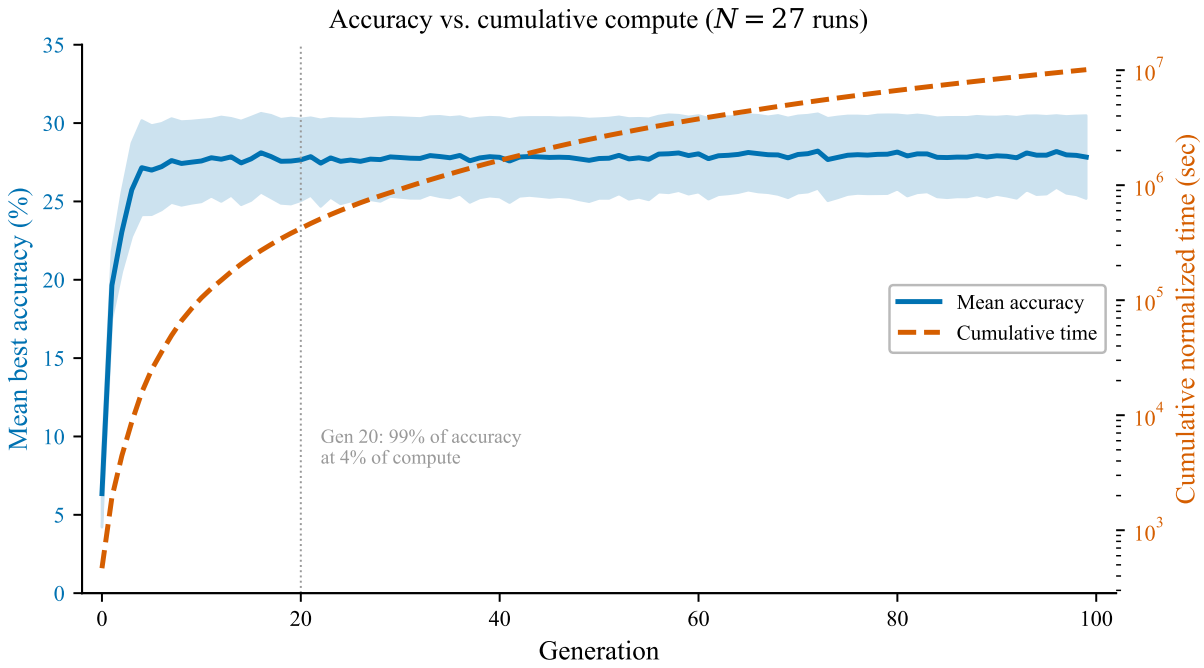


Figure 3.6: Accuracy vs. cumulative compute over 100 generations. Left axis: mean best accuracy ($N = 30$, 200-image validation batches, blue, with 95% CI band). Right axis: cumulative normalized wall-clock time ($N = 27$ runs across 14 machines, red, log scale). Generation 20 captures 88% of final accuracy at 4% of total cost.

3.2.8 Multi-Configuration Convergence

The extended generation analysis used a single configuration (softplus/clamped). A stronger test of the structural ceiling hypothesis examines whether *multiple* configurations, spanning different activation functions and connection types, converge to the same accuracy band.

The validation study evaluated eight structurally distinct configurations over 30 trials each, with per-generation fitness recorded. Figure 3.7 overlays the median fitness trajectories for seven of these configurations alongside the random search baseline over 20 generations.

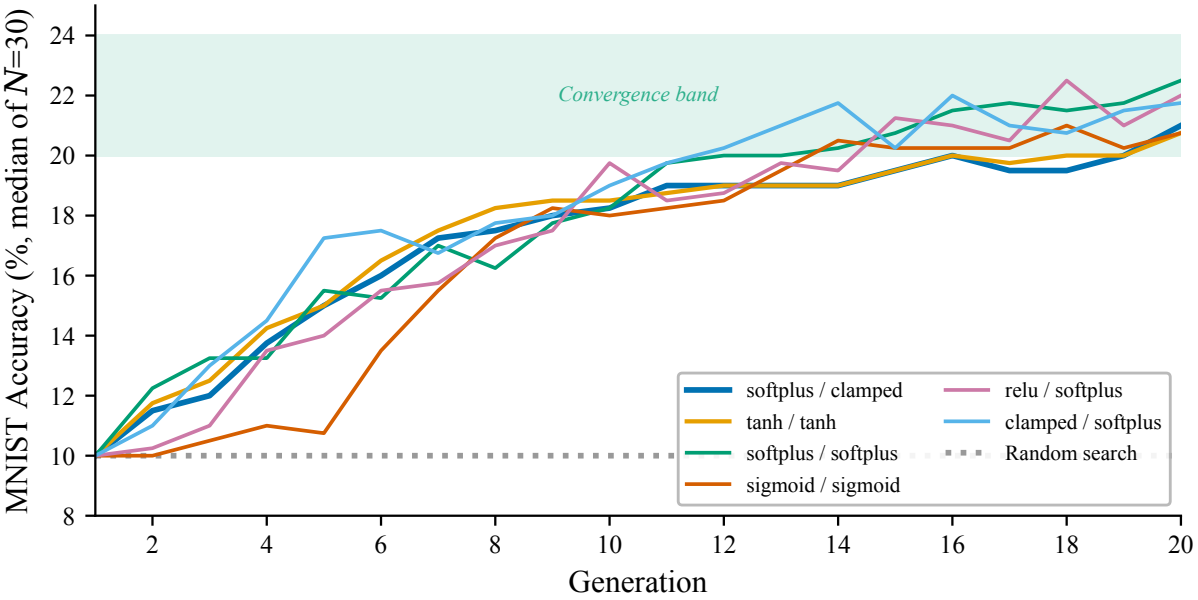


Figure 3.7: Median fitness trajectories for seven validated configurations and the random search baseline over 20 generations. Despite spanning five activation functions, two initial connection types, and multiple max depth values, all seven configurations converge to the 20–24% accuracy band. The random baseline remains at 10%.

Seven structurally distinct configurations (spanning softplus, tanh, sigmoid, clamped, and relu activation functions, both `full_direct` and `full_nodirect` initial connections, and max depths of 3–5) all converge to the same 20–24% accuracy band by generation 20. The random control remains at approximately 10%, confirming that the convergence reflects genuine learning rather than a measurement artifact.

This multi-configuration convergence provides the strongest evidence that the ceiling is architectural. Different paths through the hyperparameter space (representing different CPPN geometries, substrate densities, and evolutionary dynamics) lead to the same representational limit. No single hyperparameter choice is responsible for the ceiling; it is a property of ES-HyperNEAT's substrate formation mechanism itself.

3.3 Hyperparameter Sensitivity Analysis

The 2,013 TPE trials combined with 292 random search trials generate a dataset of 2,305 evaluated configurations. This dataset enables a systematic sensitivity analysis that goes beyond the original experiment's scope: here, the goal is not merely to identify which parameters matter, but to determine whether the performance ceiling is attributable to specific parameter choices or reflects a structural property of the algorithm.

3.3.1 Correlation Structure

Figure 3.8 shows the pairwise correlation heatmap between all sixteen hyperparameters and the resulting fitness.

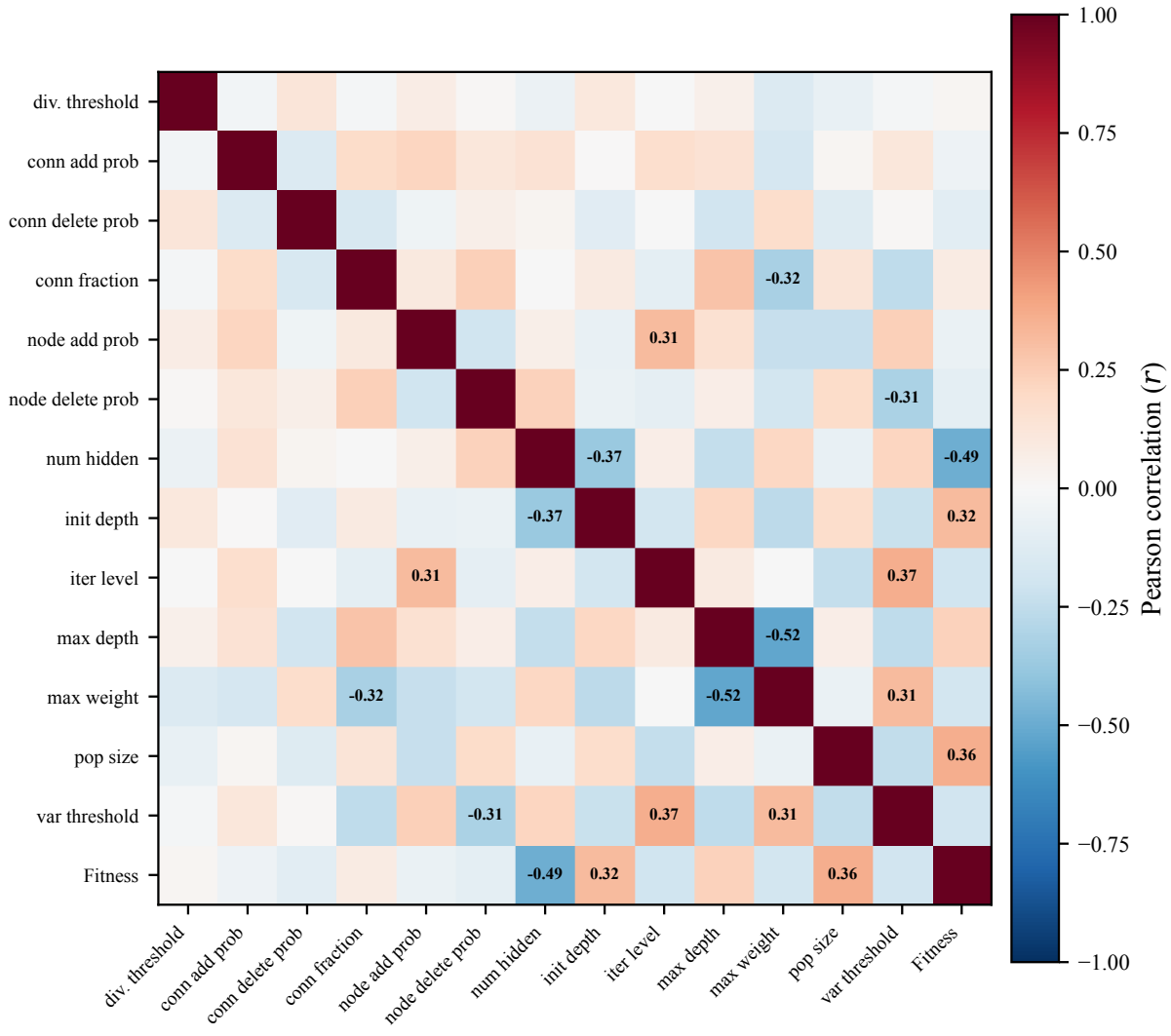


Figure 3.8: Pairwise correlation heatmap across 2,305 evaluated configurations (2,013 TPE + 292 random search). Only correlations with $|r| \geq 0.30$ are annotated. The three parameters correlated with fitness (`num_hidden` with $r = -0.49$, `pop_size` with $r = 0.36$, and `initial_depth` with $r = 0.32$) are all boundary-converged constants in the optimal configurations (Table 3.10); `var_threshold`, the parameter that varies most across successful runs (§3.3), falls below the annotation threshold.

Three parameters show $|r| \geq 0.30$ with fitness: `num_hidden` ($r = -0.49$), `pop_size` ($r = 0.36$), and `initial_depth` ($r = 0.32$). All three converge to boundary values across the five optimal configurations (§3.2.4): `num_hidden` = 0 (lower boundary), `pop_size` = 100 (upper boundary), and `initial_depth` = 2 (upper boundary). Their correlation with fitness reflects this boundary convergence rather than a continuous response to parameter variation. By contrast, the variance threshold (the parameter with the widest spread across successful runs) falls below the $|r| \geq 0.30$ annotation threshold because its influence is not monotonic: successful runs cluster at 0.01–0.03 (§3.3.3) rather than varying continuously with fitness. The sensitivity analysis that follows therefore separates two roles: parameters that must stabilize for successful evolution (the convergent recipe, §3.3.2) and parameters whose non-boundary values actively drive performance differences (§3.3.3).

3.3.2 Constant Parameters: The Convergent Recipe

Six of the sixteen parameters take identical values across all five optimal configurations (Table 3.5). Table 3.10 lists these parameters and their converged values.

Table 3.10: Hyperparameters constant across all optimal configurations. These six parameters converge to a single value, indicating low sensitivity at the optimum.

Parameter	Search Range	Value	Interpretation
`num_hidden`	0–5	0	Rely on NEAT complexification
`pop_size`	10–100	100	Maximum available
`initial_depth`	1–2	2	Standard quadtree initialization
`conn_delete_prob`	0.3–0.7	0.5	Moderate connection pruning
`division_threshold`	0.3–0.7	0.5	Moderate band-pass on quadtree splitting
`iteration_level`	0–3	0	Single-pass quadtree sufficient

The convergence pattern has a practical consequence exploited throughout this thesis: these six values can be fixed for all subsequent ES-HyperNEAT experiments, reducing the effective search space from 16 to 10 dimensions. Chapter 4 adopts this converged recipe as its default configuration, ensuring that performance differences there reflect algorithmic modifications rather than configuration artifacts; Chapters 5 and 6 sweep population and depth as independent variables.

3.3.3 Sensitive Parameters: Variance Threshold and CPPN Complexity

The remaining ten parameters vary across optimal configurations. Among these, variance threshold and maximum depth exhibit the strongest influence on fitness.

Variance threshold. This parameter controls the resolution of quadtree subdivision that determines substrate topology (Section 2.1.10). When CPPN output variance within a quadtree cell exceeds the threshold, the cell is subdivided, adding nodes to the substrate. Lower thresholds permit finer subdivision, producing denser substrates with more hidden nodes.

Across optimal configurations, variance threshold clusters at 0.01–0.03 (Table 3.5), the lower three of the four sampled values in the 0.01–0.04 search range. The mechanism

is straightforward. A variance threshold of 0.04 applies aggressive pruning, eliminating regions where the CPPN output is relatively smooth. For MNIST, where digit structure requires representing spatial features at multiple scales, such aggressive pruning discards nodes critical for discriminating digit classes. Conversely, thresholds below 0.01 would produce excessively dense substrates (not explored in this search space), likely causing overfitting to the 200-image batch and increasing computational cost without proportional accuracy gains.

From a thesis perspective, the dominance of variance threshold is revealing. It means the single most impactful configuration choice governs *how the substrate's spatial structure is resolved*, not how weights are learned or how evolution proceeds. This is the first indication that ES-HyperNEAT's performance is topology-limited, a conclusion reinforced by the structural analysis of Chapter 4.

Maximum depth. The maximum quadtree depth determines how many subdivision levels are permitted, controlling the finest spatial resolution at which the CPPN output is sampled. Optimal configurations span the full 3–5 range, reflecting a trade-off: deeper quadtrees enable more detailed substrate topologies but increase the number of nodes and connections, raising both computational cost and the risk of overfitting.

The interaction between variance threshold and maximum depth is important. A low variance threshold combined with high maximum depth produces the densest possible substrates, but two of the five optimal configurations achieve 27.5% with `max_depth` = 3 or 4 and `variance_threshold` = 0.03, showing that moderate settings suffice when the CPPN generates sufficiently structured output.

Activation functions. The CPPN and substrate activation functions vary freely across optimal configurations: softplus, clamped, and tanh all appear. This insensitivity at the optimum suggests that, once substrate topology is appropriately resolved, the specific nonlinearity used in computation is secondary. None of the optimal configurations use identity or purely linear activations, so the functions must be non-trivial, but within the set of reasonable nonlinearities, the choice does not materially affect MNIST accuracy at this scale. Whether this insensitivity persists at harder task scales remains an open question; the interaction between activation choice and task difficulty is a direction for future work (§8.4).

Connection parameters. Connection addition probability clusters at 0.5–0.7, while node deletion probability varies from 0.2 to 0.8. The initial connection type switches between `full_nodirect` and `full_direct`; the specific initial wiring matters less than the overall evolutionary dynamics that reshape it. Connection fraction and maximum weight both vary, and both interact with CPPN output characteristics rather than independently determining fitness.

3.3.4 Summary of Sensitivity Hierarchy

The sensitivity analysis reveals a clear hierarchy among the sixteen hyperparameters:

1. **Critical** (determines whether the substrate can solve the task): variance threshold, max depth.

2. **Important** (affects convergence speed and solution quality): population size, initial depth, node add probability, division threshold.

3. **Secondary** (varies across optimal configurations without dominant effect): activation functions, connection probabilities, initial connection type, connection fraction, maximum weight, iteration level.

This hierarchy has two practical implications. First, practitioners should prioritize variance threshold and maximum depth, fix the "important" parameters at the converged values (Table 3.10), and treat the rest as free variables. Second, and more significant for this thesis, the hierarchy confirms that the critical parameters control substrate *topology*, not learning dynamics. If topology is the primary determinant of performance, then improving ES-HyperNEAT requires changing how topologies are generated, not how evolution tunes them.

3.4 Transferability of Optimized Configurations

Hyperparameter optimization is expensive. If configurations optimized on one task generalize to others, the amortized cost per task decreases substantially, making the 78-day investment in MNIST optimization worthwhile beyond a single benchmark. This section examines whether the best MNIST configuration (MNIST-Config) transfers to two structurally distinct task families: six Boolean logic operations and Fashion-MNIST classification [104]. For each target task, 30 random search trials provide the baseline and 30 trials with MNIST-Config provide the transfer condition.

The transfer experiments also serve as a transferability test for the thesis. If MNIST-Config transfers well to structurally similar tasks (Fashion-MNIST) but poorly to structurally dissimilar ones (logic operations), this confirms that the optimized parameters encode *substrate-level assumptions about spatial resolution* rather than universal learning strategies—further evidence that the performance ceiling is structural.

3.4.1 Logic Operations

The six Boolean operations (XOR, OR, AND, NOR, XNOR, NAND) serve as standard ES-HyperNEAT benchmarks [78], each with two binary inputs and one binary output. Fitness measures accuracy over all four input combinations via the residual sum of squares ($1 - \text{RSS}$, §2.5.1). Trials ran for 10 generations, sufficient for these simple tasks, as established in the original paper's pilot study.

Table 3.11 reports performance for both conditions. MNIST-Config achieved perfect accuracy on OR (100.00%) and substantially raised the mean for NAND (66.00% → 87.68%) and XOR (75.19% → 83.33%). However, it never reached 100% on XOR, XNOR, or AND, where some random search trials did. This reveals a characteristic pattern: transfer raises the *floor* (worst-case performance) and sharply reduces variance (XOR SD drops from 19.28% to 1.38%), but it can also constrain the *ceiling*.

Table 3.12 quantifies statistical significance via Welch's t-tests. Transfer was statistically significant for three operations: OR ($d = 1.07$, large), NAND ($d = 0.94$, large), and XOR ($d = 0.60$, medium). For AND ($p = 0.663$), NOR ($p = 0.785$), and XNOR ($p = 0.483$), the differences were non-significant with negligible effect sizes.

The asymmetry is informative. AND and NOR are relatively easy for ES-HyperNEAT: random search already achieves mean accuracies of 86–89%, leaving little headroom for

Table 3.11: Logic operations accuracy: RandomSearch (30 trials) versus MNIST-Config (30 trials). MNIST-Config raises the mean for 5 of 6 operations but does not always reach the 100% ceiling attained by some random search trials.

Operation	Method	Mean (%)	Median (%)	SD (%)	Best (%)	Worst (%)
XOR	RandomSearch	75.19	75.27	19.28	100.00	50.00
	MNIST-Config	83.33	83.25	1.38	87.47	79.13
OR	RandomSearch	76.80	98.73	30.70	100.00	25.00
	MNIST-Config	100.00	100.00	0.00	100.00	100.00
AND	RandomSearch	88.77	87.50	10.71	100.00	75.00
	MNIST-Config	87.91	87.50	1.11	91.15	87.41
NOR	RandomSearch	85.91	77.22	12.03	100.00	75.00
	MNIST-Config	86.65	81.17	8.56	100.00	76.01
XNOR	RandomSearch	73.78	73.98	20.35	100.00	50.00
	MNIST-Config	76.43	75.00	3.07	87.53	73.96
NAND	RandomSearch	66.00	80.18	31.77	100.00	25.00
	MNIST-Config	87.68	86.49	6.78	100.00	81.14

Table 3.12: Welch's t-tests comparing MNIST-Config versus RandomSearch for each logic operation. Significant results ($p < .05$) in bold.

Operation	t	p	Cohen's d
XOR	2.31	0.025	0.60
OR	4.14	$<$ **0.001**	1.07
AND	−0.43	0.663	−0.11
NOR	0.27	0.785	0.07
XNOR	0.71	0.483	0.18
NAND	3.66	0.001	0.94

improvement. XOR, OR, and NAND present more difficulty (means of 66–77% with high variance), creating space for the transferred configuration to exploit. XNOR sits between these regimes: its random-search mean is comparably low (73.78%), but its high variance leaves the transfer improvement short of significance. Transfer helps most where the baseline struggles most—a pattern that reappears in Chapter 4's investigation of spatial partitioning, where the benefits of architectural modification are likewise greatest for the hardest tasks.

3.4.2 Fashion-MNIST

Fashion-MNIST [104] shares MNIST's image dimensions (28×28 grayscale, 10 classes) but presents a harder classification problem with more complex visual patterns (Figure 3.9). The substrate initialization, fitness evaluation, batch size (200 images), and generation count (20) were identical to the MNIST setup. Table 3.13 reports the results.

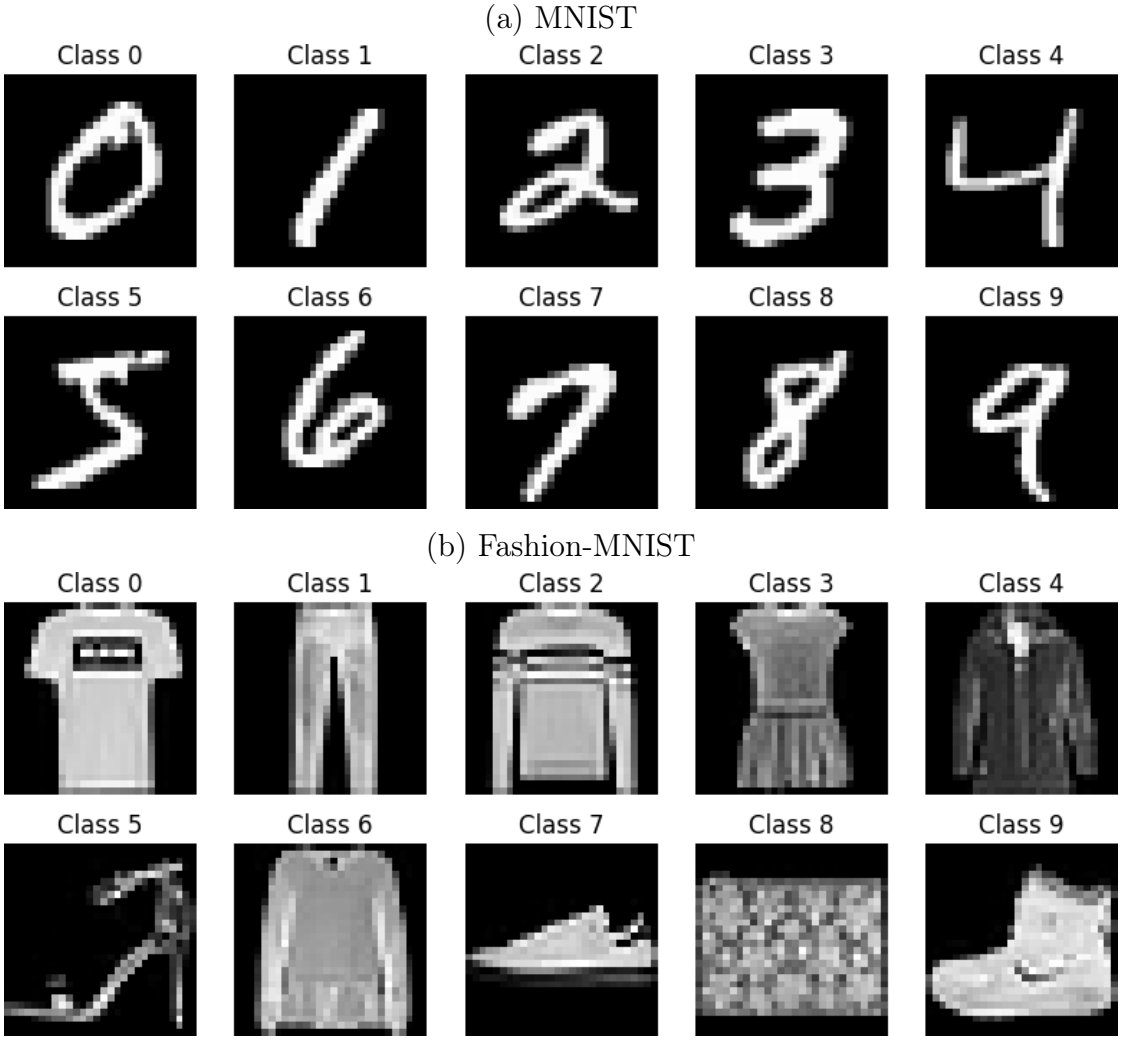


Figure 3.9: Sample images from (a) MNIST and (b) Fashion-MNIST, one per class. Both datasets share identical structure: 28×28 grayscale images across 10 classes. Fashion-MNIST presents more complex visual patterns (overlapping textures, fine-grained category distinctions) while preserving the spatial layout that enables the hyperparameter transfer studied in this subsection.

The transferred configuration raised mean accuracy to 20.00%, a 72.4% relative gain over RandomSearch-292's 11.60%. A Welch's t-test confirmed the difference ($t = 32.75$, $p < .001$, $d = 2.85$), with the effect size of 2.85 indicating that the transferred and random-search distributions occupy almost entirely non-overlapping performance regions.

The transfer exhibited the same floor-raising, ceiling-constraining pattern observed in logic operations. RandomSearch-292's best trial reached 23.50%, while MNIST-Config's best was 22.50%, marginally lower. But RandomSearch-292's worst was 10.00% (random guessing), while MNIST-Config's worst was 18.50%, nearly double. The transferred configuration provided consistent, reliable performance (SD = 1.00%) at the cost of slightly reduced peak accuracy.

Table 3.13: Fashion-MNIST classification accuracy: RandomSearch-292 (292 trials) versus MNIST-Config (30 trials). Transfer raises the mean from near-chance to 20.0% but lowers the peak.

Method	Mean (%)	Median (%)	SD (%)	Best (%)	Worst (%)
RandomSearch-292	11.60	10.00	3.08	23.50	10.00
MNIST-Config	20.00	20.00	1.00	22.50	18.50

The most consequential transfer effect is not the accuracy improvement but the elimination of catastrophic failure. Under random search, 68.4% of Fashion-MNIST trials stagnate at the 10% chance level, producing substrates that learn nothing at all. With the transferred MNIST configuration, *zero* trials fail: every run produces a functioning substrate with accuracy between 18.5% and 22.5%. The transferred hyperparameters do not merely improve mean performance; they shift the entire operating regime from "usually fails" to "always works." This $68\% \rightarrow 0\%$ failure rate reduction is a more practically significant result than the 8.4pp mean improvement, because it eliminates the need for manual inspection of trial quality: a prerequisite for automated optimization pipelines.

3.4.3 Interpretation: When and Why Transfer Works

The transfer results reveal the nature of the optimized hyperparameters and, by extension, the nature of the performance ceiling.

MNIST → Fashion-MNIST (strong transfer). Both tasks involve 28×28 grayscale image classification into 10 classes. They share identical input dimensionality, output dimensionality, and spatial structure. The substrate topology requirements (how many hidden nodes are needed, at what spatial resolution, with what connectivity) are similar enough that a configuration optimized for one task produces effective substrates for the other. The reduced variance (SD = 1.00% versus 3.08%) reflects a stable operating point in the configuration space that works across both tasks.

MNIST → Logic operations (partial transfer). Logic operations differ fundamentally from MNIST: two binary inputs versus 784 continuous pixel values, four possible input patterns versus 60,000 training images. The transferred configuration generates substrates tuned for high-dimensional spatial inputs, which is structurally mismatched with the two-dimensional binary domain. Transfer still helps for the harder logic operations (XOR, OR, NAND) because these tasks benefit from the well-tuned NEAT evolutionary

parameters (population size, mutation rates), even though the substrate-level parameters are suboptimal. For easy operations (AND, NOR), random search already finds adequate configurations, leaving no room for transfer to improve.

Transfer raises the floor, not the ceiling. Across the target tasks, the transferred configuration raised mean accuracy on five of six operations (the AND difference is statistically flat, $d = -0.11$) and reduced variance while occasionally lowering the best-case accuracy. This is a direct consequence of configuration transfer: the transferred settings lock the substrate into a regime that produces consistently adequate topologies, eliminating both the catastrophic failures and the occasional lucky hits that characterize random search. For practitioners, this is usually the preferred trade-off: reliable performance matters more than rare peaks.

Thesis-level insight: what the transfer pattern reveals. The differential transfer performance is diagnostic. Strong transfer to Fashion-MNIST (structurally similar) and weak transfer to logic operations (structurally dissimilar) confirms that the optimized parameters encode *substrate topology preferences* (specifically, assumptions about spatial resolution and input dimensionality) rather than universal evolutionary strategies. This means the 29.0% ceiling is not a failure of parameter tuning: it is a ceiling imposed by the kind of substrates that ES-HyperNEAT's quadtree mechanism can produce. Because the quadtree applies a uniform variance threshold across all regions, no choice of hyperparameters within this search space makes the substrate's resolution follow the input's information content rather than the CPPN's own output variance: the limitation is a property of the mechanism, not of the optimizer. This limitation, known as the "central bias" [78], is an architectural property that Chapter 4 addresses directly through spatial partitioning.

Retrospective: activation palette as a hidden transfer variable. In retrospect, the partial transfer to logic operations may reflect a deeper issue than structural mismatch alone. XOR and higher-parity tasks are known to benefit from oscillatory activation functions (`sin` or equivalent): a capability absent from the transferred MNIST-Config palette, which uses `softplus`/`clamped`. The transferred configuration locked the substrate into a non-oscillatory regime adequate for MNIST's spatial features but ill-suited to the structure that XOR rewards. This interpretation, unavailable at the time of the original study, reframes the XOR transfer limitation as a function palette constraint rather than solely a spatial resolution mismatch: a well-resolved substrate is still denied the function class that the companion parity evidence identifies as decisive. The broader lesson for transfer is that hyperparameter configurations encode not just structural priors but *computational capability priors* through their activation function selections: an observation that motivates per-node function evolution as a direction for future work (§8.4).

3.5 An Early-Stopping Pruner for ES-HyperNEAT

The TPE study of §3.2 demonstrated that Bayesian optimization navigates the ES-HyperNEAT hyperparameter space far more effectively than random search. However, the study also exposed a structural inefficiency that no surrogate model can eliminate: the majority of configurations (whether proposed by TPE or sampled at random) produce

substrates that stagnate at the 0.1 baseline and never exhibit learning. Each such trial still consumes a full evolutionary run before being discarded. In a 2,013-trial campaign, hundreds of dead-end runs consume budget that could instead evaluate new configurations in productive regions of the search space. This section develops a domain-specific early-stopping strategy to address this waste [12].

This waste is not incidental: it is a direct consequence of ES-HyperNEAT's binary fitness dynamics. As the bimodal distribution in Figure 3.15 will show, trials either develop viable substrate topology within the first few generations (and proceed to learn) or fail to do so (and stagnate indefinitely). There is no smooth gradient of partial learning. This binary character suggests that trial outcomes can be predicted early, turning a structural limitation into a practical advantage.

The natural follow-up question is: *can we predict, at an early generation, which trials will fail, and terminate them before completion?* The pruner developed below exploits these binary dynamics, complementing the inter-trial efficiency of TPE with intra-trial efficiency.

The problem is framed as binary classification. The input is a trial's fitness trajectory up to some early generation G; the output is a label predicting whether the trial will ultimately succeed or fail. Two quantities must be determined: the *pruning generation* G^* (how early can we intervene?) and the *fitness threshold* T^* (what minimum fitness at G^* distinguishes future successes from future failures?). The derivation uses a fresh set of 90 trials run to completion, the validation uses an independent set of 180 trials with the pruner actively deployed, and the analysis culminates in a direct comparison against the general-purpose Hyperband pruner [60].

Where the TPE study addresses inter-trial efficiency (which configurations to evaluate), this early-stopping study addresses intra-trial efficiency (how long to evaluate each configuration). The two are complementary and compose into the combined pipeline described in §3.8.

Algorithms 4 and 5 formalize the two-phase methodology. Phase 1 derives the pruning rule from a set of trials run to completion; Phase 2 deploys it on new trials. The remainder of this section details each step.

Algorithm 4 Domain-Specific Pruning Rule Derivation

Require: Derivation trials $\mathcal{D} = \{(\mathbf{c}_i, \{b_i^{(g)}\}_{g=1}^{G_{\max}})\}_{i=1}^{N_d}$; success percentile p; baseline fitness b_{baseline}

Ensure: Pruning rule (G^*, T^*)

Define $\widetilde{b}_i^{(g)} \leftarrow \text{median}\{b_i^{(1)}, \ldots, b_i^{(g)}\}$ {Median best fitness to generation g}
Label each trial: Positive if $\widetilde{b}_i^{(G_{\max})} \geq p$-th percentile of $\mathcal{D}$, else Negative
for each Positive trial i **do**
    $g_i \leftarrow$ first generation where $\widetilde{b}_i^{(g_i)} > b_{\text{baseline}}$
end for
$G^* \leftarrow \lceil \text{mean}(\{g_i\}) \rceil$
for each unique value of $\widetilde{b}_i^{(G^*)}$ across all trials as threshold τ **do**
    Classify trial i as Predicted Positive if $\widetilde{b}_i^{(G^*)} > \tau$
    Compute F_1 from the resulting confusion matrix
end for
$T^* \leftarrow \arg\max_\tau F_1$
return (G^*, T^*)

Algorithm 5 Pruning Rule Deployment

Require: Trial with fitness trajectory $\{b^{(g)}\}$; pruning rule (G^*, T^*)
1: Run trial to generation G^*
2: Compute $\widetilde{b}^{(G^*)} = \text{median}\{b^{(1)}, b^{(2)}, \ldots, b^{(G^*)}\}$
3: **if** $\widetilde{b}^{(G^*)} \leq T^*$ **then**
4: **Terminate** trial (predicted failure)
5: **else**
6: **Continue** trial to generation $G_{\max}$
7: **end if**

3.5.1 Phase 1: Deriving the Pruning Rule

Experimental Design

The derivation dataset consisted of 30 hyperparameter configurations drawn at random via Optuna's sampler, each evaluated under three independent seeds for a total of 90 trials. This layout (30 configurations times 3 replicates) distributes sampling across the search space while reducing the chance that a single unlucky evolutionary run determines a configuration's label.

All trials ran to completion at 17 generations, slightly below the 20-generation TPE budget of §3.2 but with only 2.7% accuracy loss, a trade-off that permitted more configurations within the available compute. As in the TPE campaign, the PUREPLES framework [103] provided the ES-HyperNEAT implementation; all efficiency metrics are reported in generations rather than wall-clock time to account for the heterogeneous hardware across the 14 machines used.

The hyperparameter search space retained the same 16 parameters from §3.2 but extended several ranges to increase diversity in the derivation dataset. Table 3.14 lists the complete search space.

The fitness protocol replicates §3.2: 200 MNIST images balanced across all 10 classes per evaluation batch, with fitness equal to the fraction of correct classifications. A substrate that produces random or constant predictions therefore scores 0.1, the chance-level baseline that defines the lower boundary of the binary fitness landscape.

Trial Labeling

Each trial received a binary label based on its final median best fitness. Trials whose final value fell in the 80th percentile or above were labeled *Positive* (successful); all others were labeled *Negative* (unsuccessful). The resulting split (19 Positive versus 71 Negative) follows from the 80th-percentile cutoff itself; the sensitivity analysis below reports how that percentile was selected.

The 80th percentile was selected after a sensitivity analysis across four percentile definitions (50th, 75th, 80th, 90th), each yielding a complete pruning rule that was evaluated on the independent validation set. Table 3.15 summarizes the results.

The trade-offs across percentile definitions are instructive. The 50th percentile delays intervention until generation 13 and achieves only 11.4% savings, barely worth the implementation overhead. This late trigger arises because the ES-HyperNEAT landscape is so dominated by failing configurations that even the median performer barely exceeds random guessing, preventing early differentiation. The 90th percentile achieves the same

Hyperparameter	Range
NEAT Parameters	
`num_hidden`	0, 1, 2, 3, 4, 5
`pop_size`	30, 40, 50, . . . , 120, 130
`conn_add_prob`	0.1, 0.2, . . . , 0.9
`conn_delete_prob`	0.1, 0.2, . . . , 0.9
`node_add_prob`	0.1, 0.2, . . . , 0.9
`node_delete_prob`	0.1, 0.2, . . . , 0.9
`initial_connection`	full_nodirect, full_direct, unconnected, fs_neat_hidden, fs_neat_nohidden
`connection_fraction`	0.2, 0.4, 0.6, 0.8
`activation_default`	sigmoid, gauss, clamped, relu, tanh, softplus
Substrate Parameters	
`initial_depth`	1, 2, 3, 4
`max_depth`	2, 3, 4, 5, 6
`variance_threshold`	0.005, 0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04
`division_threshold`	0.1, 0.2, . . . , 0.9
`max_weight`	1.0, 2.0, . . . , 7.0
`iteration_level`	0, 1, 2, 3, 4
`activation`	sigmoid, gauss, clamped, relu, tanh, softplus

Table 3.14: Extended hyperparameter search space for the pruner derivation study. Compared to the ranges in §3.2, the population size is widened to 30–130 (from 10–100) and the variance threshold is refined to 8 values (from 4). All other parameters not listed use the NEAT-Python XOR configuration defaults.

Success Def.	Rule (G^*, T^*)	Recall	Precision	F_1	Savings (%)
50th percentile	(13, 0.145)	0.52	1.00	0.68	11.4
75th percentile	(4, 0.143)	0.86	0.97	0.91	38.2
80th percentile	**(3, 0.140)**	**0.90**	**0.84**	**0.87**	**41.6**
90th percentile	(3, 0.140)	0.94	0.70	0.80	41.6

Table 3.15: Sensitivity analysis for the success-threshold percentile. Each rule is derived on the 90-trial derivation set and evaluated on the 180-trial validation set. The 50th percentile waits until $G^* = 13$ and captures only a fraction of the possible savings. The 90th percentile is too aggressive (precision drops to 0.70). The 80th percentile achieves high recall (0.90) while attaining the largest savings, providing the best trade-off for a pruning mechanism where minimizing false negatives is the primary safety requirement.

savings as the 80th but at the cost of precision (0.70 vs. 0.84), allowing too many unsuccessful trials through. The 75th percentile yields the highest F_1 (0.91) but with lower recall (0.86 vs. 0.90) and lower savings (38.2% vs. 41.6%). For a pruning mechanism, the cost of a false negative (discarding a trial that would have succeeded) exceeds the cost of a false positive (allowing a doomed trial to continue), making recall the priority. The 80th percentile provides the best balance of high recall and high savings.

The Pruning Metric: Median Best Fitness

The classification metric is the *median best fitness* at generation G: the median of the set $\{b_1, b_2, \ldots, b_G\}$, where b_g denotes the fitness of the best individual in generation g. This cumulative statistic has two advantages over single-generation metrics. First, it is robust to single-generation outliers: a lucky batch sample at one generation does not inflate the signal. Second, it captures the *trend* of the fitness trajectory: a trial where the best individual improves even slightly across generations will have an increasing median, while a stagnating trial will have a flat one.

The choice of median best fitness over the simpler "last best fitness" was empirically motivated. The 80th percentile of last best fitness in the derivation set was 0.1 (the random-guessing baseline), which prevents any meaningful separation between Positive and Negative trials. Median best fitness, by contrast, accumulates information from the entire trajectory and provides a threshold that separates the two classes.

To confirm that the two metrics agree as final performance estimators (differing only in their discriminative power at early generations), paired t-tests were conducted within each experimental group. Table 3.16 shows that no group exhibits a significant difference ($p > 0.05$) and all effect sizes are negligible ($|d| < 0.2$). The median best metric is therefore a valid substitute for last best fitness without introducing systematic bias.

Experimental Group	t**-statistic**	p**-value**	**Cohen's** d
Random Search ($N = 90$)	-0.88	0.378	-0.13
Random Unpruned ($N = 90$)	-0.65	0.519	-0.10
Random Pruned ($N = 90$)	-1.00	0.321	-0.15

Table 3.16: Paired-samples t-tests comparing median best fitness and last best fitness as final performance estimators. No group shows a significant difference, confirming the two metrics agree asymptotically while differing in early-generation discriminative power.

Determining the Pruning Generation G^*

The pruning generation G^* was derived by identifying, for each of the 19 Positive trials, the first generation at which its median best fitness rose above the 0.1 chance-level baseline—the earliest detectable evidence of learning. Taking the ceiling of the mean across these 19 onset generations yielded $G^* = 3$.

The result is interpretable in terms of NEAT's complexification dynamics: by generation 3, substrate topology has either differentiated sufficiently to process at least some inputs correctly, or it has not. Successful trials accumulate enough fitness improvement for their median best fitness to clear baseline, while unsuccessful trials remain flat at 0.1.

Figure 3.10 shows the mean of the median best fitness for the two classes across all 17 generations. Divergence begins at generation 2: successful trials enter a steady

upward trajectory while unsuccessful trials flatline near the 0.1 baseline. By generation 3 the gap between the two classes reaches $\Delta = 0.037$, large enough to support a reliable pruning decision. This early divergence point is characteristic of NEAT's complexification dynamics: substrates that will eventually learn show topological differentiation within the first few generations, while substrates that will stagnate remain structurally static from the start. This observation connects to the sensitivity analysis of §3.3, which showed that topology-governing parameters (variance threshold, max depth) are the dominant performance determinants: trials that fail to develop appropriate topology by generation 3 are unlikely to recover.

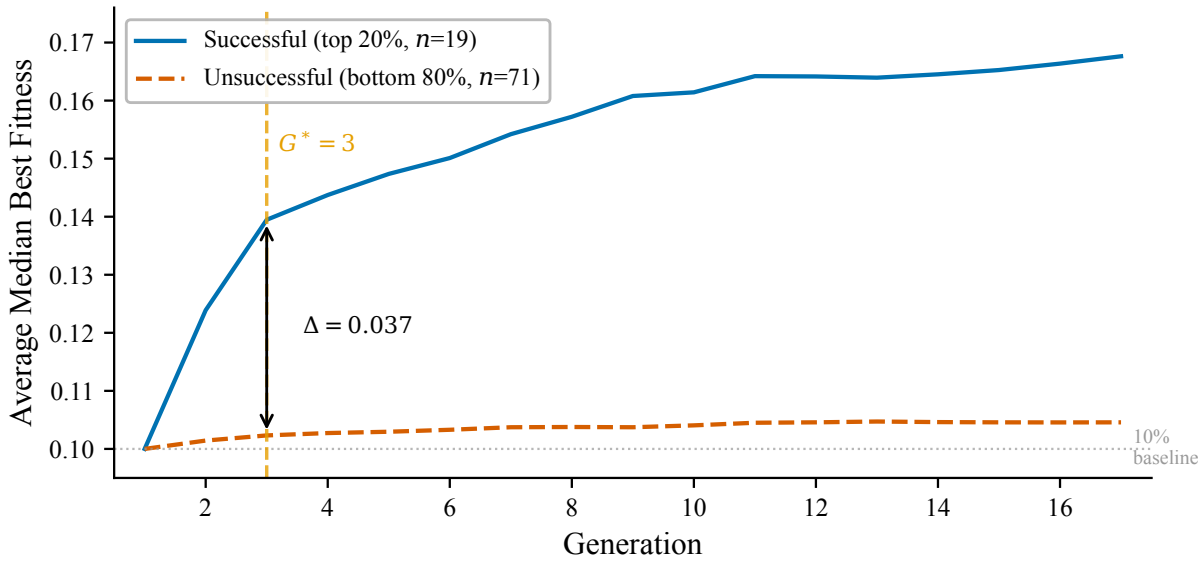


Figure 3.10: Fitness trajectories of Successful (top 20%, $n = 19$) vs. Unsuccessful (bottom 80%, $n = 71$) trials in the derivation set. Each curve plots the mean of the median best fitness across trials in that class. Successful trials begin separating from unsuccessful trials at generation 2; by generation 3 the gap reaches $\Delta = 0.037$, large enough to support a reliable classification rule. Unsuccessful trials remain near the 0.1 baseline throughout. The vertical dashed line marks the derived pruning generation $G^* = 3$.

Determining the Pruning Threshold T^*

With $G^* = 3$ fixed, the threshold T^* was determined through an exhaustive sweep over every unique median best fitness value observed at generation 3 in the 90-trial derivation set. Each candidate τ induces a classification: trials with generation-3 median best fitness above τ are predicted successes, while those at or below τ are predicted failures. Comparing these predictions against the ground-truth labels produces a confusion matrix and its associated F_1 score. The threshold maximizing F_1 was selected as T^*.

Figure 3.11 plots F_1, precision, and recall as a function of the candidate threshold at generation 3. The optimum occurred at $T^* = 0.140$, yielding a derivation $F_1 = 0.788$.

An F_1 of 0.788 on the derivation set understates the pruner's discriminative power. Two factors pull the score down: the heavy 19–71 class imbalance and boundary fuzziness at the 80th-percentile cutoff, where trials whose final fitness is indistinguishable within noise are placed on opposite sides of the decision. The validation set produces a higher F_1, as shown next.

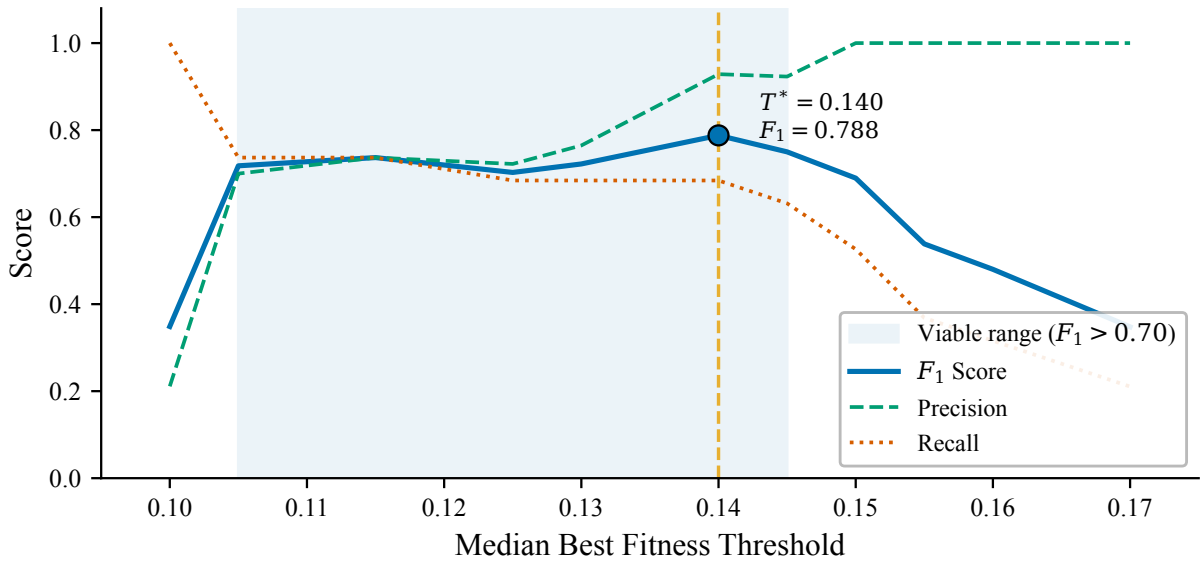


Figure 3.11: F_1 score, precision, and recall for candidate pruning thresholds at generation $G^* = 3$. The optimal threshold $T^* = 0.140$ maximizes F_1 at 0.788. Below this threshold, recall increases but precision drops sharply. Above it, precision increases but recall drops as promising "slow starters" are discarded.

The derived pruning rule (MNIST problem). Prune any trial where the median best fitness at generation 3 is ≤ 0.140.

Definition 3.1 (Domain-Specific Pruning Rule for ES-HyperNEAT)**.** *Given pruning generation G^* and threshold T^*, a trial is terminated at generation G^* if its median best fitness satisfies:*

$$\widetilde{b}_{G^*} = \text{median}\{b_1, b_2, \ldots, b_{G^*}\} \leq T^* \tag{3.1}$$

where b_g denotes the fitness of the best individual in generation g. For ES-HyperNEAT on MNIST with 200-image batches: $G^* = 3$, $T^* = 0.140$.

3.5.2 Phase 2: Validation

Generalization was tested by deploying the derived rule ($G^* = 3$, $T^* = 0.140$) on a fresh set of randomly sampled configurations from Optuna. The pruner actively terminated any trial whose median best fitness at generation 3 fell at or below 0.140.

The validation design addresses a methodological challenge specific to early-stopping evaluation: pruned trials have no ground-truth final fitness. To obtain it, 90 randomly selected pruned trials were resumed and run to full completion (generation 17), revealing what the pruner had prevented. Combined with 90 unpruned trials, this counterfactual analysis yielded 180 trials with known outcomes: enough to construct a confusion matrix and assess whether the derivation-set rule transfers reliably to unseen configurations.

Classification Performance

Table 3.17 shows the validation confusion matrix; Table 3.18 summarizes the derived classification metrics.

	Predicted Outcome	
Actual Outcome	**Successful**	**Unsuccessful (Pruned)**
Successful	75	8
Unsuccessful	14	83

Table 3.17: Confusion matrix for the pruner on the 180-trial validation set. TP=75 (successful trials correctly retained), TN=83 (unsuccessful trials correctly pruned), FP=14 (unsuccessful trials incorrectly retained), FN=8 (successful trials incorrectly pruned).

Metric	**Value**
Precision	0.843
Recall (Sensitivity)	0.904
Specificity	0.856
F_1 Score	0.872
Accuracy	0.878
Matthews Correlation Coefficient (MCC)	0.757

Table 3.18: Classification performance of the pruning rule on the validation set. The high recall (0.904) confirms that the pruner retains over 90% of trials destined for success. The MCC of 0.757 indicates reliable classification even under class imbalance.

The recall of 0.904 is the most important metric for a pruning mechanism: only 8 out of 83 successful trials were incorrectly terminated. Precision of 0.843 indicates that most trials allowed to continue were genuinely successful; the 14 false positives waste some computation but do not harm solution quality. The MCC of 0.757 provides a balanced summary that accounts for class imbalance and confirms reliable classification.

Validation F_1 (0.872) lands above the derivation F_1 (0.788), not because the rule was re-tuned (it stays fixed at $G^* = 3$, $T^* = 0.140$ per Phase 1), but because the validation evaluation uses a stratified mix of 90 resumed-pruned trials and 90 run-to-completion trials. That stratification shifts the class prior, which rebalances precision and recall relative to the derivation split.

Receiver Operating Characteristic Analysis

The pruning rule is explicitly framed as a binary classifier (Definition 3.1), so its discriminative power is naturally characterized by the receiver operating characteristic (ROC). Figure 3.12 plots the ROC curve computed on the 90-trial derivation set by sweeping the threshold across all observed generation-3 median best fitness values, together with the three validation operating points corresponding to pruning generations $G^* \in \{2, 3, 4\}$.

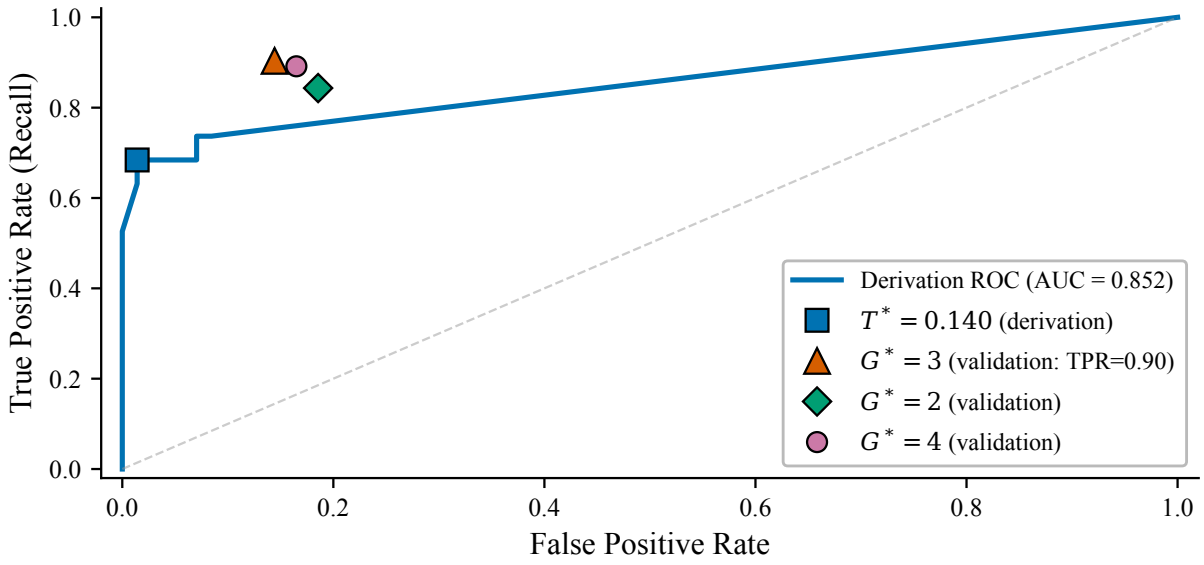


Figure 3.12: Receiver operating characteristic for the pruning rule. Blue curve: derivation-set ROC at generation 3 (area under the curve, AUC = 0.852). Red triangle: validation operating point at $G^* = 3$ (recall = 0.904, false positive rate, FPR = 0.144). Green diamond and purple circle: validation points for $G^* = 2$ and $G^* = 4$.

The AUC of 0.852 on the derivation set confirms that the generation-3 median best fitness separates successful from unsuccessful trials with good reliability. The validation operating point at $G^* = 3$ (red triangle) lies in the high-recall region of the curve, consistent with the design priority of minimizing false negatives. The close alignment between the derivation curve and the validation operating points provides further evidence that the pruner's discriminative signal generalizes beyond the derivation data.

Decision Boundary Visualization

Figure 3.13 provides a complementary view by plotting each derivation trial in the space of its generation-3 score (horizontal axis) versus its final fitness (vertical axis), with the pruning threshold $T^* = 0.140$ as a vertical decision boundary and the 80th-percentile success threshold as a horizontal line.

The four quadrants of the confusion matrix are directly visible. The upper-left quadrant contains the false negatives (trials with high final fitness but low generation-3 signal), illustrating the "slow starter" pattern explored in the next subsection. The lower-right quadrant contains the false positives: trials with high generation-3 signal that later stagnated, likely due to transient batch-sampling variance. The sharpness of the boundary between the upper-right (true positive) and lower-left (true negative) clusters confirms the binary dynamics that make single-checkpoint pruning effective.

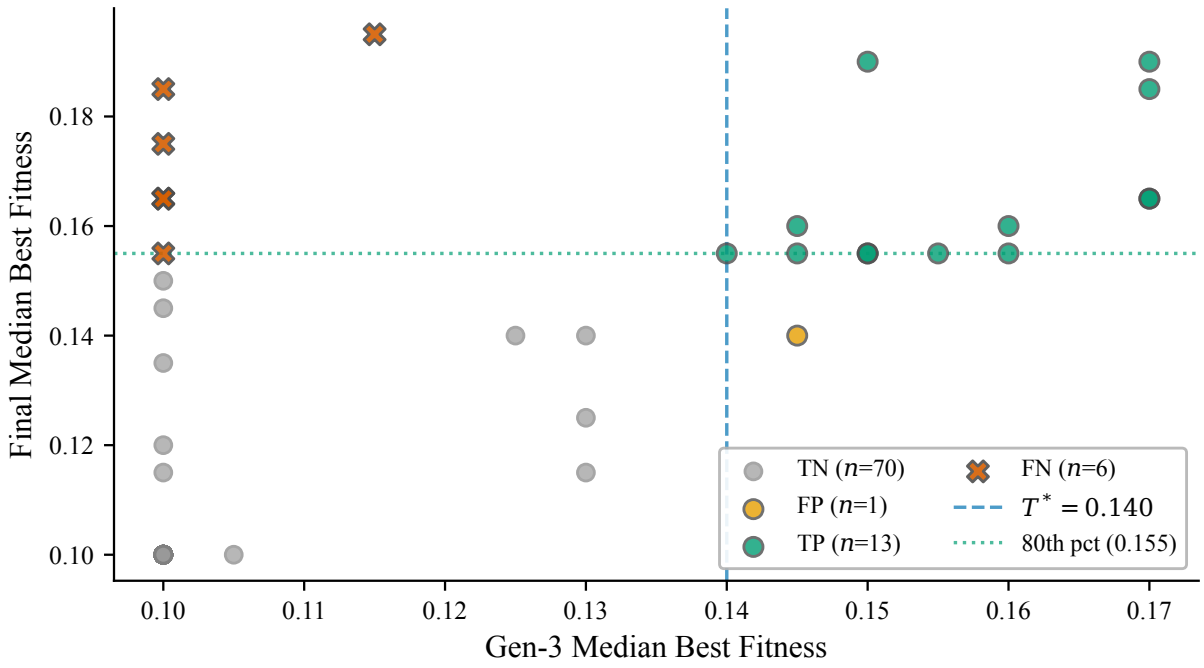


Figure 3.13: Decision boundary scatter for the 90-trial derivation set. Axes: generation-3 median best fitness (horizontal) vs. final median best fitness (vertical). Colors: true positives (green), true negatives (gray), false positives (orange), false negatives (red crosses). Vertical dashed line: $T^* = 0.140$; horizontal dotted line: 80th-percentile success threshold.

False Negative Analysis

The 8 false negatives (successful trials that the pruner incorrectly terminated) follow a consistent pattern. Figure 3.14 plots their full fitness trajectories.

These "slow starters" are trials whose learning onset is delayed beyond generation 3 but which ultimately converge to high fitness. Their existence represents the irreducible trade-off of any early-stopping strategy: an aggressive cutoff necessarily risks discarding a small number of trials with delayed learning curves. The pruner is calibrated to accept this risk (8 out of 83 successful trials, 9.6%) in exchange for the substantial computational savings quantified in §3.6.1.

The "slow starter" phenomenon connects to two broader themes of this thesis. One plausible explanation is that false negatives include trials where the CPPN has not yet discovered an appropriate activation function composition, a hypothesis consistent with the generation-3 divergence point and the sensitivity analysis showing that activation choice is secondary to topology (§3.3). Whether activation function selection can accelerate convergence is a question for future work on per-node function evolution (§8.4). Second, Chapter 4 diagnoses a central bias in ES-HyperNEAT's substrate formation: substrates either achieve sufficient spatial coverage of the input domain (learning immediately) or fail entirely (stagnating at baseline). The pruner detects precisely this structural dichotomy: its generation-3 checkpoint captures whether the substrate has developed viable topology, not merely whether the CPPN weights have converged. Both connections are explored further in §3.6.3.

Trajectory analysis of the 8 validation false negatives confirms the "slow starter" mechanism quantitatively. All 8 trials eventually exceed $T^* = 0.140$, but their median learning

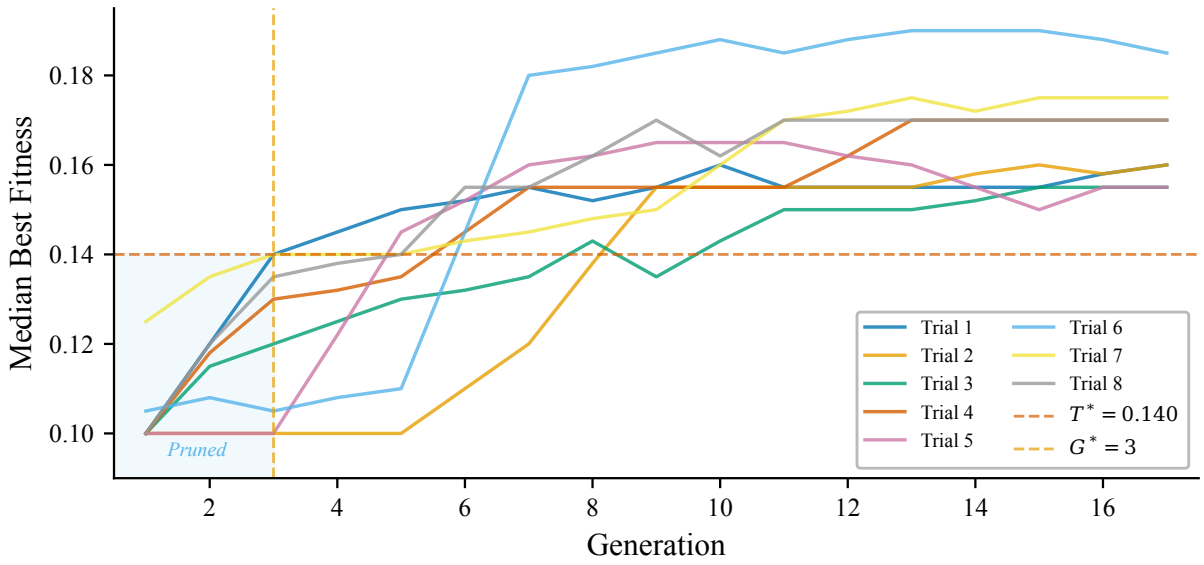


Figure 3.14: Fitness trajectories of the 8 successful trials incorrectly pruned (false negatives). The horizontal dashed line marks the pruning threshold $T^* = 0.140$; the vertical dashed line marks the pruning generation $G^* = 3$. All 8 trials are "slow starters": their median best fitness remains below 0.140 at generation 3 but eventually rises to competitive final fitness. These delayed learning curves represent the fundamental trade-off of any early-stopping strategy.

onset is generation 6 (range 4–9), compared to generation 2 (range 1–3) for the true positives—a delay of 4 generations. At the pruning checkpoint ($G^* = 3$), the false negatives show median fitness 0.125 (range 0.100–0.140), below the threshold though with a median above baseline; they are configurations where learning has begun but has not yet crossed the decision boundary. The true positives, by contrast, show median gen-3 fitness of 0.180, well above the threshold. This 4-generation onset gap is consistent with the sensitivity hierarchy from §3.3: configurations requiring denser substrates (high `max_depth`, fine `variance_threshold`) need more generations to build sufficient topology before learning becomes detectable. With only $N = 8$ false negatives, these observations are suggestive rather than conclusive, but the pattern is clear: the pruner's errors are delayed learners, not non-learners.

The 14 false positives (unsuccessful trials that passed the pruner) result from batch-sampling variance: these trials showed transient fitness above 0.140 at generation 3 due to a favorable random batch, but their learning stagnated in subsequent generations. False positives reduce computational savings rather than solution quality, and their cost is reflected in the precision of 0.843.

Statistical Validation of $G^* = 3$

To confirm that $G^* = 3$ is genuinely optimal and not an artifact of the derivation dataset, its performance was validated against adjacent generations ($G^* = 2$ and $G^* = 4$) on the 180-trial validation set. For each candidate generation, the threshold derived from the full derivation analysis was used ($T^* = 0.122$ for $G = 2$, $T^* = 0.140$ for $G = 3$, $T^* = 0.143$ for $G = 4$). Table 3.19 presents the F_1 scores with bootstrap 95% confidence intervals (1,000 resamples).

Pruning Gen. (G^*)	**F_1 Score**	**95% CI**
2	0.819	[0.754, 0.878]
3	**0.872**	**[0.817, 0.924]**
4	0.869	[0.812, 0.917]
Paired difference ($G^*=3$ minus $G^*=2$): 0.053 [0.012, 0.098]		

Table 3.19: F_1 scores and bootstrap 95% confidence intervals for the pruning rule at adjacent generations, evaluated on the 180-trial validation set. $G^* = 3$ significantly outperforms $G^* = 2$ (paired difference CI excludes zero; McNemar's $p = 0.022$). $G^* = 3$ vs. $G^* = 4$ shows no significant difference (McNemar's $p = 0.375$), but $G^* = 3$ intervenes one generation earlier.

$G^* = 3$ significantly outperformed $G^* = 2$. McNemar's exact test confirmed a significant difference ($p = 0.022$): $G^* = 3$ correctly classified 11 trials that $G^* = 2$ misclassified, while only 2 trials showed the reverse. The paired bootstrap test corroborated this: the 95% confidence interval for the F_1 difference [0.012, 0.098] excludes zero ($p = 0.003$). Waiting until generation 3 yields a significantly more reliable pruning decision than intervening at generation 2.

$G^* = 3$ and $G^* = 4$ showed no significant difference (McNemar's $p = 0.375$), but $G^* = 3$ intervenes one generation earlier, saving an additional $\frac{1}{17} \approx 5.9\%$ of computational budget per pruned trial. $G^* = 3$ therefore represents the statistically validated optimal trade-off between early intervention and reliable identification.

The robustness of $G^* = 3$ across the validation data provides confidence that this generation reflects a genuine property of NEAT's complexification dynamics rather than a dataset artifact. Substrates that will eventually learn develop distinguishable topological signatures within the first three generations, a property that may generalize to other NEAT-based algorithms, as discussed in §3.6.3.

Threshold Robustness

The preceding analysis validated $G^* = 3$ against adjacent generations. A complementary question is whether $T^* = 0.140$ is robust to perturbation: if the threshold were shifted to 0.130 or 0.150, would validation performance degrade catastrophically?

Table 3.20 addresses this by sweeping the threshold at fixed $G^* = 3$ on the validation set. For thresholds below 0.140, additional pruned trials (originally terminated at $T^* = 0.140$) are reclassified as passing, and their true labels, obtained through the counterfactual resumption protocol, determine the resulting confusion matrix.

F_1 varies smoothly across the tested range, remaining above 0.83 for all thresholds between 0.100 and 0.145. The peak validation F_1 (0.883) occurs at $T = 0.130$, within 0.003 of the operating point at $T^* = 0.140$. This near-optimality is reassuring: $T^* = 0.140$, derived from the 90-trial derivation set, transfers to the validation set without requiring recalibration. The trade-off between the two quantities ($T = 0.130$ gains 2.3 pp of recall at the cost of 1.4 pp of precision and 1.9 pp of savings) is marginal; small perturbations to the threshold do not substantially alter the pruner's behavior. Combined with the G^* robustness analysis above, these results establish that neither component of the pruning rule ($G^* = 3$, $T^* = 0.140$) is overfit to the derivation data.

Threshold T	F_1	Recall	Precision	Savings (%)
0.100	0.832	0.976	0.726	30.4
0.110	0.862	0.964	0.779	34.5
0.120	0.860	0.952	0.784	35.4
0.130	0.883	0.940	0.832	38.6
0.140	**0.880**	**0.917**	**0.846**	**40.5**
0.145	0.867	0.893	0.843	41.4

Table 3.20: Threshold robustness analysis at fixed $G^* = 3$ on the validation set. F_1 remains above 0.83 across the entire range $T \in [0.100, 0.145]$, confirming that $T^* = 0.140$ is not a fragile knife-edge. The peak F_1 (0.883) occurs at $T = 0.130$, within 0.003 of the operating point. Lower thresholds increase recall at the cost of precision and savings; higher thresholds increase savings at the cost of recall. Note: the $F_1 = 0.880$ at $T^* = 0.140$ differs from the confusion-matrix $F_1 = 0.872$ (Table 3.18) because the robustness sweep recomputes predictions on the stratified validation set (90 resumed + 90 unpruned), whose class balance shifts when threshold-dependent reclassification is applied.

3.6 Generalization and Boundary Conditions

The preceding section established that the pruning rule ($G^* = 3$, $T^* = 0.140$) works as a binary classifier: it retains 90.4% of successful trials while correctly pruning 85.6% of unsuccessful ones, with an AUC of 0.852 and robustness to both generation and threshold perturbation. The pruning rule exists; the question is now whether it translates into meaningful computational savings, and where its assumptions break down. This section answers both questions. The results complete the intra-trial component of TRQ1 and inform the deployment guidance of the combined pipeline (§3.8).

3.6.1 Computational Cost Reduction

A trial terminated by the pruner runs for 3 generations; a completed trial runs for 17. On the 180-trial validation set, total cost without pruning was:

$$C_{\text{baseline}} = 180 \times 17 = 3{,}060 \text{ generations} \tag{3.2}$$

With the pruner deployed, 91 trials were terminated at generation 3 and 89 trials ran to completion:

$$C_{\text{pruned}} = (91 \times 3) + (89 \times 17) = 273 + 1{,}513 = 1{,}786 \text{ generations} \tag{3.3}$$

This represents a 41.6% reduction in total computational cost ($\frac{3{,}060-1{,}786}{3{,}060} = 0.416$), while retaining 90.4% of successful trials (recall = 0.904).

The savings compound across large-scale campaigns. In a 2,000-trial TPE search like the one in §3.2, applying the pruner would redirect thousands of otherwise wasted generations toward evaluating new configurations, effectively increasing the throughput of the surrogate model. Section 3.8 quantifies this compounding effect in the combined pipeline. These generation-based savings also understate the wall-clock benefit. Figure 3.4 shows that per-generation cost increases steadily as NEAT complexifies the CPPN (§3.2.7); by terminating at generation 3, the pruner avoids the most expensive portion of each failed trial.

3.6.2 Comparison with Hyperband

A natural question is how the domain-specific pruner compares to Hyperband [60], the most widely used general-purpose early-stopping method. Hyperband allocates budget through successive halving rounds (§2.2.3), progressively eliminating the worst-performing fraction of trials at fixed checkpoints. For this comparison, Optuna's HyperbandPruner was configured with `min_resource=2`, `max_resource=17`, and `reduction_factor=2`, creating four successive evaluation rungs: the standard parameterization for a 17-generation budget. Both pruning methods were paired with TPE sampling to isolate the pruning strategy as the independent variable; the unpruned TPE baseline from §3.2 served as the control.

Three findings emerge from the comparison (Table 3.21).

Metric	Our Method	Hyperband	Baseline (No Pruner)
Best final fitness found	0.225	0.250	0.275
Avg. fitness of top 5% trials	0.206	0.195	0.241
Total cost (generations)	10,656	29,574	40,260
Trials to reach 0.25 fitness	Not reached	569	359

Table 3.21: Comparison of the domain-specific pruner, Hyperband, and the unpruned TPE baseline. The domain-specific pruner achieves 64% fewer generations than Hyperband while maintaining higher mean fitness among the top 5% of trials. Hyperband discovers a higher peak (0.250 vs. 0.225) at 2.8× more computation. The unpruned baseline finds the highest peak (0.275) at 3.8× more computation (3.2× at a matched 17-generation budget).

Efficiency. The domain-specific pruner reduced total cost to 10,656 generations, compared to 29,574 for Hyperband and 40,260 for the unpruned baseline: 64% fewer generations than Hyperband and 73.5% fewer than baseline. The baseline column is the 20-generation campaign of §3.2, while both pruned arms cap trials at 17 generations, so part of that gap reflects the longer per-trial budget; matched at 17 generations per trial, the baseline reduction is 68.9% and the cost ratio ∼3.2×. This efficiency gap arises because the domain-specific rule terminates trials at a single fixed checkpoint ($G^* = 3$), while Hyperband's multi-rung schedule evaluates trials at multiple intermediate points before fully eliminating them. For ES-HyperNEAT's binary fitness dynamics, where trials either show learning early or never learn at all, a single-checkpoint strategy is better calibrated than a gradual-elimination schedule designed for smoothly improving learning curves.

Figure 3.15 provides direct evidence for this binary character. The histogram of final accuracies across all 2,013 TPE trials from §3.2 is sharply bimodal: 335 trials (16.6%) stagnate at the 10% baseline (no learning at all) and 1,508 (74.9%) form a broad distribution from 15% to 27.5%, with only 170 trials (8.4%) falling in between. The population-level fitness distribution is sharply bimodal (trials either stagnate near baseline or achieve measurable accuracy), but among those that learn, a continuous performance gradient exists (as the ROC curve in Figure 3.12 confirms). This binary character is precisely why Hyperband's multi-rung schedule, designed for learning curves that improve gradually, underperforms the domain-specific single-checkpoint rule. The comparison in Table 3.21 favors the domain-specific approach: the pruner exploits ES-HyperNEAT's bimodal dynamics, whereas Hyperband uses task-agnostic successive halving brackets. A tuned Hy-

perband configuration might narrow the 64% generation gap, though the magnitude of the reduction suggests that the domain-specific advantage is substantial.

Consistency. The average fitness of the top 5% of trials was higher for the domain-specific pruner (0.206) than for Hyperband (0.195). While the pruner explored fewer total configurations, it more reliably identified and retained high-quality ones. The domain-specific threshold, calibrated to the empirical learning dynamics of ES-HyperNEAT, made better pruning decisions than Hyperband's domain-agnostic successive halving.

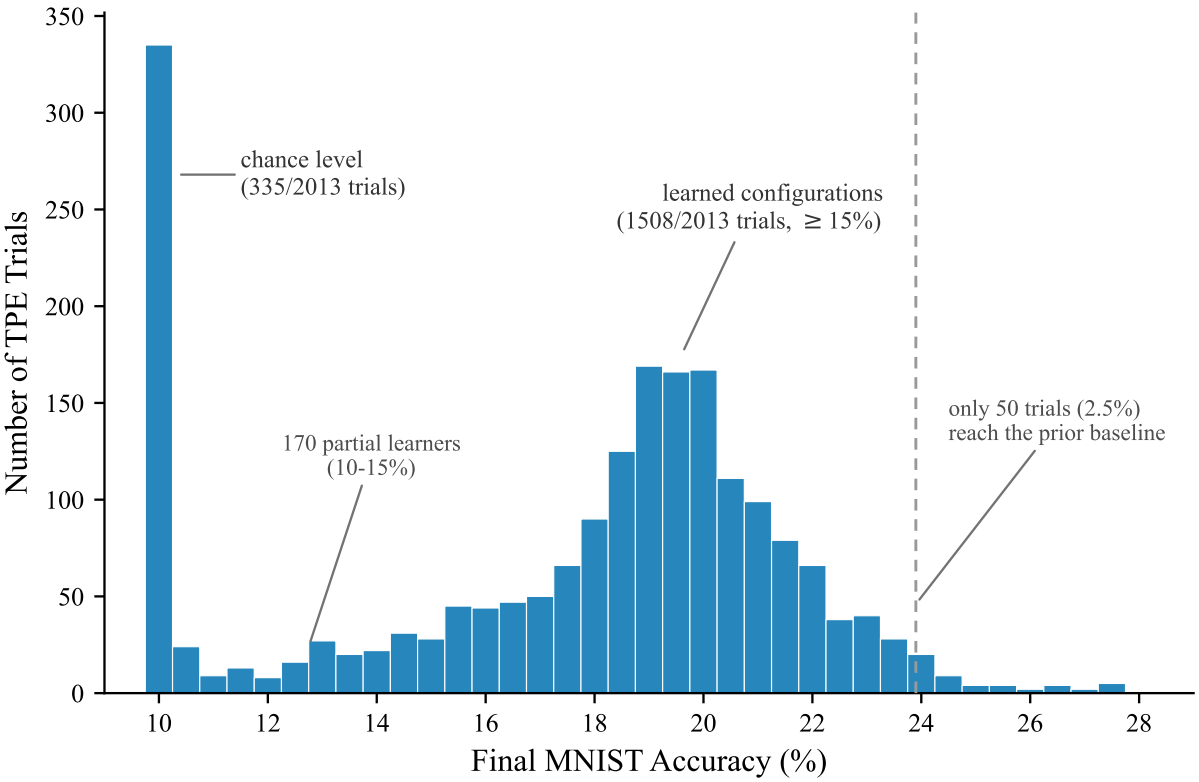


Figure 3.15: Distribution of final MNIST accuracy across 2,013 TPE trials. The distribution is sharply bimodal: 16.6% of trials stagnate at the 10% chance level (no learning) and 74.9% reach 15% or above, leaving only 8.4% in between. That deep trough, rather than a smooth gradient of partial learners, is what makes single-checkpoint pruning effective: with few exceptions, trials either learn or they do not. The dashed line marks the previous HyperNEAT baseline of 23.9%, which only 50 trials (2.5%) reach.

Peak discovery. Hyperband discovered a higher peak fitness (0.250 vs. 0.225), and the unpruned baseline discovered the highest (0.275). This is the expected cost of aggressive early pruning: a small number of "slow starter" trials that would have reached exceptional fitness were terminated. The unpruned baseline, having the luxury of running every trial to completion, had the highest probability of discovering outlier peaks; its longer 20-generation per-trial budget (against the pruned arms' 17) also favors it on the peak and trials-to-threshold rows of Table 3.21.

Statistical Analysis

A rigorous comparison of the three methods was conducted by analyzing the distributions of all final fitness scores from every completed trial. Combining the current study's trials

with the 2,013 trials from §3.2, a one-way ANOVA revealed a significant difference in mean final fitness across the three methods ($F(2, 3676) = 1090.7$, $p < .001$). Table 3.22 presents the Tukey HSD post hoc comparisons.

The domain-specific pruner significantly outperformed Hyperband in mean population fitness (Cohen's $d = 1.884$), and this consistency advantage extends across the full fitness distribution, not merely the top 5%. Both the pruner and the unpruned baseline significantly outperformed Hyperband. The root cause is a mismatch between Hyperband's assumptions and ES-HyperNEAT's dynamics: Hyperband presupposes smoothly improving learning curves where intermediate evaluations provide useful ranking information, but ES-HyperNEAT exhibits the binary fitness dynamics shown in Figure 3.15: trials either learn immediately or never learn at all, making multi-rung evaluation schedules wasteful. This finding reinforces a broader thesis theme: domain-specific methods outperform general-purpose alternatives when the problem structure departs from standard assumptions.

Comparison	**Mean Diff.**	p**-value**	**Significant**	**Cohen's** d
Our method vs. Hyperband	0.052	$< .001$	Yes	1.884
Baseline vs. Hyperband	0.058	$< .001$	Yes	1.555
Our method vs. Baseline	-0.006	0.069	No	—

Table 3.22: Tukey HSD post hoc comparisons of mean final fitness. The domain-specific pruner significantly outperforms Hyperband in mean population fitness ($d = 1.884$, a large effect). No significant difference exists between the pruner and the unpruned baseline ($p = 0.069$).

The comparison between the pruner and the unpruned baseline was not significant ($p = 0.069$), but a non-significant p-value does not constitute evidence of equivalence. To make a rigorous equivalence claim, a Two One-Sided Tests (TOST) analysis was performed with an equivalence margin of $\Delta = \pm 0.02$ absolute fitness, corresponding to approximately four additional correctly classified images per 200-image batch (four times the single-image granularity of the evaluation protocol).

Equivalence Test Metric	**Value**
Equivalence margin (Δ)	± 0.020
Observed mean difference	-0.0059
TOST p-value	$< .001$
Conclusion	**Statistically equivalent**

Table 3.23: Two One-Sided Tests (TOST) for equivalence between the domain-specific pruner and the unpruned baseline. The significant p-value ($< .001$) rejects non-equivalence within the ± 0.02 margin, confirming that the pruner achieves statistically equivalent search performance at a fraction of the cost (a 41.6% reduction, validated on the 180-trial set; §3.6.1).

The TOST analysis (Table 3.23) yielded $p < .001$, formally rejecting the hypothesis of non-equivalence. The domain-specific pruner achieves statistically equivalent search performance to the full, unpruned baseline, while cutting the validated per-campaign cost

by 41.6% (§3.6.1). The pruner's contribution is not finding better solutions; it is finding equivalent solutions with substantially less computation.

The conceptual significance of this equivalence extends beyond the cost saving itself. Non-significant p-values from standard tests (such as the Tukey HSD $p = 0.069$ above) establish only that we cannot reject the null hypothesis of no difference; they do not constitute evidence of equivalence. The TOST framework inverts this logic: by defining a substantively meaningful equivalence margin (± 0.02, corresponding to approximately four additional correctly classified images per 200-image batch) and rejecting non-equivalence within that margin, the analysis provides positive evidence that the pruner preserves search quality rather than merely failing to detect harm. This distinction matters because the pruner removes a large share of the computational effort — 41.6% in the validated 180-trial run (§3.6.1), 68.9% against the budget-matched baseline of this comparison (§3.6.2) — a level of compression that would be expected to degrade search quality if the pruned trials contained any useful information. That it does not degrade quality confirms that the pruned trials are genuinely uninformative: they occupy dead zones of the hyperparameter space from which no subsequent evolution can recover, consistent with the binary dynamics characterized in Figure 3.15.

This equivalence result has practical implications for the remaining experiments in this thesis. Chapters 4–6 involve large-scale hyperparameter searches where the pruner could substantially reduce computational requirements without compromising the validity of comparisons. The deployment guidance for integrating the pruner into subsequent experimental campaigns is provided in §3.8.

3.6.3 Limitation: Converged Search Spaces

The pruning rule is derived from a random initial population and calibrated for the scenario where most configurations fail. A natural question is whether the rule remains effective later in a TPE campaign, when the search has converged toward promising regions and a higher proportion of configurations show genuine learning.

To test this boundary condition, the pruner was simulated on the 824 late-stage trials from §3.2 that contained per-generation data. These trials come from the latter half of the 2,013-trial TPE campaign, where the sampler has already identified productive hyperparameter regions. Table 3.24 summarizes the results.

Late-Stage Simulation Metric	**Value**
Simulated computational savings	73.2%
F_1 score on late-stage trials	0.402
Recall on top 20% late-stage trials	31.1%
Mean fitness of kept trials	0.1870
Mean fitness of pruned trials	0.1500
Statistical separation (t-test p-value)	$< .001$

Table 3.24: Counterfactual simulation of the pruning rule on 824 late-stage trials from §3.2. The rule remains statistically separable ($p < .001$) but recall drops to 31.1%: the threshold calibrated for random populations prunes too many "slow starters" common in converged search spaces.

The rule retained its ability to separate higher-fitness from lower-fitness trials ($t(822) = 15.9$, $p < .001$), and simulated savings rose to 73.2% because late-stage trials are more

likely to show learning at generation 3. However, the rule became too aggressive for this population: recall dropped to 31.1%, meaning 69% of the top-20% trials would be incorrectly terminated. The best-performing trial in the late-stage dataset would itself be pruned, reducing peak fitness from 27.5% to 24.5%.

Figure 3.16 illustrates the consequence: applying the static rule to a converged search space clips the upper tail of the fitness distribution, sacrificing the very outliers that TPE's converged sampling is designed to discover.

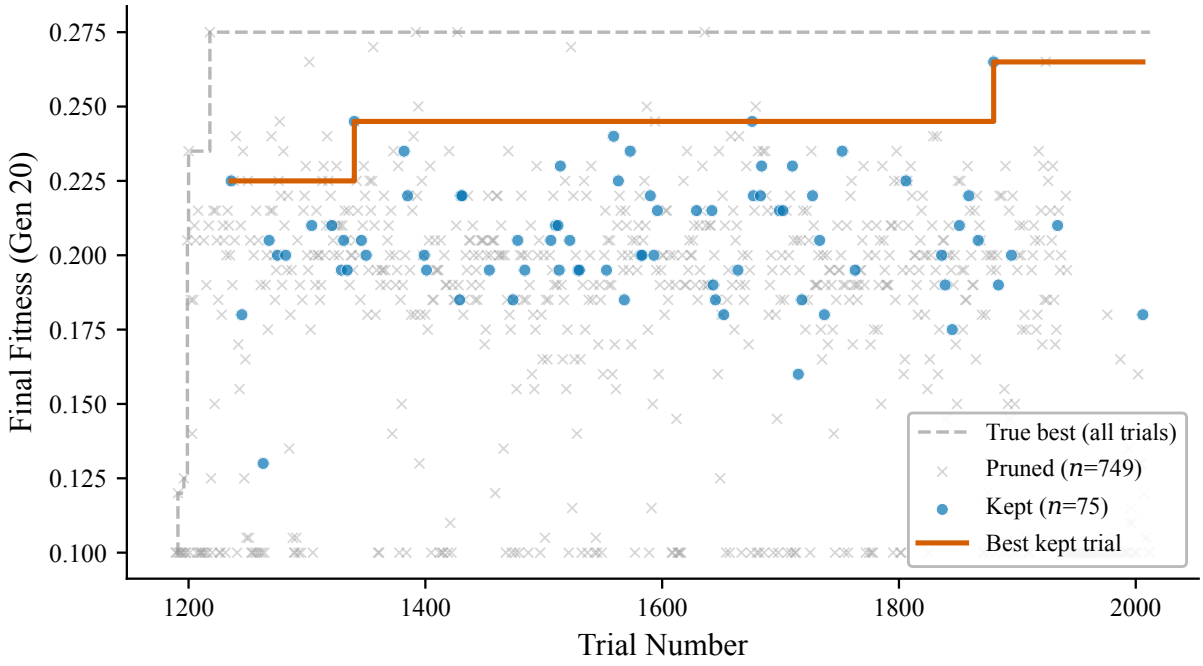


Figure 3.16: TPE search progress with the pruner simulated on the 824 late-stage trials from §3.2. The dashed line shows the true best fitness trajectory (unpruned). The solid line shows the best fitness that would have been retained had the pruner been active. The gap represents the cost of applying a static pruning rule to a converged search space: "slow starter" outlier trials that would have achieved the highest fitness are incorrectly terminated.

Figure 3.17 shows the individual fitness trajectories of the incorrectly pruned trials from this simulation. Unlike the early-stage false negatives in Figure 3.14, which are isolated slow starters in a predominantly failing population, the late-stage false negatives form a dense cluster of trials that genuinely learn but whose learning onset falls after generation 3. The contrast between the two figures illustrates the fundamental shift in the pruner's operating conditions: in a converged search space, the "slow starter" pattern is no longer an exception but the norm.

The onset distribution of these late-stage false negatives quantifies the shift. Among the 142 incorrectly pruned trials, 135 (95.1%) eventually exceed $T^* = 0.140$, with a median learning onset at generation 7 (IQR: 6–11, range: 1–17). By contrast, the 8 early-stage false negatives from §3.5.2 show median onset at generation 6 (range: 4–9) with $N = 8$: a similar delay but in a population where such delays are rare. In the converged space, 98.5% of false negatives begin learning after generation 3, and 41.5% remain at baseline at the pruning checkpoint. The failure mode is diagnostic and reveals a property of ES-HyperNEAT's evolutionary dynamics.

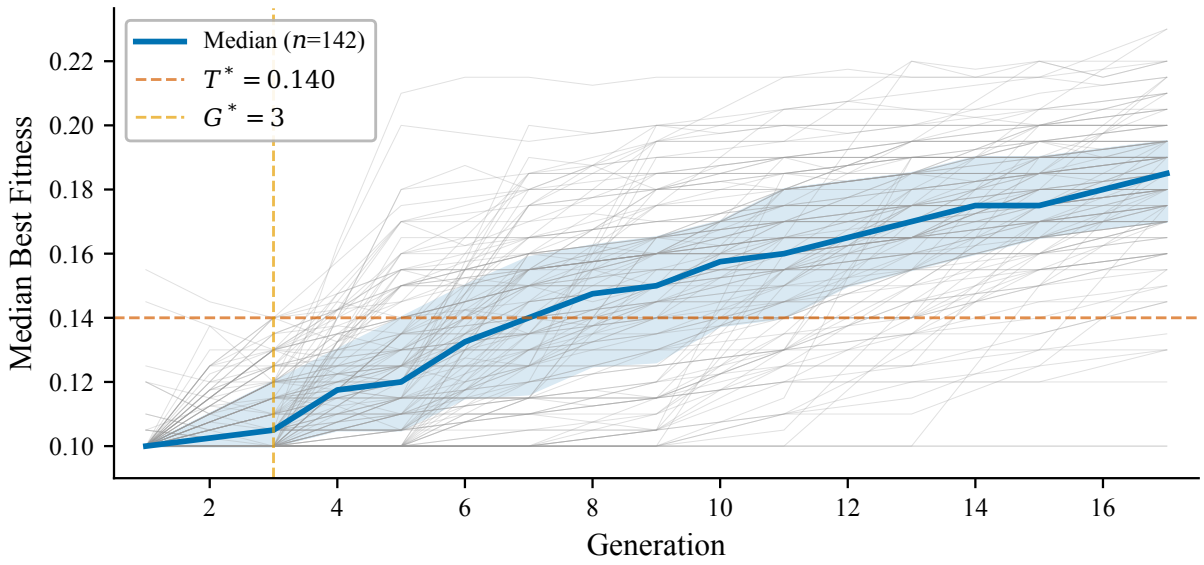


Figure 3.17: Individual fitness trajectories of late-stage trials that the pruner would have incorrectly terminated in the converged-space simulation. Unlike the 8 early-stage false negatives (Figure 3.14), these trials represent a substantial fraction of the converged population. Their delayed learning onset drives the recall collapse from 90.4% to 31.1%; the static threshold becomes systematically miscalibrated once the search concentrates on challenging but ultimately productive hyperparameter regions.

In a random initial population, most trials that show low fitness at generation 3 are genuinely stagnating; the pruner's assumption holds. In a converged population, many trials show low *early* fitness not because they are stagnating but because the TPE sampler is exploring more challenging hyperparameter combinations that require longer to develop. Configurations with large population sizes, deep substrates, or fine variance thresholds often need more generations to build sufficient topology before learning begins. These "slow starters" are exactly the configurations that the sensitivity analysis of §3.3 identified as most promising: the very trials the search should be encouraging rather than eliminating. The extended generation analysis (§3.2.6) and runtime scaling data (§3.2.7) quantify this trade-off: configurations that develop slowly through generation 20 contribute less than 3 pp of additional accuracy while incurring per-generation costs that grow 383-fold.

This limitation motivates two directions for future work. First, *adaptive pruning* mechanisms that begin aggressively and relax the threshold as the search converges, potentially tracking the running distribution of early-generation fitness across completed trials [1]. Second, *learning curve forecasting* methods that extrapolate the fitness trajectory rather than applying a fixed threshold, such as freeze-thaw Bayesian optimization [92] or hybrid methods like BOHB [28] and DEHB [2] that reallocate budget to late bloomers. Both approaches address the core limitation: the pruner's static threshold cannot distinguish between "genuinely stagnating" and "slow starting" trials, and this distinction becomes critical precisely when the search space converges.

For this thesis, the practical consequence is clear: the pruner should be deployed during the early, exploratory phase of a TPE campaign and disabled once the search begins to converge. The combined pipeline of §3.8 formalizes this recommendation.

3.7 Bridge: Why Two Methods Are Needed

Before combining the TPE and pruner into a unified pipeline, we explain why neither method alone suffices, and why their composition is natural rather than opportunistic.

TPE addresses the *inter-trial* problem: which of these 3×10^9 possible configurations should we evaluate? Without guidance, the vast majority of evaluations land in dead zones where substrates stagnate at baseline fitness. TPE's surrogate model concentrates evaluations in productive regions, but it cannot help with a complementary waste: the doomed trials that *start* in a productive region but reveal their futility only after several generations of evolutionary stagnation. A trial whose best fitness is still at the 0.100 chance-level baseline at generation 3 is far more likely to stagnate than to recover (the generation-3 rule reaches $F_1 = 0.872$ on the validation set, though §3.6.3 shows slow starters exist), yet TPE has no mechanism to exploit this because its model operates between trials, not within them.

The pruner addresses the *intra-trial* problem: given that we have chosen to evaluate a configuration, how early can we tell that it will fail? The generation-3 divergence (§3.5) provides a reliable signal, but the pruner says nothing about *which* configurations to try next. Applied to random search, the pruner would save 41.6% of generation cost but waste the freed budget on random configurations instead of informed ones.

The two methods thus operate on orthogonal axes: TPE selects *what* to evaluate, and the pruner determines *how long* to evaluate it. Their composition is multiplicative, not additive, because the savings from one method are reinvested by the other.

A worked example makes the compounding concrete. Consider a 1,000-trial optimization campaign with a 17-generation budget per trial:

- **Random search alone**: $1{,}000 \times 17 = 17{,}000$ generations. Over 90% of trials produce baseline accuracy.
- **TPE alone**: same 17,000 generations, but configurations concentrate in productive regions, and mean accuracy rises from 11.4% to 17.4%.
- **Pruner alone (with random search)**: 506 trials pruned at generation 3, 494 run to completion. Total: $(506 \times 3) + (494 \times 17) = 9{,}916$ generations (41.7% of the budget saved, matching the measured 41.6% within rounding). The freed 7,084 generations fund ∼716 additional random trials at the same pruned average cost of 9.9 generations per trial, but these are unguided.
- **TPE + Pruner**: same 9,916 generations for 1,000 initial configs, and the same freed budget funds ∼716 additional trials, now *TPE-guided*. Total: ∼1,716 configurations explored in the same 17,000-generation budget, each guided by the progressively refined surrogate model.

The compounding is now visible: pruning multiplies how many configurations the budget evaluates (∼1,716 in place of 1,000, a 72% increase at unchanged total computation, in either arm), and TPE multiplies what each evaluation is worth, spending the same freed budget on guided rather than unguided trials.

3.8 Combined Optimization Pipeline

The preceding sections developed two optimization strategies independently: TPE for inter-trial guidance (§3.2) and the early-stopping pruner for intra-trial cost reduction (§3.5). This section integrates them into a unified pipeline and quantifies the *projected* combined effect.

Important caveat. The combined pipeline described below was not empirically validated as an end-to-end system.

The TPE and pruner were developed and evaluated independently: TPE across the original 2,013-trial campaign, and the pruner on a separate 270-trial study. The cost projections below are derived analytically from the independently measured savings (41.6% pruning rate, 50.6% trial pruning ratio) under the assumption that the methods compose without interference. This assumption is reasonable (the pruner operates within individual trials while TPE operates between them, so their decision boundaries are non-overlapping), but it has not been verified by running a full combined campaign. We present the projected figures transparently and note where empirical validation would strengthen the claims. End-to-end empirical validation of the combined pipeline (running a full TPE campaign with the pruner active from the outset) represents a natural next step for future work.

3.8.1 Unified Framework

The two methods operate at different granularities and are naturally complementary:

Inter-trial optimization (TPE).
Between trials, TPE selects hyperparameter configurations using a surrogate model fitted to all previous trial outcomes. Each new configuration is drawn from the region of the 16-parameter search subspace that the surrogate predicts will yield high fitness. This concentrates the search budget: TPE evaluates more configurations in productive regions and fewer in the vast dead zones where substrates stagnate at baseline.

Intra-trial optimization (Pruner).
Within each trial, the pruner evaluates the fitness trajectory at generation 3 and terminates the trial if the median best fitness does not exceed $T^* = 0.140$. This eliminates doomed trials before they consume a full 17-generation run, freeing the saved generations for additional TPE-guided evaluations.

The two methods compose multiplicatively. Let α denote the fraction of total generations saved by the pruner (41.6%) and let β denote the number of additional configurations that can be evaluated with the freed budget. In a campaign of N trials without pruning, the combined pipeline runs N trials with pruning and then allocates the saved generations to β additional TPE-guided trials. Because the pruner and TPE operate on orthogonal axes (the pruner shortens individual trials while TPE selects which trials to run), there is no interference between them.

This composition supports two operational modes. In *cost-reduction mode*, the pipeline evaluates the same configurations TPE would have evaluated alone but consumes 41.6% fewer generations, completing the campaign in 58% of the time at unchanged solution quality. In *budget-matched mode*, the saved generations fund β additional TPE-guided trials at the empirical average of 9.9 generations per trial, yielding approximately 72% more evaluated configurations within the same total budget; because TPE's surrogate model improves with each completed trial, these additional evaluations are guided by a progressively refined model rather than allocated randomly. The choice between modes depends on whether computational budget or solution quality is the binding constraint; the 1,000-trial worked example in §3.7 illustrates both pathways.

Figure 3.18 illustrates the combined pipeline.

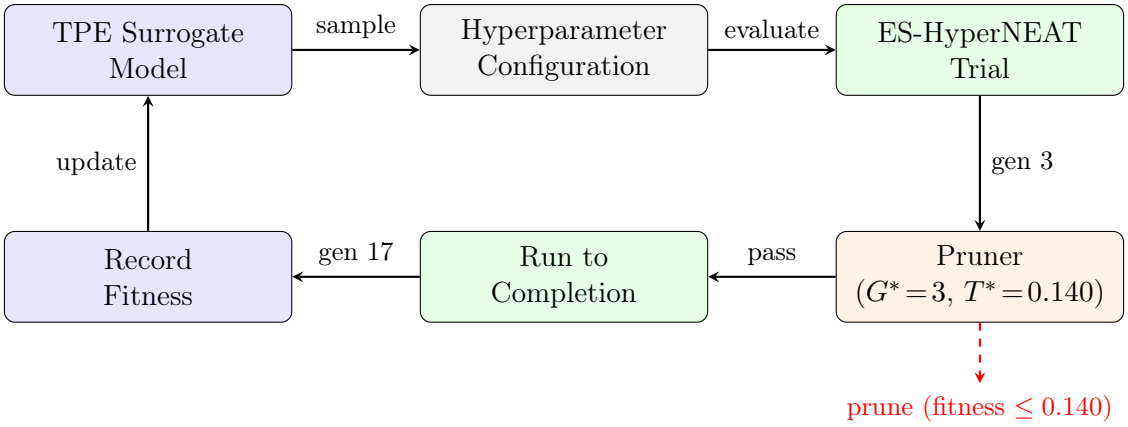


Figure 3.18: Combined optimization pipeline. TPE selects hyperparameter configurations (inter-trial); the pruner terminates unpromising trials at generation 3 (intra-trial). Pruned trials free computational budget for additional TPE-guided evaluations. Dashed red arrow indicates early termination.

3.8.2 Deployment Guidance

The combined pipeline requires two decisions from practitioners:

1. **Generation budget.** The pruner is calibrated for 17-generation trials. For substantially longer runs, the pruning generation G^* and threshold T^* should be re-derived on a pilot sample, because the relationship between early-generation fitness and final fitness may shift when the evolutionary budget is larger.

2. **Search phase.** The converged-space limitation (§3.6.3) means the static pruner should be disabled once the TPE search has converged, roughly when the running standard deviation of recent trial fitness drops below a task-specific threshold. In the MNIST campaign, convergence begins around trial 800 (Figure 3.1). A practical heuristic: enable the pruner for the first 50–70% of the campaign and disable it for the remainder, allowing the converged surrogate to explore subtle variations without risk of pruning slow starters.

Beyond these two decisions, the pipeline requires no task-specific tuning. The TPE surrogate adapts automatically to the search space. The pruner's threshold $T^* = 0.140$ is calibrated for MNIST's 200-image stratified batches (20 per class), where it corresponds to correctly classifying approximately 28 images; other tasks with different fitness scales require T^* re-calibration. The generation checkpoint $G^* = 3$ was derived from MNIST experiments only; we hypothesize that it may be more task-robust than T^* because it reflects NEAT's complexification dynamics rather than task-specific fitness scale, but cross-task validation is left for future work.

3.9 Synthesis: What Optimization Reveals About Architecture

The converged recipe as architectural prior. The TPE campaign produced more than a single best configuration. It produced a *converged recipe*: six hyperparameters that

stabilized across all high-performing trials (Table 3.10) while the remaining ten continued to vary. These six parameters (NEAT's initial topology `num_hidden` = 0 and population dynamics `pop_size` = 100, `conn_delete_prob` = 0.5, together with the substrate's initial depth, subdivision threshold, and iteration level) define the architectural prior for the ES-HyperNEAT experiments of Chapter 4; Chapters 5 and 6 instead vary population size and substrate depth deliberately as independent variables. Without this prior, the Chapter 4 results would carry the caveat that a different hyperparameter setting might have yielded a different outcome.

The cost scaling as GPU incompatibility evidence. The 100-generation runtime analysis (Table 3.9, Figure 3.4) provides the empirical foundation for the computational analysis in Chapter 5. Per-generation wall-clock time increases 383-fold from generation 0 to generation 89, and the cost-efficiency ratio collapses by roughly 1,300× between the first and last evaluation windows: the first four generations consume less than 0.1% of total compute while delivering 2.4 pp of accuracy; the last 50 generations consume 74.2% while delivering only 1.4 pp. This is not merely diminishing returns: it is a quantitative demonstration that NEAT's complexification pressure builds ever-larger CPPNs that increase computational cost without expanding representational power. Meanwhile, population genetic distance *increases* from 1.39 to 2.43 over 100 generations (Figure 3.5): the population diverges structurally while converging functionally, confirming that NEAT exhausts the productive topology space early and then explores genotype variations that produce no phenotypic improvement. Chapter 5 (§5.6) formalizes this scaling as a structural limitation of the quadtree substrate formation mechanism.

The structural ceiling as diagnostic instrument. The 29.0% ceiling is not a failure of the optimization methodology: it is the optimization's most important finding. By systematically eliminating parametric causes (2,013 configurations, sensitivity analysis, extended generations, multi-configuration convergence), the TPE campaign transforms the accuracy barrier from an unexplained disappointment into a diagnosed structural limitation. The per-digit analysis (Figure 3.3) makes this concrete: all ten digit classes collapse to a near-uniform output dominated by class 0, with per-digit mean cosine distances spanning only 0.69 to 0.76 (grand mean 0.73). The substrate does not fail on specific digits while succeeding on others; it fails uniformly, producing functionally identical outputs regardless of input class. This global failure pattern confirms that the limitation is substrate resolution (the quadtree cannot place nodes where they are needed), not class-specific learning difficulty. Chapter 4 identifies the cause. The convergence velocity data (Table 3.8) completes the diagnostic: the median seed plateaus at generation 24 (IQR [16, 30]), and generation 20 captures 88% of the final accuracy at only 4% of total computational cost. Together, these analyses transform the thesis narrative from "how to optimize ES-HyperNEAT" to "what happens when optimization succeeds and the algorithm still fails," motivating the architectural innovations that occupy the remainder of this thesis.

The fitness landscape behind the pruner. The pruner's effectiveness at generation 3 is itself evidence about the shape of the ES-HyperNEAT fitness surface—evidence that the paper-length treatment could not explore but that the thesis has room to interpret. The binary fitness dynamics (Figure 3.15), where trials either learn within the first three generations or never learn at all, are consistent with a *bimodal* landscape: a large, shallow

basin of unproductive configurations separated from a narrow productive region by what the generation-3 decision boundary effectively marks as a saddle point. The 4-generation onset gap between slow starters and true positives (§3.5.2) suggests this saddle is not a hard wall but a probabilistic transition zone: configurations in its neighborhood may cross into the productive basin given a few extra generations, which is precisely what the 8 false negatives do.

The population-level genetic distance data (Figure 3.5) adds a complementary perspective. That the population diverges structurally (median genetic distance rising from 1.39 to 2.43) while converging functionally (fitness plateaus by generation 20) implies that the productive basin is topologically sparse: many structurally distinct CPPNs achieve similar low fitness, so NEAT explores a wide genotype frontier without escaping the basin. If the productive region were topologically broad, one would expect structural convergence alongside functional convergence; the observed pattern indicates instead that successful topologies occupy a narrow corridor in genome space.

This interpretation is qualitative and rests on aggregate statistics rather than direct landscape measurement. Three experiments would strengthen it into a quantitative claim. First, measuring the basin of attraction by perturbing successful CPPN initializations and observing how far they can be displaced before fitness collapses would establish the productive region's geometric width. Second, comparing the genetic distance between initialization genomes of stuck and escaped trials, rather than the population-level average, would test whether the two basins are distinguishable at birth or only after the first few generations of complexification. Third, interpolating fitness between adjacent hyperparameter configurations in the TPE surrogate would reveal whether the transition between basins is smooth or discontinuous. None of these measurements was within the scope of the optimization study, but the pruner provides the experimental framework to conduct them: the generation-3 classification into productive and unproductive trials is exactly the partition a landscape study would need.

Connection to GEENNS. The binary fitness dynamics observed in this chapter (Figure 3.15), where trials either learn immediately or never learn, foreshadow GEENNS's design principle (§1.6): specialist grids are either capable or not, and the mapper routes around incapable ones rather than repairing them. The pruner's generation-3 checkpoint embodies the same principle: identify capability early and reinvest resources.

3.10 Chapter Summary

This chapter answered TRQ1 by developing and validating two complementary optimization strategies for ES-HyperNEAT's hyperparameter space. Table 3.25 summarizes the key results.

The answer to TRQ1. The hyperparameter space of ES-HyperNEAT can be navigated efficiently by combining TPE's Bayesian surrogate model for inter-trial guidance with a domain-specific early-stopping pruner for intra-trial cost reduction. TPE concentrates evaluations in productive regions of the 16-parameter search subspace, achieving a mean accuracy of 17.41% (versus 11.41% for random search, $d = 1.47$) and a peak of 27.5% across 2,013 trials. The pruner terminates unpromising trials at generation 3 based on a single fitness threshold ($T^* = 0.140$), reducing computational cost by 41.6% while

achieving statistically equivalent search performance (TOST $p < .001$). Together, the two methods compose multiplicatively: TPE selects what to evaluate, and the pruner determines how long to evaluate it.

The 29.0% ceiling. The most important finding of this chapter is not the optimization methodology itself but the performance ceiling it reveals. Despite systematically exploring over 2,000 configurations across the 16-parameter search subspace, the best MNIST accuracy achieved is 29.0% (validated peak from 30 runs of the best configuration; 27.5% during search): roughly one-quarter of the digits correctly classified. This ceiling is robust: it persists across all five optimal configurations (Table 3.5), is confirmed by the validation study (§3.2.5), and is independent of the specific hyperparameter settings.

We hypothesize that the ceiling is *structural, not parametric*: the limitation lies in ES-HyperNEAT's architecture, not in our ability to tune its parameters. Six pieces of evidence support this hypothesis, developed across this chapter (§3.9 interprets the ceiling they establish): sensitivity analysis confirms topology dominance; transferability shows spatial assumptions are encoded; extended evolution rules out temporal limitations; multi-configuration convergence shows path independence; binary fitness dynamics indicate a hard boundary; and cost scaling reveals exhausted representational capacity. The hypothesis rests, at present, on the two search families this campaign ran, random search and TPE, and extends to untested optimizer families as a hypothesis rather than a measurement.

If the structural hypothesis is correct, the most likely architectural cause is the *central bias*: ES-HyperNEAT's substrates concentrate density near the coordinate origin, a consequence of CPPN initialization geometry and quadtree variance thresholds. Chapter 4 diagnoses this directly and demonstrates that spatial decomposition (partitioned experts) overcomes the ceiling.

Transition to Chapter 4. If the performance ceiling is structural rather than parametric, then improving ES-HyperNEAT requires modifying the algorithm's architecture, not further tuning its parameters. The next chapter investigates the central bias directly by partitioning the input space into spatial regions, each served by a specialized ES-HyperNEAT instance, breaking the assumption that a single substrate topology must serve the entire input domain.

Table 3.25: Summary of Chapter 3 key results, with their significance for the broader thesis argument.

Result	Value	Section	Thesis Significance
Inter-trial optimization (TPE)			
Best MNIST accuracy (search)	27.5%	§3.2	Search peak; validated to 29.0% in §3.2.5
Best MNIST accuracy (validated)	29.0%	§3.2.5	Highest validated 20-gen accuracy for ES-HyperNEAT
100-gen extended evaluation	23.6% (gens 80–100 mean)	§3.2.6	Proves ceiling is architectural, not temporal
Multi-config convergence	7 configs → 20–24%	§3.2.8	Ceiling is independent of hyperparameter path
TPE-Best vs. random search	$d = 4.34$	§3.2	Validates TPE for all subsequent experiments
Most sensitive parameter	variance threshold	§3.3	Topology, not weights, limits performance
Fashion-MNIST transfer	$d = 2.85$	§3.4	Optimized params encode spatial assumptions
Intra-trial optimization (Pruner)			
Pruning generation G^*	3	§3.5	NEAT's topology emerges in $\leq$3 generations
Pruning threshold T^*	0.140	§3.5	Binary dynamics: learn early or never
Validation F_1	0.872	§3.5.2	Reliable early in a search; degrades once converged (§3.6.3)
Computational savings	41.6%	§3.6.1	Enables large-scale experiments in Ch. 4–6
Pruner vs. Hyperband	64% fewer gens	§3.6.2	Domain-specific rules cost less than general ones
TOST equivalence	$p < .001$	§3.6.2	Mean-equivalent quality; peaks can dip
Combined pipeline			
Cost reduction mode	41.6% savings (analytic)	§3.8	Same quality, 58% of the time
Budget-matched mode (analytic)	+72% configs	§3.8	More exploration in same budget
Architecture constraints (thesis-exclusive)			
Per-gen runtime scaling	383×	§3.2.7	Empirical basis for quadtree–GPU incompatibility (Ch. 5)
Speciation	median 1 → 5, stepwise	§C.7	Species still fragmenting after node count settles
Gen 20 accuracy/compute ratio	88%/4%	§3.2.7	Quantifies the diminishing-returns asymmetry

Chapter 4

Breaking the Central Bias

> The eye sees only what the mind is prepared to comprehend.
>
> Robertson Davies

This chapter addresses two thesis research questions:

TRQ2. Can ES-HyperNEAT's fundamental limitations be diagnosed empirically?

TRQ4. Can input decomposition extend adaptive substrates to high-dimensional problems?

Chapter 3 established a 29.0% MNIST ceiling that persists across all optimal configurations (§3.10)—a structural limitation, not a search failure. This raised the question: what in ES-HyperNEAT's architecture produces it?

Our experiments compare against the replication-validated monolithic baseline established in Chapter 3: the best configuration from the Tree-structured Parzen Estimator (TPE) campaign [11], re-evaluated there over 30 independent runs under corrected code, with a mean accuracy of 20.95% ± 2.41 and a peak of 29.0% (§3.2.5). All performance comparisons use this baseline; the $N_e = 1$ condition is this chapter's independent replication of it.

We identify the structural cause and resolve it through architectural intervention. We first diagnose *why* the performance ceiling exists (§4.1). We then propose a partitioned-experts architecture inspired by Mixture-of-Experts (MoE), which divides the input space into non-overlapping segments, each served by a specialized ES-HyperNEAT instance (§4.2), and investigate aggregation strategies across a range of expert counts (§4.3–§4.5). Receptive field analysis reveals the mechanism: partitioning breaks premature convergence and expands coverage from 3.6% to a peak of 85% of pixels (§4.6). Section 4.7 synthesizes the findings in the context of the broader thesis, connecting the partitioned experts to the GEENNS vision of specialist grids and evolved mappers, and §4.8 summarizes the chapter's contributions and provides the answers to TRQ2 and TRQ4.

The work in §4.1–§4.6 is based on Paper 3 [13]. Author contributions for all papers are detailed in the List of Publications (p. 31).

4.1 The Central Bias Problem

The sensitivity analysis in §3.3 traced this ceiling to substrate-topology parameters (variance threshold and quadtree depth) rather than to Compositional Pattern-Producing Network (CPPN) activation functions or connection probabilities. That result concerns variation *among* palette members; §2.1.9 shows that the origin concentration is a property most of them *share*, which is why substituting one for another leaves the ceiling in place. After 2,013 TPE trials and a 30-run validation of the best configuration, even the validated peak of 29.0% leaves fewer than one in three digits correctly classified.

With the ceiling firmly established, the question shifts from *whether* a bottleneck exists to *what structural mechanism produces it*. Receptive field analysis of the best-performing networks from the 30-run validation study (§3.2.5) provided the answer. The monolithic baseline's composite receptive field at $N_e = 1$ (§4.4.4) activates only 28 of the 784 available input pixels (3.6% coverage), clustered tightly around the image center. To that network, the remaining 96% of the input space is effectively invisible. This pronounced locality, which we term the *central bias*, constitutes the structural bottleneck that hyperparameter optimization alone cannot overcome.

4.1.1 Root Cause: CPPN Geometry and Quadtree Interaction

Building on the CPPN-geometry and quadtree-refinement mechanism formalized in §2.1.9, two properties of ES-HyperNEAT's optimized substrates make the bias persistent. First, the CPPN's geometric concentration operates independently of the variance threshold θ: lowering θ permits finer subdivision everywhere but does not alter the *relative* density advantage of central regions. Second, all five optimal configurations from Chapter 3 (Table 3.5) exhibit central clustering, confirming that the bias is structural rather than parametric.

4.1.2 Hypothesis

This diagnosis led to a testable prediction: if the central bias reflects a structural constraint of monolithic substrates rather than an optimal evolved strategy, then replacing the single substrate with an ensemble of spatially partitioned specialists, each confined to a distinct input region, should force evolution to discover features across the full image, breaking the central convergence and improving classification accuracy.

This hypothesis shifts the research question from parametric optimization (the focus of Chapter 3) to architectural design: instead of tuning the existing substrate formation process, we modify the structure of the input pipeline itself. The intervention is detailed in the following section.

4.2 Partitioned-Experts Architecture (MoE-inspired)

This section presents, to our knowledge, the first application of spatial partitioned-experts decomposition (inspired by Mixture-of-Experts) within indirect encoding, and connects it to the broader thesis arc by testing whether specialist subnetworks (the building blocks that GEENNS in Chapter 7 will later formalize as frozen grids) can outperform monolithic substrates on a task that no amount of hyperparameter tuning could solve (Chapter 3).

Two variants were evaluated (Independent and Shared), distinguished by whether the segments are processed by separate populations or by a single shared network.

4.2.1 Input Partitioning Strategy

Each 28×28 MNIST image is read row-by-row into a 784-element vector and divided into N_e disjoint, equal-length segments; every segment is then assigned to exactly one expert. Figure 4.1 illustrates the scheme for $N_e = 4$.

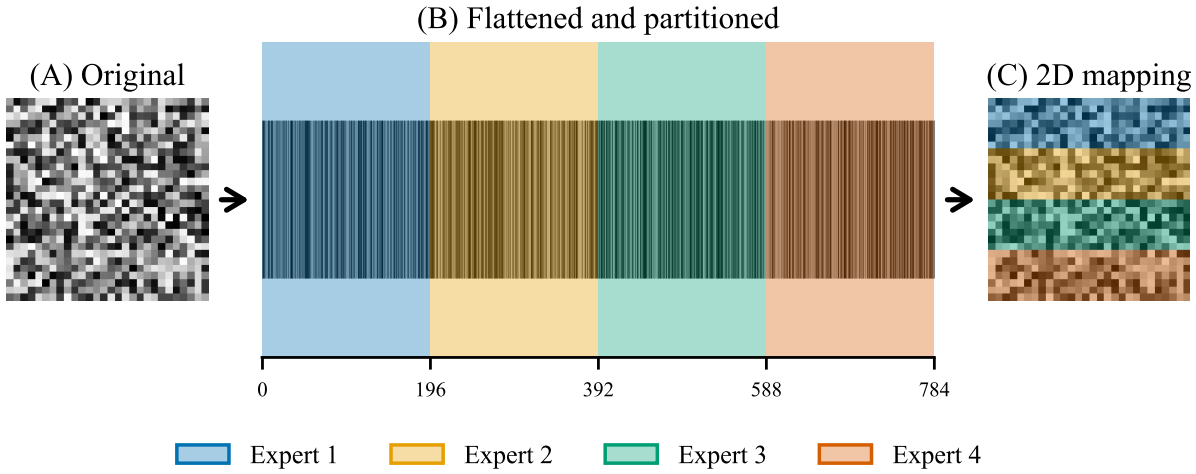


Figure 4.1: Input partitioning illustrated for $N_e = 4$. **(A)** Each 28×28 MNIST image is read row-by-row into a 784-element vector. **(B)** The vector is divided into four contiguous, equal-length segments; a barcode visualization shows the pixel intensities retained by each. **(C)** Mapping the segments back onto the 2D grid reveals that each expert receives a horizontal band: Expert 1 covers rows 0–6 (pixels 0–195, light blue), Expert 2 rows 7–13 (pixels 196–391, light orange), Expert 3 rows 14–20 (pixels 392–587, light green), Expert 4 rows 21–27 (pixels 588–783, light pink).

Expert counts range from $N_e = 1$ to $N_e = 17$. These bounds serve specific roles: $N_e = 1$ replicates the monolithic baseline from Chapter 3, while $N_e = 17$ yields segments of roughly 48 pixels, comparable in scale to the receptive field observed for the prior monolithic campaign's champion network [11]. This range tests whether forced partitioning at the scale of the observed central bias is beneficial.

4.2.2 Independent variant

In the Independent variant, N_e separate ES-HyperNEAT populations evolve in parallel, each operating exclusively on its assigned input partition. Every expert maintains its own CPPN population, substrate, and fitness evaluation: no weights, evolutionary history, or information are shared during evolution. The experts co-evolve only in the cooperative sense that their outputs are later combined for classification.

This design draws on the principles of Cooperative Co-evolutionary Algorithms [73]: the input space is partitioned along the pixel dimension, and each expert operates as an independently evolving population that specializes on its assigned region. The credit assignment challenge inherent in co-evolution is deferred to the post-evolution aggregation

stage (§4.3). In the context of this thesis, the Independent variant serves as a controlled test of the decomposition hypothesis: if independent specialists outperform the monolithic baseline, then the central bias is an architectural constraint rather than an evolutionary optimum.

4.2.3 Shared variant

In the Shared variant, a single ES-HyperNEAT population is evolved, and the resulting substrate processes each of the N_e input partitions sequentially. The same CPPN-derived network is queried N_e times, producing N_e output vectors that are then aggregated. This variant isolates the contribution of spatial decomposition from independent specialization: if the Shared variant matches the Independent variant, partitioning alone suffices; if it falls short, parallel specialization is the critical factor.

4.3 Aggregation Strategies

The aggregation step, combining expert outputs into a single prediction, is where the partitioned-experts architecture's advantage must be realized. Without effective aggregation, the specialist subnetworks produce independent outputs that may conflict, and naive combination strategies cannot resolve these conflicts as expert count grows (a pattern that foreshadows the need for evolved mappers in GEENNS, Chapter 7). Every expert e emits a raw score matrix S_e of dimension $B \times C$ (samples by classes). Fourteen methods for synthesizing these matrices into a single prediction are considered, organized into three families; the principal variants are summarized below, and Figure 4.5 compares all fourteen.

4.3.1 Simple Heuristics

The `Avg`, `Sum`, and `Max` methods perform unweighted combinations (averaging, summing, or taking the element-wise maximum across expert scores per class). These methods are computationally cheap but treat all expert opinions equally regardless of demonstrated competence.

4.3.2 Mask-Based Heuristics

The `Connection-Masked`, `Connection-Weighted`, and `Threshold-Masked` strategies apply binary or weighted masks M_e derived from structural connectivity between experts and output classes, silencing experts that lack connections to certain classes:

$$S_{\text{agg}} = \sum_{e=1}^{N_e} \big(S_e \odot M_e\big) \tag{4.1}$$

where $\odot$ denotes element-wise multiplication. The `Connection-Masked` variant uses binary masks; `Connection-Weighted` and `Threshold-Masked` incorporate connection-strength information.

4.3.3 Performance-Weighted Methods

Performance-weighted strategies compute weights $W_{e,c}$ representing expert e's competence at predicting class c, derived from per-class F_1-scores on a validation set. The core method, `Performance-Weighted`, aggregates predictions as:

$$S_{\text{agg}}[c] = \sum_{e=1}^{N_e} W_{e,c} \cdot S_e[c] \tag{4.2}$$

where $W_{e,c}$ is expert e's F_1-score for class c. Variants include `Structurally-Gated` (requiring structural connectivity), `Top-Expert` (selecting the single highest-confidence expert per sample), and `Evolved-Weights` (rescaling each $W_{e,c}$ by a per-expert scalar $w_e^{\text{exp}} \in [0, 1]$; the scalar is fixed manually in this chapter, with evolutionary optimization left to future work).

The 43.07% mean and 50% maximum accuracy (the maximum across control variants) reported in this chapter were achieved with `Performance-Weighted`, demonstrating that data-driven aggregation (weighting each expert's contribution by empirically measured competence) is essential for translating expert specialization into classification performance.

From the perspective of the broader thesis, these performance-weighted strategies function as *proto-mappers*: they route and compose specialist outputs based on measured competency, foreshadowing the evolved mapper design central to the GEENNS architecture (§4.7). The key difference is that `Performance-Weighted` weights are computed from held-out validation data after evolution, whereas GEENNS mappers would evolve alongside the specialist grids.

4.4 Experimental Design

4.4.1 Control Factors

Parallelized evolution and input partitioning introduce potential confounds that must be controlled. Three methodological control switches isolate the architectural effects of the partitioned-experts framework; conditions labeled *Default* in what follows enable none of them, so every individual is evaluated on its own freshly drawn random batch of 200 MNIST images:

Same-Batch-per-Generation (SBG).
All individuals in a generation see an identical, once-drawn batch, removing inter-individual fitness variance from random data sampling.

Same-Batch-per-Expert (SBE).
Applicable only to the Independent variant, SBE extends SBG so that every expert (across all individuals and parallel processes) receives the same data batch. Enabling SBE automatically enforces SBG.

Shared-Mix (SM).
Used exclusively with the Shared variant, SM decorrelates inputs by independently shuffling batch indices for each expert slot, preventing co-adaptation between segments.

Table 4.1 summarizes all experimental conditions.

Table 4.1: Summary of all experimental conditions and their control factors. Each condition was run for 30 independent replicates.

Experimental Run	Architecture	Evaluation Controls	Replicates
Independent variant			
Default	Independent	None	30
+ Same-Batch-per-Generation	Independent	Same-Batch-per-Generation	30
+ Same-Batch-per-Expert	Independent	Same-Batch-per-Expert (incl. Same-Batch-per-Generation)	30
Shared variant			
Default	Shared	None	30
+ Same-Batch-per-Generation	Shared	Same-Batch-per-Generation	30
+ Shared-Mix	Shared	Shared-Mix	30
+ SBG + SM	Shared	Same-Batch-per-Generation, Shared-Mix	30

4.4.2 Hyperparameter Configuration

A deliberate design choice was to freeze all ES-HyperNEAT hyperparameters at the optimal configuration identified by the Chapter 3 TPE campaign (Table 3.5). The NEAT parameters were: population size 100, no initial hidden nodes, connection addition and deletion probabilities of 0.5, node addition probability of 0.8 and deletion probability of 0.2, full direct initial connections with 80% connection fraction, and tanh activation. The substrate parameters were: initial depth 2, maximum depth 5, variance threshold 0.01, division threshold 0.5, maximum weight 3.0, iteration level 0, and tanh activation. Unspecified parameters used NEAT-Python defaults.

This configuration was optimized for the monolithic architecture processing all 784 pixels simultaneously. Applying the same unmodified configuration to the partitioned-experts models provides a conservative test: any performance gains must be attributable to the architectural change rather than to hyperparameter re-tuning. As §4.7 discusses, per-expert hyperparameter optimization would likely improve results further, making the 43.07% figure a lower bound on partitioned-experts performance.

4.4.3 Evaluation Protocol

Each condition was evaluated across the full range of expert counts ($N_e = 1$ to $N_e = 17$) with 30 independent seeds per configuration, running 20 generations per trial on MNIST. Fitness equals the fraction of correctly classified images in a 200-image batch stratified across 10 digit classes. The implementation uses the PUREPLES framework [103], consistent with Chapter 3.

4.4.4 Receptive Field Analysis Methodology

To test whether the partitioned-experts architecture changes how networks process the input space, we propose a receptive field tracking methodology. At the end of each generation, the best-performing individual's phenotype is inspected to determine its *active receptive field*: the set of input pixels that have at least one non-zero-weight connection to a hidden or output neuron. This metric enables both quantitative measurement of re-

ceptive field size across evolutionary time and qualitative visualization of spatial coverage patterns.

The monolithic baseline's active pixel count varies across independent trials, with a per-run mean of approximately 55 pixels across the 30-run replication. For narrative consistency we use two fixed anchors: one from this distribution, one from prior work. *Coverage* refers to the $N_e = 1$ point of the composite visualization (Figure 4.11), reported as 28 pixels (3.6% of 784); it is the convention used in §4.1 and §4.6 when the point is the spatial extent of the central focus. *Density* refers to the ∼48-pixel receptive field of the prior campaign's champion network [11] (6.1% of 784); it is the convention used in §4.6.4 as the per-trial anchor for per-expert density comparisons. This chapter's own replication does not converge to a single density (final-generation counts have median ∼29 and range 1–465), a spread that itself illustrates the premature-convergence variability analyzed below. Both anchors summarize the same sparse central focus, one as this chapter's spatial extent, the other as the prior campaign's per-trial wiring density.

4.5 Results

The central finding is decisive: the Independent partitioned-experts architecture produces a qualitative shift in ES-HyperNEAT performance on MNIST, while the Shared variant achieves only modest gains. Where Chapter 3 exhausted the parametric design space, this architectural intervention lifts the ceiling entirely. The mechanism, detailed in §4.6, is that forced spatial partitioning prevents the convergence trap, expanding pixel coverage from 3.6% to a peak of 85% and converting the central bias from a liability into a per-partition asset.

4.5.1 Performance Comparison

Table 4.2 summarizes the high-level performance across the three architectural variants.

Table 4.2: Summary of mean performance (± std. dev.) over 30 runs with the `Performance-Weighted` aggregation method, where N_e is the number of experts.

Architecture	**Mean Accuracy (%)**	**Optimal N_e**
Independent variant (Default)	**43.07 ± 2.45**	**13**
Monolithic (Baseline)	20.95 ± 2.41	1
Shared variant (Best config)	26.03 ± 2.06	8

The default Independent variant with $N_e = 13$ achieves 43.07% mean accuracy, a 105% relative improvement over the 20.95% monolithic baseline established in Chapter 3. This result answers TRQ4 affirmatively: structured input decomposition does improve evolved substrate performance, and by a margin that no parametric tuning can match. Architecture choice dominates performance: a two-way ANOVA yields $F(1, 3536) = 11{,}915.88$ ($p < .001$), and the Independent variant surpasses the Shared variant and monolithic baseline in every experimental condition.

The Shared variant peaks at 26.03% ($N_e = 8$), a 24% improvement over the monolithic model but substantially below the Independent variant. Sequential processing of disjoint segments through a single substrate creates a representational bottleneck: the shared

network cannot maintain distinct feature representations for spatially separated regions. This result establishes that spatial decomposition alone is necessary but not sufficient—independent, parallel specialization is the critical factor.

An important validation result concerns the $N_e = 1$ condition, which is architecturally identical to the Chapter 3 baseline. No significant performance difference was found ($t = -0.68$, $p = .502$, $|d| = 0.17$), consistent with the partitioned-experts framework reproducing the monolithic baseline.

The Independent variant's 22-percentage-point gain over the monolithic baseline is attributable to the coverage expansion documented in §4.6, not to changes in evolutionary search parameters. The fixed partitioning here establishes a lower bound on what evolved mapper-based partitioning in GEENNS (Chapter 7) could achieve.

Three patterns emerge from Table 4.3 that warrant emphasis in the thesis context. First, the Independent variant is *consistently* superior: its worst-performing Default run (38.00%), and indeed its worst run in any condition of this table (35.00%), substantially exceeds the best monolithic run (29.00%), meaning the distributions at each condition's optimal N_e do not overlap. Second, the performance ceiling rises from 29.00% to 50.00%, demonstrating headroom that the monolithic architecture cannot access. Third, the tight alignment between mean and median (43.07% versus 42.75%, SD = 2.45%) confirms that the improvement is consistent across seeds, not driven by outliers.

To isolate architectural contributions from aggregation effects, Table 4.4 repeats the analysis with naive `Avg` aggregation.

Table 4.3: Detailed performance summary for the optimal configuration of each condition using `Performance-Weighted` aggregation (30 runs). The worst *Default* Independent run (38.00%) exceeds the best monolithic run (29.00%); no Independent run in any condition of this table falls below 35.00%.

Method	Mean ± SD (%)	Median ± MAD (%)	Best / Worst
Independent variant			
Default	**43.07 ± 2.45**	**42.75 ± 1.75**	49.00 / **38.00**
Same-Batch-per-Generation	41.23 ± 3.65	41.25 ± 1.75	**50.00** / 35.00
Same-Batch-per-Expert	41.90 ± 3.26	41.25 ± 1.75	**50.00** / 35.50
Shared variant			
Default	26.03 ± 2.06	25.50 ± 1.25	30.50 / 22.00
Same-Batch-per-Generation	23.88 ± 1.50	23.75 ± 1.00	27.00 / 21.00
Shared-Mix	16.75 ± 2.20	17.00 ± 2.00	21.00 / 12.50
SBG + SM	19.22 ± 2.50	19.50 ± 1.75	25.50 / 15.00
Baseline			
Monolithic	20.95 ± 2.41	20.75 ± 1.25	29.00 / 17.00

Table 4.4: Performance with naive `Avg` aggregation. The SBG Independent variant (35.60%) still outperforms the monolithic baseline (20.95%), so partitioning's benefit is independent of aggregation choice.

Method	Mean ± SD (%)	Median ± MAD (%)	Best / Worst
Independent variant			
Default	32.93 ± 3.29	32.50 ± 2.50	39.00 / 27.00
Same-Batch-per-Generation	**35.60 ± 3.31**	**36.00 ± 2.00**	**42.00 / 29.00**
Same-Batch-per-Expert	29.60 ± 2.24	29.75 ± 1.25	33.50 / 24.50
Shared variant			
Default	22.52 ± 1.03	22.50 ± 0.50	24.00 / 20.00
Same-Batch-per-Generation	23.15 ± 2.57	23.25 ± 1.75	28.50 / 19.00
Shared-Mix	13.93 ± 2.64	13.75 ± 1.75	19.00 / 9.00
SBG + SM	13.27 ± 1.84	13.50 ± 1.00	16.50 / 8.50
Baseline			
Monolithic	20.95 ± 2.41	20.75 ± 1.25	29.00 / 17.00

4.5.2 Expert Granularity

The performance improvement is not uniform across expert counts. Figure 4.2 plots mean test accuracy as a function of N_e, revealing the contrasting scaling behaviors of the two architectures.

Beyond $N_e = 5$, the Independent variant pulls decisively ahead of the baseline, and the gap widens with finer partitioning. This architecture×granularity interaction is highly significant ($F(16, 3536) = 114.43$, $p < .001$): accuracy scales with partition count for independent experts but remains flat for the shared variant.

The optimum occurs at $N_e = 13$: mean accuracy 43.07% (SD = 2.45%), confirmed as the unique peak by a one-way ANOVA ($F(16, 493) = 230.78$, $p < .001$) followed by Tukey HSD post hoc comparisons showing significant differences against every other granularity.

The $N_e = 13$ optimum has a natural interpretation. At this granularity, each expert's input partition contains approximately $784/13 \approx 60$ pixels, close to the ∼48 active pixels of the prior campaign's champion network (§4.6.4), the receptive field it discovered when unconstrained. This suggests that the champion network's 48-pixel central focus reflects not a limit on how many pixels evolution can wire, but a ceiling on how many input dimensions the evolutionary loop can exploit *informatively* under the 20-generation budget, roughly 50–60. The distinction matters: at $N_e = 3$, individual experts frequently wire over 250 pixels within their segments (§4.6.4), so connection count is not the binding constraint; informative use of those connections is. Partitioning to segments of comparable size lets each expert place nearly all of its exploitable capacity on a different region, collectively covering the full image.

By contrast, the Shared variant never meaningfully exceeds the 20.95% baseline, confirming that spatial decomposition alone is insufficient: the processing mode (independent parallel specialists versus a shared sequential processor) determines whether partitioning translates into accuracy gains.

Figure 4.2 reports aggregated ensemble accuracy; the per-expert fitness distributions (Figure 4.3) reveal how individual specialist performance scales with partition count. While ensemble accuracy climbs with N_e toward its N_e=13 peak, individual experts remain near the monolithic baseline regardless of partition count, confirming that the

ensemble gain stems from spatial coverage rather than per-expert improvement.

Figure 4.4 traces the mean per-expert fitness over 20 generations. Approximately half of the per-expert improvement occurs by generation 5 (mean 53%, range 48–67% across panels); the remaining gain accrues gradually across the rest of the run, with the three variants converging to similar final fitness values at each expert count.

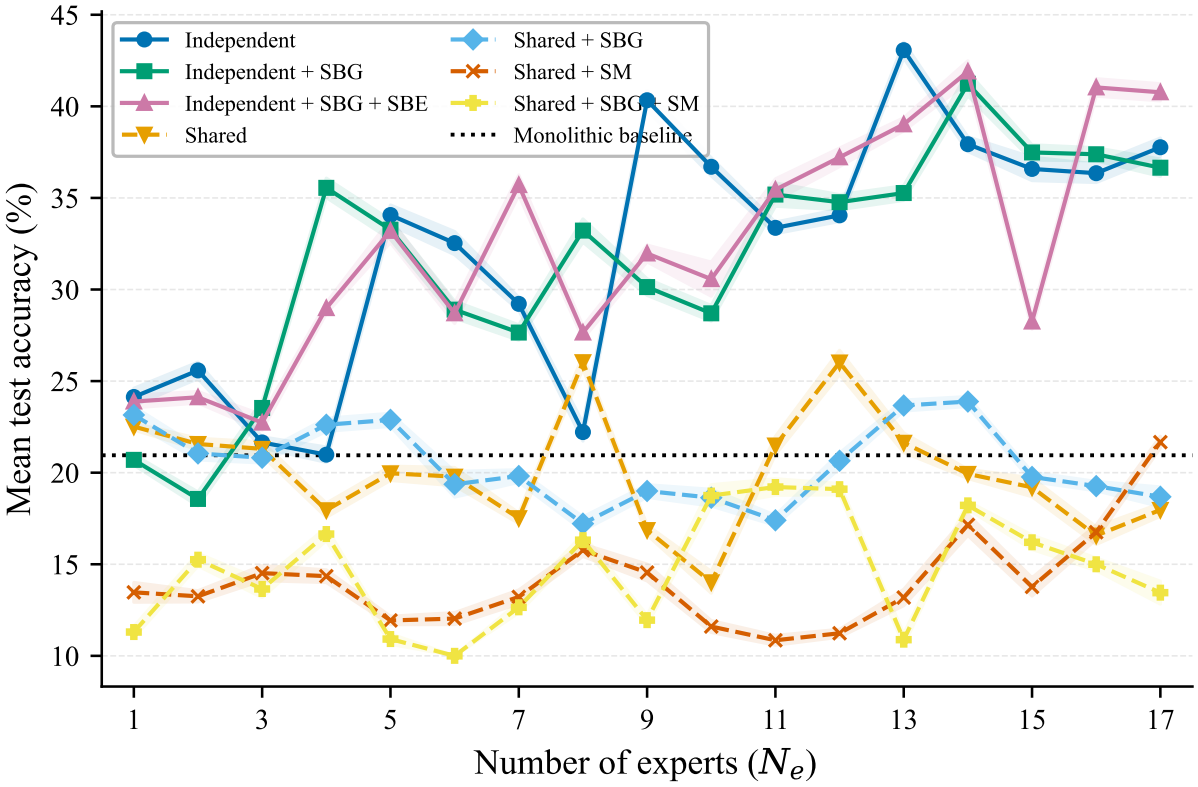


Figure 4.2: Mean test accuracy as a function of N_e. Independent variants (blue) show a significant upward trend, peaking at $N_e = 13$ (43.07%). Shared variants (orange/yellow) remain near or below the monolithic baseline (black dotted line).

4.5.3 Statistical Significance

Table 4.5 reports pairwise Welch's t-tests for the key comparisons, with Cohen's d as the effect size measure.

Table 4.5: Key statistical comparisons. The $N_e = 1$ replication shows no significant difference from the original monolithic baseline ($p = .502$). All architectural and aggregation comparisons yield very large effect sizes.

Comparison	**t-statistic**	**p-value**	**Cohen's d**
$N_e = 1$ vs. Monolithic baseline	−0.68	.502	−0.17
$N_e = 13$ vs. $N_e = 1$	33.69	$< .001$	8.70
Independent best vs. Shared best	29.19	$< .001$	7.54
`Performance-Weighted` vs. `Avg` ($N_e = 13$)	28.20	$< .001$	7.28

The effect sizes merit explicit comment. The partitioning benefit ($N_e = 13$ vs. $N_e = 1$, $d = 8.70$) is not a marginal improvement but a regime change: the performance distributions are completely separated. The architectural superiority of Independent over Shared variants ($d = 7.54$) confirms that parallel specialization, not just spatial decomposition, drives the gains. The aggregation comparison ($d = 7.28$) establishes that competency weighting contributes a further, separable gain on top of partitioning (§4.5.4). In the context of the thesis, these are among the largest effect sizes reported across all experimental chapters.

4.5.4 Aggregation Comparison

Figure 4.5 compares all 14 aggregation methods across the seven experimental conditions at the accuracy optimum $N_e = 13$.

A one-way ANOVA confirms that aggregation strategy has a highly significant effect on accuracy ($F(13, 406) = 798.06$, $p < .001$). The 14 strategies stratify into three tiers:

1. **F_1-weighted strategies** (`Performance-Weighted`, `Top-Expert`, `Evolved-Weights`) consistently outperform the baseline and improve with granularity, peaking at $N_e = 10$–13. Their success stems from using each expert's measured per-class competence.

2. **Mask-based methods** provide moderate improvement by silencing structurally disconnected expert–class pairs, but cannot capture quantitative differences in expert reliability.

3. **Simple heuristics** (`Avg`, `Sum`, `Max`) slightly exceed the baseline at low N_e but show erratic behavior as expert count grows. As the number of specialists increases, so does prediction conflict, and naive heuristics cannot resolve it.

At $N_e = 13$, the gap between the best and simplest aggregation spans 17.69 percentage points (pp): `Performance-Weighted` achieves 43.07% while `Avg` reaches only 25.38% ($t = 28.20$, $p < .001$, $d = 7.28$). The failure of naive heuristics at high N_e foreshadows a challenge central to the GEENNS architecture (§4.7): as the number of specialist grids grows, evolved aggregation (mappers) becomes essential, because static combination cannot resolve conflicting specialist outputs.

However, even with naive `Avg` aggregation, the best Independent variant reaches 35.60% (Table 4.4), a 70% improvement over the monolithic baseline. This disentangles the two contributions: partitioning alone provides a 70% relative gain even under naive averaging, and competency-weighted aggregation adds another 7.5 pp on top (43.07% vs. 35.60%, each lever measured at its best condition). Both the architectural shift and the aggregation mechanism contribute independently to the accuracy improvement.

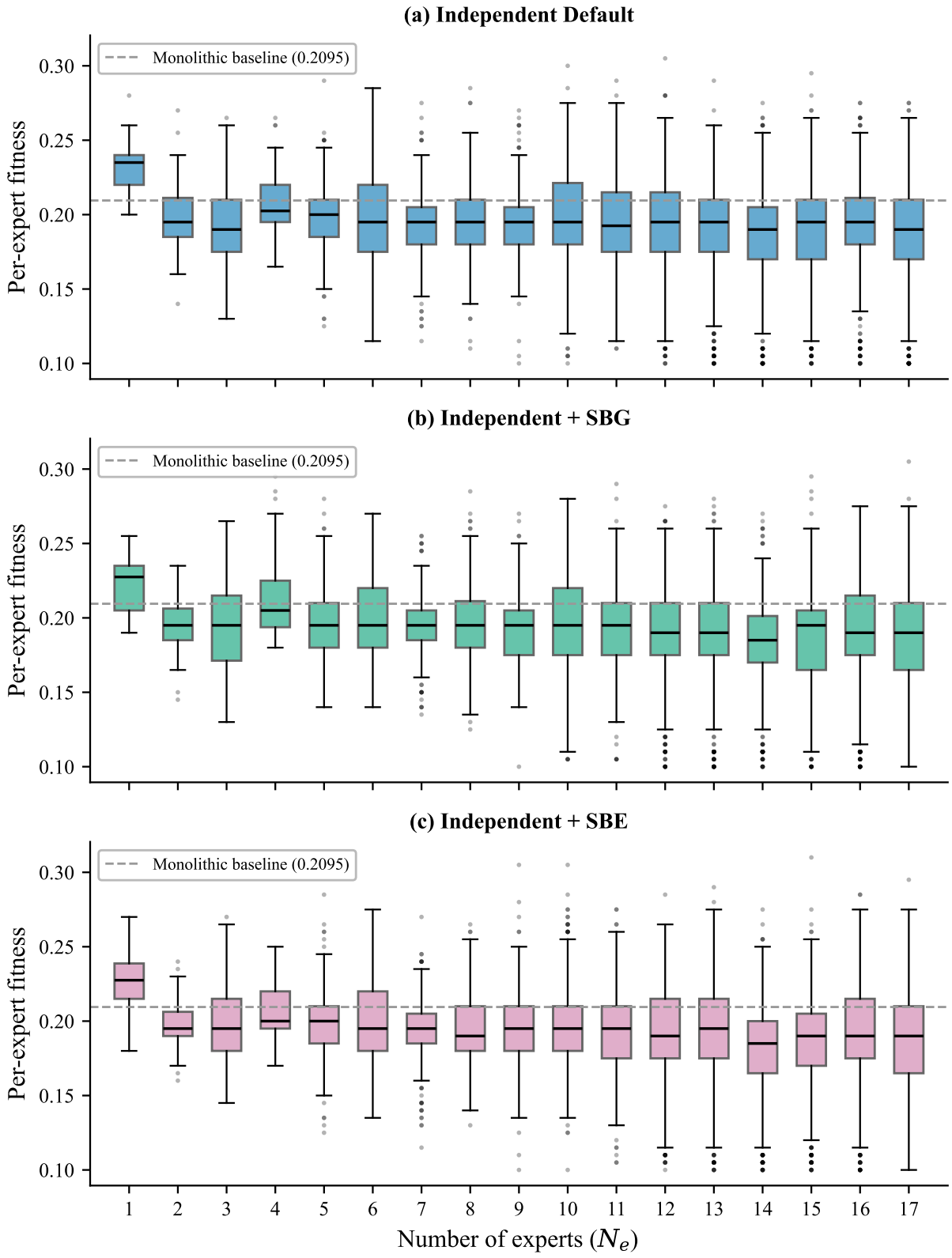


Figure 4.3: Per-expert final best fitness distributions across $N_e = 1$–17 for each Independent variant. Each box aggregates $N_e \times 30$ individual expert observations (e.g., $N_e = 13$ yields $13 \times 30 = 390$ data points). The dashed line marks the monolithic baseline (0.2095). Individual expert fitness remains near or slightly below the baseline across all partition counts, indicating that ensemble accuracy gains arise from spatial complementarity rather than per-expert improvement.

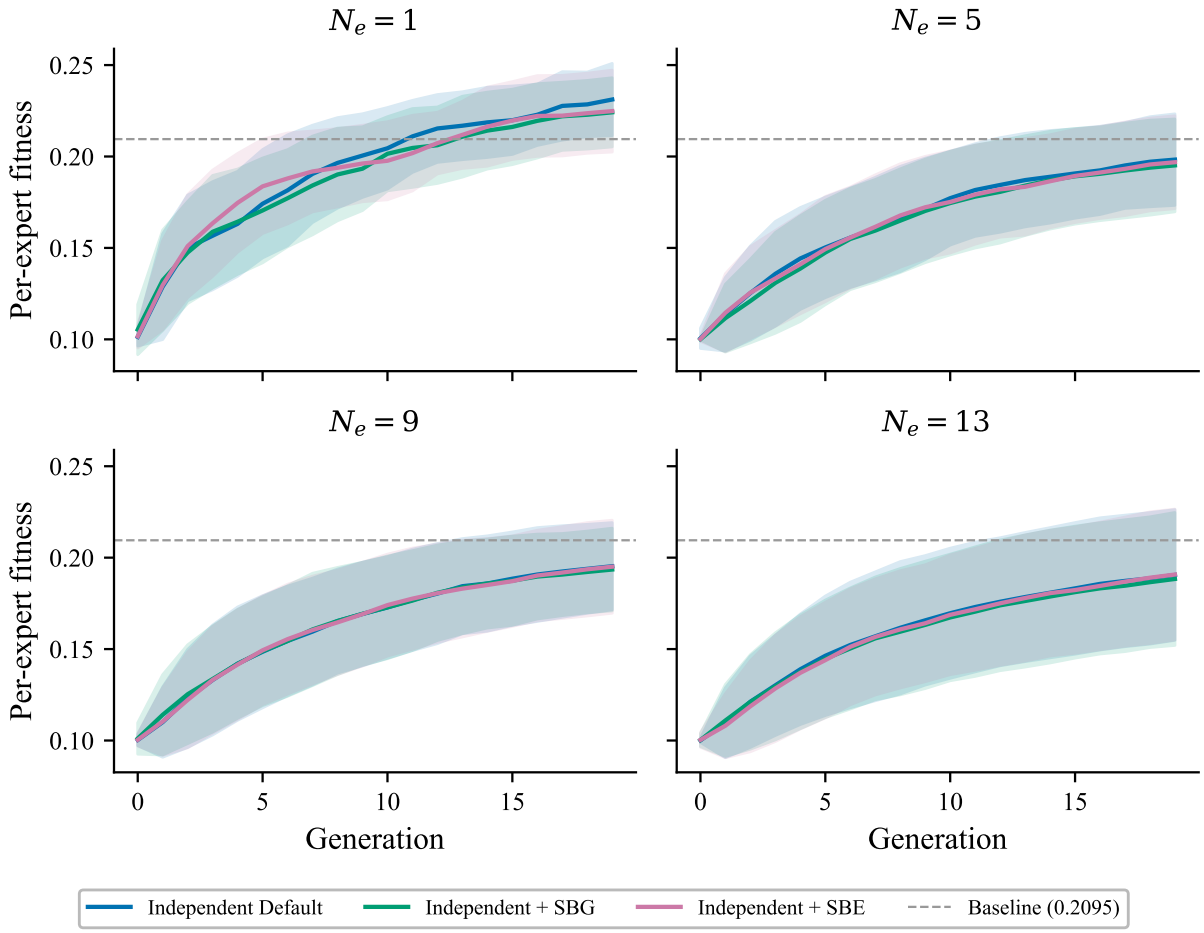


Figure 4.4: Mean per-expert fitness ± 1 s.d. over 20 generations for $N_e \in \{1, 5, 9, 13\}$. Shaded regions denote one standard deviation across all experts and trials. All three methods follow nearly identical convergence trajectories, with the majority of improvement occurring in the first five generations. The dashed line marks the monolithic baseline (0.2095).

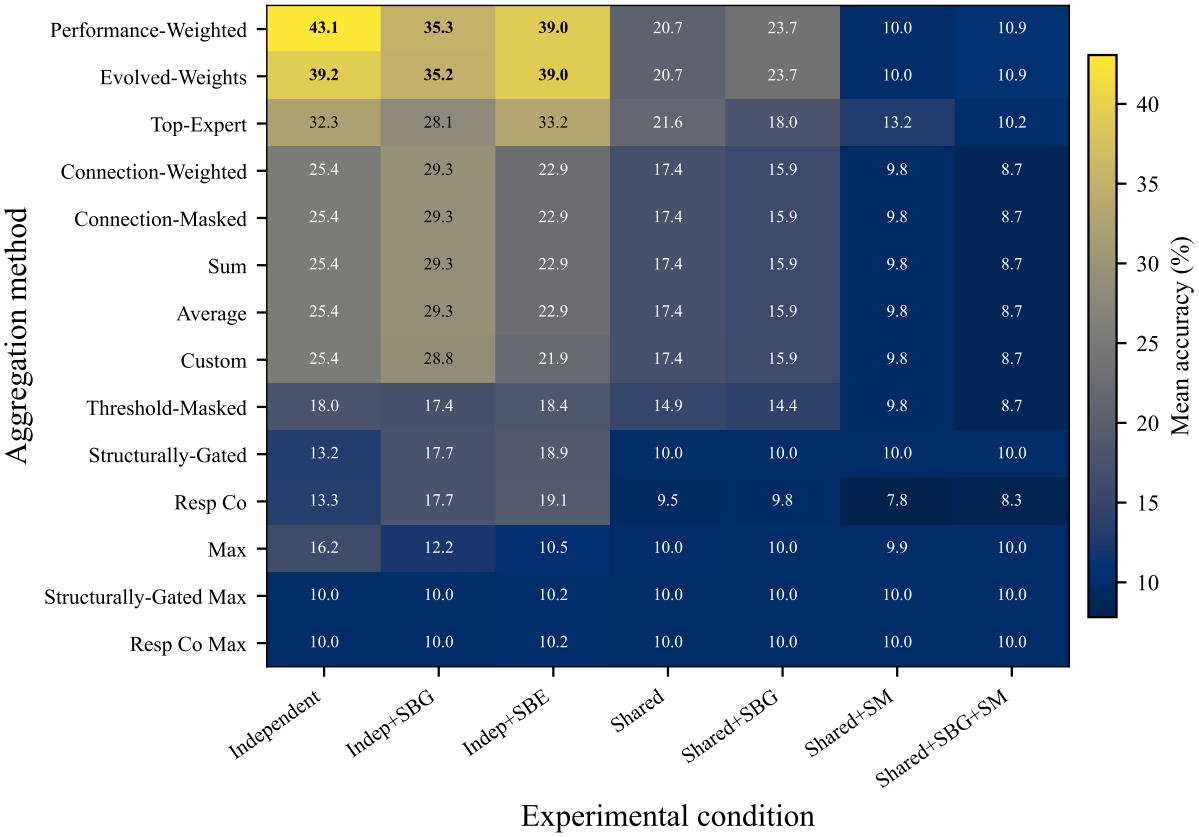


Figure 4.5: Impact of aggregation strategy on performance. Three F_1-weighted strategies (in color) outperform the remaining eleven methods (in gray) across the experimental conditions; cells are mean accuracy at $N_e = 13$.

4.5.5 Per-Class Prediction Analysis

To understand what 43% accuracy represents mechanistically, we decompose the prediction behavior by digit class across three analyses. For each of the 200 test samples per generation, every genome produces a single predicted digit class; the *prediction distribution* records how many of those 200 predictions are assigned to each class 0–9.

Prediction distribution. Figure 4.6 compares the fraction of 200 test-sample predictions assigned to each digit class. The monolithic substrate predicts a mean of 3.7 distinct classes at the final generation (median 3, range 2–9 across the top 30 trials), with the top three classes (0, 1, and 9) accounting for 55.7% of all predictions. Class 0 alone dominates at 30.5%, consistent with a substrate that has learned to distinguish a single geometrically simple digit (a closed oval) from the rest. The partitioned experts individually predict 4.4 classes on average (median 4, range 2–9), and their collective union covers all ten digit classes, compared to the monolithic's coverage of all ten only when aggregated across 30 independent trials.

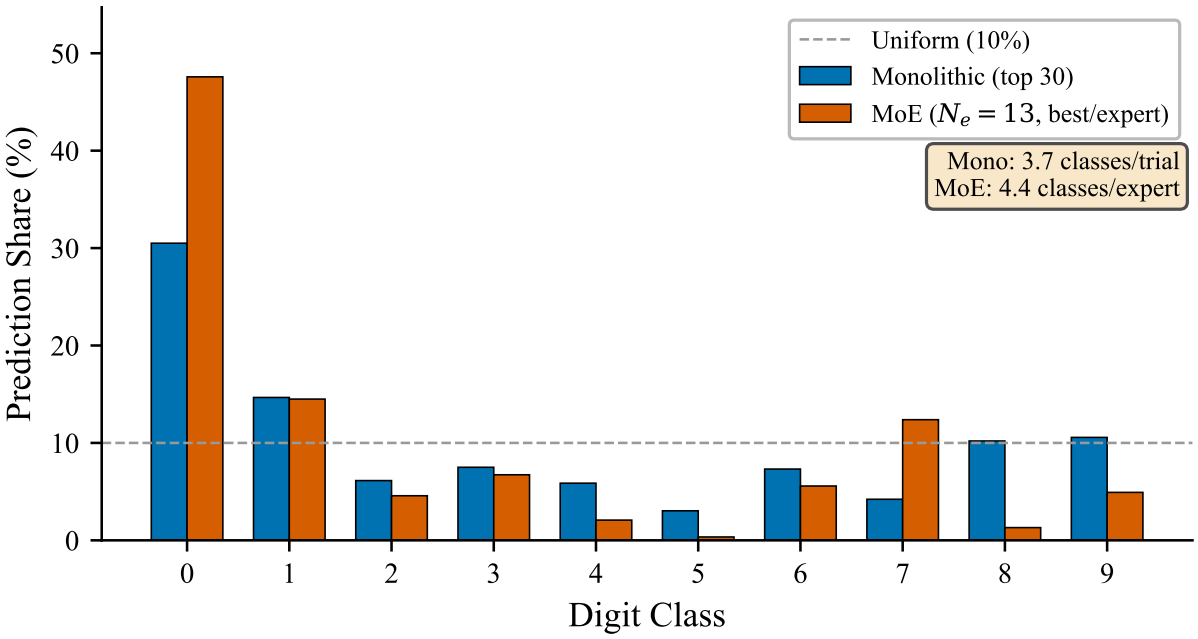


Figure 4.6: Prediction distribution by digit class, aggregated over the top 30 monolithic trials and the best trial per partitioned expert ($N_e = 13$). Both architectures concentrate predictions on class 0, but the partitioned-experts architecture distributes predictions more broadly across classes 1, 7, and 6. Dashed line marks the uniform baseline (10%). Text box reports the mean number of distinct classes per trial/expert.

Prediction diversity evolution. Figure 4.7 reveals a qualitative difference in how class differentiation develops. Monolithic networks rise from 1.4 classes at generation 0 to approximately 3.4 classes by generation 1, then fluctuate within a 3.3–3.8 band for the remaining run, gaining only +2.2 classes across the full evolutionary run. The early plateau reflects the central bias: once a small cluster of central pixels provides sufficient fitness signal for a few easily separable classes, evolution has no pressure to expand coverage. Partitioned experts start at 1.0 class (all predictions initially assigned to class 0) and

expand by approximately +3.0 classes, reaching 4.0 by the final generation with a peak of 4.1 at generation 18 (pooling all 390 expert-trials rather than the best per expert); the trajectory through the peak is upward, suggesting further improvement would occur with additional generations.

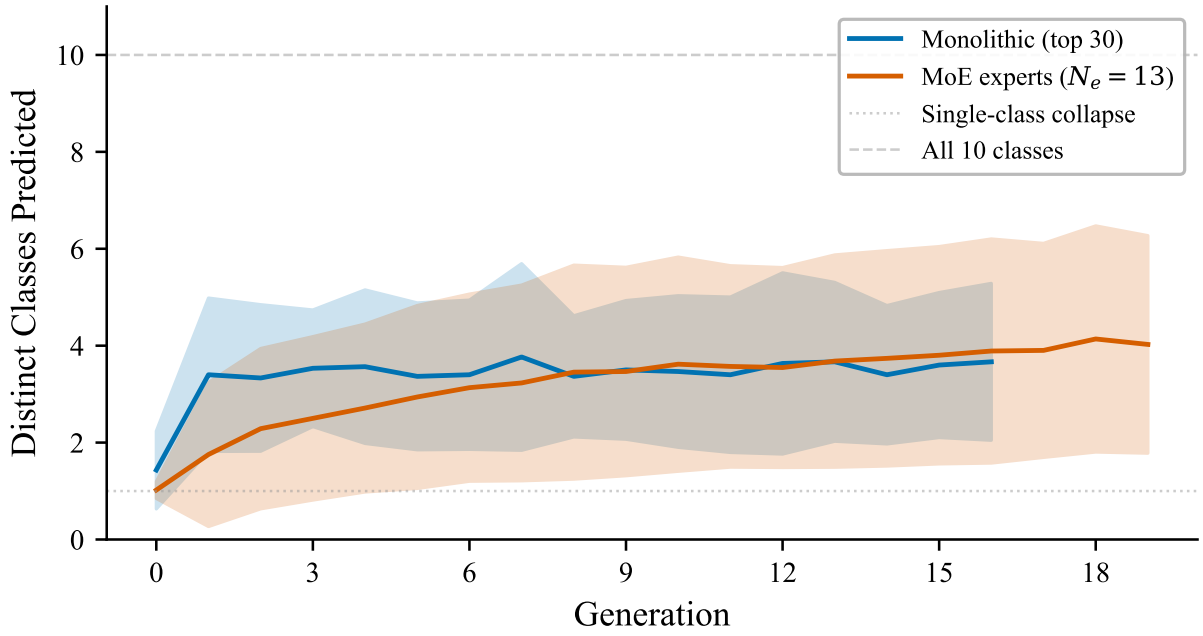


Figure 4.7: Number of distinct digit classes predicted over generations (mean ± 1 s.d.). Monolithic networks (top 30 trials) reach approximately 3.4 classes by generation 1 and then fluctuate within a 3.3–3.8 band; partitioned experts ($N_e = 13$, all 390 expert-trials pooled) start at 1 class and reach ∼4.0 by the final generation (peak ∼4.1 at generation 18).

Expert specialization. Figure 4.8 reveals that different spatial partitions develop distinct class preferences. Expert 7 (E7) allocates 62% of its predictions to class 9, Expert 3 (E3) allocates 52% to class 7, and Expert 5 (E5) concentrates 52% on class 1. Expert 6 (E6), which achieves the highest individual fitness (0.280), distributes predictions across 9 classes with no single class exceeding 38%. The peripheral experts (Expert 0 and Expert 12) predict only 2 classes, allocating 90–98% to class 0, reflecting the sparser discriminative information available at image edges.

This class-level decomposition confirms that 43% accuracy is not random guessing. The substrate develops spatially grounded, class-specific prediction strategies, a necessary precondition for the compositional aggregation that the `Performance-Weighted` method exploits when combining expert outputs.

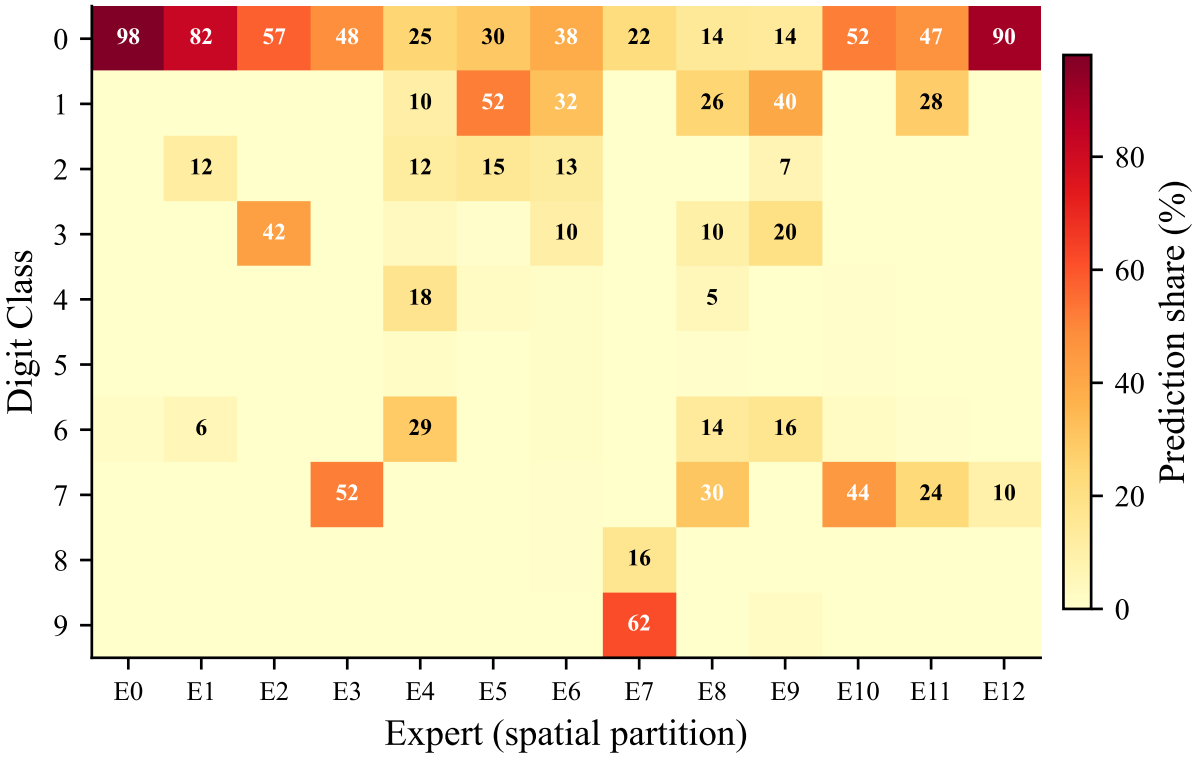


Figure 4.8: Per-expert prediction specialization heatmap ($N_e = 13$, best trial per expert, final generation). Each column is one spatial partition; cell values show the percentage of 200 test-sample predictions assigned to each digit class. Different experts develop distinct class preferences: E7 allocates 62% of predictions to class 9, E3 allocates 52% to class 7, while E6 distributes across 9 classes.

4.5.6 Control Factor Effects

An unexpected result concerns the experimental controls. Table 4.3 shows that the default Independent variant (43.07%) outperforms the Same-Batch-per-Generation variant (41.23%), a statistically significant difference ($t = 2.29$, $p = .026$, $d = 0.59$), though the difference with Same-Batch-per-Expert (41.90%) is not significant ($p = .122$).

The explanation is counterintuitive but has a direct analogue in gradient-based optimization, where minibatch sampling noise is understood to aid escape from sharp minima: stochastic batch sampling during fitness evaluation may *benefit* evolutionary search. When each individual encounters a different random batch, the resulting fitness landscape noise helps the population escape local optima and selects implicitly for generalization, since individuals must perform well across variable data samples rather than memorizing features of a fixed batch. While the control factors provide methodological rigor for fair comparison, the default stochastic evaluation maximizes performance in practice.

This stochastic-beats-deterministic finding suggests that noise and diversity in the search process benefit neuroevolution, a pattern that may extend to other levels of the architecture, such as activation function selection, which future work on per-node function evolution could investigate (§8.4).

4.6 Mechanistic Analysis: Receptive Field Evolution

The preceding sections established *that* the Independent variant improves performance; this section explains *why*. Receptive field analysis provides the causal link between the architectural intervention and the accuracy gain, closing the diagnostic loop that Chapter 3 opened and this chapter's central bias diagnosis (§4.1) sharpened.

4.6.1 Coverage Expansion

Figure 4.9 compares the receptive fields across three architectural conditions.

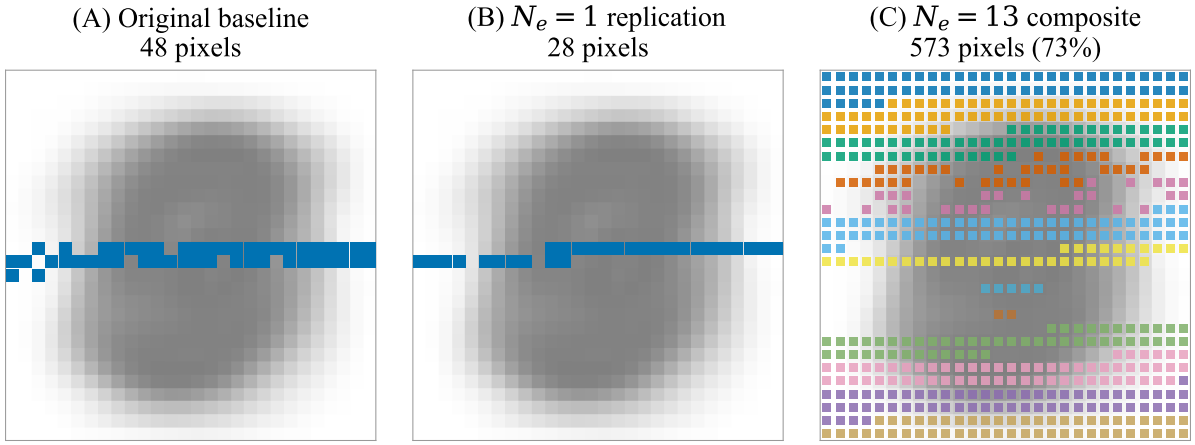


Figure 4.9: Direct comparison of active receptive fields. The background shows the per-pixel MNIST standard-deviation heatmap (darker regions indicate higher variance). **(A)** The prior campaign's champion network (Chapter 3), showing a sparse central focus. **(B)** The median run of this chapter's $N_e = 1$ replication, which reproduces the same centrally biased behavior (accuracy-equivalence $t = -0.68$, $p = .502$). **(C)** The composite field of the 13-expert partitioned-experts ensemble (colored squares), achieving near-complete coverage of the input space.

Panel A confirms the central bias diagnosed in §4.1: the original monolithic substrate confines its active pixels to a sparse, centrally located cluster. Panel B validates our experimental framework by independently reproducing this behavior. Panel C reveals the architectural intervention's effect: the composite receptive field of the 13-expert ensemble covers nearly the entire input space, with each color representing a single expert's active pixels. The ensemble forces evolution to discover features across the full image instead of converging on the minimal central solution.

4.6.2 Evolution Dynamics Within a Single Trial

Before examining the aggregate coverage statistics, it is instructive to observe how receptive fields evolve *within* a single partitioned-experts trial. Figure 4.10 shows per-generation receptive field snapshots for a representative $N_e = 3$ trial, illustrating the four-step mechanism described in §4.6.5.

Figure 4.10: Per-generation receptive field evolution for a representative $N_e = 3$ trial (trial 1134). The background shows the per-pixel MNIST variance heatmap for spatial context. Each row shows one expert's active input pixels across evolutionary generations. All three experts start with sparse, centrally biased connectivity within their assigned partition and progressively fill their segment as evolution discovers useful peripheral features. This temporal expansion directly illustrates how partitioning converts the CPPN's geometric bias from a global liability into a per-partition asset: the bias toward the origin now operates *within* each segment, seeding initial connectivity that evolution then extends. By generation 11 each expert's receptive field has largely saturated its partition; the visual similarity between generations 11 and 19 reflects the shift from spatial expansion to weight refinement once coverage is achieved.

The dynamics visible in Figure 4.10 are consistent across trials: early generations show sparse central connectivity within each partition, followed by progressive expansion as NEAT's complexification adds connections to peripheral pixels. By the final generation, each expert achieves dense coverage of its assigned region, a pattern not observed in any of the 30 monolithic baseline runs, where the central bias traps evolution in a globally suboptimal configuration. The near-identical receptive fields at generations 11 and 19 confirm that spatial expansion saturates early: once the partition is covered, subsequent generations refine connection weights rather than discovering new input pixels.

4.6.3 Receptive Field Evolution Across Expert Counts

Figure 4.11 visualizes how the composite receptive field expands as expert count increases.

The coverage expansion is substantial. At $N_e = 1$, the monolithic substrate uses only 28 pixels (3.6%). Even minimal partitioning disrupts the central convergence: by $N_e = 3$, coverage jumps to roughly 74%. Coverage continues to rise non-monotonically and reaches tied peaks of 85% at $N_e = 8$ (98-pixel segments) and $N_e = 14$ (56-pixel segments), bracketing the highest-accuracy model ($N_e = 13$, 60-pixel segments) at 73%. These two coverage peaks sit within and above the search-capacity range identified in the granularity analysis (§4.5.2): the smaller-segment configuration sits within the capacity range, the larger-segment configuration above it. The 3.6% $\rightarrow$ 85% trajectory demonstrates that the central bias is not an inherent limitation of evolutionary search: it is an artifact of the monolithic architecture's search-space overwhelm.

The quadtree geometry amplifies this effect: as Chapter 5 will show, the quadtree's top-down traversal concentrates CPPN queries near the coordinate origin, reinforcing the same spatial bias that the CPPN activation functions already impose.

The coverage expansion is not monotonic. The largest jump is immediate: $N_e = 2$ already lifts coverage from 3.6% to 57%, and $N_e = 3$ to 74%. Beyond that the series fluctuates without tracking segment size ($N_e = 4$ falls to 50% despite 196-pixel segments, below $N_e = 2$'s 392-pixel segments), so segment size alone does not order coverage. Part of the fluctuation also reflects the figure's mixed panel-selection rules (Figure 4.11): the $N_e = 3$ panel unions each expert's champion trial while the $N_e = 2$ and $N_e = 4$ panels show single complete runs, so point-to-point changes across that boundary are not like-for-like; the $N_e = 1$ panel is in turn a median rather than a best-run selection, so the 3.6% $\rightarrow$ 85% span likewise compares across rules. What the series does show is that any partitioning at all breaks the monolithic central convergence, while the specific coverage attained depends on where each segment boundary falls relative to the digit's informative rows.

4.6.4 Receptive Field Density

Using the density convention defined in §4.4.4, the density anchor is approximately 48 active pixels out of 784 (6.1% utilization), carried over from the prior campaign's champion network; this chapter's own replication straddles rather than converges to it. By contrast, at $N_e = 3$, individual experts frequently use over 250 active pixels within their $\sim$261-pixel segments ($>96\%$ utilization). At $N_e = 13$, each expert operates on $\sim$60 pixels and achieves near-complete density within its segment. In both cases, expert-level utilization rates dwarf the $\sim$6.1% density anchor, demonstrating that partitioning transforms the relationship between evolutionary capacity and input dimensionality. Density mea-

sures wiring, not exploitation: a pixel counts as active with a single non-zero connection, whether or not that connection carries discriminative signal. Why the two quantities separate in the data, and why the prior campaign champion's 48-pixel stop is a satisficing point rather than a capacity wall, is analyzed in Appendix D.1.

This finding addresses a question left open in §4.1: is the monolithic network's sparse receptive field an optimal feature selection strategy, or an artifact of search-space overwhelm? The density result answers decisively. Constraining each expert to a smaller input region allows evolution to explore and exploit that region more thoroughly, discovering denser and more informative connectivity patterns. The baseline's sparsity reflects premature convergence, not intelligent selectivity.

4.6.5 The Mechanism: Why Partitioning Works

The receptive field analysis provides a complete mechanistic account of the Independent partitioned-experts architecture's success, which can be summarized in four steps:

1. **Breaking the central bias.** Each expert operates in a coordinate space where its own partition occupies the center. The CPPN's geometric bias toward the origin, a liability for the monolithic substrate, now operates *within* each partition, encouraging dense feature extraction in every region of the image.

2. **Reducing premature convergence.** The monolithic network converges early to a minimal central receptive field because evolution discovers that a small cluster of informative pixels provides sufficient fitness signal. With partitioned inputs, each expert's fitness depends exclusively on features in its assigned region, preventing this early convergence to a globally suboptimal solution.

3. **Enabling denser connectivity.** Smaller input dimensionality per expert ($\approx 784/N_e$ pixels) reduces the effective search space, allowing NEAT's complexification dynamics to explore connectivity more thoroughly. The result is denser, more informative substrates per expert.

4. **Composing specialist knowledge.** The `Performance-Weighted` aggregation strategy turns diverse specialist knowledge into classification accuracy by weighting each expert's contribution by its demonstrated per-class competence, resolving the credit assignment problem inherent in cooperative co-evolution.

The $N_e = 13$ optimum reflects a balance: enough experts to ensure broad spatial coverage (73% of pixels), but not so many that each expert's partition becomes too small to contain discriminative features. Coverage is non-monotonic, with tied peaks of 85% at $N_e = 8$ and $N_e = 14$ bracketing the $N_e = 13$ accuracy peak (73% coverage); this decoupling shows that coverage quantity alone does not predict accuracy: broader coverage at $N_e = 8$ or $N_e = 14$ does not translate into higher accuracy than $N_e = 13$'s 73%. Which pixels a partition scheme exposes evidently matters, though this chapter measures coverage extent rather than coverage informativeness.

This four-step mechanism has direct implications for the GEENNS vision. Each expert functions as a *proto-grid*: a specialist subnetwork processing a constrained input region. The aggregation strategy functions as a *proto-mapper*: a composition mechanism that weights specialist outputs by competence. The key gap between this partitioned-experts architecture and full GEENNS is that the partitioning and the aggregation are fixed rather

than evolved, and the experts share no links at all; Section 4.7 examines these connections and limitations in detail.

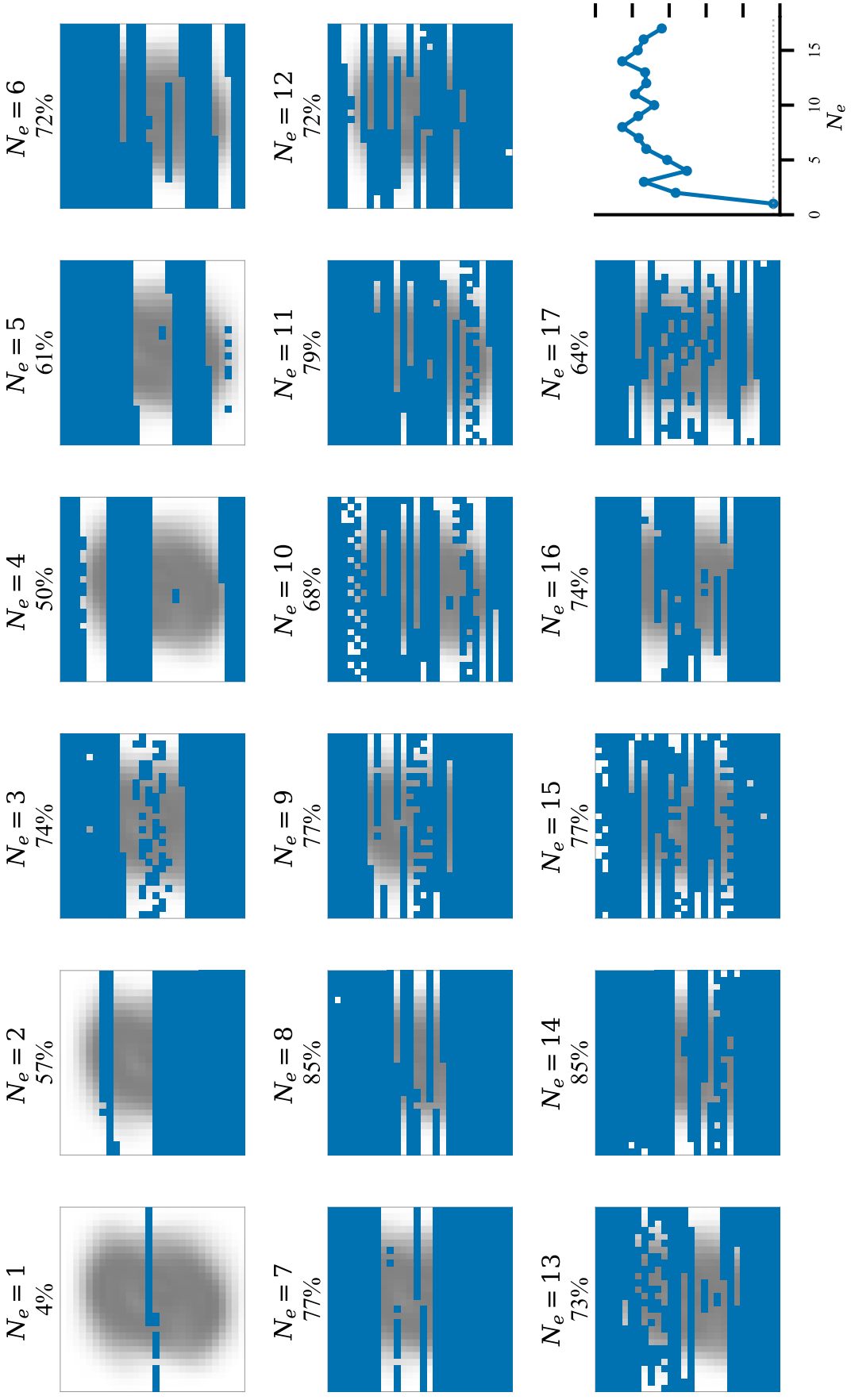


Figure 4.11: Evolution of the composite receptive field as N_e increases. The background is an MNIST pixel-variance heatmap (brighter areas indicate higher information content). Coverage expands from 3.6% at $N_e = 1$ to a non-monotonic peak of 85% (tied at $N_e = 8$ and $N_e = 14$), with the highest-accuracy model ($N_e = 13$) at 73%. Panel selection is three-way: the $N_e = 1$ panel shows the median of the 40 groupable runs; the $N_e \in \{3, 9, 11, 17\}$ panels union each expert's champion trial at its best generation; the remaining panels show each configuration's best complete run.

4.7 Synthesis: From Diagnosis to Decomposition

Before interpreting the partitioned-experts results, we note the scale of the accuracy gap. A single-hidden-layer MLP with 256 units trained by gradient descent achieves approximately 98% on MNIST; LeNet-5 reaches approximately 99%. Our 43% accuracy falls approximately 55 pp below these baselines. Throughout this thesis, classification accuracy is a *diagnostic measure*, not a performance target. The partitioned-experts architecture's contribution is demonstrating that spatial decomposition breaks the central bias and recovers up to 24× more input coverage (20× at the configuration that reaches 43%), establishing the architectural principle that GEENNS requires, not competing with gradient-trained classifiers.

4.7.1 Central Bias as an Encoding-Geometry Failure Mode

The central bias is not a quirk of MNIST classification. It is the downstream signature of three compounding properties of ES-HyperNEAT's encoding that together nudge every substrate toward the same narrow central basin before the evolutionary search has a chance to explore alternatives.

The first property is *coordinate-system symmetry*. The substrate embeds every input position into the normalized range $[-1, 1]$, placing the geometric origin at the center of the image. Any connection-weight function whose magnitude peaks at zero therefore generates its strongest responses for pixels near the image center, independent of what the task actually requires. The second is *symmetric CPPN primitives*. The activation functions typically used in CPPNs concentrate output variance near the coordinate origin (§2.1.9), so even a well-formed CPPN produces its largest output variances near $(0, 0)$. A bias input could in principle shift these responses off-center, but evolution does not reliably discover that shift once a locally adequate solution has been found inside the central basin. The third is the *quadtree's origin prior*. ES-HyperNEAT's recursive subdivision begins with coarse queries whose density is highest near the center of the coordinate region; node placement therefore inherits a central tilt before any CPPN query has been scored.

The three effects compound. The CPPN's output is loudest at the origin, the quadtree probes the origin most densely, and the evolutionary search finds a locally sufficient solution in the central basin with no gradient pulling it outward. That the bias reproduces across all 30 replications of Chapter 3's best configuration tells us it is not a tuning accident or an unlucky seed: it is a deterministic consequence of the encoding's geometry, and no hyperparameter choice within the viable region of the 16-parameter space dislodges it.

A confound this thesis cannot disentangle. Our diagnosis comes with a caveat that MNIST itself introduces. The informative pixels of a handwritten digit lie near the image center, so the substrate's coordinate-system bias coincides with the task's information-density bias. On this benchmark we cannot tell whether evolution converges to the central cluster *because* the substrate geometry pushes it there or *because* the task actually rewards attending to the center. A clean disambiguation would require stimuli whose informative regions are spatially offset from the substrate origin: for example, MNIST digits translated into an image corner, or object-at-edge detection tasks in which the diagnostic feature never overlaps the center. We leave that experiment to future work (§4.7.6). What we can say on MNIST is that the partitioning intervention removes the central bias *as a global constraint* regardless of which mechanism dominates, because it makes a single image-

wide central basin unreachable by construction: no single expert sees the full image, and no expert can solve its partition by attending to pixels outside its slice.

Why partitioning works. Reducing each expert's input from 784 dimensions to roughly $784/N_e \approx 60$ dimensions at $N_e = 13$ converts an intractable global search into 13 tractable local ones. This is a form of *divide-and-conquer for evolutionary search*: rather than hoping that a single population discovers useful connectivity across the entire coordinate space simultaneously, the architecture constrains each population to a region small enough that the connectivity it discovers is also informatively exploitable within the evolutionary budget. The 105% accuracy improvement is not a property of the partitioned-experts concept per se: it is a consequence of matching the search space to the evolutionary algorithm's capacity *and* of forcing coverage of regions that the coordinate-system bias would otherwise suppress.

Scope of the diagnosis. Both the failure and its remedy are specific to coordinate-based indirect encodings. The three compounding mechanisms all live in the encoding's geometry (the normalized substrate coordinates, the symmetric CPPN primitives, and the quadtree's recursive subdivision), so neuroevolutionary methods that use direct encodings, or indirect encodings whose generator is not origin-symmetric, will not exhibit the same pathology and will not benefit from the same fix. The claim this chapter defends is therefore narrow but sharp: within the ES-HyperNEAT family and its descendants (including Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT) in Chapter 6), structured input decomposition is a necessary architectural intervention. GEENNS, because it inherits its substrate geometry from ES-HyperNEAT, inherits the bias as well, and this chapter is the first architectural evidence that the bias can be neutralized as a global constraint without abandoning indirect encoding altogether.

4.7.2 Why $N_e = 13$ Is Optimal

The accuracy peak at $N_e = 13$ is not an arbitrary data point: it reflects a match between the expert's receptive field size and the evolutionary algorithm's search capacity. At $N_e = 13$, each expert receives approximately $784/13 \approx 60$ input pixels. This is close to the ∼48-pixel receptive field of the prior campaign's champion network [11] (this chapter's own 30-run replication does not reproduce that value: median ∼29 pixels, range 1–465; §4.4.4), suggesting that ∼50–60 input dimensions is the range the evolutionary loop can exploit *informatively* within the standard 20-generation budget, not the range over which it can wire connections at all (§4.6.4). Below this segment size ($N_e > 13$), experts receive too few pixels to extract discriminative features; a thin horizontal band of the digit may be insufficient to distinguish "1" from "7." Above this segment size ($N_e < 13$), experts face the same search-space overwhelm as the monolithic baseline: the central bias re-emerges within each segment.

The peak at $N_e = 13$ therefore encodes a property of NEAT's search capacity under ES-HyperNEAT's indirect encoding: approximately 60 input dimensions is the largest domain a single population can exploit informatively within a reasonable evolutionary budget; wiring extends well beyond it (§4.6.4). This has direct implications for GEENNS grid design, where each specialist grid's input dimensionality should be calibrated to this empirical capacity bound.

4.7.3 Comparison to Classical Mixture-of-Experts

Our spatial partitioned-experts architecture differs from the classical MoE framework of Jacobs et al. [43] and its modern descendants [84] in three ways. In classical MoE, a *gating network* learns to route each input to the most appropriate experts, and the gate weights are trained jointly with the experts via backpropagation. The gating is dynamic, so the same expert may process different inputs with different weights. Our architecture differs as follows:

1. **Static vs. dynamic gating.** Our assignment of input regions to experts is fixed and human-designed: each expert always sees the same pixel segment regardless of the input. Classical MoE learns which expert should handle which input through gradient-based gating. Our static assignment eliminates the gating search space entirely, which is advantageous when the optimizer is evolutionary (and therefore orders of magnitude slower than gradient descent) but sacrifices the ability to discover non-obvious input–expert affinities.

2. **No inter-expert gradient flow.** In classical MoE, the gating function provides gradient paths that allow experts to specialize cooperatively. Our experts evolve in complete isolation; specialization arises from the input partition, not from competitive pressure. This simplifies the evolutionary dynamics but prevents experts from discovering complementary representations.

3. **Post-hoc vs. learned aggregation.** The `Performance-Weighted` (F_1-weighted) strategy computes aggregation weights from held-out validation accuracy, a supervised post-processing step. Classical MoE learns aggregation weights end-to-end. Our approach is simpler but cannot adapt aggregation to individual inputs.

These differences define the gap between our partitioned-experts prototype and the full GEENNS architecture. GEENNS would require evolved gating (mappers), inter-grid communication, and end-to-end fitness-driven aggregation, closing all three gaps simultaneously through evolutionary rather than gradient-based optimization.

Why this is not just ensembling. A natural objection is that any ensemble of 13 independent models would outperform a single one, and that what this chapter attributes to spatial decomposition might be no more than averaging across 13 near-independent classifiers. The receptive-field data rule that out directly. Across all 30 monolithic baseline replications, the individual receptive fields are small and centrally anchored: the median run wires 29 pixels, every run's row centroid falls inside rows 10–16 (a central horizontal band that covers only 25% of the image height), and 77% of all wiring mass lands inside that band. The fields are not copies of one another (their union reaches 474 of the 784 input pixels, 60.5%, spanning rows 3–22, driven by a few atypically broad runs), but the union saturates the central band while 69% of its peripheral pixels are touched by exactly one run. An ensemble of monolithic models therefore inherits the central concentration rather than escaping it: unions of 13 baselines drawn at random from the 30 cover a median of 278 pixels (35.5% of the image), far from the near-complete coverage that partitioning guarantees by construction. Under the fixed-batch Same-Batch-per-Expert control (§4.4), the collapse is tighter still: its 30 monolithic runs stay entirely inside rows 10–16, with a 30-run union of just 169 pixels (21.6%). Ensemble theory predicts exactly this degeneracy. The classical decomposition of Krogh and Vedelsby [53] shows that an

ensemble's error equals the average individual error minus a diversity term, and that diversity term goes to zero when every member converges to the same features. What the Independent variant provides is not more classifiers but *structurally forced diversity*: because each expert is shown a different slice of the image, the population cannot collapse into the central basin even if the encoding geometry would otherwise pull it there. The Shared variant makes the same point from the opposite direction. It receives partitioned inputs but evolves a single genome shared across all partitions, so it cannot specialize on any one slice, and its 26.03% accuracy falls far short of the Independent variant's 43.07%. Spatial diversity of inputs is necessary; independent evolution within each partition is what turns that diversity into a performance gain.

4.7.4 Connection to the GEENNS Vision

As introduced in Section 1.6, GEENNS proposes compositional reasoning through frozen specialist grids coordinated by evolved mappers. The partitioned-experts architecture, though designed as a practical intervention, provides empirical evidence for two principles central to that framework:

Specialist subnetworks process distinct input regions.
Each partitioned expert functions as a *proto-grid*: a specialist subnetwork that evolves features within a constrained input partition. Decomposition into specialists is not merely viable but decisively superior to monolithic processing, at least when the search space is large relative to evolutionary capacity.

Competency-based aggregation outperforms naive combination.
The `Performance-Weighted` strategy, which weights each expert's contribution by its demonstrated F_1-score on each class, functions as a *proto-mapper*: it routes and combines specialist outputs based on measured competency rather than treating all opinions equally. The $d = 7.28$ effect size separating `Performance-Weighted` from naive averaging shows that how specialists are combined is a second, separable lever: partitioning alone contributes 14.65 pp and competency weighting a further 7.47 pp, measured across the best conditions of each lever (§4.5.4).

These proto-grids and proto-mappers provide empirical support for the decomposition principle at the core of GEENNS. However, the analogy has clear limits: the partitioning and the aggregation are fixed here rather than evolved, and the experts have no inter-expert links at all. The gap between fixed spatial decomposition and evolved compositional reasoning remains the central open problem for GEENNS.

Chapter 6 extends the platform side of the decomposition argument: the CCM financial benchmark (94,464 observations, 5 input features) sustains a 5.5× per-generation speedup at depth 6 with population 100, evidence that the tensor reformulation scales evolved substrates to real-world data volumes, not only pedagogical benchmarks.

4.7.5 From Fixed to Evolved Decomposition

The gap between the partitioned-experts prototype and the GEENNS vision can be organized into three concrete requirements that GEENNS must satisfy but this chapter's architecture does not:

Evolved partitioning.
This chapter uses a fixed, linear partitioning of a flattened pixel vector. GEENNS requires the decomposition itself to be evolved: mappers must discover which input features should be routed to which grid based on task structure, not based on predetermined pixel adjacency. The $N_e = 13$ peak suggests that segment size matters, but the optimal decomposition is almost certainly task-dependent and non-linear.

Inter-grid communication.
Partitioned experts evolve in complete isolation. GEENNS grids must communicate during substrate formation: a grid specializing in vertical strokes may need to query a grid specializing in curves to resolve ambiguous inputs. The connection type taxonomy developed in Chapter 6 provides the vocabulary for such inter-grid connectivity, but the coordination mechanism remains an open problem.

Compositional aggregation.
`Performance-Weighted` aggregation is static and computed offline. GEENNS requires an evolved aggregation function (the output mapper) that learns to compose grid outputs in a task-specific, input-dependent manner. The 43% accuracy with static aggregation provides a lower bound on what evolved aggregation could achieve; the magnitude of the gap is an empirical question that the current architecture cannot answer.

Each of these requirements corresponds to a component of the GEENNS architecture that has not yet been validated at scale. The prototype results (Chapter 7) demonstrate that frozen specialist grids combined with evolved mappers can solve 3-task compositions at 100% success (providing partial evidence for the first and third requirements), but the inter-grid communication problem remains fully open.

The partitioned-experts results also provide indirect evidence for the emergence thesis developed in Section 7.3.3. Expert *class* specialization was not explicitly programmed: the input partition was fixed by design, but nothing assigned digit classes to experts, and each expert evolved independently under the same evolutionary pressure. Yet the ensemble exhibits emergent division of labor: different experts develop different internal representations suited to the digit features visible in their partition, and the weighted ensemble outperforms any individual expert. The accuracy gap documented in §4.5 quantifies how much more capable an ensemble becomes when components are structurally constrained to specialize. Whether this emergent specialization scales to the full GEENNS architecture, where the partitioning itself is evolved rather than fixed, and where inter-grid communication replaces static aggregation, remains an open empirical question.

4.7.6 Scope Limitations

Several limitations constrain the generalizability of these results:

1. **Linear partitioning of a flattened vector.** The input image is flattened to a 784-element vector and split into contiguous segments. This produces horizontal slices of the original 28×28 image, ignoring 2D spatial structure entirely. A partitioning scheme that respects pixel adjacency (e.g., rectangular patches or superpixels) might yield better results by preserving local spatial correlations within each expert's input.

2. **No inter-expert communication.** Experts evolve in complete isolation. There is no mechanism for one expert to inform another about features it has discovered, which may limit performance on tasks requiring global spatial integration (e.g., recognizing a digit that spans multiple partitions). The aggregation step combines outputs but cannot compensate for information that was never extracted because no expert had access to the relevant context.

3. **Fixed expert-to-region assignment.** The mapping from input partition to expert is predetermined and static. A learned gating function, as in classical MoE architectures [43, 84], could potentially discover better assignments, though at the cost of additional evolutionary complexity and a gating search space that grows with N_e.

4. **Untested causal hypotheses.** The three-mechanism account of central bias above is a hypothesis, not a theorem. Two targeted experiments would test it directly. The first would swap ES-HyperNEAT's symmetric CPPN primitives for asymmetric alternatives (shifted sigmoids, learned radial centers, or primitives whose peaks are not anchored to the origin) and check whether the central basin survives when no primitive is maximal at the substrate center. The second would retrain the baseline on stimuli whose informative regions are deliberately offset from the substrate center, for instance MNIST digits translated into an image corner, or object-at-edge detection tasks, and observe whether the receptive field tracks the data or stays at the origin. A positive result in the first experiment would confirm the primitive-symmetry mechanism; a positive result in the second would disentangle coordinate-system bias from information-density bias. Neither experiment is within the scope of this thesis, but both are natural extensions of the spatial-decomposition result and would sharpen the causal claim from a reasoned hypothesis into a tested explanation.

Thesis-level limitations, including benchmark scope and statistical considerations, are consolidated in Section 8.3.1.

4.7.7 Cost–Benefit Analysis

The partitioned-experts architecture trades computational cost for accuracy:

- **Networks evolved:** 13× more than the monolithic baseline (one per expert).
- **Search space per expert:** $784/13 \approx 60$ pixels, versus 784 for the monolithic network, a 13× reduction in input dimensionality.
- **Wall-clock cost:** Approximately 2× the monolithic baseline when experts evolve in parallel, not 13×, because expert populations are independent and can be distributed across cores.
- **Accuracy gain:** 20.95% → 43.07%, a 105% relative improvement ($d = 8.70$).

The cost–benefit trade-off is favorable: a 2× increase in wall-clock time purchases a 2× increase in accuracy with an effect size exceeding 8 standard deviations. The smaller per-expert search space means that each expert's evolution is easier (fewer dead-end configurations, faster convergence), which partially offsets the multiplied network count.

This cost structure also has implications for future work. The partitioned-experts overhead scales linearly with N_e, but the per-expert search space shrinks proportionally. If expert evolution were combined with the early-stopping pruner from Chapter 3, the 41.6% intra-trial savings would apply independently to each expert, further reducing the aggregate cost. The combined pipeline (TPE + pruner + partitioned experts) was not tested in the experiments reported here, but the methods are architecturally compatible and could compose multiplicatively.

4.8 Chapter Summary

This chapter answered TRQ2 and TRQ4 by diagnosing the central bias as the structural limitation behind ES-HyperNEAT's performance ceiling and demonstrating that a spatially partitioned experts architecture (MoE-inspired) can overcome it. Across 3,570 experimental runs (7 conditions × 17 expert counts × 30 seeds), evaluated under 14 aggregation strategies, the results consistently show that decomposition into specialist subnetworks, combined with competency-based aggregation, outperforms monolithic substrate evolution on high-dimensional tasks. Table 4.6 summarizes the key results.

Table 4.6: Summary of Chapter 4 key results.

Result	**Value**	**Section**
Central bias diagnosis		
Baseline pixel coverage	28/784 (3.6% coverage)	§4.1
Peak partitioned-experts coverage	85% (tied at $N_e = 8$, $N_e = 14$)	§4.6
Partitioned-experts performance		
Best Independent variant accuracy (mean)	43.07%	§4.5
Peak single-run accuracy	50.00%	§4.5
Independent vs. Monolithic	$d = 8.70$	§4.5
Independent vs. Shared	$d = 7.54$	§4.5
Optimal expert count	$N_e = 13$	§4.5
Aggregation		
F1-weighted vs. naive (Avg)	$d = 7.28$	§4.5
Aggregation ANOVA	$F(13, 406) = 798.06,\ p < .001$	§4.5
Replication		
$N_e = 1$ vs. original baseline	$p = .502$ (no significant difference)	§4.5

The answer to TRQ2. The fundamental limitation of ES-HyperNEAT's substrate formation is the *central bias*: evolved substrates concentrate their connectivity near the coordinate origin, using only 3.6% of the available input pixels regardless of hyperparameter settings. Three compounding mechanisms produce this outcome: the normalized substrate coordinates place the origin at the image center, the CPPN activation palette concentrates output variance at that same origin (§2.1.9), and the quadtree's recursive subdivision probes the origin most densely. The failure mode is therefore a deterministic consequence of the encoding's geometry rather than a tuning artifact, and no amount of hyperparameter optimization overcomes it. This scopes the diagnosis precisely: the central bias is a property of coordinate-based indirect encodings with symmetric generators

on geometric substrates, so the ES-HyperNEAT family and its descendants (including EMR-HyperNEAT in Chapter 6) inherit it, whereas direct encodings and non-origin-symmetric generators do not. On MNIST, however, the coordinate-system bias and the task's own information-density bias coincide (digits are centrally located), so this chapter cannot by itself disentangle which of the two drives the convergence; we flag targeted experiments that would disambiguate them in §4.7.6. The replication control ($N_e = 1$ vs. original baseline, $p = .502$) is consistent with the limitation being reproducible rather than an artifact of our experimental framework, completing the Chapter 3 transition from parametric ceiling to architectural diagnosis.

The answer to TRQ4. Structured input decomposition improves evolved substrate performance. The Independent partitioned-experts architecture more than doubles the monolithic baseline accuracy (Table 4.6) by forcing each expert to discover features within its assigned partition. The mechanism is receptive field expansion: partitioning breaks premature convergence, expanding coverage from the central cluster identified in §4.1 to most of the input space. Competency-based aggregation (the `Performance-Weighted` strategy, which uses per-class F_1-scores as expert weights) is essential for realizing this potential; naive combination strategies fail to resolve conflicting specialist outputs as expert count increases. The results provide empirical support for the decomposition-and-composition principle at the core of the GEENNS vision, though the partitioning here is fixed rather than evolved. The non-overlapping distributions between the optimal partitioned-experts configurations and monolithic runs (worst 35.00% vs. best 29.00%) demonstrate that the architectural advantage is consistent at those configurations rather than dependent on favorable random seeds.

Transition to Chapter 5. If the central bias demonstrates that ES-HyperNEAT's quadtree-based substrate formation is the bottleneck, a natural question arises: can the quadtree itself be accelerated or replaced? All improvements in this chapter (13 expert networks, F1-weighted aggregation, 43.07% accuracy) still rely on the original CPU-bound quadtree implementation. Chapter 5 investigates the fundamental computational limitations of ES-HyperNEAT's quadtree on modern GPU hardware, establishing the incompatibility result that motivates a complete architectural replacement.

Part III

Beyond the Quadtree

Chapter 5

ES-HyperNEAT's Quadtree Problem

> The snake which cannot cast its skin has to die. As well the minds which are prevented from changing their opinions; they cease to be mind.
>
> Friedrich Nietzsche

This chapter addresses two thesis research questions:

TRQ2. Can ES-HyperNEAT's fundamental limitations be diagnosed empirically?

TRQ3. Can adaptive substrate discovery be redesigned for massively parallel execution without sacrificing adaptivity?

This chapter addresses the *diagnostic* half of TRQ3: whether ES-HyperNEAT's existing quadtree-based discovery can itself be parallelized on GPUs, as a precondition to any broader redesign. Chapter 6 supplies the constructive half (a tensor-native approach that replaces the quadtree while preserving adaptivity). The immediate motivation is Chapter 4's partitioned-experts architecture, which overcomes the central bias ($d = 8.70$) but multiplies the quadtree cost by $N_e = 13$; we first ask whether JAX-based GPU acceleration can alleviate that bottleneck. This chapter establishes that it cannot: the quadtree refines the variance landscape of each Compositional Pattern-Producing Network (CPPN) into a unique topology, leaving no population-level tensor of uniform shape for GPU vectorization.

We begin by formalizing the computational architecture of ES-HyperNEAT's quadtree and identifying the structural incompatibilities with GPU parallelism (§5.1–§5.3). We then evaluate the Hierarchical Spatial Hash Grid (HSHG) as an alternative (§5.4), present empirical results across 456 baseline XOR trials and a JAX-based GPU reimplementation, with cross-task validation on Parity-3, circle classification, sine regression, and cart-pole control (§5.5), and analyze why JAX-based GPU acceleration plateaus despite aggressive optimization (§5.6). Section 5.7 synthesizes the findings within the broader thesis narrative, connecting the Single Instruction, Multiple Threads (SIMT) incompatibility result to the GEENNS vision of compositional reasoning, and §5.8 summarizes the chapter's contributions and delivers the answer to TRQ2 together with the diagnostic half of TRQ3.

The work in §5.1–§5.6 is based on Paper 4 [17]. Author contributions for all papers are detailed in the List of Publications (p. 31).

5.1 The Quadtree Bottleneck Hypothesis

The preceding two chapters established complementary results: Chapter 3 showed that systematic hyperparameter search can explore ES-HyperNEAT's large configuration space, while Chapter 4 demonstrated that spatially partitioned expert ensembles can overcome the central bias inherent in single-network substrates. Both lines of work share a practical constraint: the 2,013-trial Tree-structured Parzen Estimator (TPE) campaign required months of distributed CPU evaluation, and the 13-expert partitioned-experts architecture multiplies that cost by the number of experts. Scaling to the dozens of specialist networks that the GEENNS compositional architecture requires (Section 1.6) is computationally prohibitive without GPU acceleration.

This chapter asks whether GPU parallelism, the standard remedy for compute-bound algorithms, can resolve this bottleneck, and establishes that it cannot. The investigation targets the quadtree at the core of substrate discovery in ES-HyperNEAT, the algorithm originally introduced by Risi and Stanley [78]. As described in §5.2, ES-HyperNEAT uses adaptive quadtree subdivision [30] to place hidden nodes where the CPPN's output landscape exhibits high variance. This adaptive mechanism eliminates the need for hand-designed network topologies, but no prior work has established whether ES-HyperNEAT's adaptive substrate discovery is compatible with modern GPU hardware.

The central hypothesis is that the quadtree imposes a *structural* bottleneck that no implementation-level optimization can overcome. Because quadtree subdivision follows each individual CPPN's variance landscape, every genome produces a substrate with different node counts, positions, and connections. No uniform tensor exists across the population: the fixed-shape data structure that GPU parallelism requires. GPU utilization therefore remains low regardless of how efficiently any single substrate is constructed. This hypothesis, if confirmed, has direct consequences for the thesis roadmap: it would rule out incremental optimization as a path to the GEENNS multi-grid architecture and require a fundamentally different substrate discovery mechanism.

To test this hypothesis, the chapter addresses three research questions that map directly onto TRQ2 and TRQ3 from the thesis introduction:

RQ1. Does within-individual acceleration of the quadtree traversal close the gap left by the population-level serial dependency, or does a residual bottleneck survive every implementation effort?

RQ2. If the algorithm is retargeted onto a GPU-backed stack (CPPN evolution driven by TensorNEAT, substrate construction executed in JAX), do per-depth runtimes and solve rates shift enough to recover the practical utility absent from the CPU reference?

RQ3. At what depth does adaptive quadtree discovery stop being runnable in practice, and what structural property of the algorithm fixes that ceiling?

If the quadtree bottleneck proves fundamental, then the GEENNS roadmap must include an entirely different substrate discovery mechanism: one that produces fixed-shape tensors suitable for GPU vectorization while preserving the adaptive sparsity that makes ES-HyperNEAT's topologies useful. Chapter 6 develops exactly such a mechanism.

5.2 Implementation Architecture

To evaluate GPU acceleration of ES-HyperNEAT, we developed a two-level evolutionary system called *JAX-ESHN* (JAX-based ES-HyperNEAT). The outer level uses TensorNEAT [100] to evolve a population of CPPN genomes via the standard NEAT algorithm. The inner level implements the same adaptive quadtree logic as the in-framework reimplementation of the PUREPLES algorithm [103] (hereafter "Baseline") but executes it through JAX and Accelerated Linear Algebra (XLA) compilation rather than CPU-bound Python. This design isolates the execution substrate as the independent variable: the algorithmic logic is shared, but the Baseline runs entirely on CPU using neat-python [66], while JAX-ESHN targets GPU. However, the two implementations differ in CPPN initialization distributions, default NEAT parameter values, and internal threshold conventions, so solve rate discrepancies (Section 5.5) cannot be attributed solely to the hardware change.

5.2.1 System Overview

Algorithm 6 presents the per-generation loop that both implementations share in structure.

Algorithm 6 ES-HyperNEAT Generation Loop

Require: State s, Problem P, Population size N

1: cppns $\leftarrow$ TensorNEAT.ask(s) {Sample CPPN population}
2: cppns_transformed $\leftarrow$ vmap(transform)(s, cppns)
3: **for** $i = 1$ **to** N **do**
4: cppn $\leftarrow$ cppns_transformed[i]
5: $H, C \leftarrow$ DiscoverSubstrate(cppn)
6: net $\leftarrow$ BuildNetwork(H, C)
7: $f_i \leftarrow$ Evaluate(net, P)
8: **end for**
9: $s' \leftarrow$ TensorNEAT.tell(s, $\mathbf{f}$)
10: **return** s'

Line 2 converts TensorNEAT's internal genome encoding into explicit weight matrices that the substrate discovery pipeline can query; this conversion step *is* vectorizable via `vmap`. The inner loop at lines 3–8, however, cannot be vectorized: each iteration runs the full substrate discovery pipeline for a single CPPN, and the resulting topology differs from individual to individual. This sequential inner loop is the structural bottleneck under investigation. Figure 5.1 illustrates the overall architecture.

5.2.2 Three-Phase Substrate Discovery

Substrate discovery proceeds in three phases (Algorithm 7), building the substrate incrementally from inputs through hidden nodes to outputs, following the protocol defined by Risi and Stanley [78].

- **Phase 1 (Input → Hidden).** Seeds the initial hidden layer by discovering connections from input nodes to potential hidden positions.

- **Phase 2 (Hidden → Hidden).** Iterates over newly discovered hidden nodes for L levels, allowing hidden-to-hidden connectivity to emerge. This capability matters for tasks requiring deep compositional reasoning across multiple hidden layers; the two-layer JAX-ESHN implementation sacrifices it for parallelizability (§5.6.6).
- **Phase 3 (Hidden → Output).** Reverses the discovery direction: queries are anchored at each output with outgoing = False, so that connections run from discovered hidden positions to the output, subject to the reversed feedforward constraint ($\ell.y < a.y$).

Each phase calls two sub-procedures (DivisionInit for quadtree subdivision and PruningExtract for band detection), described below.

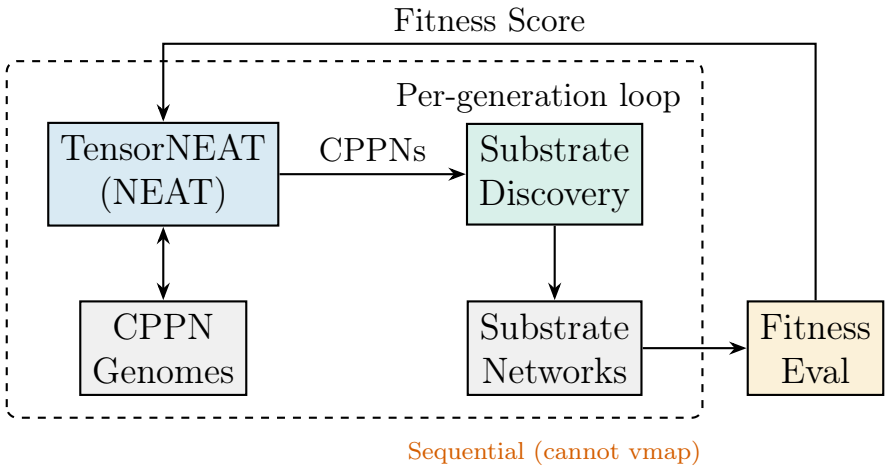


Figure 5.1: ES-HyperNEAT system architecture. TensorNEAT evolves CPPN genomes (left), each processed by substrate discovery (center) to build a network via quadtree subdivision. Fitness scores (right) drive selection. The bottleneck: substrate discovery runs sequentially per CPPN because each CPPN produces a unique topology.

5.2.3 Division Initialization (Quadtree Subdivision)

Throughout this chapter, QueryCPPN(θ, a, t, d) evaluates the CPPN so that the first two inputs are always the *connection source* and the last two are the *connection target*: $\theta(a.x, a.y, t.x, t.y)$ when d = outgoing (the anchor is the source), and $\theta(t.x, t.y, a.x, a.y)$ when d = incoming (the anchor is the target).

The quadtree subdivision procedure (Algorithm 8) recursively partitions the 2D substrate space *for a single anchor node*. At each level, it creates four child nodes at the quadrant positions, queries the CPPN for the connection weight from the anchor to each child, and computes the variance across these four weights. If this variance exceeds the division threshold θ_{div} (default 0.5, representing 50% of the maximum theoretical variance for CPPN outputs in $[-1, 1]$), the quadrant is subdivided further. Below D_{init}, subdivision is unconditional, ensuring a minimum resolution; above $D_{\max}$, subdivision halts regardless of variance.

Figure 5.2 illustrates the subdivision process for a region where one quadrant exhibits high variance.

Algorithm 7 Three-Phase Substrate Discovery

Require: CPPN θ, Inputs I, Outputs O, Iteration level L

$H \leftarrow \emptyset$, $C \leftarrow \emptyset$
{Phase 1: Input $\rightarrow$ Hidden}
for each $i \in I$ **do**
  root $\leftarrow$ DivisionInit(θ, i, outgoing = True)
  $C_1 \leftarrow$ PruningExtract(θ, i, root, outgoing)
  $C \leftarrow C \cup C_1$
  $H \leftarrow H \cup$ targets(C_1)
end for
{Phase 2: Hidden $\rightarrow$ Hidden}
$U \leftarrow H$ {Unexplored nodes}
for $\ell = 1$ **to** L **do**
  for each $h \in U$ **do**
    root $\leftarrow$ DivisionInit(θ, h, outgoing = True)
    $C_2 \leftarrow$ PruningExtract(θ, h, root, outgoing)
    $C \leftarrow C \cup C_2$
    $H_{\text{new}} \leftarrow$ targets(C_2) $\setminus H$
    $H \leftarrow H \cup H_{\text{new}}$
  end for
  $U \leftarrow H_{\text{new}}$
end for
{Phase 3: Hidden $\rightarrow$ Output}
for each $o \in O$ **do**
  root $\leftarrow$ DivisionInit(θ, o, outgoing = False)
  $C_3 \leftarrow$ PruningExtract(θ, o, root, incoming)
  $C \leftarrow C \cup C_3$
end for
$H, C \leftarrow$ CleanNetwork(H, C, I, O)
return H, C

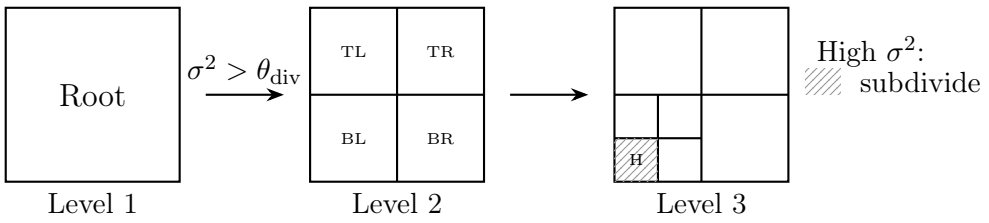


Figure 5.2: Quadtree subdivision for hidden node discovery. High-variance regions ($\sigma^2 > \theta_{\text{div}}$) are recursively subdivided; the hatched region (H) spawns a hidden node.

Algorithm 8 DivisionInit (Baseline)

Require: CPPN θ, Anchor coord a, Direction flag d
Require: D_{init}, $D_{\max}$, θ_{div}
root $\leftarrow$ QuadPoint$(0, 0, 1.0, \text{level} = 1)$
$Q \leftarrow$ [root] {BFS queue}
while $Q \neq \emptyset$ **do**
  $p \leftarrow Q$.pop()
  $w \leftarrow p$.width$/2$
  {Create 4 children at quadrant positions}
  children $\leftarrow$ [QuadPoint$(p.x - w, p.y - w, w, p.\text{level} + 1)$,
    QuadPoint$(p.x - w, p.y + w, w, p.\text{level} + 1)$,
    QuadPoint$(p.x + w, p.y + w, w, p.\text{level} + 1)$,
    QuadPoint$(p.x + w, p.y - w, w, p.\text{level} + 1)$]
  for each $c \in$ children **do**
    c.weight $\leftarrow$ QueryCPPN$(\theta, a, (c.x, c.y), d)$
  end for
  $\sigma^2 \leftarrow$ Var([children[i].weight])
  if p.level $< D_{\text{init}}$ **or** (p.level $< D_{\max}$ **and** $\sigma^2 > \theta_{\text{div}}$) **then**
    p.children $\leftarrow$ children
    Q.extend(children)
  end if
end while
return root

5.2.4 Pruning Extraction (Band Detection)

After quadtree construction, the pruning extraction step (Algorithm 9) separates genuine connection sites from artifacts of uniform CPPN output. As specified by Risi and Stanley [78], a leaf node is retained only if it sits at a *discontinuity*: a "band" where the CPPN's output changes sharply.

The algorithm queries the CPPN at four neighboring positions around each leaf (left, right, top, bottom) and computes the absolute difference between each neighbor's CPPN output and the leaf's own stored connection weight ($\ell.w$, assigned during subdivision) per direction. These are combined into a single band metric: $\max(\min(d_T, d_B), \min(d_L, d_R))$. The inner min ensures that the weight changes on *both* sides along at least one axis, confirming the node lies *between* different weight regions rather than at an edge; the outer max accepts discontinuities along either axis. Leaves whose band metric exceeds θ_{band} are retained, subject to a feedforward constraint ($a.y < \ell.y$ for outgoing connections in Phases 1–2; $\ell.y < a.y$ for incoming connections in Phase 3). A NormalizeWeight step applies dead-zone thresholding: weights with $|w| \leq w_0$ are mapped to zero, and the remainder is rescaled as $(w \pm w_0)/(1 - w_0)$, where $w_0 = 0.2$ in the reference implementation. This dead-zone mechanism is conceptually similar to the variance thresholding in the EMR reformulation (Chapter 6), where a different filtering strategy achieves the same goal of eliminating weak connections.

Algorithm 9 PruningExtract (Baseline)

Require: CPPN θ, Anchor a, Quadtree root, Direction d
Require: Band threshold θ_{band}
$C \leftarrow \emptyset$
leaves $\leftarrow$ CollectLeaves(root)
for each leaf $\ell \in$ leaves **do**
  $w_p \leftarrow \ell.\text{width} \times 2$ {Parent width}
  {Query 4 neighbors}
  $d_L \leftarrow |\ell.w - \text{QueryCPPN}(\theta, a, (\ell.x - w_p, \ell.y), d)|$
  $d_R \leftarrow |\ell.w - \text{QueryCPPN}(\theta, a, (\ell.x + w_p, \ell.y), d)|$
  $d_T \leftarrow |\ell.w - \text{QueryCPPN}(\theta, a, (\ell.x, \ell.y + w_p), d)|$
  $d_B \leftarrow |\ell.w - \text{QueryCPPN}(\theta, a, (\ell.x, \ell.y - w_p), d)|$
  {ES-HyperNEAT band metric}
  band $\leftarrow \max(\min(d_T, d_B), \min(d_L, d_R))$
  if band $> \theta_{\text{band}}$ **then**
    $w \leftarrow$ NormalizeWeight(ℓ.weight)
    if $w \neq 0$ **and** $d =$ outgoing **and** $a.y < \ell.y$ **then**
      $C \leftarrow C \cup \{(a, (\ell.x, \ell.y), w)\}$ {Anchor below leaf}
    end if
    if $w \neq 0$ **and** $d =$ incoming **and** $\ell.y < a.y$ **then**
      $C \leftarrow C \cup \{((\ell.x, \ell.y), a, w)\}$ {Hidden below output}
    end if
  end if
end for
return C

5.2.5 Network Cleaning

After all three discovery phases complete, a cleaning pass eliminates dead-end topology. The procedure performs two breadth-first traversals: one starting from inputs to identify all nodes reachable in the forward direction, and another starting from outputs to identify all nodes whose activity can reach an output. Only nodes appearing in both sets are retained; all other nodes and their incident connections are removed.

5.3 Optimization Strategies

Given the sequential inner loop identified in Algorithm 6, we developed four optimizations targeting the primary computational bottlenecks *within* each individual's substrate construction. These optimizations are deliberately limited in scope: they can reduce per-individual cost but cannot break the population-level sequentiality that is the core barrier. The distinction matters for the thesis argument: if aggressive within-individual optimization still yields only modest speedups, the bottleneck is confirmed as structural. All speedup measurements are relative to a non-optimized JAX-ESHN baseline (TensorNEAT + sequential quadtree queries + standard BFS cleaning + sequential fitness evaluation) on XOR at depth 2 with population 500.

O1: Batched Division Queries. In the non-optimized implementation, Algorithm 8 issues a separate CPPN query for each child node at each subdivision level. O1 (Algorithm 10) restructures the loop to collect all children at a given quadtree level into a single coordinate array and issue one batched CPPN evaluation via `vmap`. This reduces the number of CPPN invocations from $4 \times |\text{nodes at level}|$ to one per level.

O2: Batched Pruning Queries. The band detection step (Algorithm 9, lines 5–8) issues four neighbor queries per leaf node. O2 (Algorithm 11) accumulates all neighbor coordinates across all leaves into a single array and issues one batched CPPN evaluation, reducing $4 \times |\text{leaves}|$ individual queries to a single call.

O3: vmap Substrate Evaluation. Without optimization, the substrate is evaluated on each test input one at a time (Algorithm 6, line 7). O3 instead concatenates all test inputs into a single array and vectorizes the forward pass with `vmap`, evaluating the entire batch in one call.

O4: Precomputed Offsets. Each subdivision step in Algorithm 8 (lines 5–10) recalculates the same four quadrant offsets $(\pm 0.5, \pm 0.5)$ relative to the current width. O4 eliminates this redundancy by storing these offsets in a constant matrix $\mathbf{\Delta} \in \mathbb{R}^{4\times 2}$ and computing all child positions in a single broadcasted operation, $\mathbf{pos} = (p_x, p_y) + \mathbf{\Delta} \times w$.

Table 5.1 reports the cumulative effect of these four strategies.[1]

The cumulative 1.65× improvement in the reference configuration is the ceiling for within-individual optimization in this configuration. By comparison, full GPU vectorization across a population of 500 individuals could theoretically yield 500× speedup, three orders of magnitude beyond what the quadtree's variable-topology output permits. These results confirm that the bottleneck lies not in the efficiency of individual substrate constructions but in the inability to batch across the population.

Table 5.1: Optimization speedups (XOR, depth 2, population 500).

Optimization	Incremental	Cumulative
Baseline (BFS cleaning)	1.00×	1.00×
+ O1 Batch division	1.15×	1.15×
+ O2 Batch pruning	1.07×	1.23×
+ O3 vmap evaluation	1.06×	1.30×
+ O4 Precomputed offsets	–	**1.65**×

[1]The cumulative speedups are not strictly additive because the optimizations interact. In isolation at population 1000, O4 contributes only 1.01×, yet in combination with O1–O3 it raises the total from 1.30× to 1.65×, a 27% apparent gain attributable to improved memory locality when precomputed offsets align with batched data layouts. An ablation at population 1000 found O2+O3 alone reaching 1.73×, indicating that interaction magnitudes are population-dependent.

Algorithm 10 O1: Batched Division Queries

Require: CPPN θ, Anchor a, Direction d

root $\leftarrow$ QuadPoint$(0, 0, 1.0, 1)$
$Q_{\text{level}} \leftarrow$ [root]
while $Q_{\text{level}} \neq \emptyset$ **do**
  {Collect ALL children for entire level}
  coords $\leftarrow$ []
  all_children $\leftarrow$ []
  for each $p \in Q_{\text{level}}$ **do**
    children $\leftarrow$ CreateChildren(p)
    all_children.extend(children)
    coords.extend([$(c.x, c.y)$ for $c \in$ children])
  end for
  {SINGLE batched CPPN query via vmap}
  $\mathbf{w} \leftarrow$ vmap(QueryCPPN)$(\theta, a, \text{coords}, d)$
  for $i = 1$ **to** |all_children| **do**
    all_children$[i].w \leftarrow \mathbf{w}[i]$
  end for
  {Filter by variance threshold}
  $Q_{\text{next}} \leftarrow$ []
  for each $p \in Q_{\text{level}}$ **do**
    $\sigma^2 \leftarrow$ Var([$c.w$ for $c \in p$.children])
    if ShouldSubdivide(p, σ^2) **then**
      Q_{next}.extend(p.children)
    end if
  end for
  $Q_{\text{level}} \leftarrow Q_{\text{next}}$
end while
return root

Algorithm 11 O2: Batched Pruning Queries

Require: CPPN θ, Anchor a, Root node, Direction d, Band threshold θ_{band}

$L \leftarrow$ CollectLeaves(root)
{Collect ALL neighbor coords for ALL leaves}
neighbors $\leftarrow$ []
for each $\ell \in L$ **do**
  $w_p \leftarrow \ell.\text{width} \times 2$
  neighbors.extend($[(\ell.x - w_p, \ell.y), (\ell.x + w_p, \ell.y),$
    $(\ell.x, \ell.y + w_p), (\ell.x, \ell.y - w_p)]$)
end for
{SINGLE mega-batch query ($4N \rightarrow 1$ batch)}
$\mathbf{w}_n \leftarrow \text{vmap}(\text{QueryCPPN})(\theta, a, \text{neighbors}, d)$
{Process with band detection}
$C \leftarrow \emptyset$
for $i = 1$ **to** $|L|$ **do**
  $d_L \leftarrow |L[i].w - \mathbf{w}_n[4i - 3]|$
  $d_R \leftarrow |L[i].w - \mathbf{w}_n[4i - 2]|$
  $d_T \leftarrow |L[i].w - \mathbf{w}_n[4i - 1]|$
  $d_B \leftarrow |L[i].w - \mathbf{w}_n[4i]|$
  band $\leftarrow \max(\min(d_T, d_B), \min(d_L, d_R))$
  if band $> \theta_{\text{band}}$ **then**
    $C \leftarrow C \cup \{\text{CreateConnection}(a, L[i])\}$
  end if
end for
return C

5.4 The HSHG Alternative

With within-individual optimization exhausted, a natural follow-up is whether a different spatial data structure could replace the quadtree entirely. This section investigates a Hierarchical Spatial Hash Grid (HSHG) based on spatial hashing techniques [95], testing whether $O(1)$ hash-based neighbor lookups combined with fixed-size arrays for just-in-time (JIT) compatibility could enable GPU acceleration where the quadtree cannot.

This approach was unsuccessful. Although HSHG achieves $O(1)$ neighbor queries and produces JIT-compatible data structures, it suffers from a deeper architectural mismatch with ES-HyperNEAT's discovery approach that no parameter tuning can resolve. The negative result is nonetheless instructive: it clarifies what properties any quadtree replacement must preserve and informs the "evaluate everywhere, filter afterward" strategy that EMR-HyperNEAT (Chapter 6) later adopts successfully.

5.4.1 Spatial Hash Function

HSHG partitions the substrate space into a grid of cells and assigns each node to a cell via a spatial hash using large primes [95]:

$$h(x, y) = (\lfloor x/c \rfloor \cdot 73856093 \oplus \lfloor y/c \rfloor \cdot 19349663) \bmod M \tag{5.1}$$

where c is cell size and M is hash table size (prime, e.g., 1009).

5.4.2 Fixed-Shape State for JIT Compatibility

Unlike the quadtree, which allocates nodes dynamically as subdivision proceeds, HSHG preallocates all arrays at initialization (Algorithm 12). This fixed-shape representation is directly compatible with JAX's JIT compilation, which requires static array dimensions.

Algorithm 12 HSHG State Structure

HSHGState := {
  positions : $\mathbb{R}^{N_{\max} \times 2}$ {Node coordinates}
  weights : $\mathbb{R}^{N_{\max}}$ {CPPN outputs}
  levels : $\mathbb{Z}^{N_{\max}}$ {Hierarchy level per node}
  valid_mask : $\{0, 1\}^{N_{\max}}$ {Active node flags}
  n : $\mathbb{Z}$ {Current node count}
  {Per-hierarchy-level cell storage}
  cells : $\mathbb{Z}^{L \times M \times K}$ {L levels, M cells, K capacity}
  cell_counts : $\mathbb{Z}^{L \times M}$ {Nodes per cell}
  overflow : $\{0, 1\}$ {Buffer overflow flag}
}

5.4.3 O(1) Radius Query

Neighbor queries check a fixed 5×5 grid of candidate cells centered on the query position (Algorithm 13), then filter candidates by Euclidean distance. The constant number of cells checked (25) makes the query $O(1)$ in the average case, regardless of the total number of nodes in the structure.

Algorithm 13 HSHG O(1) Radius Query

Require: State S, Center $\mathbf{c}$, Radius r, Cell size σ

{Generate 5×5 candidate cell hashes}
$c_x, c_y \leftarrow \lfloor \mathbf{c}_x/\sigma \rfloor, \lfloor \mathbf{c}_y/\sigma \rfloor$
$\mathcal{H} \leftarrow \emptyset$
for $\delta_x = -2$ **to** 2 **do**
  for $\delta_y = -2$ **to** 2 **do**
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{\text{SpatialHash}(c_x + \delta_x, c_y + \delta_y)\}$
  end for
end for
{Gather nodes from 25 candidate cells}
candidates $\leftarrow S.\text{cells}[\mathcal{H}]$ $\{[25 \times K]\}$
{Filter by actual Euclidean distance}
$\mathbf{P} \leftarrow S.\text{positions}[\text{candidates.flatten}()]$
$\mathbf{d} \leftarrow \|\mathbf{P} - \mathbf{c}\|_2$
valid $\leftarrow (\mathbf{d} \leq r)$
return candidates[valid], $\mathbf{d}$[valid]

Figure 5.3 illustrates the query mechanism.

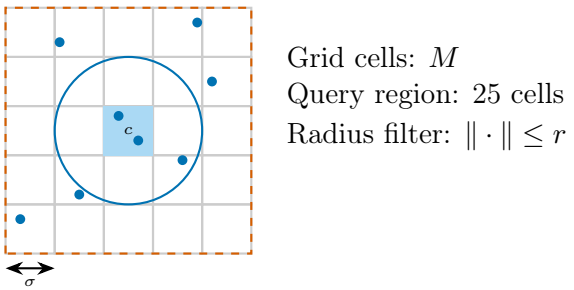


Figure 5.3: Hierarchical Spatial Hash Grid (HSHG) for $O(1)$ neighbor queries. Nodes hash into cells of size σ. Radius queries check a fixed 5×5 region (vermillion dashed), then filter by Euclidean distance (blue circle), replacing $O(\log n)$ quadtree lookups.

5.4.4 Dense Grid Pre-Generation

The quadtree and HSHG differ not only in their computational primitives but in *when* candidate positions come into existence. Under the quadtree, a position is produced only after the CPPN's variance at that location has been inspected and found to exceed the subdivision threshold; positions that fail the test are never materialized. HSHG reverses the timing: every candidate the data structure could contain (across every level of resolution, whether or not the CPPN will eventually have anything interesting to say there) is allocated up front. The depth parameter picks the finest resolution, and the total candidate count follows the standard expression for a quadtree filled to depth d (including the root level):

$$\text{Positions for max_depth } d = \sum_{i=0}^{d} 4^i = \frac{4^{d+1} - 1}{3} \tag{5.2}$$

Setting $d = 3$ yields $1 + 4 + 16 + 64 = 85$ preallocated positions. Once the grid is in place the CPPN is evaluated in a single batched pass, and a variance filter then selects which candidates become active nodes. This evaluate-then-filter pattern is what makes HSHG JIT-friendly, but it is also the source of the problem examined in the next subsection.

5.4.5 Why HSHG Fails: The Discovery Approach Mismatch

The quadtree and HSHG embody opposite discovery approaches, a distinction that proves decisive for the HSHG method:

- **Quadtree** (Algorithm 8): subdivides, beyond the initial depth, only where variance exceeds a threshold, producing sparse, adaptive exploration.
- **HSHG**: pre-generates candidate positions at all hash-grid levels, then filters by variance. The uniform pre-generation systematically over-discovers, as described below.

The root cause is that pre-generated positions at fixed locations sample the CPPN output in overlapping neighborhoods. When a single informative feature spans a region wider than the inter-position spacing, multiple adjacent positions detect high variance independently and all pass the threshold. The quadtree avoids this duplication by converging *toward* the feature, placing exactly one node at the point of maximum information density.

Evaluation across variance thresholds from 0.03 to 0.5 and grid depths 2–4 on 50 CPPNs confirmed the problem: HSHG placed 29–63 hidden nodes per substrate versus the quadtree's 1–2. Fitness peaked at 0.746 for a substrate with a single hidden node; substrates with 62 nodes scored only 0.500 (chance on XOR), confirming that over-discovery degrades performance rather than aiding it. No threshold or depth setting recovered the quadtree's selectivity. Figure 5.4, presented below after the comparative summary, illustrates the over-discovery mechanism.

5.4.6 HSHG vs. Quadtree Comparison

Table 5.2 summarizes the theoretical trade-offs between the two spatial data structures. On every axis relevant to GPU acceleration (lookup complexity, JIT compatibility, batch parallelism), HSHG is superior. But these advantages are nullified by the discovery-approach mismatch: the eager evaluation that makes HSHG JIT-compatible is the same property that causes over-discovery. This is not a parameter tuning problem: it is a structural incompatibility between eager position generation and adaptive sparsity.

A further complication revealed by the HSHG experiments is the *threshold mismatch* between the two implementations. The Baseline, an in-framework reimplementation of the PUREPLES algorithm [103], uses a `division_threshold` of 0.5 to decide whether a quadrant warrants further subdivision (representing 50% of the maximum theoretical variance for CPPN outputs in $[-1, 1]$). HSHG, by contrast, uses a `variance_threshold` of 0.03 for position activation, a 16.7× more sensitive setting. This difference is not a tuning error; it reflects fundamentally different roles: the quadtree threshold controls *whether to explore further*, while the HSHG threshold controls *whether a pre-explored position is active*. The much lower HSHG threshold is necessary because the hierarchical grid has already committed to exploring everywhere, and setting a high threshold would

eliminate most positions and destroy the grid structure. This asymmetry confirms that the two approaches are not interchangeable.

The lesson from HSHG is that replacing the quadtree's data structure is insufficient: any successful replacement must also redesign the discovery *approach*. The challenge is to find a method that evaluates eagerly (for JIT compatibility) while filtering adaptively (for sparse topologies). Chapter 6 shows that the EMR reformulation (eager evaluation on a fixed multi-resolution grid followed by variance-based filtering) achieves exactly this.

Table 5.2: HSHG vs. Quadtree comparison.

Aspect	Quadtree	HSHG
Neighbor lookup	$O(\log n)$	$O(1)$ average
JAX JIT compat.	Limited	Excellent
Batch parallelism	Difficult	Native vmap
Memory pattern	Dynamic	Fixed alloc.
Adaptivity	Full	Medium

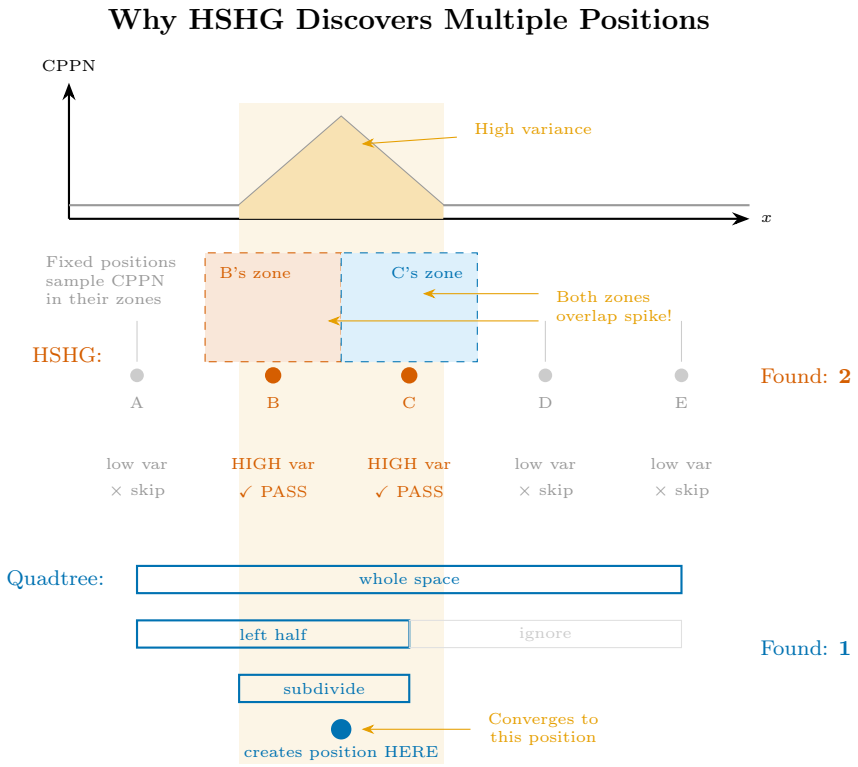


Figure 5.4: Why fixed-grid sampling over-counts an informative feature. **Top:** a CPPN output with one narrow spike. **Middle:** HSHG's pre-generated positions B and C both detect the spike within their neighborhoods and emit two nodes. **Bottom:** the quadtree converges adaptively and terminates at a single position.

5.4.7 Evolutionary Confirmation

The 50-CPPN sweep above establishes the over-discovery problem at the level of individual substrates, but it leaves open whether NEAT-driven evolutionary search could, in principle, navigate around the bloated topologies and find a working configuration. To close that loophole, we re-ran HSHG through the full evolutionary pipeline on four configurations chosen to span Boolean depth (XOR at depths 2 and 3), Boolean width (Parity-3 at depth 2), and continuous regression (Sine at depth 2), each at population 50 with 30 seeds and task-appropriate solve thresholds (0.975 for Boolean tasks, 0.95 for Sine regression).

HSHG uses a 50-generation cap (HSHG is orders of magnitude slower per generation, capping practical wall time) and the Quadtree Baseline uses a 100-generation cap, which is non-binding because every Quadtree XOR seed solves by generation 14. This campaign protocol differs both from the pop 150, 100-generation Multi-Benchmark Validation in §5.5.8 and from the Tier 1 Pop 50 cells of Figure 5.5 ($n = 3$, threshold 0.99, 30-generation cap, original GECCO config); the Quadtree solve rates in Table 5.3 accordingly diverge from those Tier 1 cells, which is one of the library-protocol effects flagged in §5.5.2. Within Table 5.3 both columns share the same protocol, so the within-row Quadtree-vs-HSHG comparison is direct.

The result is unambiguous: 0 of 120 seeds solved their task across the four conditions (Table 5.3). Two distinct failure modes appear. In the first, the substrate is so bloated that no valid network can be assembled (fitness $-\infty$): 5 of 30 seeds for XOR at depth 2 and 10 of 30 for each of the other three conditions. In the second, evolution does produce a functional substrate but cannot escape a fitness plateau: 0.750 for the Boolean tasks, which is exactly 3 of 4 patterns correct and corresponds to a majority-vote solution that cannot represent XOR's nonlinearity; and 0.769 for Sine, below the success threshold of 0.95 and only marginally above the Quadtree Baseline mean of 0.737. The 0.746 fitness reported above for an individually selected single-node substrate lands on the same 3-of-4 plateau under a different protocol, indicating that both protocols hit the same representational ceiling on XOR.

Table 5.3: HSHG vs. Quadtree Baseline solve rates under the full evolutionary pipeline (pop 50, 30 seeds; 50-generation cap for HSHG, 100 for Quadtree). HSHG fails on every condition; the Boolean-task plateau at 0.750 corresponds to a majority-vote solution.

Task	Depth	Quadtree	HSHG
XOR	2	30/30 (100%)	0/30 (0%)
XOR	3	30/30 (100%)	0/30 (0%)
Parity-3	2	23/30 (76.7%)	0/30 (0%)
Sine	2	0/30 (fit 0.737)	0/30 (0%)

The evolutionary failure agrees with the structural diagnosis already established and confirms that no parameter setting or seed sample recovers ES-HyperNEAT's sparse selectivity once positions are pre-generated.

5.5 Experimental Results

Having exhausted both within-individual optimization (§5.3) and data-structure replacement (§5.4), we now characterize the quadtree's empirical scaling directly, across the full depth × population grid.

5.5.1 Benchmark Configuration

All experiments use XOR, the canonical hard problem for non-deep architectures [68], following the standard ES-HyperNEAT benchmark protocol [78]. XOR was chosen because it requires a non-linear hidden layer, exercising the substrate discovery pipeline, while remaining simple enough that solve rate differences reflect computational constraints rather than task difficulty. The experimental grid spans the following configurations:

- Depths: 1–7 (5 to 21,845 candidate positions)
- Population sizes: Baseline 50, 100, 200, 300, 400, 500, 750, 1000 (8 values); JAX-ESHN 50, 100, 150, 200, 250, 300, 500, 750, 1000 (9 values)
- Maximum generations: 30 (Baseline measured; JAX-ESHN projected to 30 from 10-generation runs, §5.5.6; Tier 1 solve analysis extends to 300)
- Iteration level: 1 ($L = 1$ in Algorithm 7; one pass of hidden-to-hidden expansion)
- Fitness threshold: 0.98 for JAX-ESHN (the Baseline run records use 0.99)[2]
- Hardware: Baseline on Apple M4 Max (JAX CPU backend); JAX-ESHN on NVIDIA RTX 2080 Ti 11 GB (JAX CUDA backend)
- Optimizations: All JAX-ESHN results include O1–O4 from Section 5.3

Each configuration was replicated with 3 random seeds. Depths D1–D6 were tested at three iteration levels each (1, 2, 3); depth D7, due to its computational cost, was tested at iteration level 1 only. The full grid spans 8 population sizes (50–1000) × $[(6 \times 3) + (1 \times 1)]$ depth–iteration configurations × 3 seeds = 456 baseline trials. The replication count reflects computational constraints: at depth 7 with population 1000, a single Baseline run averages 34 minutes per generation, so larger replication counts are prohibitively expensive. The Tier 1 results (iteration_level = 1) presented in the tables below include 189 JAX-ESHN trials (9 populations × 7 depths × 3 seeds) and 168 Baseline trials (8 populations).[3] When the text refers to the aggregate 52.8% solve rate at D3, this reflects all 456 trials across all three iteration levels; the Tier 1 rate at D3 is 62.5%.

5.5.2 Implementation Comparison Framework

Section 5.5.1 above lists the experimental grid. Baseline and JAX-ESHN are not simply CPU and GPU runs of the same algorithm: they use different NEAT libraries underneath the quadtree (neat-python on the Baseline side and TensorNEAT on the JAX-ESHN side),

[2]The thesis uses several convergence thresholds depending on context (0.80, 0.85, 0.90, 0.95, 0.975, 0.98, 0.99, 0.995); see Appendix C.11 for the inventory and justification.

[3]Pop 300 was benchmarked with a 10-generation limit; its D1 (0/3) and D4 (2/3) solve rates may be underreported.

and the two libraries differ in several choices that affect what substrates end up being discovered. As Section 5.2 noted, the two libraries diverge in CPPN initialization and on at least a dozen default mutation and speciation parameters, and their activation-function menus carry slightly different primitive sets and normalization conventions. Each of these differences is small in isolation, but in combination they shift the fitness landscape enough that the two implementations can classify the same task as "solvable" or "unsolvable" for reasons that have nothing to do with GPU vectorization.

This is a confound, not a measurement: a controlled CPU-vs-GPU comparison would require running both implementations against the same underlying NEAT library, a cross-library port well beyond the scope of a diagnostic study whose only purpose is to measure the scaling exponent of the quadtree traversal itself.

The reading rule that follows from this structure is simple, and every numerical comparison in the remainder of this chapter and in Chapter 6 respects it:

- *Scaling exponents* (the 65–71× Baseline ratio, the 4.8–5.1× JAX-ESHN ratio) are genuine cross-implementation quantities: they measure how wall-clock cost responds to depth independently of which substrates end up solving the task, and they compare cleanly across the Baseline–JAX-ESHN boundary. The 1.65× optimization ceiling for the reference configuration and the JIT plateau growth factors are within-implementation scaling quantities, measured entirely inside JAX-ESHN.

- *Absolute solve rates* are *not* cross-implementation quantities. They are properties of each implementation's NEAT configuration and should be read only within an implementation, never across the boundary. When the multi-benchmark results in §5.5.8 show Parity-3 favoring the Baseline and sine regression favoring JAX-ESHN, those reversals reflect the NEAT-library defaults driving each implementation, not the hardware.

With this framework in place, the results in the remainder of §5.5 can be read on their proper terms.

5.5.3 Solve Rates by Depth

Two findings emerge from the full solve-rate measurements across all tested population sizes and depths, both sharpening the incompatibility argument. First, when given sufficient time (300-generation limit), JAX-ESHN achieves near-universal success: 182 of 189 Tier 1 trials (96.3%) reach the fitness threshold. The few failures (Pop 50 at D1, Pop 200 at D4, Pop 300 at D1 and D4) reflect specific population–depth combinations, not systematic inability. At population sizes ≥ 750, XOR is solved in the warm-up generation that JIT compilation already covers at depths 4–7, a result that reframes the problem entirely: the evolutionary search is not the bottleneck; the *construction* of the substrate is.

Second, the Baseline exhibits systematic collapse with depth: from 62.5% at D3 (Tier 1) to just 4.2% at D7, a 14.9× decay. This is not a smooth gradient: the sharpest drop occurs between D5 (33.3%) and D6 (8.3%), where the 4× increase in candidate positions (from 1,365 to 5,461) pushes most configurations past the 30-generation timeout. Figure 5.5 visualizes this contrast. The Baseline and JAX-ESHN use different population sets (400 vs. 150, 250, respectively) due to independent benchmark campaigns; comparison across the seven shared populations (50, 100, 200, 300, 500, 750, 1000) shows the same pattern.

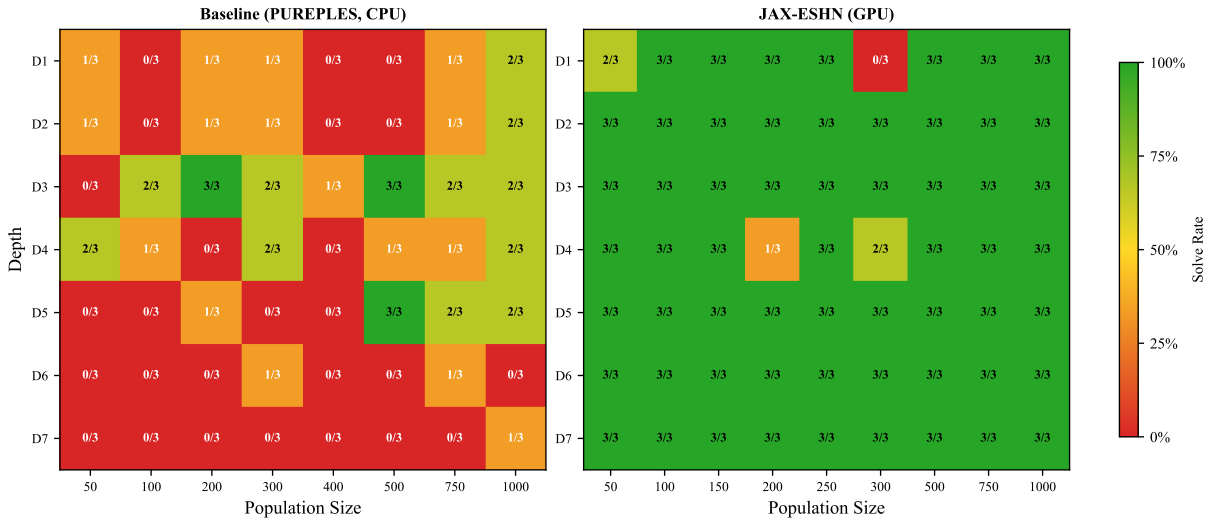


Figure 5.5: Solve rate by depth and population: Baseline (left, reimplemented PUREPLES algorithm, CPU, 30-generation limit) vs. JAX-ESHN (right, GPU, 300-generation limit). Each cell shows the fraction of seeds solving XOR (fitness ≥ 0.98 for JAX-ESHN; the Baseline run records use ≥ 0.99). The Baseline exhibits systematic collapse with depth (62.5% at D3 to 4.2% at D7), while JAX-ESHN achieves near-universal success.

5.5.4 Convergence Speed by Population

Beyond solve rates, convergence speed exposes how population size, depth, and search difficulty interact. This is the strongest evidence for the construction-cost hypothesis. Figure 5.6 reports the average generation to solve XOR for all JAX-ESHN configurations, and the data reveal two clear patterns.

First, for populations ≥ 750, all depths D4–D7 are solved at generation 1 (Pop 1000 extends this to D8–D10, not tabulated). This means that once the substrate is constructed (a process that takes hours of JIT compilation) the evolutionary search finds a solution immediately. XOR is not hard for the evolved substrate; it is hard for the *construction* of that substrate.

Second, the relationship between depth and difficulty is inverted relative to the Baseline. In the Baseline, deeper depths are progressively harder (62.5% at D3 to 4.2% at D7), while in JAX-ESHN, deeper substrates with more degrees of freedom often converge *faster* than shallow ones (Pop 50: D7 averages 6.0 generations vs. D1's 155.3). The larger substrate provides more representational capacity, allowing the evolutionary search to find solutions with less optimization.

5.5.5 Generation Time Scaling

Table 5.4 quantifies how Baseline runtime scales with depth and population. The scaling is the product of two factors: linear in population (doubling the population approximately doubles runtime, since each individual is processed sequentially) and exponential in depth (approximately 7× per depth level, driven by the 4^d growth in quadtree positions from Equation 5.2). At D7 with population 500, a complete 30-generation run averages 14 hours.

To contextualize this cost for the broader thesis: the 2,013-trial TPE campaign of Chapter 3 used depth 3, where a single run completes in under a minute. Repeating

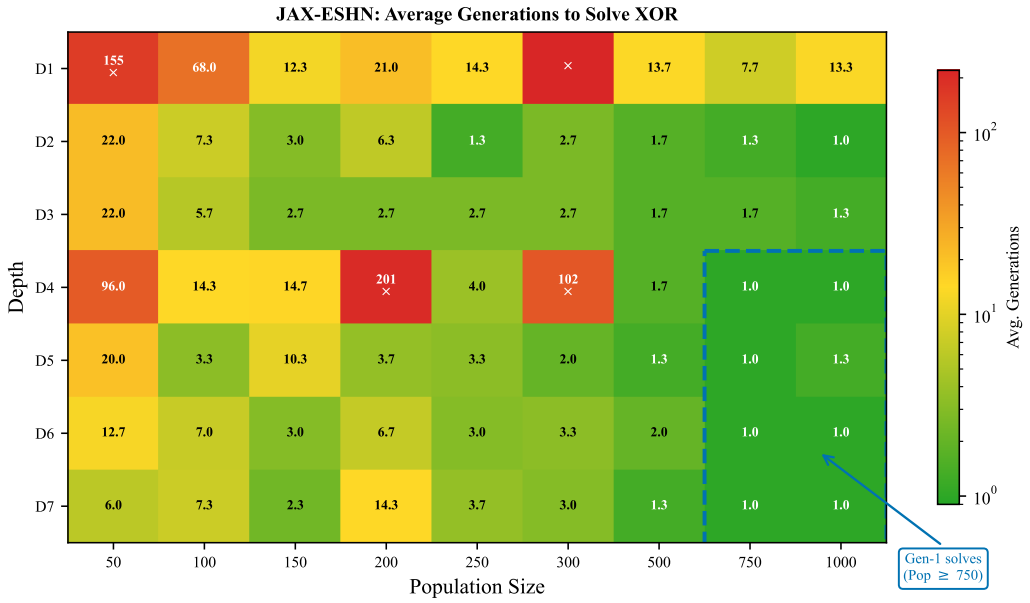


Figure 5.6: JAX-ESHN convergence speed: average generations to solve XOR by depth and population ($n = 3$ seeds per cell, max 300 generations; unsolved seeds counted as 300). Green cells indicate fast convergence (1–2 generations) and red cells indicate slow or failed convergence; the four cells where fewer than 3 seeds solved are D1 P50, D1 P300, D4 P200, and D4 P300, marked with "×". The dashed rectangle marks the gen-1 solve region (Pop $\geq$ 750, D4–D7), where XOR is solved immediately after substrate construction. Pop 200 at D4 (201 avg. gens, 1/3 solved) is the sole outlier among populations $\geq$ 200. Pop 300 ran with a 10-generation budget rather than 300, so its D1 (3/3 failed) and D4 (1/3 failed) cells may be underreported relative to populations that used the full 300-generation budget. Per-cell solve fractions are shown in Figure 5.5.

that campaign at depth 5, which is necessary for the deeper substrates that complex tasks demand, would multiply wall-clock time by roughly 50×. The partitioned-experts architecture of Chapter 4 further multiplies by $N_e = 13$ experts, making the combined cost impractical without GPU acceleration.

5.5.6 JIT Compilation Overhead

Table 5.6 reports JAX-ESHN total run time projected to a 30-generation budget, matching the Baseline evaluation window. This projection enables apples-to-apples comparison; in practice, JAX-ESHN solves XOR at generation 1 for most (depth, population) configurations at $D \geq 4$, and because generation 1 is evaluated inside the JIT warm-up step, the actual wall-clock time per run reduces to the JIT compile time alone. Both regimes are relevant to the chapter argument: the 30-generation projection (Table 5.6, Figure 5.7) characterizes JAX-ESHN's cost surface at matched budget; the wall-clock-spent reality (Table 5.5) characterizes user experience.

Both JIT compilation cost and post-JIT per-generation cost were measured directly for each (depth, population) configuration across 9 populations at D1–D7. The per-generation timing was collected over 10 evolutionary steps per configuration. A convergence analysis (Pop 200, D1–D5) indicates that 10-step measurements track longer runs closely at shallow depths (D1–D3), where per-generation timing stabilizes by step 10 with drift below ±5%

Table 5.4: Baseline total run time (minutes) by depth and population size (Apple M4 Max, mean ± std, 30 generations, $n = 3$). Time scales linearly with population and exponentially with depth due to quadtree's 4^d position growth. Per-population timing is shown for four representative populations; at D1–D2 the Baseline completes in under a minute at every population. At D7 Pop 1000, a single run averages 16.8 hours.

D	Pop 200	Pop 500	Pop 750	Pop 1000
3	0.1 ± 0.0	0.3 ± 0.1	0.3 ± 0.0	0.5 ± 0.0
4	0.6 ± 0.1	1 ± 1	2 ± 1	2 ± 1
5	5 ± 4	11 ± 4	7 ± 1	23 ± 5
6	53 ± 32	110 ± 80	106 ± 8	100 ± 26
7	291 ± 97	824 ± 528	645 ± 225	$1{,}010 \pm 497$

through step 30; at deeper substrates the checks are thinner and drift larger (+7.3% at D4 with $n = 3$; +20.9% at D5 with a single run). At D6–D7 no direct convergence validation was conducted, but the monotonic D1–D5 trend (stable → bounded drift) supports treating deep-substrate per-generation estimates as conservative — actual per-generation costs at D6–D7 may be underestimated by up to roughly a fifth. The 30-generation total reported in Table 5.6 is JIT compile time plus 29 post-JIT generations: the harness evaluates generation 1 inside the JIT warm-up step and divides the remaining evolution time by generations − 1, so a 30-generation budget consists of the warm-up generation plus 29 further ones.

JIT compilation time itself grows with both depth and population size (Table 5.5). At shallow depths the overhead is manageable (D1: 86 s, D2: 321 s for Pop 1000), but at depth 6, compilation alone takes nearly 4 hours before any evolutionary step executes. Contrary to the expectation that XLA compilation should depend only on tensor shapes (which are depth-determined), JIT time also scales with population. Across depths D3–D7, Pop 1000 JIT times are approximately 18–23× larger than Pop 50 at the same depth, indicating a near-linear population scaling factor layered on top of depth-driven exponential growth. This likely reflects unrolling of the outer population loop during compilation, or memory allocation patterns that grow with population count.

Table 5.5: JIT compilation time (seconds) by depth and population (RTX 2080 Ti). JIT scales near-linearly with population and exponentially with depth through D5 (∼3× per level), before entering a plateau at D6–D7. JIT compile time measured at the start of each evolutionary run (D1–D7).

D	P50	P100	P150	P200	P250	P300	P500	P750	P1000
1	19	22	26	31	98	35	44	61	86
2	29	52	71	91	228	112	169	237	321
3	55	102	149	213	386	265	464	701	975
4	152	289	436	634	1,113	741	1,423	2,175	3,110
5	437	857	1,327	1,903	3,753	2,202	4,467	6,728	9,986
6	708	1,241	2,004	2,538	4,447	3,124	6,024	9,008	13,700
7	740	1,302	2,100	2,827	4,610	3,328	6,356	9,498	16,151

Table 5.6: JAX-ESHN total run time (seconds) for 30-generation evolution by depth and population (RTX 2080 Ti, mean ± std, $n = 3$ seeds per cell). Per-population solve rates are reported separately in Figure 5.5. P250 and P300 are non-monotonic relative to neighboring populations due to a JIT-compile-cache outlier in one P250 seed and a different GPU slot used for the P300 campaign; the deviation does not affect the qualitative scaling pattern.

D	**P50**	**P100**	**P150**	**P200**	**P250**	**P300**	**P500**	**P750**	**P1000**
1	138 ± 20	224 ± 18	308 ± 19	455 ± 5	652 ± 89	569 ± 24	899 ± 2	1,337 ± 22	2,097 ± 8
2	405 ± 9	759 ± 12	1,045 ± 42	1,539 ± 81	2,731 ± 359	2,140 ± 77	3,294 ± 239	4,837 ± 129	6,177 ± 118
3	1,136 ± 32	2,406 ± 75	3,604 ± 127	4,795 ± 327	7,513 ± 864	6,206 ± 378	10,822 ± 358	13,953 ± 187	19,695 ± 770
4	4,080 ± 254	7,544 ± 768	10,377 ± 689	15,132 ± 563	23,939 ± 336	15,370 ± 805	26,013 ± 2,843	36,212 ± 3,390	53,938 ± 1,594
5	14,085 ± 3,217	23,460 ± 616	44,037 ± 3,259	56,326 ± 1,683	92,483 ± 7,849	54,595 ± 3,423	97,056 ± 27,702	117,700 ± 6,686	212,659 ± 19,977
6	27,110 ± 9,305	45,380 ± 9,282	59,270 ± 13,904	100,832 ± 9,527	209,458 ± 76,547	105,838 ± 7,160	150,009 ± 31,244	292,604 ± 49,005	325,370 ± 19,509
7	20,673 ± 5,119	49,006 ± 10,641	82,481 ± 28,796	96,657 ± 30,298	201,718 ± 64,575	105,341 ± 21,868	212,423 ± 33,735	251,049 ± 47,752	499,874 ± 140,856

Figure 5.7 presents the runtime comparison visually, with interquartile ranges (IQR) across the population sizes tested for each implementation (8 for the Baseline, 9 for JAX-ESHN) and for JIT compilation alone.

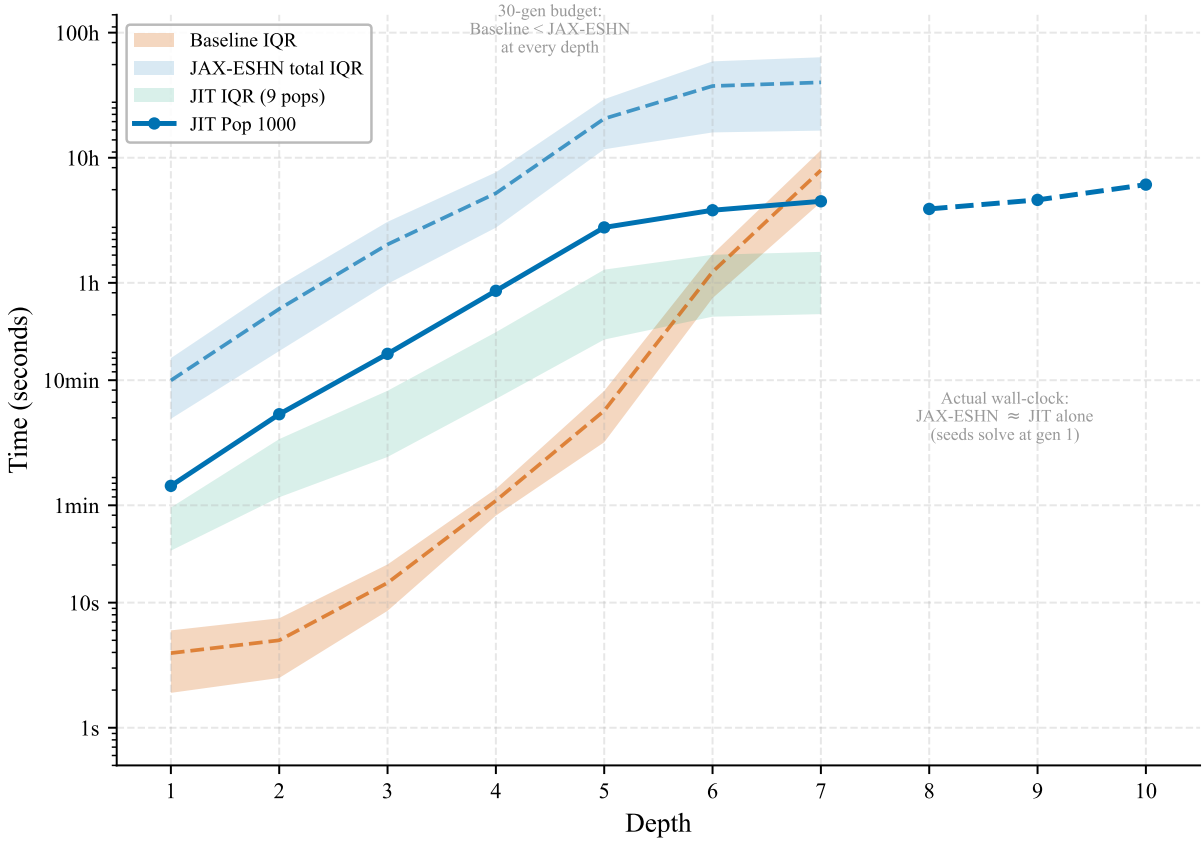


Figure 5.7: Total runtime comparison (log scale, D1–D7). Shaded bands are IQR across population sizes ($n = 3$ per cell): Baseline CPU (vermillion, 8 pops, 30 generations), JAX-ESHN GPU (blue, 9 pops, 30-generation budget), and JIT alone (green, 9 pops). The blue line traces Pop 1000 JIT compile time: solid through D7, dashed for D8–D10 (separate 300-generation campaign). At a 30-generation budget, JAX-ESHN's total exceeds Baseline at every tested depth: the per-generation slowdown (Figure 5.11) is no longer offset by JIT amortization. In practice, JAX-ESHN solves XOR at generation 1 for most configurations at $D \geq 4$, so the wall-clock time actually spent per run reduces to approximately the JIT compile time (green band) — which alone reaches hours at deep substrates and dominates user experience.

Figure 5.8 disaggregates JIT compilation time by population for all nine population sizes and shows near-linear population scaling at most depths. A pronounced plateau appears at D6–D7, where JIT growth per depth level slows from ∼3× (D4→D5) to 1.37× (D5→D6) and 1.18× (D6→D7). The dashed extension at D8–D10 comes from a separate 300-generation Pop-1000-only campaign ($n = 3$) and is plotted without a connecting line to D7 because the two datasets used different measurement conditions. The apparent drop in the D8 marker below D7 in Figure 5.8 is an artifact of this dataset switch, not a real scaling reversal: the 300-generation campaign yielded somewhat lower per-depth JIT times than the multi-depth measurement campaign, so the two series cannot be compared vertex-to-vertex. Within the 300-generation campaign itself, JIT time grows

monotonically through D10.

The JIT cost amortizes quickly within JAX-ESHN at deeper substrates, but as Section 5.6.3 establishes, internal amortization does not translate into a wall-clock advantage over Baseline. Figure 5.9 shows the break-even generation count: how many post-JIT generations are needed to amortize the compilation cost. All configurations amortize well within the 30-generation benchmark limit (maximum break-even ≈ 5 generations at D1).

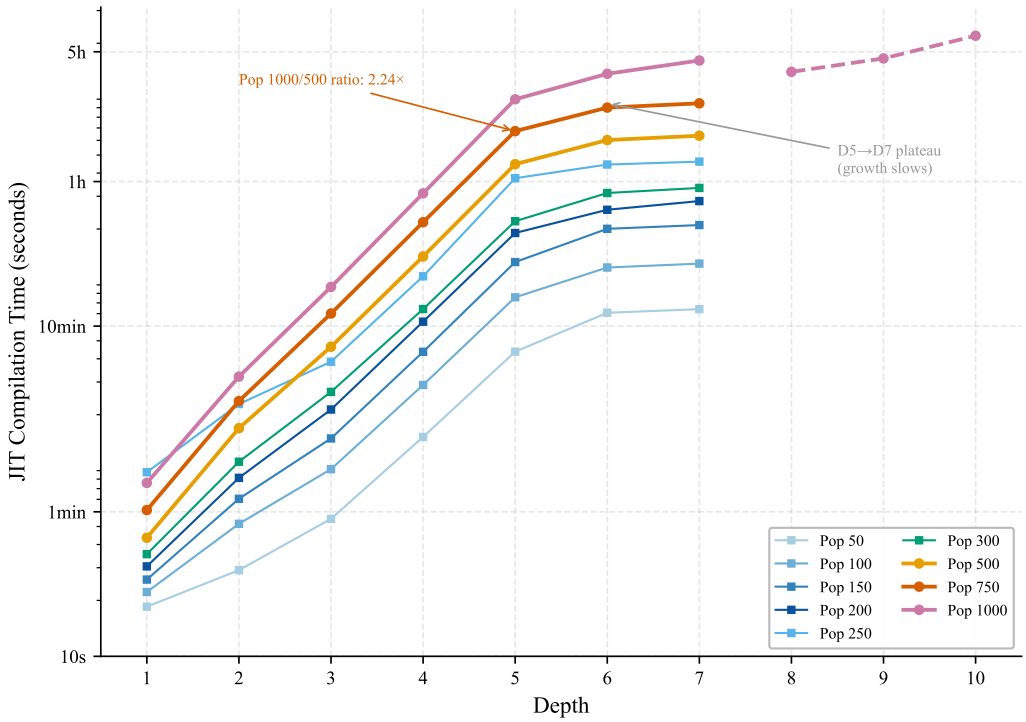


Figure 5.8: JIT compilation time vs. depth across 9 populations (D1–D7); Pop 1000 D8–D10 (dashed) come from a separate 300-generation campaign ($n = 3$) and are plotted as a disconnected extension. Scaling is near-linear with population (Pop 1000/Pop 500 = 2.24× at D5; 1.9–2.5× across D1–D7) and exponential with depth through D5, plateauing at D6–D7.

5.5.7 Statistical Validation

To move beyond the qualitative patterns visible in Figures 5.7–5.9, we subjected the full 456-trial Baseline dataset (7 depths × 8 population sizes × 3 replications × up to 3 iteration levels) to three confirmatory tests that quantify the depth–cost relationship central to the bottleneck hypothesis.

- **Depth as the dominant cost driver.** A one-way ANOVA on generation time with depth as the factor produces $F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$. Over half of the variance in per-generation runtime is attributable to depth alone, consistent with the $\mathcal{O}(4^d)$ position growth of Equation 5.2.

- **Depth-dependent solve rates.** A chi-square test of independence between depth and solve rate yields $\chi^2 = 42.73$, df $= 6$, $p < 10^{-7}$, establishing that the probability of solving XOR within 30 generations is not uniform across depths. This effect is

distinct from the timing result above: deeper quadtrees are not only slower but also less reliable.

- **Population as a partial mitigator.** A second chi-square test ($\chi^2 = 45.04$, df $= 7$, $p < 10^{-7}$) confirms that larger populations improve solve rates (14.0% at population 50 versus 59.6% at population 1000, $r = 0.27$, a 4.3× improvement). However, this moderate correlation indicates that population size compensates for, but does not eliminate, the depth-induced difficulty.

Pairwise correlation analysis sharpens the diagnosis further. The strongest relationship is between the number of candidate positions, set by depth via the 4^d branching factor, and generation time ($r = 0.73$), providing direct statistical evidence that quadtree expansion is the primary bottleneck, not CPPN evaluation or fitness computation per se. The weaker depth–time correlation ($r = 0.44$) reflects the nonlinearity of the depth–time relationship: runtime tracks the 4^d position count rather than depth itself, so the position count is the better linear predictor; while the population–solve rate correlation ($r = 0.27$) echoes the chi-square result above. A polynomial regression model incorporating both depth and population as predictors achieves $R^2 = 0.95$ (5-fold cross-validation: 0.85 ± 0.11), indicating that the computational overhead is almost entirely deterministic and predictable from these two experimental factors. This predictability is itself a key finding: it means the bottleneck is structural, not stochastic, and therefore cannot be resolved by longer runs or luckier initializations. The EMR reformulation presented in Chapter 6 exploits this predictability by pre-computing the full 4^d position grid eagerly, trading redundant computation for complete parallelizability.

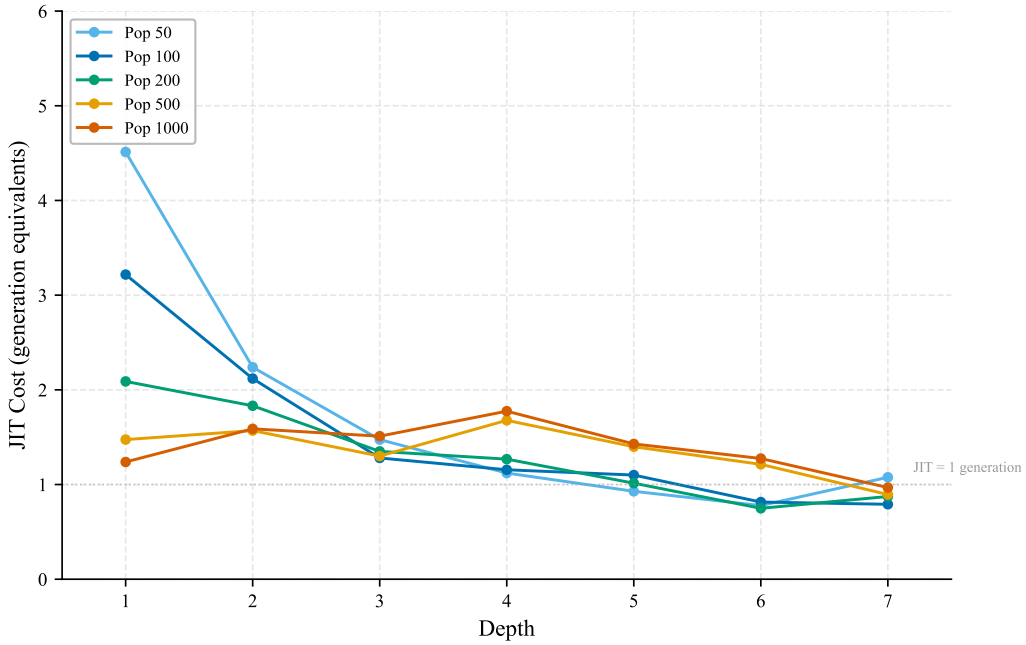


Figure 5.9: JIT amortization ratio (JIT compilation time ÷ post-JIT per-generation time) for five representative populations. A ratio of 1.0 means the one-time JIT cost equals exactly one post-JIT generation; values below 1.0 indicate JIT costs less than a single generation. All plotted configurations amortize within five generations.

5.5.8 Multi-Benchmark Validation

The statistical tests above establish the quadtree bottleneck on XOR with high confidence, but they leave one natural objection unanswered: perhaps the scaling exponents we have measured are a property of XOR's fitness landscape rather than of ES-HyperNEAT's discovery algorithm. If the bottleneck vanished on other tasks, or if its magnitude varied with input dimensionality or output type, the structural-incompatibility argument would need a much narrower formulation. To probe this possibility, we re-ran the benchmark protocol on four additional tasks that span the axes along which XOR is unusual: Parity-3 (a harder Boolean problem), circle classification (continuous binary with a non-linear decision boundary), sine regression (continuous regression with a single input), and cart-pole control (reinforcement learning with four inputs and episode-based fitness). Each task was run at depths 2–4 with 30 seeds per condition and population 150, for a total of 720 runs across both implementations (360 Baseline and 360 JAX-ESHN, all 30 seeds completed at every depth).

The $d_2 \rightarrow d_4$ column of Table 5.7 reflects per-run wall time, which for JAX-ESHN is dominated by JIT compilation at every depth. The steady-state picture (how costly a single generation is once the substrate has been JIT-compiled) tells the complementary story required for a fair runtime comparison. Table 5.8 reports this figure for each task at depth 4, alongside the one-time JIT cost that must be amortized before the per-generation speedup becomes net-positive.

Two patterns emerge that bear directly on the structural claim. The first pattern is the stability of the Baseline scaling exponent across fundamentally different fitness landscapes. Parity-3, circle classification, and sine regression differ in input dimensionality, output type, and loss shape, yet their Baseline depth-scaling ratios cluster tightly at 65–71×, the same order as the ≈7×-per-depth-level scaling measured on XOR in Table 5.4 (≈49× over two depth levels). Because the quadtree traversal dominates wall-clock time at every depth level tested, the scaling exponent is governed almost entirely by how many positions the quadtree allocates and almost not at all by the task being solved. Cart-pole's 377× ratio is the sole outlier, and it remains consistent with the structural claim: the scaling exponent now measures *quadtree cost compounded with RL episode length*, and the compounding is itself depth-dependent because deeper substrates run longer episodes before triggering termination. On a per-generation basis the Baseline's cart-pole cost grows 177× from D2 to D4; the 377× per-run figure is larger because deeper substrates also take more generations to solve.

The second pattern, visible in the JAX-ESHN columns of Table 5.7, mirrors the first from the opposite side of the implementation boundary. JAX-ESHN's depth-scaling ratios on the same four tasks collapse into a narrow 4.8–5.1× band that has no relationship to the fitness landscape and an obvious relationship to JIT compilation time: the implementation is bounded by how quickly XLA can trace a deeper computation graph, not by how quickly the resulting graph evaluates. The cart-pole exception is again revealing. Its 2.5× ratio is lower than the other tasks because each generation already incurs heavy episode-evaluation overhead that swamps the incremental depth cost, flattening the curve at the top. Both implementations, in other words, scale according to their own dominant cost center (quadtree traversal for Baseline, JIT compilation for JAX-ESHN), and neither scaling exponent depends on the task. This is exactly the task-independence that the structural argument of §5.1 predicted, and it extends the XOR-only evidence of the preceding subsections into something closer to a general result for coordinate-based adaptive substrate discovery.

The solve-rate columns in Table 5.7 tell a different and more delicate story. The two implementations reach opposite conclusions about which of these tasks is actually tractable. Parity-3 is mostly solved by the Baseline (76.7–100%) yet remains out of reach for JAX-ESHN, which tops out at 13.3–23.3% (Fisher's exact $p < 10^{-6}$ at each depth). Sine regression runs the split in reverse: JAX-ESHN clears the success threshold on almost every seed (96.7–100%) while the Baseline never crosses it at all ($p < 10^{-15}$). Circle classification defeats both systems at every depth tested. These disagreements are precisely what the framework in §5.5.2 predicts: the NEAT-library defaults driving each implementation classify the same task as solvable or unsolvable independently of the hardware. The Parity-3 and sine-regression reversals are properties of the NEAT configurations, not of CPU-vs-GPU execution; only the scaling exponents transfer cleanly across the implementation boundary.

The multi-benchmark runs convert the XOR result from a single-task observation into a scaling claim that generalizes across task types. The quadtree's per-depth cost ratio depends on the discovery algorithm, not on the problem being solved, and the 4.8–5.1× JIT-bound growth of JAX-ESHN confirms that no amount of vectorization at the implementation level can flatten the exponent without changing the discovery algorithm itself. The remainder of this chapter builds on that conclusion.

Table 5.7: Solve rate and depth-scaling ratio for the Baseline and JAX-ESHN implementations on four tasks beyond XOR. Each cell reports the fraction of 30 seeds that solved the task within the 100-generation budget (or within the task-specific success threshold for sine regression and cart-pole). The rightmost column gives the ratio of mean per-run wall-clock time at depth 4 relative to depth 2, which isolates the depth-scaling exponent from absolute runtime. The Baseline ratios cluster in the 65–71× range for regression and classification tasks, the same order as the scaling measured on XOR; JAX-ESHN's ratios cluster between 4.8× and 5.1× and reflect JIT-dominated rather than traversal-dominated growth. Cart-pole sits outside both clusters because each fitness evaluation involves an RL episode, compounding the per-depth quadtree cost with episode length.

	Baseline				**JAX-ESHN**			
Task	D2	D3	D4	d2→d4	D2	D3	D4	d2→d4
Parity-3	76.7%	100%	96.7%	71×	13.3%	23.3%	16.7%	4.9×
Circle	0%	0%	0%	65×	0%	0%	0%	4.8×
Sine	0%	0%	0%	67×	96.7%	100%	100%	5.1×
Cart-pole	100%	100%	100%	377×	100%	100%	100%	2.5×

Table 5.8: Steady-state (post-JIT) per-generation cost at depth 4 (Pop 150, $n = 30$ per cell). Baseline per-generation time is the mean of AVG_GEN_TIME_S across runs; JAX-ESHN per-generation time excludes JIT compilation. The JIT column lists the one-time compilation cost that must be paid before JAX-ESHN's per-generation advantage applies. The speedup column is the ratio of Baseline to JAX-ESHN per-generation time. Once JIT is amortized, JAX-ESHN runs 1.2–6.1× faster per generation across tasks; the exact speedup depends on whether the task's fitness evaluation is itself cheap (Parity-3) or heavy (Cart-pole).

Task	Baseline/gen	JAX post-JIT/gen	JIT (one-time)	Speedup
Parity-3	401.4 s	65.9 s	419.3 s	6.1×
Circle	188.1 s	78.1 s	384.6 s	2.4×
Sine	163.5 s	65.8 s	328.8 s	2.5×
Cart-pole	1,175.7 s	964.9 s	1,579.3 s	1.2×

5.6 Analysis: The Impossibility Result

5.6.1 The Fundamental Limitation

The experimental evidence converges on a single conclusion: ES-HyperNEAT's adaptive quadtree creates an insurmountable barrier to GPU acceleration. The barrier is not computational cost per se (individual substrate constructions can be accelerated) but *heterogeneity*: each CPPN in the population produces a unique topology, and no fixed-shape representation can capture all topologies simultaneously. The 1.65× speedup from within-individual batching (Section 5.3) and the $O(1)$ HSHG queries (Section 5.4) both improve individual operations but leave the population-level sequential loop untouched.

The HSHG results sharpen the diagnosis further. Replacing the quadtree with a GPU-friendly data structure does not suffice: eager position pre-generation undermines the variance-driven selectivity that produces useful topologies (29–63 hidden nodes vs. 1–2). The quadtree's lazy, variance-driven refinement is not an implementation detail to be optimized away but a core algorithmic property that any replacement must preserve. The implication, that a fundamentally different substrate-discovery mechanism is required, is the key contribution of this chapter and the premise on which EMR-HyperNEAT (Chapter 6) is built.

5.6.2 JIT Compilation Plateau

An unexpected secondary finding is a plateau in JIT compilation time. Through depth 5, compilation time grows exponentially at approximately 3× per depth level (Pop 1000: 3,110 → 9,986 s, 3.21×). At D6–D7, growth slows sharply: 1.37× (D5→D6) and 1.18× (D6→D7). This pattern is consistent across all nine population sizes in the 10-generation timing reruns. Several mechanisms could explain this plateau:

- **Saturation of the XLA graph.** Beyond a certain size, the traced graph's tracing cost no longer grows meaningfully with the number of new positions: additional leaves enter the graph but the work XLA does per leaf has already flattened.
- **Fixed-cost dominance.** Compilation phases whose cost is independent of graph

size (startup, pass scheduling, optimization-pipeline setup) begin to swamp the parts that do scale, so the depth-dependent portion of total JIT time shrinks as a share.

- **Redundant-structure reuse.** The XLA compiler can plausibly detect quadtree sub-structures it has already traced and short-circuit the repeated work, especially once deeper levels are largely repetitions of shallower ones.

- **Memory-system ceilings.** Cache residency and memory bandwidth during compilation are not free scalings; once those limits are hit the compiler can no longer trade more positions for more throughput at the same rate.

Pop 1000 experiments from the original 300-generation campaign extend through D10 (dashed in Figure 5.8) and independently corroborate the finding: growth ratios remain subdued throughout D8–D10 (1.18× at D8→D9, 1.32× at D9→D10), well below the ∼3× per depth observed at D1–D5. Absolute times from this campaign confirm the plateau: D8 = 14,034±766 s (∼3.9 h), D9 = 16,575±473 s (∼4.6 h, only +18% over D8 despite 4× more positions), and D10 = 21,955 ± 332 s (∼6.1 h, +32% over D9, within the plateau regime). The two benchmark series were conducted weeks apart under different system conditions, yet both exhibit the same plateau, strengthening confidence that the phenomenon reflects a genuine property of XLA compilation rather than a measurement artifact. Regardless of its cause, the plateau is a partial reprieve rather than an additional barrier: from D6 through D10 the JIT cost grows at 1.18–1.37× per depth level, well below the ∼3× per level seen at D1–D5 and far below the 4^d growth in quadtree positions. Bounded growth in this regime eases the dominant cost driver of JAX-ESHN within the tested depth range. The trade-off is for performance modeling: a single-growth-rate exponential fit $\text{cost}(d) = a\,b^d$ calibrated on shallow-depth data will over-predict cost throughout the plateau, since the empirical b decreases with depth. The plateau's behavior beyond D10 is not characterized by the present experiments.

5.6.3 Post-JIT Performance Characteristics

Despite the JIT overhead, the compilation cost amortizes rapidly: break-even occurs within five generations at all depths (Figure 5.9). However, the per-generation cost remains substantial because each individual's unique substrate graph must still be processed sequentially. At depth 7 with Pop 1000, each post-JIT generation takes 16,680 s, exceeding the Baseline's 2,021 s per generation. The JIT investment is amortized within JAX-ESHN within five generations but does not translate into a wall-clock advantage versus Baseline: even with JIT cost fully absorbed, JAX-ESHN's per-generation processing remains 4–8× slower than Baseline at deep substrates due to the sequential individual loop, so the 30-generation total exceeds Baseline at every depth tested. The exception is the gen-1-solve regime: at $D \geq 4$ with Pop ≥ 750, all seeds solve XOR in the warm-up generation that JIT compilation already covers (Section 5.5.4), so the actual wall-clock cost reduces to the JIT compile time alone — a regime where JAX-ESHN is a few times faster than Baseline but still far short of order-of-magnitude practical acceleration (see the total wall-clock comparison below).

Figure 5.10 visualizes the per-generation cost surface, showing approximately linear scaling with population at each depth and super-linear scaling with depth. The speedup ratio (Baseline per-generation time divided by JAX-ESHN post-JIT per-generation time) is below 1.0 at all tested XOR configurations (Figure 5.11), confirming that the Baseline

is faster per generation at every depth and population tested on XOR. The gap narrows at deeper substrates (from ∼500–800× slower at D3 to ∼4–8× slower at D7), but on the XOR grid JAX-ESHN never achieves per-generation parity due to the sequential individual processing constraint; the four cross-task benchmarks, measured in a separate campaign, show the opposite per-generation ordering (Table 5.8).

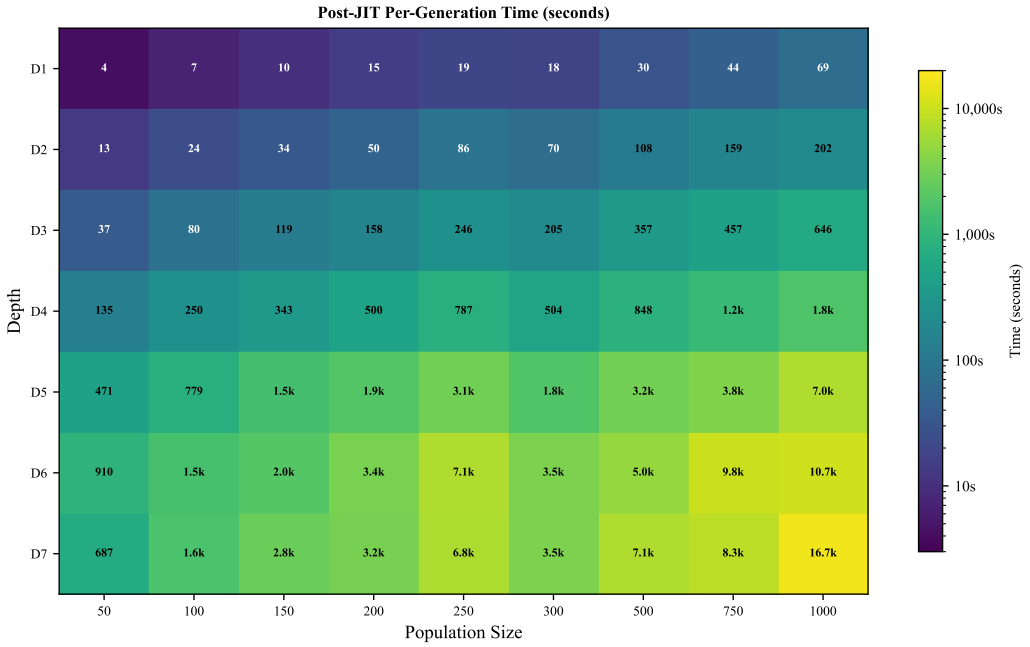


Figure 5.10: Post-JIT per-generation time across depth and population. Values range from 4.1 s (D1, Pop 50) to 16,680 s (D7, Pop 1000), scaling approximately linearly with population at each depth.

JAX's persistent compilation cache does not help here, because each CPPN's unique topology produces unique tensor shapes that require separate compilation traces. The practical implication is that JAX-ESHN provides no consistent wall-clock advantage over Baseline at the depth/population grid tested (8.2× slower at 30-generation budget, 3.75× faster only in the gen-1-solve regime); its value lies in the JIT compilation plateau (Section 5.6.2) and the architectural insights that inform the EMR reformulation presented in Chapter 6.

Total wall-clock comparison. Two comparisons against Baseline are relevant: a matched 30-generation budget that quantifies JAX-ESHN's full cost surface, and the actual wall-clock spent given JAX-ESHN's gen-1-solve behavior at deep substrates. Under a 30-generation budget at depth 7 with population 1000, the Baseline averages 1,010 minutes (Table 5.4) while JAX-ESHN's projected total averages approximately 8,330 minutes (Table 5.6) — an ∼8.2× slowdown rather than a speedup, because the per-generation cost is JAX-ESHN's binding constraint. In practice, however, JAX-ESHN does not run to a 30-generation budget at this configuration: all seeds solve XOR in the warm-up generation that JIT compilation already covers, so the actual wall-clock cost reduces to the JIT compile time alone (≈ 269 minutes), giving JAX-ESHN a ∼3.75× speedup over Baseline in actual wall-clock terms — still below the order-of-magnitude improvement that would justify the reimplementation effort as a practical solution. Either way, the conclusion is

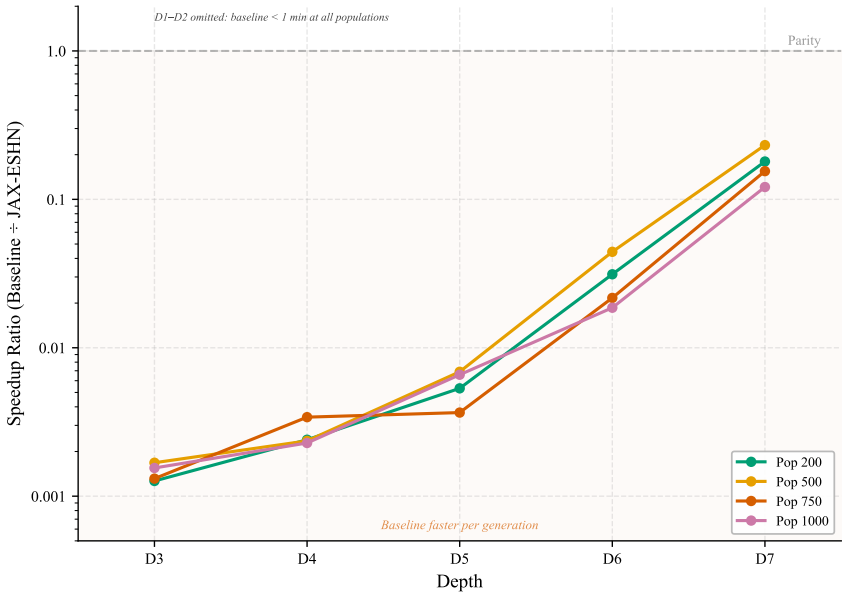


Figure 5.11: Per-generation speedup ratio (Baseline ÷ JAX-ESHN post-JIT) for D3–D7. Ratios below 1.0 indicate the Baseline is faster; on the XOR sweep, JAX-ESHN never reaches per-generation parity due to sequential individual processing. D1–D2 and Pop 50–100 are omitted (sub-minute runtimes).

the same: even three seeds per (depth, population) cell cost hours of wall-clock time, explaining the three-replication design, and the JAX-ESHN reimplementation was not pursued as a practical acceleration but as a diagnostic tool whose architectural insights (particularly the identification of topology uniqueness as the fundamental barrier) directly informed the EMR redesign of Chapter 6.

5.6.4 Implementation Comparison Caveats

The framework set out in §5.5.2 applies throughout this analysis and the constructive arguments of Chapter 6. Scaling exponents are cross-implementation properties of the hardware and the parallelization strategy; absolute solve rates depend on each implementation's NEAT configuration and read only within an implementation. Every numerical claim above respects this rule.

5.6.5 Additional Implementation Optimizations

Beyond the four quadtree-level optimizations (O1–O4) documented in Section 5.3, the JAX-ESHN implementation uses six further optimizations targeting the evaluation pipeline. Two are architecturally significant; the remainder are minor but collectively contribute to the overall performance profile.

O5: Hierarchical Multi-Resolution Grid. The quadtree's recursive data structure is replaced with a pre-computed fixed-size hierarchical grid, where all positions at each resolution level are stored as dense JAX arrays (pseudocode in Appendix C.13, Algorithm 19). Level 0 contains 4 cells, Level 1 contains 16, Level k contains 4^{k+1}, with parent–child relationships pre-computed as static index arrays. This enables batch CPPN queries via

`vmap` over all positions at a given level simultaneously, eliminating per-cell overhead from recursive node creation and traversal.

O6: GPU-Resident Multi-Generation Loop. The standard evolutionary loop alternates between GPU computation (CPPN evaluation, fitness) and CPU coordination (selection, logging) every generation. O6 wraps the entire generation step (ask, transform, CPPN queries, variance computation, network evaluation, tell) into a pure JAX function executable inside `jax.lax.while_loop` (pseudocode in Appendix C.13, Algorithm 20). This eliminates per-generation host–device synchronization, keeping the evolution loop GPU-resident until a fitness threshold is reached or a generation limit is hit.

O7–O8: Unified Queries and Sparse Evaluation. O7 (Unified Input→Output Query) batches all source–target coordinate pairs into a single double-`vmap` CPPN call, replacing thousands of individual Python function calls with one vectorized tensor operation. O8 (Sparse Forward Pass) computes an activity mask across all genomes and slices weight matrices to include only active hidden positions, reducing matrix multiplication size proportionally to the spatial sparsity of the discovered substrate.

Minor optimizations. O9 (Batched Metrics Extraction) consolidates variance and mask computation into single-pass JIT-compatible functions that stay entirely on-device, eliminating intermediate CPU–GPU synchronization points. O10 (JIT-Compatible Static Metadata) stores hierarchical grid metadata (level counts, offsets, grid dimensions) as Python tuples rather than JAX arrays, enabling their use inside `jax.lax.while_loop` without concretization errors, a prerequisite for O6.

5.6.6 Architectural Limitation: Two-Layer Substrates

The JAX-ESHN implementation supports only Input→Hidden→Output substrates (no Hidden→Hidden connections). The original ES-HyperNEAT discovers three connection types: Input→Hidden, Hidden→Hidden, and Hidden→Output. The Hidden→Hidden layer is explored `iteration_level` times. This implementation omits Hidden→Hidden connections because iterative propagation through recurrent hidden layers breaks JAX `vmap` vectorization: each propagation step depends on the previous, requiring sequential execution incompatible with batch parallelism.

As a consequence, the `iteration_level` parameter has *no effect* in the JAX-ESHN implementation. This trade-off limits substrate expressiveness (problems requiring deep compositional reasoning through multiple hidden layers cannot be solved), but does not affect benchmark tasks (XOR, simple classification) that are solvable with two-layer capacity. Chapter 6 (§6.5.11) later quantifies the empirical contribution of H→H connectivity once EMR makes recurrence cheaply parallelizable: enabling H→H roughly doubles XOR's solve rate, confirming that H→H provides search-space enrichment without being strictly required.

5.6.7 Implications for the Thesis Roadmap

The impossibility result carries consequences that extend beyond this chapter. The 2,013-trial campaign of Chapter 3 was already CPU-constrained at depth 3; repeating it at

depth 5 would require roughly 50× more wall-clock time. The 13-expert partitioned-experts architecture of Chapter 4 applies a 13× multiplier on top of that. And the GEENNS vision (Section 1.6) requires evolving several dozen CPPNs (34 for a modest 5-grid, 10-task configuration per the $4 + k(k-1)/2 + 2T$ formula in Appendix B), making the combined cost at quadtree speeds entirely infeasible within any practical timeline. Even the best-case 1.65× optimization would merely reduce this from prohibitive to still-prohibitive.

This impossibility motivates the lazy-to-eager redesign developed in Chapter 6. Rather than asking "where should I look?" before evaluating the CPPN (the quadtree approach), the eager method evaluates the CPPN on a fixed multi-resolution grid and then asks "what did I find?" afterward. Because the grid is fixed, every individual in the population produces tensors of the same shape, enabling full `vmap` parallelization and achieving a 12–34× per-generation GPU speedup on XOR at depths 5–7 (population 1000): the kind of acceleration that the quadtree's variable-topology output fundamentally prevents. The eager approach trades redundant computation (evaluating all 4^d positions when only a fraction survive filtering) for the uniform tensor shapes that GPU hardware requires. Chapter 6 quantifies this tradeoff and shows that the redundancy cost is far smaller than the parallelism gain.

5.7 Synthesis: From SIMT Incompatibility to Architectural Replacement

This section interprets the findings within the broader thesis narrative: what the incompatibility result means for indirect encoding in general, what it reveals about the tension between adaptive discovery and hardware parallelism, and how it motivates the architectural replacement developed in Chapter 6.

5.7.1 The Incompatibility as a Design Constraint

The core finding is not merely that the quadtree is slow: it is that the quadtree's adaptive substrate discovery is *fundamentally incompatible* with the SIMT parallelism that makes GPUs fast: each CPPN produces a unique variance landscape refined into a unique topology (different node counts, different spatial positions, different connection structures), leaving no population-level tensor of uniform shape to vectorize over. The four optimizations attempted in §5.3 apply the principal implementation-level improvements available to the quadtree traversal and yield a cumulative speedup of only 1.65× on the reference XOR-at-depth-2 configuration (the magnitude is task-, depth-, and population-dependent), confirming that the bottleneck is architectural, not an implementation issue. The multi-benchmark validation in §5.5.8 extends this finding to four tasks beyond XOR — Parity-3, circle classification, sine regression, and cart-pole — and shows that each implementation's depth-scaling factor is determined by its dominant cost center (quadtree traversal for the Baseline, JIT compilation for JAX-ESHN), independent of task type, input dimensionality, or output structure. Better SIMT hardware will not fix this; the constraint is architectural within the SIMT paradigm, and only algorithmic redesign will resolve it.

The gen-1 solve phenomenon (Figure 5.6) makes the diagnosis especially sharp: at populations ≥ 750, XOR is solved *immediately* after substrate construction at D4–D7

(and Pop 1000 extends this to D10). The evolutionary search is not the bottleneck: the construction of the substrate is. This reframes the problem: if the search finds a solution in the first generation, then the entire JIT compilation cost (hours at deep substrates) is spent enabling a computation that completes in seconds.

This incompatibility compounds with the Chapter 4 results. The partitioned-experts architecture multiplies the number of ES-HyperNEAT instances by the expert count ($N_e = 13$), and each instance must run through the same CPU-bound quadtree. A 13× cost multiplier on an already slow algorithm makes the partitioned-experts pipeline impractical for iterative development, let alone for deployment in a system requiring dozens of specialist subnetworks.

5.7.2 Why HSHG Over-Discovers

The Hierarchical Spatial Hash Grid (HSHG) evaluation (§5.4) illustrates a deeper lesson about algorithmic replacement: data structure substitution is not algorithmic redesign. HSHG pre-computes spatial queries at all resolution levels, replacing the quadtree's lazy refinement with eager discovery. The result is *over-discovery*: 29–63 hidden nodes where the quadtree discovers 1–2 (§5.4).

The root cause of over-discovery is that HSHG queries a *fixed grid at every resolution level simultaneously*. At depth 7, this means querying at $4^0, 4^1, \ldots, 4^7$ positions across resolution levels, summing to 21,845 candidate positions (Equation 5.2). Any position where the CPPN produces non-trivial output becomes a candidate hidden node, regardless of whether a coarser-resolution node already covers the same spatial region. The quadtree avoids this by querying each resolution level only within regions where the *previous* level showed sufficient variance, a conditional refinement that automatically prunes spatially redundant nodes. HSHG lacks this conditional logic; it queries everywhere and keeps everything above threshold, producing the order-of-magnitude node inflation observed.

The excess nodes do not improve performance; they degrade it: the CPPN must now encode a coherent weight pattern across a far larger substrate. With 63 hidden nodes instead of 2, the number of expressed connections grows quadratically, so a single CPPN mutation must produce useful weights across roughly 4,000 connection sites at once, a search problem far harder than the original XOR task. This is the optimizer-topology mismatch: HSHG creates substrates that are appropriate for the task in terms of spatial coverage but severely overparameterized for NEAT's incremental complexification strategy.

The evolutionary pipeline confirms this structurally diagnosed failure: across four task variants (XOR at D2/D3, Parity-3, Sine), HSHG achieved 0 solves in 120 seeds (§5.4.7), establishing that the bloating is insurmountable even with adaptive NEAT-driven search.

The quadtree's laziness (its decision to subdivide, beyond the initial depth, only where CPPN variance exceeds a threshold) is therefore not an implementation detail. It is the mechanism that produces sparse, task-appropriate topologies. Any replacement must preserve this adaptive sparsity while enabling the fixed-shape tensor operations that GPU parallelism demands. The core tension is between *adaptivity* (discovering topology from the CPPN's variance landscape) and *vectorizability* (requiring uniform tensor shapes across the population).

5.7.3 Connecting Chapters 4 and 5: WHERE and WHY

Chapters 4 and 5 diagnose complementary facets of the same underlying problem.

Chapter 4 showed *where* evolution fails: the substrate covers only 28 of 784 pixel positions (3.6% of the input), missing the peripheral features that distinguish many digit classes. The diagnosis is spatial: evolution converges to a small region and cannot escape.

This chapter established *why* scaling the spatial resolution does not help: increasing quadtree depth should, in principle, enable finer-grained coverage, yet each additional depth level multiplies the diversity of topologies the population contains, preventing GPU vectorization and collapsing solve rates from 52.8% at depth 3 to 4.2% at depth 7. The computational cost of deeper quadtrees makes it impossible to scale the resolution needed to cover the full 784-pixel input.

Together, the two chapters establish a double bind: the substrate is too narrow (central bias) and the mechanism for making it wider is too expensive (quadtree scaling). The partitioned-experts architecture of Chapter 4 sidesteps the central bias by partitioning the input, but it *multiplies* the quadtree cost by $N_e = 13$. Solving both problems at once requires changes beyond input partitioning: the substrate discovery mechanism itself has to change, to something that scales without the quadtree's unique-topology problem. Chapter 6 provides exactly this.

5.7.4 Connection to the GEENNS Vision

The SIMT incompatibility is the barrier that made the GEENNS multi-grid architecture computationally infeasible before this thesis; at a modest (k=5, T=10) scale the architecture would require 34 CPPNs, and Table B.3 (Appendix B) derives the arithmetic.

Thesis-level limitations, including hardware scope and replication counts, are consolidated in Section 8.3.1.

5.8 Chapter Summary

This chapter answered TRQ2 and delivered the diagnostic half of TRQ3 by establishing that ES-HyperNEAT's quadtree-based substrate discovery is fundamentally incompatible with parallel hardware—a structural incompatibility, not an implementation shortcoming (§5.7). Chapter 6 supplies the constructive half of TRQ3. The result is drawn from 456 Baseline trials and 189 JAX-ESHN Tier 1 trials across seven depth levels, augmented by a four-strategy within-individual optimization study (O1–O4) on JAX-ESHN. Table 5.9 summarizes the key results.

The answer to TRQ2. The fundamental computational limitation of ES-HyperNEAT's substrate formation is the *topology uniqueness problem*: the quadtree refines each CPPN's variance landscape into a unique network topology with unique node counts, positions, and connections. These unique topologies prevent population-level vectorization and make computational cost scale exponentially with resolution depth, compounding the central bias diagnosed in Chapter 4 with a computational bottleneck that constrains the practical utility of any quadtree-based approach.

TRQ3 (diagnostic half). Massively parallel execution cannot overcome the quadtree bottleneck within the existing algorithmic framework. Four independent optimizations

Table 5.9: Summary of Chapter 5 key results.

Result	**Value**	**Section**
Depth–performance relationship		
Depth effect (ANOVA)	$F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$	§5.5
Baseline solve rate (D3, Tier 1)	62.5% (52.8% all iterations)	§5.5
Baseline solve rate (D7)	4.2%	§5.5
Solve rate variation (χ^2)	42.73, $df = 6$, $p < 10^{-7}$	§5.5
Regression model fit	$R^2 = 0.95$	§5.5
GPU acceleration attempts		
Cumulative optimization ceiling (reference configuration)	1.65×	§5.3
Post-JIT per-gen at D7 (XOR)	Baseline 4–8× faster (Pop 500–1000)	§5.6
JAX-ESHN Tier 1 solve rate	96.3% (182/189)	§5.5
Gen-1 solve rate (Pop $\geq$ 750)	100% at D4–D7	§5.5.4
JIT–population scaling factor	2.24× at D5 (Pop 1000/Pop 500)	§5.5.6
HSHG alternative		
Hidden node over-discovery	29–63 vs. 1–2	§5.4
Threshold mismatch (HSHG/Quadtree)	16.7× (0.03 vs. 0.5)	§5.4
Evolutionary solve rate (pop 50, 30 seeds)	0/120 across 4 tasks	§5.4.7
Experimental scope		
Total baseline trials	456	§5.5
Total JAX-ESHN Tier 1 trials	189	§5.5
Depth scaling (positions)	$\frac{4^{d+1}-1}{3}$: 5 (D1) to ∼1.4M (D10)	§5.5

yield a cumulative speedup of only 1.65× on the reference XOR-at-depth-2, population-500 configuration; the magnitude is task-, depth-, and population-dependent, and at deeper substrates JIT overhead inverts the gain so that JAX-ESHN's per-generation cost actually exceeds the Baseline. The ceiling is well below the order-of-magnitude improvement required for practical use. JAX accelerates individual CPPN evaluations on GPU, but this cannot be lifted to the population level because each individual produces a different topology, requiring a separate JIT compilation for each unique substrate shape. The HSHG alternative fails for a complementary reason: eager pre-computation eliminates the adaptive sparsity that makes the quadtree's topologies useful, producing 29–63 hidden nodes where 1–2 suffice and degrading solve rates. The bottleneck is not the data structure or the hardware but the *discovery approach*—the sequential, topology-dependent nature of adaptive substrate formation.

Transition to Chapter 6. This incompatibility motivates a rethinking of substrate discovery. If the quadtree cannot be accelerated because it produces unique topologies, and if it cannot be replaced by eager pre-computation because that eliminates adaptive sparsity, then the solution must reconcile both requirements: fixed-shape tensors for parallel execution *and* adaptive sparsity for task-appropriate topologies. Chapter 6 introduces EMR-HyperNEAT (Eager Multi-Resolution HyperNEAT), which resolves this tension by evaluating the CPPN on a fixed multi-resolution grid and filtering by variance threshold afterward, preserving adaptivity within a vectorizable framework and achieving a 12–34× per-generation GPU speedup over the quadtree baseline on XOR at depths 5–7 (population 1000).

Chapter 6

EMR-HyperNEAT: Eager Multi-Resolution Substrate Discovery

> Make it work, make it right, make it fast.
>
> Kent Beck

This chapter completes the answer to the thesis research question left open by Chapter 5:

> **TRQ3.** Can adaptive substrate discovery be redesigned for massively parallel execution without sacrificing adaptivity?

Chapter 5 proved that the quadtree's adaptive discovery is incompatible with massively parallel hardware: four optimization strategies exhaust at 1.65× on the reference configuration (§5.3), and HSHG (Hierarchical Spatial Hash Grid) over-discovers by an order of magnitude. The core tension between adaptive sparsity and fixed-shape tensors appeared irreconcilable.

This chapter resolves it. EMR-HyperNEAT inverts the evaluation order from *lazy* to *eager*: evaluate the Compositional Pattern-Producing Network (CPPN) everywhere on a fixed multi-resolution grid, then filter by variance afterward. The fixed grid produces uniform tensors by construction, enabling batched tensor vectorization across the entire population, while the variance filter preserves adaptive sparsity.

We present the EMR algorithm and its formal relationship to the quadtree (§6.1–§6.2), introduce a connection type taxonomy enabling systematic exploration of recurrence (§6.3), detail the implementation (§6.4), report empirical results across depth levels, connection types, and a real-world financial benchmark (§6.5), present cross-domain validation across 15 problems in 4 domains (§6.5.10), and analyze the scaling properties and limitations (§6.6). Section 6.7 synthesizes the findings within the broader thesis narrative, and §6.8 summarizes the chapter's contributions and completes the answer to TRQ3.

The work in §6.1–§6.6 is based on Paper 5 [14]. Author contributions for all papers are detailed in the List of Publications (p. 31).

6.1 The Eager Evaluation Hypothesis

Chapter 5 concluded with a negative result for TRQ3: massively parallel execution cannot overcome the quadtree bottleneck. The JAX implementation of the quadtree was unable to provide a consistent wall-clock advantage over the CPU baseline: under a matched 30-generation budget, JAX-ESHN ran slower than Baseline at every tested depth (Section 5.6.3), far below the order-of-magnitude speedups typical of GPU-accelerated workloads: roughly 10–70× for traditional deep learning training [10, 75] and up to 500× reported for GPU-accelerated neuroevolution on expensive control tasks [100]. The root cause was structural, not implementational: each CPPN produces a unique subdivision tree, and no implementation-level optimization can reshape unique topologies into the uniform tensors GPU kernels require.

The question is whether a different approach can preserve the quadtree's adaptive sparsity while producing the fixed-shape data structures GPUs demand. This chapter answers in the affirmative. We replace the quadtree's operational semantics (lazy, top-down, conditional traversal) with an eager, bottom-up, unconditional evaluation pipeline that we call Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT, or EMR) [14]. The core idea is a single change in the order of operations:

ES-HyperNEAT (Lazy): `subdivide_if(var > θ)`
EMR-HyperNEAT (Eager): `eval_all(); filter(var > θ)`

Under the lazy regime, the CPPN is queried only at positions whose parent region exceeded the variance threshold; under the eager regime, the CPPN is queried at every grid cell without consulting its parent, and the variance criterion is applied only after the full weight tensor has been populated. The inversion trades redundant queries for regular memory access patterns. It is valid because the CPPN output at coordinate (x, y) depends on the coordinate alone (the parent's variance carries no information that could change the value), and the resulting pointwise independence means every cell of the tensor can be filled in parallel. Adaptive sparsity survives either way: low-variance positions end up discarded regardless of whether they were reached conditionally or unconditionally. What changes is the control flow. The quadtree's top-down conditional descent, which Chapter 5 identified as incompatible with Single Instruction, Multiple Threads (SIMT) execution, is replaced by a bottom-up unconditional sweep followed by top-down masking, both expressed as uniform tensor operations that just-in-time (JIT) compilation can trace end-to-end.

Figure 6.1 contrasts the two approaches; Figure 6.2 provides an isometric visualization. ES-HyperNEAT (left) subdivides sequentially: each depth level must complete before children are evaluated, producing tree-shaped computations that differ per CPPN. EMR (right) precomputes regular grids at every depth level, evaluates all positions simultaneously, then applies variance filtering to produce a sparse substrate (identical to ES-HyperNEAT's under matched thresholds and substrate-selection rule; Theorem 6.3). The regular grid structure yields fixed-shape tensors amenable to JIT compilation and `vmap` vectorization.

Where Chapter 5 asked "can we accelerate the quadtree?" and found the answer was no, this chapter asks three questions that arise from abandoning the quadtree entirely:

RQ1. Does eager evaluation achieve GPU-parallel scalability while preserving the adaptive substrate discovery that makes ES-HyperNEAT effective?

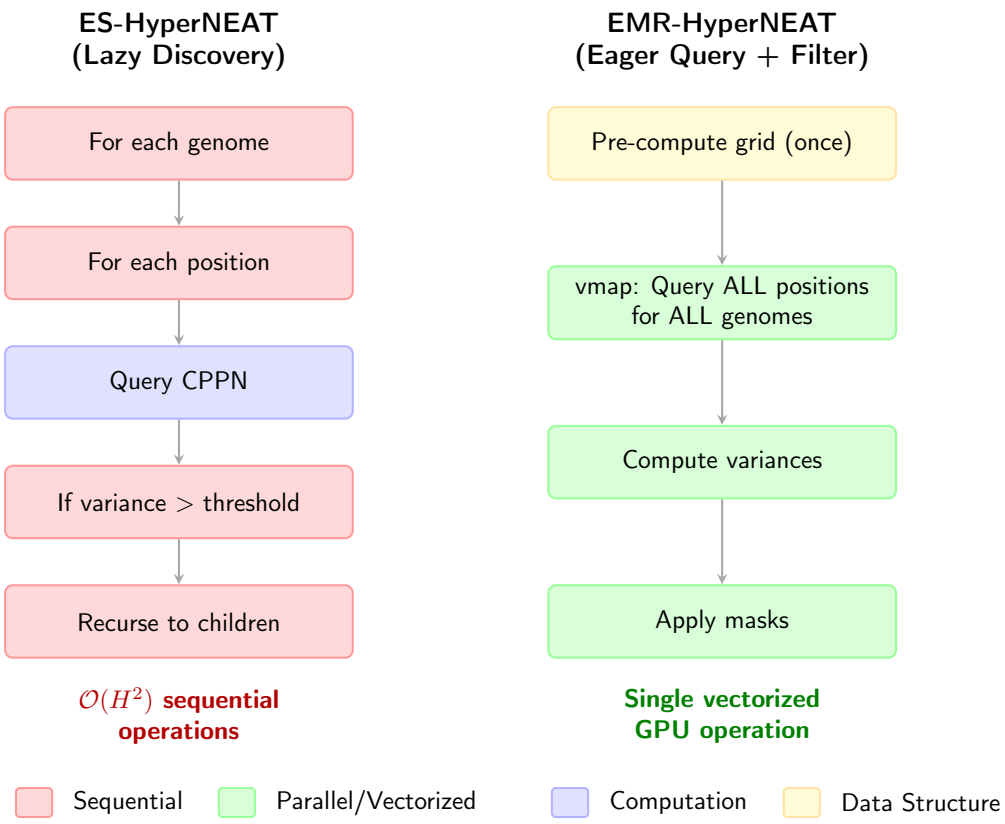


Figure 6.1: Lazy versus eager substrate discovery. **Left:** ES-HyperNEAT subdivides sequentially; each depth must complete before children are evaluated. **Right:** EMR-HyperNEAT evaluates all depths in parallel, then filters by variance. Both discover sparse substrates from CPPN variance, but EMR produces fixed-shape tensors suitable for GPU execution.

RQ2. What speedup does EMR achieve over ES-HyperNEAT at increasing substrate depths, and at what depth does the crossover occur?

RQ3. Does eager evaluation affect solution quality, or does the bottom-up variance computation discover equivalent (or better) substrates?

The following sections address each question in turn, and §6.8 returns to all three. Sections 6.2–6.3 describe the algorithm and its connection type taxonomy. Section 6.4 details the implementation and execution strategies. Section 6.5 presents the empirical evaluation, and Section 6.6 analyzes the scaling properties and trade-offs.

6.2 The EMR Algorithm

The quadtree analysis in Chapter 5 established two facts: the quadtree's variable-topology output is structurally incompatible with GPU parallelization, and four optimization strategies exhaust at a cumulative 1.65× ceiling on the reference XOR-at-depth-2 configuration (§5.3). This section presents the algorithmic redesign that resolves both problems. Rather than optimizing the quadtree's sequential traversal, we replace it entirely with a three-stage pipeline [14]: (1) unconditional evaluation of all grid positions followed by

variance-based filtering; (2) precomputation of static multiresolution grids that enable JIT compilation; and (3) decomposition of the discovery process into three vectorized stages amenable to `vmap`. Each stage produces fixed-shape tensors by construction: the structural property that the quadtree lacks and that Chapter 5 identified as the root obstacle. The following subsections formalize each component.

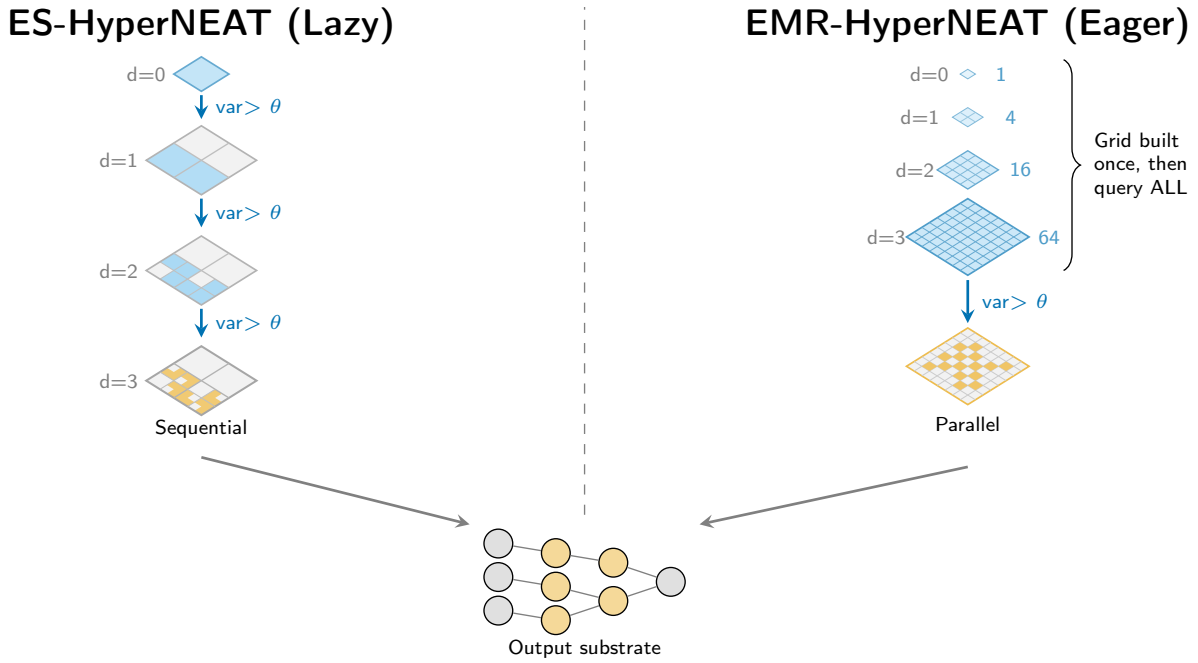


Figure 6.2: Isometric visualization of lazy versus eager substrate discovery. **Left:** ES-HyperNEAT subdivides sequentially from depth 0 to 3; blue cells continue subdividing, gray cells are pruned. **Right:** EMR-HyperNEAT evaluates all depths in parallel (brace), then filters by variance to produce a sparse active grid (orange cells). **Bottom:** schematic output substrate. Both algorithms apply the same parent-conditioned predicate, so under matched thresholds and substrate-selection rule, EMR-HyperNEAT and ES-HyperNEAT discover identical position sets (Theorem 6.3). The difference between the two algorithms is execution (EMR-HyperNEAT's eager fixed-shape grid versus the quadtree's sequential BFS) and substrate selection (EMR-HyperNEAT retains intermediate parents that the quadtree's leaf-only output discards), not the predicate itself.

6.2.1 Hierarchical Grid Definition

The quadtree constructs its spatial decomposition dynamically during evolution, producing a different tree for each CPPN. This data-dependent structure is the property that Chapter 5 proved prevents massively parallel execution. EMR replaces it with a single static grid computed once before evolution begins. The grid contains every position that a depth-D quadtree *could* visit, regardless of variance, preempting the data-dependent branching entirely.

Definition 6.1 (Hierarchical Grid)**.** *For maximum depth D, the hierarchical grid $\mathcal{G}_D$ contains all positions:*

$$\mathcal{G}_D = \bigcup_{d=0}^{D} \text{Grid}_d, \quad \text{Grid}_d = \left\{ \left(\tfrac{2i+1}{2^{d+1}} - 1,\ \tfrac{2j+1}{2^{d+1}} - 1 \right) : i, j \in [0, 2^{d+1}) \right\} \tag{6.1}$$

Each depth level d contributes 4^{d+1} uniformly spaced positions within $[-1, 1]^2$. The total position count across all levels follows a geometric series:

$$N_{\text{total}} = \sum_{d=0}^{D} 4^{d+1} = \frac{4^{D+2} - 4}{3} \tag{6.2}$$

Table 6.1 shows the concrete scaling. The grid data structure stores four components: position coordinates in $[-1, 1]^2$, depth indices, parent-child relationships as index arrays, and level offsets. Because every component has a shape determined solely by D, the entire structure is known at compile time, enabling JAX's JIT compiler to trace through all operations without recompilation. This is the structural property that the quadtree lacks and that Chapter 5 identified as the root obstacle to massively parallel execution.

Figure 6.3 illustrates the hierarchical grid structure, showing how each depth level subdivides the coordinate space into progressively finer grids.

Table 6.1: Hierarchical grid position counts for depths 1–13. Each level d contains 4^{d+1} positions; the total across all levels is $(4^{D+2} - 4)/3$. GPU memory scales linearly with total positions × population size (Section 6.6).

Depth D	Finest Level (4^{D+1})	Total Positions
1	16	20
2	64	84
3	256	340
4	1,024	1,364
5	4,096	5,460
6	16,384	21,844
7	65,536	87,380
8	262,144	349,524
9	1,048,576	1,398,100
10	4,194,304	5,592,404
11	16,777,216	22,369,620
12	67,108,864	89,478,484
13	268,435,456	357,913,940

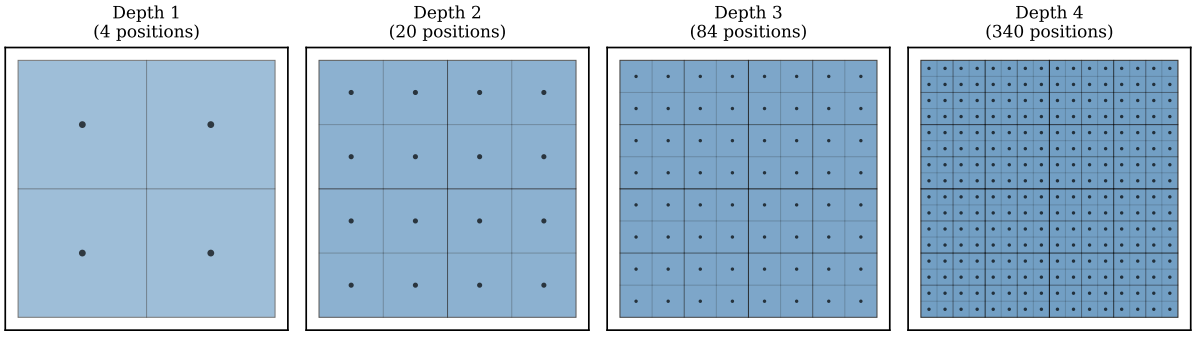


Figure 6.3: Hierarchical grid structure. Each depth level subdivides the $[-1, 1]^2$ coordinate space into a regular grid. Depth 0 contains 4 positions, depth 1 contains 16, and depth d contains 4^{d+1}. The complete grid $\mathcal{G}_D$ is the union of all levels and is precomputed once before evolution begins.

6.2.2 Variance-Based Filtering

Because the grid is fixed in advance, all CPPN queries are independent: the trivially parallel structure that quadtree branching eliminated [54]. Given one source s and a target set T of size N, a single `vmap` call produces the weight vector $\mathbf{W}$ in $\mathcal{O}(N/P)$ wall-clock time on P cores rather than $\mathcal{O}(N)$ sequentially. A second, outer `vmap` over the population extends this to all genomes at once, yielding a three-dimensional weight tensor indexed by genome, source, and target. This two-level vectorization is the direct answer to Chapter 5's finding that population-level batching is impossible when each genome produces a unique topology.

Variance computation mirrors the quadtree's division-step subdivision criterion (variance over the four immediate children) but operates on pre-existing data instead of conditionally generated data. For each position at depth $d \in [1, D-1]$, the variance is computed from the CPPN outputs at its four children at depth $d+1$. At the root layer ($d = 0$), where no parent layer exists above, a single "root variance" is computed over the four level-0 cells themselves, mirroring ES-HyperNEAT's root-variance gate. EMR uses this 4-child variance from depth 1 onward; ES-HyperNEAT switches to variance over all leaf descendants in its later pruning step (discussed in the "Implementation differences" paragraph below). Because the hierarchical grid is static, the parent-child relationships are deterministic index mappings precomputed at initialization: $\text{children}[d][i] = \{j : \text{pos}_j \text{ is child of } \text{pos}_i\}$. Algorithm 14 computes variance at every non-leaf depth: a single root variance at d=0, then per-position variance via vectorized gather-and-reduce at depths 1 through $D-1$, with one depth processed per loop iteration and work within a depth distributed across parallel lanes.

Algorithm 14 Bottom-Up Variance Computation

Require: Weight tensor $\mathbf{W}$, children index arrays children[·], maximum depth D
Ensure: Variance var[d] at each non-leaf depth

```
var[0] ← Var(W[level-0 cells]) {single root variance over the 4 level-0 cells (no parent layer above)}
for d = 1 to D−1 do
    W_children ← gather(W, children[d]) {shape: (positions, 4)}
    var[d] ← Var(W_children, axis=1) {parallel over positions}
end for
```

The subdivision decision then propagates top-down as a binary mask:

$$\text{active}[d][i] = \begin{cases} \text{True} & d = 0 \\ \text{active}[d-1][p_i] \wedge (\text{var}[d-1][p_i] > \theta_{\text{div}}) & \text{else} \end{cases} \tag{6.3}$$

where $p_i = \text{parent}(i)$. Each parent controls all four of its children as a block, so a position is active only if its parent is active *and* the parent's variance exceeds the threshold. Algorithm 15 realizes this recurrence as a top-down sweep over the D depth levels. The recurrence is sequential across depths (the active mask at depth d is gathered from the mask and variance at depth $d-1$, so the D levels must be processed in order), but each depth level is fully vectorizable across positions: the $d-1$ active mask and variance vectors are gathered to child positions, then combined by a pointwise AND ($\wedge$ in Eq. 6.3) with the threshold comparison, with no within-level dependencies. The total elapsed cost is therefore D sequential gather-and-AND steps, each parallelized across positions and across the population via a single outer `vmap`.

Algorithm 15 Top-Down Active Mask Propagation

Require: Variance var[·], grid $\mathcal{G}_D$, threshold θ_{div}
Ensure: Active mask active[d] for each depth $d \leq D$
1: active[0] ← True {root always active}
2: **for** $d = 1$ **to** D **do**
3: parent_active ← gather(active[$d-1$], $\mathcal{G}_D$.parents[d])
4: parent_var ← gather(var[$d-1$], $\mathcal{G}_D$.parents[d])
5: active[d] ← parent_active ∧ (parent_var > θ_{div})
6: **end for**

Figure 6.4 illustrates the complete pipeline and Figure 6.5 provides a detailed schematic: CPPN outputs are gathered bottom-up to compute variance, then the active mask propagates top-down, inheriting parent decisions to children.

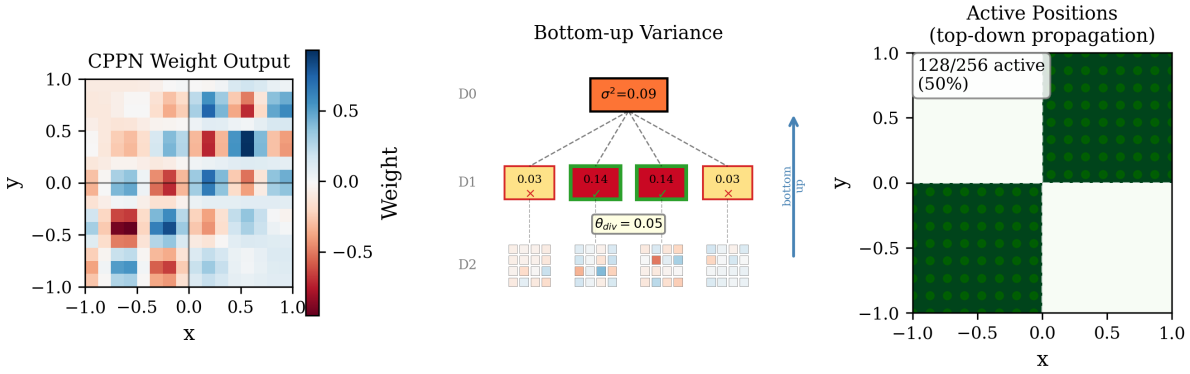


Figure 6.4: Hierarchical variance thresholding. **Left:** CPPN weight tensor **W** at depths 0–2. **Middle:** Variance computed bottom-up (upward arrows); color intensity indicates magnitude. **Right:** Active mask propagated top-down (downward arrows); positions meeting the threshold condition (Eq. 6.3) are active, others pruned. Dashed lines show parent-child inheritance.

The following theorems formalize the computational advantage and correctness of the eager approach relative to the quadtree. Table 6.2 summarizes the complexity comparison.

Table 6.2: Computational complexity comparison. D = maximum depth, H = discovered hidden positions, P = parallel execution units. ES-HyperNEAT operations are shown separately to highlight what EMR parallelizes; in ES-HyperNEAT, these are interleaved during quadtree traversal rather than distinct passes.

Operation	ES-HyperNEAT	EMR-HyperNEAT
Grid Construction	—	$\mathcal{O}(4^D)$ once
CPPN Query (Phase 1)	$\mathcal{O}(4^D)$ seq.	$\mathcal{O}(4^D/P)$
Variance Comp.	$\mathcal{O}(4^D)$ seq.	$\mathcal{O}(4^D/P)$
Mask Propagation	$\mathcal{O}(4^D)$ seq.	$\mathcal{O}(4^D/P)$
h→h Query (Phase 2)	$\mathcal{O}(H^2)$ seq.	$\mathcal{O}(H^2/P)$
Total	$\mathcal{O}(4^D + H^2)$	$\mathcal{O}((4^D + H^2)/P)$

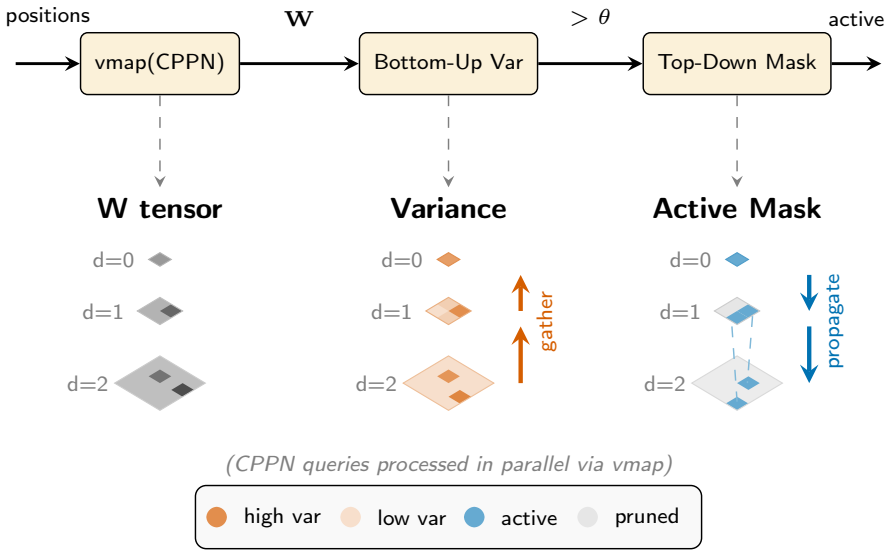


Figure 6.5: Hierarchical variance thresholding pipeline. **Top:** Processing stages. Positions enter `vmap(CPPN)`, producing weight tensor $\mathbf{W}$; variance is computed bottom-up; active mask propagates top-down. **Bottom:** Isometric grids at depths 0–2. Left column shows CPPN weights (grayscale); middle column shows variance gathered from leaves to root (vermillion arrows); right column shows the active mask propagated from root to leaves (blue arrows). Dashed lines indicate parent-child inheritance.

Theorem 6.1 (Sequential Complexity)**.** *ES-HyperNEAT's Phase 1 requires $\mathcal{O}(4^D)$ sequential CPPN queries in the worst case, and Phase 2 requires $\mathcal{O}(H^2)$ sequential queries where $H \leq 4^D$ is the number of discovered hidden positions.*

Proof. Phase 1: In the worst case (all positions active), the quadtree explores all $\sum_{d=0}^{D} 4^d = \mathcal{O}(4^D)$ nodes, each requiring sequential evaluation due to parent dependency. Phase 2: For H hidden positions, all-pairs connectivity requires H^2 queries, each sequential due to the same dependency structure. □

Note. $\mathcal{O}(H^2)$ is the count of source–target pairs, not the count of traversals. In practice, Phase 2 launches H quadtree traversals, and each one can visit up to $\mathcal{O}(4^D)$ positions before terminating. With $H \leq 4^D$, the per-pair and per-traversal accounting converge to the same asymptotic figure.

Theorem 6.2 (Parallel Complexity)**.** *EMR achieves $\mathcal{O}(4^D/P)$ elapsed time for Phase 1 and $\mathcal{O}(H^2/P)$ for Phase 2, where P is the number of parallel execution units.*

Proof. Phase 1: The precomputed grid has $\mathcal{O}(4^D)$ positions. Batch CPPN query via `vmap` evaluates all positions simultaneously across P cores, achieving $\mathcal{O}(4^D/P)$ elapsed time. Variance computation parallelizes across levels (each level depends only on CPPN outputs, not on other levels' variances). Mask propagation is D sequential steps because the active mask at depth d depends on the mask at depth $d-1$; however, each step is parallel across positions, and the deepest level's $\mathcal{O}(4^D)$ positions dominate, giving $\mathcal{O}(4^D/P + D) = \mathcal{O}(4^D/P)$ elapsed time for $P \ll 4^D$. Phase 2: Constructing an $H \times H$ query batch and applying `vmap` yields $\mathcal{O}(H^2/P)$ elapsed time. □

Theorem 6.3 (Set Equality under Matched Settings)**.** *Under identical variance thresholds θ_{div}, band thresholds θ_{band}, and substrate-selection rule, EMR-HyperNEAT and ES-HyperNEAT discover identical position sets and produce identical connection sets.*

Proof. Both algorithms admit positions under the same parent-conditioned predicate. ES-HyperNEAT's quadtree visits a position p at depth d iff every ancestor in its chain (root through p's parent) had 4-child variance above θ_{div}; the BFS queue realizes this top-down conditional descent. EMR's Algorithm 15 computes $\text{active}[d][p] = \text{active}[d-1][\text{parent}(p)] \wedge (\text{var}[d-1][\text{parent}(p)] > \theta_{\text{div}})$, which by induction yields $\text{active}[d][p] = \text{True}$ iff every ancestor in p's chain had 4-child variance above θ_{div}. The two characterizations are identical, so under matched thresholds and matched substrate-selection rule the two algorithms identify identical position sets. With matched band thresholds, both algorithms then produce identical connection sets. □

Practical superset relation. In the standard implementations, the two algorithms differ in their substrate-selection rule (*which* active positions to retain). EMR retains every reached position across every depth level (the multi-resolution union), while ES-HyperNEAT retains only the quadtree's leaves — positions where subdivision stopped because variance fell below θ_{div} or the maximum depth was reached. The intermediate-level parents whose own variance was high are subdivided past in ES-HyperNEAT and dropped from the substrate, but retained in EMR's multi-resolution view. EMR's substrate is therefore a strict superset of ES-HyperNEAT's under the default implementation choices. This is a substrate-selection difference, not a difference in the parent-conditioned predicate itself.

This result clarifies a question raised in Chapter 5: does the quadtree's lazy descent fundamentally lose positions that an eager evaluation would discover? Theorem 6.3 shows the answer is no: both algorithms apply the same parent-conditioned predicate, so under matched thresholds and substrate-selection rule they identify the same set of admissible positions through the same conditional descent. The quadtree's true cost is therefore architectural rather than positional: its top-down BFS is structurally incompatible with SIMT parallelism (Section 5.6), while EMR's eager evaluation produces fixed-shape tensors that vectorize across the population. In the standard implementations, EMR also retains intermediate-level parents that ES-HyperNEAT's leaf-only selection discards, giving EMR a richer multi-resolution substrate — a substrate-selection choice rather than an algorithmic recovery of unreachable positions.

Chapter 5's HSHG attempt (Section 5.4) tested an evaluate-then-filter design superficially similar to EMR's — pre-allocated positions, batched CPPN evaluation, variance threshold — yet over-discovered systematically (29–63 hidden nodes per substrate versus the quadtree's 1–2) because variance was measured at each position over a local neighborhood and adjacent neighborhoods overlapped, so a single informative feature triggered multiple adjacent positions to pass the threshold independently. EMR avoids that failure by re-coupling variance to the grid's hierarchy: variance at each depth is computed over four non-overlapping cells (Algorithm 14) and Algorithm 15 activates a position only when its parent was active and the parent's variance exceeded the threshold. The same feature passes the threshold at exactly the parent that subsumes it, recovering the quadtree's parent-conditioned selectivity within a fixed-shape parallel computation. Where HSHG retained parallelism but lost selectivity, and the JAX-ESHN port retained the quadtree's selectivity but inherited its heterogeneity-bound 1.65× ceiling (Section 5.3), EMR is the design that obtains both.

The practical consequence is most visible at `initial_depth=0`. Under ES-HyperNEAT's top-down traversal, the quadtree begins at the root and descends only when root variance exceeds the threshold; randomly initialized CPPNs whose outputs are uniformly low therefore produce *no hidden nodes at all*, a cold-start failure mode documented in

Chapter 5. EMR relaxes this failure through two implementation choices. First, EMR's multi-resolution substrate retains the always-active root level, so EMR yields at least the four root positions even when root variance is below threshold. Second, EMR's lower variance threshold (0.03 on raw CPPN outputs versus 0.5 on scaled substrate weights) admits subdivisions where the quadtree would have halted. Together these two choices — mask selection and threshold scale — produce valid substrates where ES-HyperNEAT yields none. Section 6.5 quantifies the resulting solve-rate improvement.

Implementation differences. Theorem 6.3 assumes matched thresholds and matched substrate-selection rule. The real implementations match neither. Subdivision in EMR fires when variance on raw CPPN outputs exceeds 0.03; the quadtree fires at 0.5 on scaled substrate weights. The two signal ranges differ as well ($[0, 4]$ for raw outputs versus $[0, 25]$ for scaled weights), and the threshold values themselves differ by 16.7×. Together, these two gaps make EMR admit subdivisions that the quadtree would have pruned, so EMR operates in a more permissive discovery regime on the same CPPN. The two algorithms also evaluate variance against the threshold over different sample sets: EMR uses 4-child variance at every level, while ES-HyperNEAT uses 4-child variance during its division step (matching EMR) but switches to variance over all leaf descendants of each subtree during its later pruning step. ES-HyperNEAT's pruning step further filters which division-admitted positions become substrate nodes; EMR retains all admitted positions without this additional filter. The asymmetry carries through to connection filtering: EMR keeps a connection whenever $|w| > 0.1$, a purely local test, whereas the quadtree inspects a spatial gradient across neighboring positions at band threshold 0.3, a non-local test. The first is pointwise arithmetic and batches cleanly across SIMT lanes; the second is position-dependent control flow and does not.

Algorithm 16 gives the complete substrate discovery procedure.

Algorithm 16 EMR Substrate Discovery

Require: Grid $\mathcal{G}_D$, CPPN parameters, sources S, threshold θ_{div}
Ensure: Active positions, weight tensor $\mathbf{W}$

// **Phase 1: Batch CPPN query**
$\mathbf{W} \leftarrow \texttt{vmap}(\text{CPPN})(S,\ \mathcal{G}_D.\text{positions})$ {Shape: $(|S|, |\mathcal{G}_D|)$}

// **Phase 2: Bottom-up variance**
$\text{var}[0] \leftarrow \text{Var}(\mathbf{W}[\text{level-0 cells}])$ {single root variance (no parent layer above level 0)}
for $d = 1$ **to** $D - 1$ **do**
  $W_{\text{ch}} \leftarrow \text{gather}(\mathbf{W},\ \text{children}[d])$ {$(|S|, \text{positions}_d, 4)$}
  $\text{var}[d] \leftarrow \text{Var}(W_{\text{ch}},\ \text{axis}{=}{-1})$
end for

// **Phase 3: Top-down mask propagation**
$\text{active}[0] \leftarrow \text{True}$ {Root always active}
for $d = 1$ **to** D **do**
  $\text{active}[d] \leftarrow \text{gather}(\text{active}[d{-}1],\ \mathcal{G}_D.\text{parents}[d]) \wedge (\text{gather}(\text{var}[d{-}1],\ \mathcal{G}_D.\text{parents}[d]) > \theta_{\text{div}})$
end for

return active, $\mathbf{W}$

6.2.3 Connection Discovery

Once the active hidden positions have been identified, EMR discovers connections in three phases, all of which are fully vectorizable.

Phase 1: Forward connections (I→H, H→O). CPPN queries for all input–hidden and hidden–output pairs are batched into two `vmap` calls. Connections are retained where the absolute CPPN output exceeds a weight threshold: $|\text{CPPN}(x_s, y_s, x_t, y_t)| > \theta_w$. Both source and target sets are known before evaluation, so no sequential dependency arises.

Phase 2: Hidden-to-hidden connections (h→h). This phase constructs the full $H \times H$ query batch and evaluates it via `vmap`, achieving $\mathcal{O}(H^2/P)$ elapsed time. The connection type configuration (Section 6.3) determines which h→h connections are permitted; a mask derived from the Y-coordinates of source and target positions filters the result tensor without branching.

Phase 3: Network cleaning. Dead-end nodes (those with no path from input to output) are removed, and the network is topologically sorted for feedforward evaluation or prepared for iterative evaluation in recurrent configurations. All hidden nodes use the substrate's default activation function (tanh in all experiments reported in this thesis). The EMR architecture's batch evaluation extends to per-node function assignment via additional CPPN output dimensions, a capability validated in companion work (§8.4).

6.3 Connection Type Taxonomy

Standard ES-HyperNEAT discovers only forward h→h connections; backward, lateral, and self-loop variants were never part of the algorithm. Each additional connection type would have multiplied Phase 2's already-sequential $\mathcal{O}(H^2)$ cost, making systematic exploration of these recurrence patterns computationally impractical with the quadtree. EMR's eager evaluation changes this calculus: the cost of Phase 2 is bounded by the fixed grid size rather than by the unpredictable quadtree expansion, making systematic exploration of recurrence patterns tractable for the first time. This section introduces a taxonomy of six substrate configurations based on four connection type primitives: a design vocabulary that directly maps to the GEENNS multi-grid architecture described in Chapter 7.

Substrates in the HyperNEAT family are defined in $[-1, 1]^2$, with the y axis reserved for processing depth: the input layer is pinned at $y = -1.0$, the output layer at $y = +1.0$, and any hidden node takes an intermediate value. A connection between a source $s = (x_s, y_s)$ and target $t = (x_t, y_t)$ can therefore be typed from the relative positions of its endpoints on that axis, yielding four primitives:

Definition 6.2 (Connection Types)**.** *For source position $s = (x_s, y_s)$ and target position $t = (x_t, y_t)$:*

- ***Forward**: $y_s < y_t$ (signal propagates toward the output layer)*
- ***Backward**: $y_s > y_t$ (recurrent feedback toward earlier layers)*
- ***Lateral**: $y_s = y_t \wedge s \neq t$ (within-layer connections between distinct nodes)*
- ***Self-loop**: $s = t$ (a node's output feeds back to itself)*

Algorithm 17 shows how these four primitives enter the connection discovery pipeline (§6.2.3): Phase 2's FilterByType routine masks the $H \times H$ weight tensor according to the requested subset, without branching.

Enabling or disabling subsets of these four primitives yields six substrate configurations (Table 6.3). Feedforward disables Phase 2 entirely; Hidden-to-Hidden enables only forward h→h connections (matching ES-HyperNEAT's default behavior); the remaining four configurations add backward, lateral, self-loop, or all three.
Figure 6.6 visualizes the four primitives.

The `iteration_level` parameter controls multihop h→h expansion. At level 0, no h→h connections are formed (feedforward only). At level 1, only direct h→h connections are retained. At level k, k-hop expansion applies a discounted power series [50]: $A + \gamma A^2 + \gamma^2 A^3 + \cdots + \gamma^{k-1} A^k$, where $\gamma = 0.8$ prevents weight explosion across hops.

Table 6.3: Six substrate configurations. F = Forward (always enabled for I→H, H→O). H indicates hidden-to-hidden connectivity; B/L/S specify connection type primitives when H is enabled (Definition 6.2).

Configuration	F	H	B	L	S	Notes
Feedforward	✓					Fastest; skips Phase 2 entirely
Hidden-to-Hidden	✓	✓				ES-HyperNEAT default; forward h→h only
Backward	✓	✓	✓			Adds feedback from higher to lower layers
Lateral	✓	✓		✓		Adds same-layer connections between distinct nodes
Self	✓	✓			✓	Adds self-connections for recurrent memory
Full Recurrent	✓	✓	✓	✓	✓	All primitives enabled; requires fixed iteration count

Semantic difference from ES-HyperNEAT. EMR's `iteration_level` controls signal *propagation* depth through the discovered connections, whereas ES-HyperNEAT uses the same parameter to control position *discovery* rounds: each round launches a fresh quadtree traversal seeded from the previous round's hidden nodes. Despite this difference in mechanism, both parameters modulate network complexity on a comparable scale: higher values yield deeper processing hierarchies. Matching numerical values (1–3) across algorithms compares substrates at comparable complexity scales. The pairing is approximate because the underlying mechanisms differ, but it provides a defensible baseline for the cross-algorithm efficiency measurements that follow.

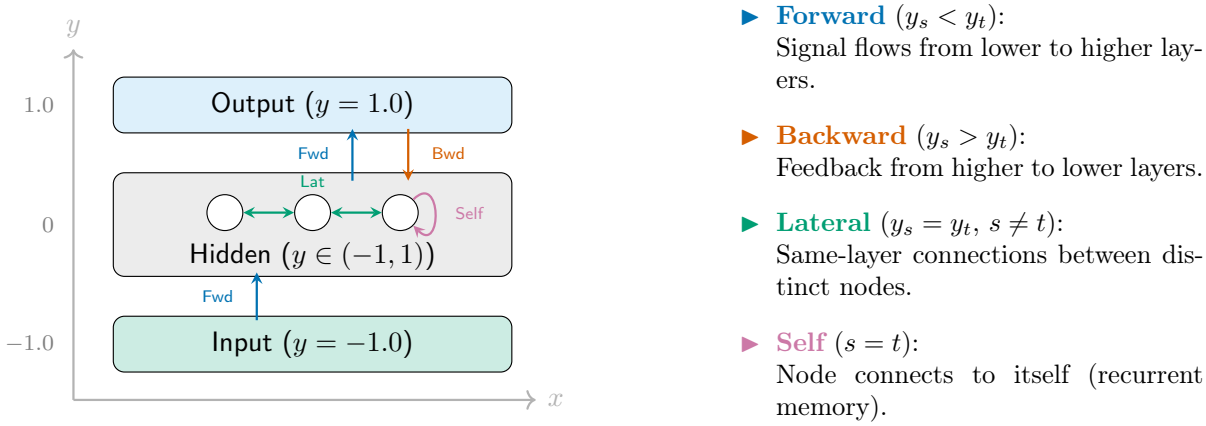


Figure 6.6: Connection type primitives in the EMR substrate: y encodes processing depth; arrows show the four classes, classified by source/target y-coordinates.

Algorithm 17 EMR Connection Discovery

Require: Active positions H, input positions I, output positions O, CPPN parameters, connection config $\mathcal{C}$, weight threshold θ_w

Ensure: Connection set E (all hidden nodes use default activation σ_0)

```
// Phase 1: Forward connections
W_ih ← vmap(CPPN)(I, H) {(|I|, |H|)}
W_ho ← vmap(CPPN)(H, O) {(|H|, |O|)}
E_fwd ← {(s, t, w) : |w| > θ_w, w ∈ W_ih ∪ W_ho}

// Phase 2: Hidden-to-hidden (connection-type dependent)
if C.hh_enabled then
    W_hh ← vmap(CPPN)(H, H) {(|H|, |H|)}
    E_hh ← FilterByType(W_hh, H, C, θ_w) {Apply connection type mask}
else
    E_hh ← ∅
end if

// Phase 3: Network cleaning
E ← RemoveDeadEnds(E_fwd ∪ E_hh)

return E
```

6.4 Implementation and Execution Strategies

The EMR implementation uses JAX/Python and extends the TensorNEAT library [100], continuing the JAX-based stack introduced in Chapter 5's JAX-ESHN system; Chapters 3 and 4 used the CPU-bound PUREPLES framework [103] instead. All evolutionary and substrate state resides in JAX arrays to keep execution GPU-resident throughout the evolutionary loop, eliminating the CPU-GPU transfer overhead that Chapter 5 identified as a secondary bottleneck.

Caching optimizations operate at two independent levels (JAX compilation caching and algorithmic h→h connection caching), detailed in Section 6.4.1. The h→h connection cache contributes 1.01–1.53× (Table 6.4); the larger EMR-CPU vs. ES-HyperNEAT ratio of 4.6–7.2× at depths 5–6 (Table 6.6) combines this cache contribution with the lazy-to-eager evaluation order and `vmap` parallelism.

The grid's independence from problem data, I/O configuration, or CPPN weights means it is fully reusable: e.g., a depth-7 grid (87,380 positions) serves unlimited evolutionary runs across arbitrary problems. For the GEENNS architecture, where multiple CPPNs share the same spatial substrate, this property eliminates redundant grid construction entirely.

The implementation exposes five execution strategies, each matched to a different point on the speed/memory-footprint trade-off. The speed factors reported below are relative to Single Device (1.0× baseline) running EMR on the same task; cross-algorithm comparisons against ES-HyperNEAT appear in Section 6.5.3:

1. **Single Device** (1.0× baseline speed, GPU memory). The entire evolutionary loop executes on a single accelerator, avoiding inter-device data movement between gen-

erations. When the grid fits in device memory this is the fastest strategy and is backend-agnostic (CPU, GPU, or TPU).

2. **Memory Streaming** (0.2–1.0× speed, CPU/disk). When the full grid exceeds GPU memory, a hierarchical memory tier (Algorithm 18) enables substrates of arbitrary depth. CPU RAM holds weight accumulators; the GPU processes CPPN queries in double-chunked batches over both population and position axes, with chunk sizes auto-configured to fit available VRAM. Each chunk's results are offloaded to CPU immediately, bounding peak GPU allocation to $\mathcal{O}(\max_d |\text{Grid}_d|)$ instead of the full $\mathcal{O}(\sum_d |\text{Grid}_d|)$. At depths where CPU RAM is also insufficient (e.g., depth 13: ∼61 GiB for 358M positions across two weight matrices per population chunk at float32 precision, exceeding our GPU server's 32 GB RAM), memory-mapped file arrays (hereafter `memmap`) spill to disk transparently: the operating system pages portions of the weight matrices between disk and RAM on demand, so that arrays whose aggregate size exceeds physical memory can be accessed as if they were fully resident. A variant of this tier, *depth-offload*, stages the evaluation one depth level at a time with population chunking, trading a higher per-generation cost for a minimal resident working set; at run time the framework's *automatic execution selection* chooses among the single-device, streaming, and depth-offload paths per configuration. To make the scale concrete: depth 7 (87,380 positions) with population 500 on XOR (3 inputs, 1 output) produces an input-to-hidden matrix W_1 of ∼525 MB ($500 \times 3 \times 87{,}380 \times 4$ B) and a hidden-to-output matrix W_2 of ∼175 MB. Including variance arrays, masks, CPPN intermediates, and Accelerated Linear Algebra (XLA) buffers, peak GPU usage reaches approximately 8.3 GB (Figure 6.15).

3. **Data Parallel** (0.6–0.7× speed, distributed memory). Distributes the dataset across GPUs while replicating the population.

4. **Evaluation Parallel** (0.6–0.7× speed, distributed memory). Fitness evaluation is distributed across every device while the h→h cache is kept resident on a single GPU.

5. **Population Parallel** (0.6–0.7× speed, distributed memory). The population is sharded across GPUs and each shard runs the complete discovery pipeline on every input sample.

All three multi-GPU strategies were validated with 2× NVIDIA RTX 2080 Ti 11 GB (JAX CUDA backend). Figure 6.7 illustrates the chunking and offload architecture.

Algorithm 18 Memory Streaming: Hierarchical Memory Tiering

Require: Grid $\mathcal{G}_D$, CPPN params, threshold θ_{div}, chunk size S
Ensure: W_1^{cpu}, W_2^{cpu}, active mask M

$W_1^{\text{cpu}}, W_2^{\text{cpu}} \leftarrow$ `np.zeros`$(\ldots)$ {CPU (or disk-backed)}
reached$[0] \leftarrow \mathbf{1}$
for $d = 0$ **to** D **do**
  for positions in chunks of S **do**
    $w^{\text{gpu}} \leftarrow$ `vmap`(CPPN)(chunk) {GPU query}
    $W^{\text{cpu}}[\ldots] \leftarrow$ `np.asarray`(w^{gpu}) {Offload to CPU}
    del w^{gpu} {Free GPU memory}
  end for
  if $d > 0$ **then**
    reached$[d] \leftarrow$ (reached$[d{-}1]$ $\wedge$ var$[d{-}1] > \theta_{\text{div}})[\mathcal{G}_D.\text{parents}]$
  end if
end for
return W_1^{cpu}, W_2^{cpu}, concat(reached)

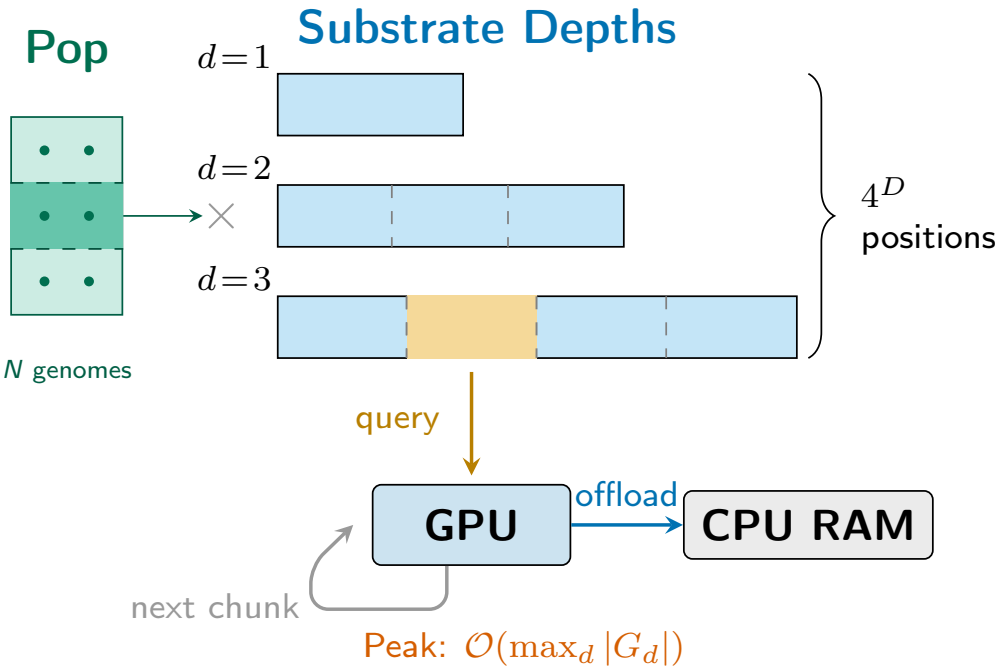


Figure 6.7: Memory management architecture for EMR-HyperNEAT. Depth levels (right) scale as 4^D; each level is chunked and processed on GPU, with results offloaded to CPU RAM. Population members (left) are similarly chunked. Peak GPU memory is bounded by the largest single chunk, not the total grid size.

6.4.1 Two-Level Caching

EMR uses two independent caching layers that address orthogonal bottlenecks.

Level 1: JAX compilation cache. JAX's XLA compiler translates Python-traced functions into optimized machine code at first invocation. The compilation result is keyed by function signature and array shapes, so subsequent calls with matching signatures skip compilation entirely. EMR enables a persistent disk cache (`/tmp/jax_cache`) that

survives across evolutionary runs, reducing startup at shallow depths from 3–7 s (full recompilation) to ∼100 ms (cache load); at depth 7 the avoided cost reaches 39–60 s (Table 6.6).

Level 2: h→h connection cache (HHCacheManager). During substrate development, h→h connections discovered via CPPN queries are cached in memory and reused across generations. A hybrid refresh policy invalidates the cache either *periodically* (every k generations, default k=5) or *reactively* (when the active-node mask changes by more than a threshold δ, default δ=0.1):

$$\texttt{refresh} \leftarrow (g \bmod k = 0) \vee \left(\|m_g - m_{\text{cached}}\|_1 / |m_g| > \delta\right) \tag{6.4}$$

When the cache is valid, h→h discovery time drops from ∼27 s to ∼3 ms, a negligible fraction of generation cost. The cached representation stores sparse connection triples (source indices, target indices, weights) plus a validity mask, avoiding the full CPPN query for $\mathcal{O}(N^2)$ position pairs.

The two levels are orthogonal: Level 1 eliminates JIT overhead regardless of evolutionary state, while Level 2 amortizes the most expensive per-generation operation (h→h discovery) across generations where the substrate topology is stable.

Table 6.4 and Figure 6.8 quantify the impact using a controlled CPU benchmark (Apple M4, 2,016 configurations, 30 generations) across two substrate tiers: T1 Sparse (few h→h connections) and T2 Dense (millions of h→h connections).

Three patterns emerge:

1. **Cache benefit is tier-dependent.** The T1 hidden-to-hidden row shows negligible cache speedup (Table 6.4; 1.01×) because few h→h connections exist to cache, while T2 dense substrates, where millions of connections are discovered per generation, achieve 1.42–1.53× speedup.

2. **The cache reduces h→h exploration.** Cached generations reuse prior connections instead of re-querying the CPPN, resulting in 2.2–5.5× fewer unique h→h connections discovered. For Full Recurrent at T1, this trade-off is favorable: fewer but higher-quality connections yield a +3% solve rate improvement (93% vs. 90%). For Hidden-to-Hidden at T1, the reduced exploration hurts (−6% solve rate). Hidden-to-Hidden mode discovers only forward h→h connections (Section 6.3), so its substrate structure depends on per-generation re-discovery to track the evolving CPPN; caching freezes this structure under stale weights, suppressing the exploration that Hidden-to-Hidden mode relies on. Full Recurrent's three additional connection types may diversify the substrate enough to absorb individual stale discoveries, though we did not isolate the asymmetry's mechanism experimentally.

3. **Solve rates are preserved in T2 regardless of cache setting** (95%/95% Hidden-to-Hidden, 79–80% Full Recurrent); the cache does not sacrifice solution quality at high connection density.

Table 6.4: h→h cache impact on steady-state performance (Apple M4 CPU, 30 generations, 168 configs per row). Speedup = $\text{time}_{\text{no-cache}}$ / $\text{time}_{\text{cache}}$. Feedforward rows (no h→h discovery) serve as baselines.

Tier	Mode	Cache	Steady (s)	Solve %	Avg h→h	Speedup
T1	Feedforward	—	6.7	43	0	—
T1	Hidden-to-Hidden	N	15.7	92	172K	—
T1	Hidden-to-Hidden	Y	15.5	86	63K	1.01×
T1	Full Recurrent	N	17.6	90	275K	—
T1	Full Recurrent	Y	13.1	93	50K	1.34×
T2	Feedforward	—	10.3	43	0	—
T2	Hidden-to-Hidden	N	19.6	95	2.7M	—
T2	Hidden-to-Hidden	Y	13.8	95	2.5M	1.42×
T2	Full Recurrent	N	18.3	80	2.8M	—
T2	Full Recurrent	Y	12.0	79	1.3M	1.53×

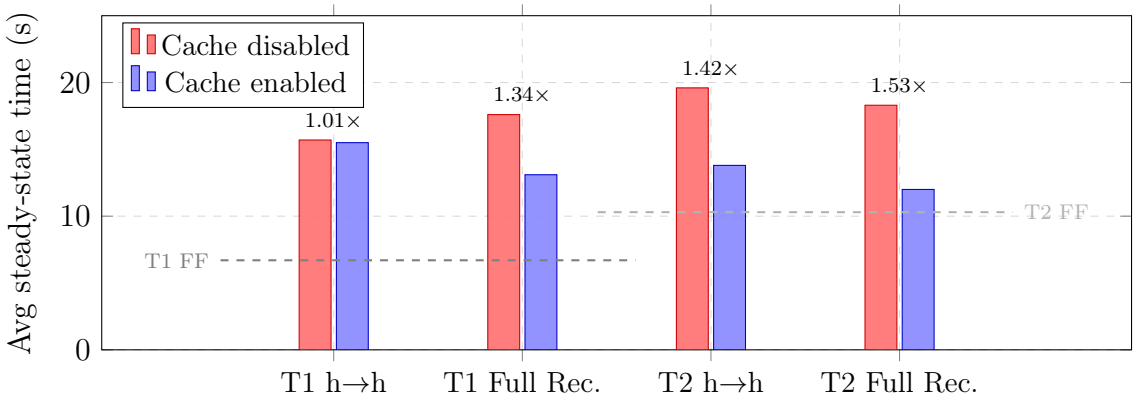


Figure 6.8: h→h connection cache impact on steady-state generation time (Apple M4 CPU, 30 generations). Speedup ratios shown above each pair. Dashed lines indicate feedforward baselines (no h→h discovery). Cache benefit increases with connection density: T1 sparse shows minimal Hidden-to-Hidden benefit (1.01×) but T2 dense achieves 1.42–1.53× speedup.

6.5 Experimental Results

This section evaluates EMR against ES-HyperNEAT on three dimensions: (1) scalability to depths that the quadtree cannot reach, (2) per-generation and cumulative speedup, and (3) solution quality. The comparison uses Hidden-to-Hidden (h→h) connections throughout, matching ES-HyperNEAT's default configuration. Both algorithms begin at `initial_depth=0`, so the topology must emerge entirely through evolutionary search rather than through a seeded starting structure. We deliberately chose this configuration — harder than the standard ES-HyperNEAT literature default of `initial_depth`>0, which seeds an initial substrate — because it is the setting where Chapter 5 documented the cold-start failure mode: randomly initialized CPPNs with low root variance produce empty substrates under top-down traversal. If EMR overcomes this failure mode while also delivering speedup, this provides the strongest support for the lazy-to-eager reformulation.

6.5.1 Experimental Setup

Hardware. All *CPU* timings were collected on an Apple M4 Max running JAX's CPU backend; all *GPU* timings on an NVIDIA RTX 2080 Ti (11 GB, JAX CUDA backend).

Baseline. The ES-HyperNEAT arm uses the same in-framework reimplementation of the PUREPLES algorithm [103] that drove the quadtree diagnostics of Chapter 5; tables abbreviate it as "ES-HN."

Problem. XOR benchmark (4 input patterns), the standard neuroevolution test case [88].

Common parameters. To isolate the effect of the eager reformulation from confounding algorithmic differences, we held as many parameters constant as possible across both implementations. Both algorithms run for 30 generations over the same 8 population sizes ($\{50, 100, 200, 300, 400, 500, 750, 1000\}$), depths 1–7, `initial_depth` 0, iteration levels 1–3, and 3 replications (seeds 42, 123, 456). The replication count is deliberately small: as documented in Chapter 5, a single ES-HyperNEAT run at depth 7 can exceed three hours, making large-N designs impractical for the baseline. The XOR target fitness is 0.99[1], the band threshold 0.3, and the variance/division thresholds are set to the respective defaults of each implementation.

Threshold usage differs and this distinction merits attention. ES-HyperNEAT uses `division_threshold` (0.5) to gate quadtree subdivision; EMR uses `variance_threshold` (0.03) as the analogous mask cutoff for its bottom-up evaluation. The 16.7× numerical gap reflects the different variance scales: ES-HyperNEAT computes variance on scaled weights (range 0–25 with `max_weight` = 5.0 [78]), whereas EMR (built on TensorNEAT [100]) operates on raw CPPN outputs (range 0–4 with EMR's default `max_weight` = 8.0). The practical consequence (more permissive topology discovery in EMR) follows from this threshold asymmetry combined with EMR's multi-resolution substrate selection (Theorem 6.3 and the "Practical superset relation" remark that follows it). The ES-HyperNEAT arm totals 456 runs (8 populations × 3 replications × 19 depth–iteration configurations: depths 1–6 at three iteration levels, depth 7 at level 1 only).

EMR specifics. Every EMR experiment ran in a fresh process to force JIT recompilation, so that all reported timings include the one-time compilation overhead analyzed in Section 5.6.2 of the preceding chapter. In production deployments, JAX's JIT cache persists across invocations and this cost is amortized away. The sparse-connection budget is capped at $C_{\max} = 10{,}000$ per genome. Because EMR's $\mathcal{O}(4^D/P)$ scaling (Theorem 6.2) makes deeper resolutions tractable, the EMR arm extends to depths 8–13, resolutions that are computationally infeasible for the ES-HyperNEAT quadtree baseline.

Experimental tiers. Tier 1 tests minimal h→h discovery (iteration level 1); Tier 2 tests maximal h→h discovery (EMR only, dense/all positions); Tier 3 tests multi-layer signal propagation (iteration levels 2–3).

Substrate selection and threshold scale. A key distinction that directly connects to the Chapter 5 analysis: when root variance falls below threshold, ES-HyperNEAT's top-down quadtree abandons the entire tree: no children are ever extracted, and the algorithm returns an empty substrate. EMR avoids this failure through two implementation choices. First, EMR's substrate is the multi-resolution union, which always retains the four root-level positions independently of variance. Second, EMR uses a more permissive variance threshold (0.03 on raw CPPN outputs versus 0.5 on scaled substrate weights),

[1] Tighter than the literature default of $\phi^* = 0.95$; see Appendix C.11 for the full threshold inventory across the thesis.

so subdivisions that the quadtree would have halted are admitted. Together these two choices yield valid substrates at `initial_depth=0` where ES-HyperNEAT yields none, a cold-start failure documented but not resolved in Chapter 5.

6.5.2 Benchmark Problems

This chapter evaluates EMR on two classes of benchmarks: *timing benchmarks* that measure computational scalability, and *cross-problem benchmarks* that validate solution quality across diverse task structures. Table 6.5 summarizes all problems and their roles.

Timing benchmarks. XOR (2 inputs, 1 output) is a standard neuroevolution test case since Stanley and Miikkulainen [88], used here because its small evaluation cost ($n = 4$ patterns) isolates substrate construction time from data processing. All timing measurements (Tables 6.6–6.11, Figure 6.13) use XOR. CRSP/Compustat Merged (CCM) financial data [9] provides a real-world regression benchmark ($n = 94{,}464$ observations) that tests whether EMR's speedup transfers to realistic data volumes (Table 6.12).

Cross-problem benchmarks. The connection type analysis (Table 6.15) evaluates all six substrate configurations across 18 problems: XOR plus four groups. *Boolean gates* (AND, OR, NAND, NOR) and *pattern tasks* (sequence prediction, receptive field) serve as complexity floors that are trivially solved by any substrate. *Visual/spatial tasks* (Visual Discrimination, Retina, Orientation) test pattern recognition at varying spatial scales (defined in Appendix C). *Parity-3* requires a multi-stage Boolean cascade. The seven *capacity-limited* tasks (Spirals, Circles, Symmetry, the pattern-recognition task, Turing, and both Hanoi variants; Table 6.15, note *c*) all achieve 0% convergence across all configurations at depth 5. This points to a representational capacity ceiling, not connection type limitations.

Table 6.5: Benchmark problems used in this chapter. Category: T = timing, C = cross-problem, B = both. "Role" indicates the problem's function in the experimental design. All problems are formally defined in §2.5.1–2.5.2 and Appendix C.10.

Problem	Category	In	Out	Role	Def.
XOR	Boolean (T)	2	1	Timing benchmark	§2.5.1
AND/OR/NAND/NOR	Boolean (C)	2	1	Complexity floor	§2.5.1
Visual Discrim.	Visual (C)	25	1	Spatial integration	§C.10.1
Retina	Visual (C)	8	2	Left/right comparison	§C.10.2
Orientation	Neurosci. (C)	27	1	Orientation tuning	§C.10.3
Parity-3–6	Logic (C)	3–6	1	Cascaded computation	§2.5.2
Two Spirals	Geom. (C)	2	1	Nonlinear separation	§C.10.4
Conc. Circles	Geom. (C)	2	1	Radial classification	§C.10.5
Symmetry Det.	Visual (C)	49	4	Global symmetry	§C.10.6
Turing Patterns	Morph. (C)	2	2	Morphogenetic dyn.	§C.10.7
Tower of Hanoi	Planning (C)	var	var	Sequential planning	§C.10.8
CartPole	Control (C)	4	1	RL control	§C.10.9
Acrobot	Control (C)	6	1	RL control	§C.10.9
MountainCar	Control (C)	2	1	RL control	§C.10.9
CCM Financial	Regression (B)	5	1	Real-world volume	§C.10.10
Earnings-Price	Regression (C)	12	1	Financial regression	§C.10.11
CalHousing	Regression (C)	8	1	Housing regression	§C.10.12

6.5.3 Scaling and Speedup

Table 6.6 presents per-generation timing at population 1000. EMR times are steady-state (after JIT); "JIT" columns show the one-time compilation overhead.

Scaling behavior. The quadtree's per-depth runtime penalty, measured in Chapter 5, ranged from 4× to 11× between depths 4 and 6. Each depth level both quadruples the candidate positions and lengthens the serial traversal path, a compounding effect that the Chapter 5 optimizations could not break. EMR decouples these two costs: position count still grows as 4^D, but parallel evaluation distributes the work across P GPU cores, converting the multiplicative traversal-depth penalty into an additive grid-size ratio. The per-level cost increase stays at 1.3–2.3× across depths 4–6. Depth 7 is the exception: the weight tensor overflows 11 GB VRAM, forcing memory streaming that pushes the per-level penalty to ∼6.7×, though the per-generation speedup over ES-HyperNEAT still reaches 34×.

The contrast with Chapter 5's 1.65× ceiling — measured on the reference XOR-at-depth-2 configuration — is sharp. That ceiling was implementation-level: four strategies optimized the existing sequential traversal and exhausted its headroom. EMR's 34× speedup at depth 7 (XOR, pop 1000) reflects an algorithmic change: the lazy-to-eager reformulation alters the order of operations rather than optimizing the existing order, and the specific ratio depends on the benchmark and configuration (Table 6.6). Where the quadtree takes 2,020 seconds per generation (pop 1000) at depth 7, EMR completes the same work in 60 seconds. At deeper substrates (Section 6.5.8), the gap widens further because the quadtree's sequential cost grows exponentially while EMR's parallel cost grows by a bounded factor per level.

Table 6.6: Average time per generation (XOR, pop = 1000). EMR uses h→h type (CPU/GPU). Speedup is per-generation (excludes JIT).

Depth	ES-HN	EMR	JIT	Speedup CPU/GPU
1	0.3 s	1 s / 0.9 s	4 s / 7 s	0.3× / 0.3×
2	0.4 s	2 s / 1.8 s	5 s / 9 s	0.2× / 0.2×
3	1.0 s	3 s / 2.3 s	8 s / 17 s	0.3× / 0.4×
4	4.27 s	5 s / 2.9 s	12 s / 18 s	0.85× / **1.5**×
5	46.3 s	10 s / 3.9 s	15 s / 19 s	**4.6**× / **12**×
6	200.4 s	28 s / 8.9 s	30 s / 24 s	**7.2**× / **23**×
7	2,020 s	165 s / 60 s†	60 s / 39 s	**12.2**× / **34**×†

Total time = JIT + (generations × time per generation).
†GPU depth 7 uses streaming mode; its 60 s/gen denominator is an estimate (measured range 50–70 s/gen on 11 GB VRAM).

6.5.4 Positioning Among Neuroevolution Frameworks

Table 6.6 compares EMR against the quadtree baseline it reformulates. To place both within the wider neuroevolution landscape, this subsection compares four frameworks under a single measurement protocol. Figure 6.9 organizes them along the two axes that matter here. The first axis is the *encoding*: in the direct encoding the genome encodes the network one gene per parameter (NEAT evolves topology and weights explicitly, and depth plays no role), while in the indirect encoding a compact CPPN genome generates

the substrate, and the framework must discover node placements and weights by querying it; substrate discovery is the step the depth axis prices. The second axis is the *implementation*: a reference implementation executes the published algorithm as interpreted Python on CPU, while a tensor reformulation restates the algorithm as batched array operations that run on either CPU or GPU. The four cells are occupied by neat-python, the reference Python implementation of NEAT [66]; the PUREPLES library, the reference implementation of ES-HyperNEAT [103] (the published library itself, not the in-framework reimplementation that Table 6.6 benchmarks); TensorNEAT, the tensor reformulation of NEAT [100]; and EMR, this chapter. The TensorNEAT arm of this comparison runs NEAT and is thus a reformulation of the *direct* encoding; EMR is the corresponding tensor reformulation for adaptive-substrate neuroevolution, a reformulation of the *indirect* encoding. EMR is itself built on TensorNEAT's NEAT layer (§6.4), so the two tensor arms share the same population machinery and JAX stack, and the comparison between them isolates the substrate-discovery layer.

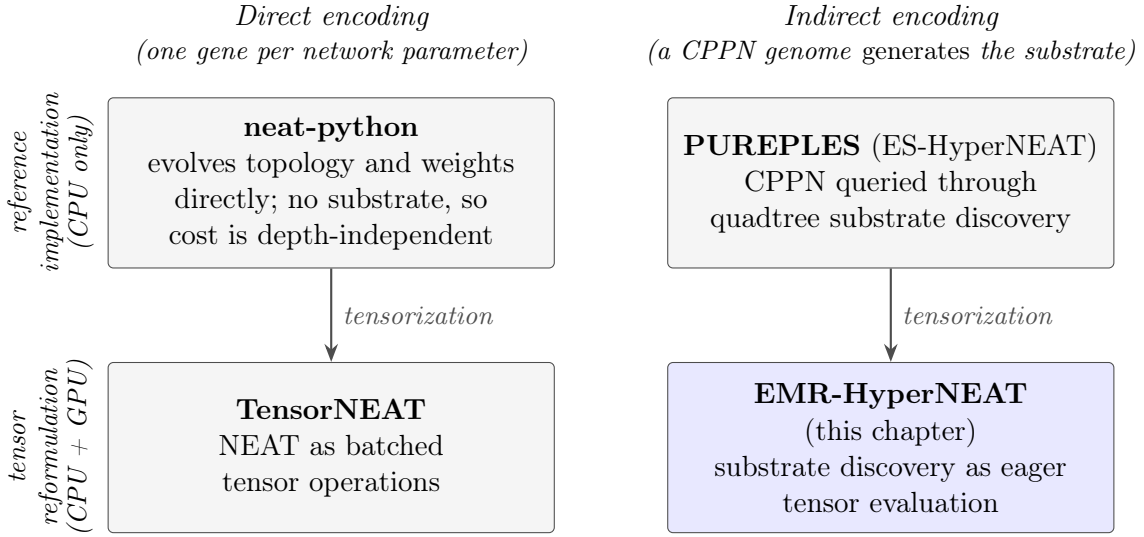


Figure 6.9: The four frameworks of the cross-framework comparison, organized by encoding (columns) and implementation strategy (rows). The CPU pass (Table 6.7) compares all four frameworks, because CPU is the only hardware all four support; the GPU pass (Table 6.8) repeats the identical protocol for the two tensor reformulations. The ES-HyperNEAT arm in this comparison is the reference PUREPLES library, distinct from the in-framework reimplementation benchmarked in Table 6.6.

All values in both tables come from one harness: the same task (XOR), population (1,000), generation budget (30, no early stopping), fitness definition (1−MSE), and CPPN search hyperparameters, with runs interleaved within each measurement batch and per-run machine state recorded. The protocol runs in two passes. The *CPU pass* (Table 6.7) compares all four frameworks; neat-python and PUREPLES have no GPU execution, so CPU is the only shared hardware. The *GPU pass* (Table 6.8) then repeats the protocol for the two tensor reformulations on an 11 GB card: identical code revision, seeds, budgets, and statistic, with only the hardware changed (the JAX version is pinned per platform, 0.6.1 on the CPU machine and 0.4.38 on the CUDA server, so absolute values embed compiler differences that within-pass comparisons do not).

Three reading conventions apply to both tables. First, each cell is the mean steady-

state time per generation in seconds: the post-warmup mean of each run, taken as the median across N=30 seeds; cells measured with fewer seeds carry a lettered note giving their count. Second, *capacity-matched* means TensorNEAT's declared node capacity is set to EMR's grid size $N(d)$ at each depth (Table 6.1), granting the direct encoding the same representational space the adaptive substrate can reach; because TensorNEAT's native use is a small fixed capacity, its own-capacity cost (`max_nodes` = 50) is reported separately. Third, NEAT evolves no substrate, so its per-generation cost does not depend on depth: the neat-python and own-capacity TensorNEAT values therefore appear as single depth-independent reference entries below the depth rows rather than as columns. Values for ES-HyperNEAT and EMR differ from Table 6.6 for two reasons of very different size. The dominant one is the implementation: the reference library is different code from the in-framework reimplementation that Table 6.6 and Chapter 5 benchmark, and runs up to two orders of magnitude slower at the same depth. Protocol differences account only for the residual factor of roughly one to three visible in the EMR columns, which carry the same implementation under both protocols. Each protocol is internally consistent but the two are not cell-comparable, which is why the positioning enters as its own tables rather than as new columns in Table 6.6.

The CPU pass supports three readings. First, the direct encoding pays for declared capacity where EMR pays per genome–position pair. A TensorNEAT genome matched to EMR's grid capacity $N(d)$ materializes a dense connection matrix per genome, and at depth 5 the population no longer fits in memory: the capacity-matched comparator drops out at exactly the scale where adaptive substrates begin to matter. Among the substrate-building arms the gap is widest at depth 4, where the reference library's 535.9 s stands against EMR h→h's 3.73 s, roughly 144× (Appendix D.2 bounds the ratio's load sensitivity: fully load-attributed, it remains roughly 120×). Second, at its own capacity TensorNEAT runs XOR about twice as slow as neat-python on CPU (0.069 s against 0.035 s per generation). This does not contradict the up to 500× speedup reported for TensorNEAT [100], which was measured on Brax robotics control tasks with expensive fitness evaluation, populations up to 10,000, and GPU execution against a CPU baseline; on a four-sample task at population 1,000 on CPU, tensorization overhead dominates. The comparison locates where tensorization pays (expensive fitness evaluation, large populations, GPU execution) rather than diminishing the reported gains. Third, among the adaptive-substrate methods, EMR in h→h mode tracks the reference PUREPLES ES-HyperNEAT at depth 1 and separates rapidly thereafter. EMR is also the only framework in the comparison whose GPU path reaches every depth, by configuration where necessary (Table 6.8).

Per-seed pairing of the two passes yields three observations (the acceleration factors below are medians of per-seed GPU/CPU ratios, which is why they differ slightly from ratios of the two tables' printed cells). First, the direct encoding gains the most from the hardware it was reformulated for: TensorNEAT's capacity-matched cells accelerate by 10.7×, 31.5×, and 58.5× at depths 1–3, and its own-capacity configuration by 15.4×. Second, EMR's gains grow with depth — from 0.69× at depth 1 (slower than CPU, as kernel-launch overhead dominates a 20-position grid) through parity near depth 2–3 to 4.8× at depth 6 — while the h→h configuration crosses later still, with the depth-4 cell statistically indistinguishable between backends. Third, and most consequentially for the capacity argument: the direct-encoding wall arrives *earlier* on the GPU. Capacity-matched depth 4, feasible on the CPU, fails on all thirty seeds on an 11 GB card because the transform pipeline's working set is roughly six times the single connection buffer

Table 6.7: Cross-framework positioning, CPU pass: mean steady-state time per generation in seconds (post-warmup mean per run, median across seeds; XOR, pop = 1000, CPU, 30 generations, no early stopping, N=30 seeds per cell unless a lettered note says otherwise). All values from one harness with matched CPPN search settings and identical 1−MSE fitness; runs interleaved within each measurement batch and per-run machine state recorded. ES-HyperNEAT is the reference PUREPLES library; EMR h→h is the mode-matched configuration (its hidden connectivity matched to the forward-only h→h links standard ES-HyperNEAT discovers), and EMR ff is the feedforward configuration. TensorNEAT declares capacity matched to EMR's grid $N(d)$; NEAT itself evolves no substrate, so the depth-independent reference entries below the depth rows give neat-python's cost and TensorNEAT's cost at its own capacity. Dashes mark cells not measured under this protocol.

	Direct encoding	Indirect encoding		
Depth	TensorNEAT (capacity-matched)	ES-HyperNEAT (PUREPLES library)	EMR h→h	EMR ff
1	0.038	0.54	0.38	0.11
2	0.33	3.49	0.61	0.15
3	3.71	45.5	2.99	0.24
4	50.6	535.9[a]	3.73	0.68
5	infeasible[b]	∼6314.3[c]	8.74	3.85
6	–	–	–	16.4[d]
7	–	–	–	101.9[d]
Depth-independent references (no substrate to discover): neat-python NEAT: 0.035 s; TensorNEAT NEAT at its own capacity (`max_nodes` = 50): 0.069 s				

[a] Median of n=5 seeds.
[b] The capacity-matched population's genome memory exceeds 111 GiB at depth 5; the requirement grows ∼16× per level, so deeper cells are not shown.
[c] Projected from the measured per-depth growth; not run.
[d] Median of n=10 seeds.

(44.77 GB requested against a 7.49 GB buffer). EMR's own feedforward depth 7 also exceeds the card under its default execution configuration, but unlike the direct encoding's wall, EMR's memory cost is per genome–position pair and therefore chunkable. Under depth-offload execution with population chunking, the same cell completes on all ten seeds at 25.4 s per generation (coefficient of variation 1.3%), 2.7× faster than the identical configuration and code revision on the CPU (68.6 s, n=3, a shared-seed subset of the ten). Against the CPU protocol-default cell of Table 6.7 (101.9 s) the contrast reads roughly 4×, configuration and hardware mixed. The chunking requires no hand-tuning: the framework's automatic execution selection completes the same cell at 13.5 s per generation (n=10, per-seed fitness identical to the hand-configured runs), faster than the manual setting itself. The direct encoding's dense per-genome transform admits no such decomposition: its working set is the declared capacity, with no axis to chunk along. On GPU, the indirect encoding does not accelerate most; its advantage is that it keeps scaling, by configuration rather than by rewrite, past the depth where the direct encoding runs out of memory.

Figure 6.10 redraws both passes from the tables' printed cells, making the crossings

Table 6.8: Cross-framework positioning, GPU pass: mean steady-state time per generation in seconds (post-warmup mean per run, median across seeds; XOR, pop = 1000, NVIDIA RTX 2080 Ti 11 GB, single GPU, 30 generations, no early stopping, N=30 seeds per cell unless a lettered note says otherwise). Same harness, code revision, seeds, fitness definition, and statistic as the CPU pass; only the hardware differs. PUREPLES ES-HyperNEAT and neat-python have no GPU path and are deliberately absent; the depth-independent reference entry gives TensorNEAT's cost at its own capacity (`max_nodes` = 50, as in Table 6.7; "stock" is the library's shipped preset). EMR cells include a ∼0.3 ms/generation measurement-side host sync that the TensorNEAT arm excludes by construction (≤0.2% of any cell). Dashes mark cells not measured under this protocol.

	Direct encoding	Indirect encoding	
Depth	TensorNEAT (capacity-matched)	EMR h→h	EMR ff
1	0.003	0.96	0.15
2	0.010	0.83	0.16
3	0.060	3.32	0.20
4	OOM[a]	3.73	0.31
5	infeasible[b]	4.54[c]	0.83
6	–	–	3.35[d]
7	–	–	25.4[e]
Depth-independent reference (no substrate to discover): TensorNEAT NEAT, own capacity: 0.004 s (stock 0.006 s)			

[a] Runs out of memory on 11 GB.
[b] Single-buffer arithmetic already exceeds 111 GiB.
[c] Median of n=23 seeds.
[d] Median of n=10 seeds.
[e] Median of n=10 seeds.

and the two memory walls visible at a glance. Measurement-protocol detail (the batch structure, the load-sensitivity measurements, and the weight-scale bound) is recorded in Appendix D.2.

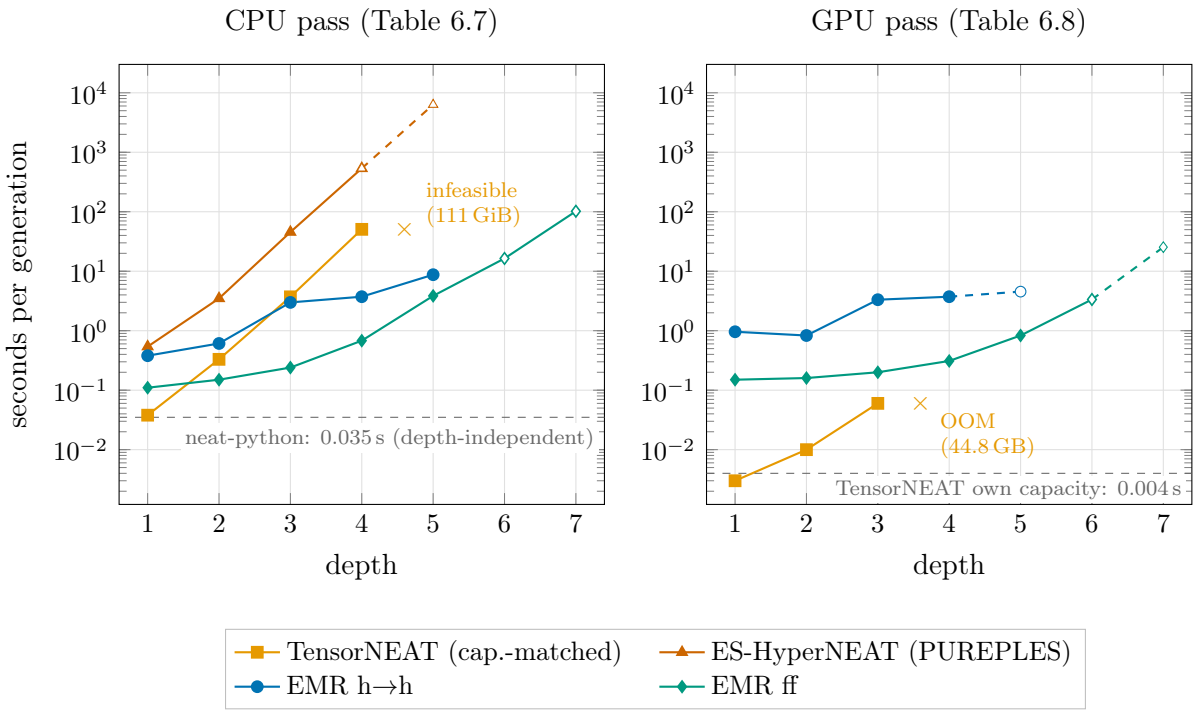


Figure 6.10: Per-generation runtime versus depth, drawn from the cells of Tables 6.7 (CPU pass, left) and 6.8 (GPU pass, right); logarithmic scale, shared axis range. Open markers flag cells that are not full 30-seed measurements (the reduced-n cells of the tables' lettered notes and the projected ES-HyperNEAT depth-5 cell), and dashed segments show the projected ES-HyperNEAT depth-5 cell, the GPU h→h depth-5 lower bound, and the depth-offload continuation of EMR ff to depth 7. Crosses mark the last feasible capacity-matched TensorNEAT cells before its memory walls (genome memory exceeds 111 GiB at CPU depth 5; out-of-memory on all seeds at GPU depth 4). Thin dashed lines give the depth-independent NEAT references. On the GPU pass the direct encoding is fastest wherever it fits but hits its wall earlier; on both passes the indirect reformulation keeps scaling past it.

6.5.5 Statistical Validation

Table 6.9 reports one-sided Mann-Whitney U tests (null hypothesis H_0: ES-HyperNEAT median runtime $\leq$ EMR median runtime) together with Cliff's δ as the effect-size measure. The 'Speedup' column reports per-seed median paired ratios; these run higher than Table 6.6's point-estimate ratios (e.g., 75× vs 34× at D7) because right-skewed runtimes amplify paired ratios at deeper substrates, so the headline figure used elsewhere in the chapter remains the conservative 34× point estimate. The pattern partitions cleanly by depth. In the shallow regime (D1–D3) ES-HyperNEAT is faster because JIT compilation still dominates EMR's total time ($p > .99$, $\delta \leq -0.89$); the gap peaks at D2 (0.1–0.2×), where the grid is trivially small yet the compiler still traces the full pipeline. D4 sits at the crossover: EMR on GPU overtakes ES-HyperNEAT (1.6×, $p < .001$, $\delta = 0.59$) while EMR on CPU remains slightly behind. From D5 onward the GPU advantage is decisive, with $\delta \geq 0.99$ in every GPU cell and $\delta \geq 0.70$ on CPU. Seven of the fourteen paired tests reject the null (D4 on GPU together with D5–D7 on both hardware paths), and every one of them survives Holm-Bonferroni correction. The speedup rises monotonically with

depth (Spearman $\rho = 0.96$, $p < .001$). The quadtree's weakest regime, per Chapter 5, is precisely where EMR's advantage peaks.

Runtime dispersion separates the two algorithms just as sharply. Using $\sigma \approx \text{IQR}/1.35$ as a robust spread estimator, the D7 standard deviations are 12,785 s for ES-HyperNEAT (about 3.6 h), 1,889 s for EMR on CPU (31 min), and ∼323 s for EMR on GPU (about 5 min), a roughly 40× contraction on the GPU path. Eliminating the quadtree's data-dependent control flow in favor of a fixed-shape grid traversal drives the contraction; the spread that remains on the GPU path tracks population size (the pooled cell holds one run per population) rather than data-dependent structure.

Table 6.9: Statistical comparison: Mann-Whitney U test (one-sided), Cliff's delta effect size. Speedup from median ratio. Significant at $\alpha = 0.05$ for GPU depths 4–7 and CPU depths 5–7 after Holm-Bonferroni correction.

	EMR GPU			EMR CPU		
Depth	p	δ	Speedup	p	δ	Speedup
1	>.99	-1.00^{L}	0.2×	>.99	-1.00^{L}	0.2×
2	>.99	-1.00^{L}	0.2×	>.99	-1.00^{L}	0.1×
3	>.99	-0.89^{L}	0.3×	>.99	-0.97^{L}	0.2×
4	<.001	0.59^{L}	**1.6**×	.915	-0.23^{S}	0.8×
5	<.001	0.99^{L}	**6.8**×	<.001	0.70^{L}	**3.5**×
6	<.001	1.00^{L}	**42**×	<.001	0.98^{L}	**11**×
7	<.001	1.00^{L}	**75**×	<.001	0.91^{L}	**15**×

Effect size: S = small, M = medium, L = large [19].

6.5.6 Solution Quality

Table 6.10 and Figure 6.11 present the full population–depth performance map across eight population sizes and seven depth levels, comparing ES-HyperNEAT and EMR GPU.

Two patterns stand out:

1. EMR's solve rate advantage grows with depth. At depth 4 both algorithms converge for most populations, but by depth 7 ES-HyperNEAT fails almost completely (0–33% across all populations) while EMR maintains 50–100% for populations ≥ 200.

2. Population size matters for both algorithms, but differently. ES-HyperNEAT's solve rate is erratic across populations (in Figure 6.11's per-seed view, pop 500 solves 100% at depth 5 but 0% at depth 6; Table 6.10's pooled rates for the same cells are 44% and 17%, and the EMR panels likewise differ from the table's pooled rates on the daggered populations), while EMR degrades with small populations (pop 50 erratically, 0–67%; pop 100 drops to 33% at depths 6–7 due to insufficient evolutionary pressure).

The erratic ES-HyperNEAT pattern reflects the data-dependent quadtree: a randomly initialized CPPN may or may not trigger root subdivision, producing an all-or-nothing failure mode that is independent of population size. EMR's deterministic grid eliminates this failure mode entirely.

Population 1000 at depth 7 exceeds 11 GB VRAM in standard GPU mode. Memory streaming (Algorithm 18) resolves this constraint: the streaming configuration is estimated at 60.0 s/gen steady-state (JIT measured at 38.7 s; the solve rate is carried from the measured populations), a 33.7× estimated speedup over ES-HyperNEAT's 2,021 s/gen (Figure 6.12). Extreme-depth experiments in Section 6.5.8 extend this approach to depths 8–13.

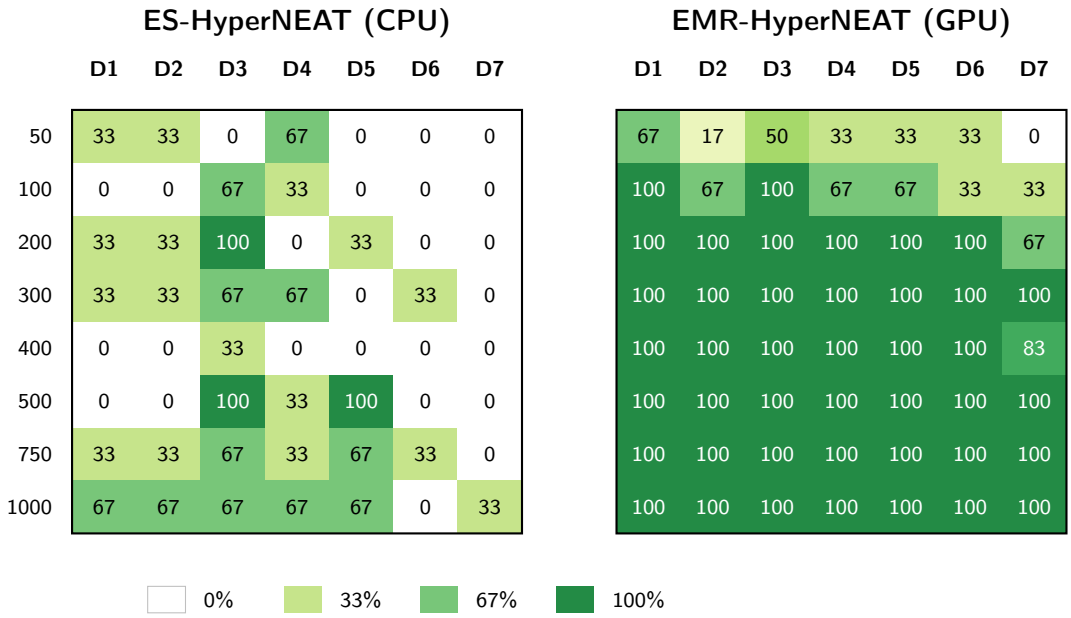


Figure 6.11: Population–depth solve rate heatmap for XOR (depths 1–7, 8 populations, 3–6 replications per cell). Left: ES-HyperNEAT shows erratic success with many zero-solve cells (white), reflecting the quadtree's data-dependent failure mode. Right: EMR-HyperNEAT achieves 100% solve rate at pop $\geq$ 300 for depths 1–6; pop 750 depth 7 timing interpolated from scaling trend. Pops 50, 400, 750 use 6 seeds from a CPU benchmark; others use 3 seeds from GPU benchmark. Table 6.10 reports the same campaign pooled across replication tiers (3–9 replications per ES-HN cell, 1–3 per EMR cell; ES-HyperNEAT pooled across all three iteration levels), so its per-cell rates differ from the 3- and 6-seed rates shown here.

Table 6.10: Population–depth performance map (XOR, depths 1–7, 8 populations; 3–9 replications per ES-HN cell, 1–3 per EMR cell). Cells show steady-state time per generation (mean ± σ across seeds) / solve rate. ES-HN = the reimplemented PUREPLES-algorithm baseline (CPU, all tiers pooled); EMR = GPU with h→h sparse discovery and cache. †Pops 50, 400, 750: EMR timing (mean and σ) interpolated from GPU population scaling trend; solve rates are from actual experiments, except depth 7 at populations 750 and 1,000, whose rates are carried over (the h→h runs at 750 exhausted GPU memory; the 1,000 cell is the streaming estimate). ‡Memory streaming mode (Section 6.5.8). JIT compilation overhead (not shown) ranges from 8–23 s and is reported separately in Table 6.6.

	Pop	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7
ES-HN	50	0.02 ± 0.00 s / 33%	0.03 ± 0.01 s / 33%	0.13 ± 0.12 s / 0%	1.0 ± 1.1 s / 22%	4.1 ± 1.8 s / 0%	25.3 ± 15.2 s / 0%	179 ± 233 s / 0%
	100	0.04 ± 0.01 s / 0%	0.05 ± 0.02 s / 0%	0.26 ± 0.09 s / 44%	1.9 ± 1.1 s / 33%	9.0 ± 6.4 s / 0%	35.2 ± 17.5 s / 0%	151 ± 116 s / 0%
	200	0.07 ± 0.00 s / 33%	0.09 ± 0.01 s / 33%	0.34 ± 0.14 s / 78%	1.7 ± 1.2 s / 0%	12.5 ± 8.8 s / 33%	116 ± 47 s / 33%	291 ± 322 s / 0%
	300	0.10 ± 0.01 s / 33%	0.12 ± 0.01 s / 33%	0.32 ± 0.06 s / 44%	1.5 ± 0.4 s / 44%	11.9 ± 7.0 s / 33%	158 ± 131 s / 0%	326 ± 360 s / 0%
	400	0.14 ± 0.01 s / 0%	0.18 ± 0.01 s / 0%	0.50 ± 0.21 s / 33%	3.2 ± 1.9 s / 44%	22.8 ± 11.0 s / 22%	173 ± 75 s / 17%	472 ± 492 s / 0%
	500	0.19 ± 0.00 s / 0%	0.24 ± 0.01 s / 0%	0.69 ± 0.14 s / 56%	4.8 ± 2.8 s / 78%	27.0 ± 11.9 s / 44%	269 ± 238 s / 17%	989 ± 1,150 s / 0%
	750	0.25 ± 0.03 s / 33%	0.31 ± 0.03 s / 33%	0.74 ± 0.09 s / 78%	3.7 ± 1.8 s / 22%	42.6 ± 41.6 s / 44%	297 ± 141 s / 11%	1,290 ± 450 s / 0%
	1000	0.30 ± 0.05 s / 67%	0.39 ± 0.06 s / 67%	1.0 ± 0.2 s / 89%	6.3 ± 2.9 s / 67%	45.9 ± 10.5 s / 56%	436 ± 242 s / 22%	2,021 ± 995 s / 33%
EMR	50†	0.20 s / 67%	0.41 s / 0%	0.33 ± 0.02 s / 33%	0.45 ± 0.05 s / 33%	0.46 ± 0.04 s / 0%	0.6 s / 33%	1.4 s / 0%
	100	0.25 ± 0.00 s / 100%	0.46 ± 0.02 s / 67%	0.44 ± 0.02 s / 100%	0.5 ± 0.0 s / 67%	0.6 ± 0.0 s / 67%	1.0 ± 0.1 s / 33%	2.7 ± 0.0 s / 33%
	200	0.33 ± 0.00 s / 100%	0.6 ± 0.1 s / 100%	0.6 ± 0.0 s / 100%	0.7 ± 0.0 s / 100%	0.9 ± 0.0 s / 100%	1.8 ± 0.1 s / 100%	5.4 ± 0.2 s / 67%
	300	0.41 ± 0.00 s / 100%	0.7 ± 0.2 s / 100%	0.8 ± 0.0 s / 100%	0.9 ± 0.1 s / 100%	1.2 ± 0.1 s / 100%	2.6 ± 0.1 s / 100%	7.9 ± 0.3 s / 100%
	400†	0.48 s / 100%	0.7 ± 0.2 s / 100%	0.9 ± 0.0 s / 100%	1.1 ± 0.1 s / 100%	1.5 ± 0.1 s / 100%	3.5 ± 0.2 s / 100%	10.4 ± 0.3 s / 50%
	500	0.5 ± 0.0 s / 100%	0.8 ± 0.1 s / 100%	1.0 ± 0.0 s / 100%	1.3 ± 0.1 s / 100%	1.9 ± 0.1 s / 100%	4.3 ± 0.2 s / 100%	13.0 ± 0.2 s / 100%
	750†	0.7 s / 100%	1.3 ± 0.4 s / 100%	1.6 ± 0.0 s / 100%	2.1 ± 0.2 s / 100%	2.9 ± 0.3 s / 100%	6.6 ± 0.2 s / 100%	36.5 s / 100%
	1000	0.9 ± 0.0 s / 100%	1.8 ± 0.6 s / 100%	2.3 ± 0.0 s / 100%	2.9 ± 0.4 s / 100%	3.9 ± 0.5 s / 100%	8.9 ± 0.2 s / 100%	60.0 s / 100%†‡

6.5.7 Population Scaling

Table 6.10 also enables the following analysis: how does EMR's speedup ratio change with population size? Figure 6.12 visualizes the per-generation speedup $s = t_{\text{ESHN}}/t_{\text{EMR}}$ for each population–depth combination ($s > 1$ indicates EMR is faster than ES-HyperNEAT).

Four patterns emerge from the expanded depth range:

1. At depths 1–3, ES-HyperNEAT outperforms EMR by 2–10×. At these shallow depths the quadtree produces small substrates where EMR's JIT-compiled batch evaluation has nothing to parallelize, and GPU kernel launch overhead dominates.

2. The crossover at depth 3–4 is population-invariant. For all five GPU-benchmarked populations (100–1000), EMR overtakes at depth 4. Comparing the CPU and GPU EMR data (Table 6.6) reveals a one-depth offset: EMR on GPU crosses the ES-HyperNEAT baseline at depth 4 (1.5×), while EMR on CPU crosses one depth later at depth 5 (4.6×). The three populations marked † (50, 400, 750) have interpolated *timing* only (their solve rates are from actual experiments) and use CPU-based EMR timing from a different benchmark strategy that inflates their per-generation cost, producing artificially depressed speedup ratios at all depths.

3. Speedup grows exponentially with depth for all populations, reflecting the $\mathcal{O}(4^D)$ cost of the quadtree versus EMR's hardware-parallelized flat evaluation.

4. Population size amplifies speedup at depth 5+. At depth 5, pop 1000 achieves 12.8× versus pop 100's 7.3×, because larger populations increase the sequential quadtree burden per generation while EMR absorbs the additional genomes into its batch `vmap` with minimal overhead.

At depth 7, pop 1000 exceeds GPU memory for batch evaluation and requires EMR's streaming mode (estimated 60.0 s/gen, 33.7×), which introduces per-level CPU↔GPU transfer overhead that caps the achievable ratio. Section 6.5.8 extends this approach to depths 8–13.

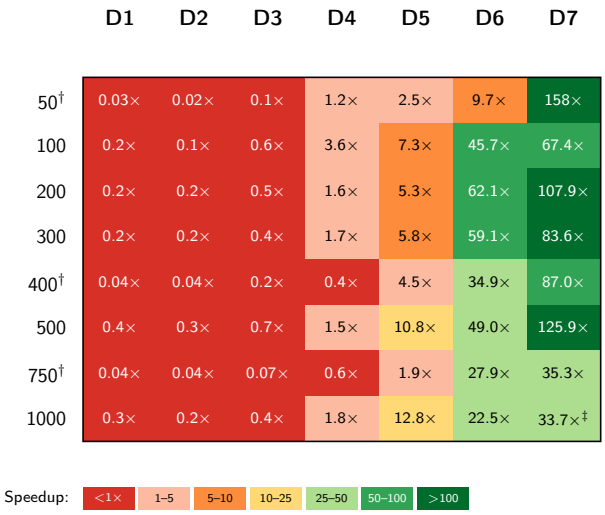


Figure 6.12: Speedup ratio heatmap: $s = t_{\text{ESHN}}/t_{\text{EMR}}$ per-generation, depths 1–7. Values > 1 indicate EMR is faster; red cells (D1–3, values < 1) indicate ES-HyperNEAT is faster. Crossover at D3–4 is population-invariant; at D6–7 speedup reaches 22–158×. †EMR timing from CPU benchmark. ‡Memory streaming.

6.5.8 Extreme-Depth Scaling

Tables 6.6 and 6.10 cover depths 1–7 with matched baselines. EMR's memory streaming (Algorithm 18) extends evaluation well beyond ES-HyperNEAT's practical ceiling. Table 6.11 presents measured timing for depths 8–13, to our knowledge the first reported substrate evolution experiments at these resolutions.

At depth 8, the grid contains 349,524 positions and a single generation takes approximately 26 seconds (pop 300), comparable to ES-HyperNEAT's per-generation cost at depth 5 (46 s at pop 1000, Table 6.6). By depth 13, the grid exceeds 357 million positions and each generation requires approximately 5.8 hours at pop 300 and ∼67.5 hours at pop 1000, the latter dominated by disk-backed memmap weight-tensor I/O. Population scaling is near-linear across depths 8–12: the pop 1000/pop 300 steady-state ratio spans 2.8–4.2×, close to the expected 3.33× from the population ratio alone (depth 12 ratio 4.2× reflects memmap overhead). Depth 13 breaks this pattern: the pop 1000/pop 300 ratio jumps to 11.6× because pop 1000 weight matrices no longer fit even in offloaded CPU RAM and must be materialized through memmap.

The JIT-to-steady-state ratio provides a practical amortization metric at extreme depths. (In production, JIT compilation is paid once per process and persisted via JAX's disk cache; the figures reported here include worst-case fresh-process overhead; see §6.5.1.) At depth 8, JIT compilation costs 1.4 generations' worth of compute; by depth 11, JIT costs less than one generation. This ratio remains near unity at extreme depths because both JIT and steady-state scale with position count; depth 12 (pop 1000) shows a JIT/SS ratio of 1.12 due to memmap serialization overhead, and depth 13 (pop 1000) settles back to 1.03 as memmap I/O dominates both the compilation and the steady-state phases.

At pop 300, depth 13 EMR completes the first generation (including JIT compilation) in ∼5.3 hours, comparable to the wall-clock time ES-HyperNEAT requires for a full 30-generation run at depth 7 (Figure 6.13), which fails to solve XOR at all. This comparison captures the lazy-to-eager reversal: the eager grid makes extreme-depth substrates cheaper than the quadtree's struggle at moderate depth. At pop 1000, depth 13 no longer fits this balance. JIT alone takes ∼69 hours as memmap serialization dominates, but the substrate remains tractable, whereas the quadtree cannot produce any depth 13 result on any available hardware.

Table 6.11: EMR timing at extreme depths (GPU with memory streaming, h→h sparse, cache). Pop 300: complete for depths 8–13 (depth 13 single seed). Pop 1000: complete for depths 8–13 (3 seeds each). "JIT/SS" is the ratio of JIT compilation time to one steady-state generation. Runs are measured over a fixed 10-generation budget to isolate per-generation timing; seed 123 at pop 300 solves XOR at generation 5 at every depth with recorded fitness (through depth 12), while the remaining seeds plateau at fitness 0.87–0.90 within the budget; the depth-13 runs record no solves (per-generation wall-clock is insensitive to convergence state). Streaming strategies escalate with depth: [a]level-by-level GPU→CPU offload, [b]CPU-resident depth accumulators, [c]CPU-resident accumulators with population chunking, [d]position chunking, [e]CPU-resident grid, [f]disk-backed (memmap) weight arrays.

Depth	Positions	JIT (s)	Steady (s/gen)	10-gen (s)	JIT/SS	Mode
Population 300						
8	349,524	36	26	∼300	1.37	[a]
9	1,398,100	90	84	∼925	1.08	[a]
10	5,592,404	380	361	∼3,991	1.05	[b]
11	22,369,620	1,184	1,214	∼13,327	0.98	[b,d]
12	89,478,484	4,360	4,751	∼51,866	0.92	[b,e]
13	357,913,940	19,122	20,873	∼227,849	0.92	[f]
Population 1000						
8	349,524	87	81	∼895	1.07	[a]
9	1,398,100	300	297	∼3,268	1.01	[a]
10	5,592,404	979	1,018	∼11,155	0.96	[b,c]
11	22,369,620	3,473	3,724	∼40,708	0.93	[b,c,d]
12	89,478,484	22,144	19,785	∼219,995	1.12	[b,c,e,f]
13	357,913,940	249,242	242,915	∼2,678,392	1.03	[f]

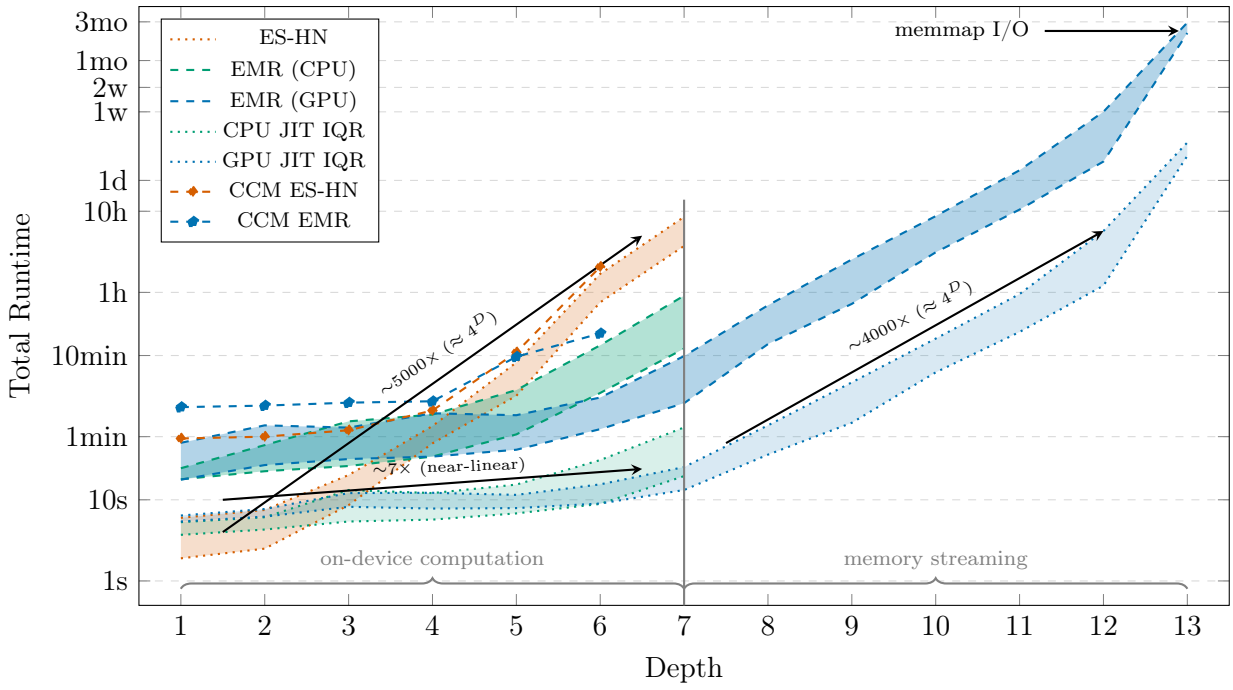


Figure 6.13: Runtime over 30 generations (logarithmic y-axis). Bands show IQR (25th–75th percentile): ES-HN and EMR CPU use 8 population sizes × 3 replications for depths 1–6; EMR GPU uses the same methodology for depths 1–7, while depths 8–13 use GPU↔RAM streaming, plotting the 30-generation total as JIT plus 30× the measured steady-state generation from a 10-generation budget (Table 6.11; pop 300 and pop 1000, 3 seeds each except depth 13 at pop 300). Depth 13 combines 1 pop 300 seed with 3 pop 1000 seeds; the widening of the GPU band at d13 reflects the onset of disk-backed memmap I/O, which dominates both JIT compilation and per-generation cost at that depth (annotated by the "memmap I/O" arrow). JIT bands show the one-time compilation overhead. CCM lines show runtime on CCM financial data [9] (94K samples, pop = 100). Trend arrows show ∼5000× scaling for ES-HN ($\approx 4^D$ per depth), ∼7× near-linear scaling for EMR on-device, and ∼4000× ($\approx 4^D$) for EMR streaming over depths 7–12.

6.5.9 GPU Scaling and Real-World Validation

Multi-GPU. Adding GPUs does not yield a speed improvement. The single-GPU pipeline compiles its generation loop down to `lax.while_loop` and runs end-to-end inside XLA. The multi-GPU pipeline cannot do the same: NEAT's cross-device fitness reduction forces a return to host-side Python between iterations, and this host-hop costs 1.5–1.7× relative to the single-GPU path. CPPN evaluation already dominates wall-clock time, so parallelizing only the evaluation step does not recover the loss. Even after absorbing the 1.7× penalty, a multi-GPU run at depth 7 still reaches roughly 20× the CPU ES-HyperNEAT baseline (34 × /1.7×); the configuration does not pay in additional speed. Capacity comes from memory streaming rather than from extra devices: staging the weight tensor out of GPU memory (to CPU RAM and, at the deepest levels, disk-backed `memmap` arrays; Table 6.11) is how substrates at depths 8–13 become addressable at all — depths that ES-HyperNEAT cannot reach in practice.

Real-world benchmark. XOR demonstrates EMR's computational advantage but involves only 4 input patterns. To validate that the speedup transfers to problems with

realistic data volumes, Table 6.12 compares performance on CRSP/Compustat Merged (CCM) financial data [9]: a time-series regression task with 94,464 observations, where the evolved substrate must learn a mapping from financial variables to a target metric. The problem exercises EMR's data parallelism (94K samples vs. 4 for XOR) and tests whether larger I/O dimensions affect the speedup profile.

ES-HyperNEAT runs on CPU (the ES-HyperNEAT baseline has no GPU path); EMR runs on GPU. The crossover pattern mirrors XOR: at depths 1–5, ES-HyperNEAT is faster due to JIT overhead, but at depth 6, EMR achieves 5.5× speedup (45 s vs. 247.9 s) *even with DepthOffload streaming* (weights on CPU RAM rather than GPU memory). ES-HyperNEAT exhibits an 11.3× slowdown from depth 5 to depth 6, the same exponential wall observed on XOR, while EMR scales by only ∼2×. This subsection validates EMR's *speed transfer* to realistic data volumes; rigorous solve-rate validation on real-world datasets remains future work. The CCM result also reveals that the crossover depth is problem-dependent: on XOR (4 samples), EMR overtakes at depth 4; on CCM (94K samples), the crossover is at depth 5–6. Larger datasets increase the per-evaluation cost for both algorithms proportionally, shifting the crossover rightward but not eliminating it. The implication for GEENNS is that multi-CPPN pipelines operating on real-world datasets will benefit from EMR at moderate depths, not only at the extreme depths where XOR shows the largest speedups.

Figure 6.13 shows the complete runtime comparison over 30 generations, including both the XOR and CCM benchmarks. The figure makes concrete what the Chapter 5 analysis predicted: ES-HyperNEAT becomes impractical at depth 6+ (IQR: 45 min–1.7 h) and infeasible at depth 7 (IQR: 3.7–8.5 h). EMR GPU extends to depth 13 via memory streaming, maintaining consistent $\mathcal{O}(4^D)$ scaling. The crossover point (where EMR becomes faster despite JIT overhead) occurs around depth 4 on XOR and shifts to depth 5–6 on CCM, precisely the depths where Chapter 5 showed the quadtree's exponential growth begins to dominate.

Table 6.12: Per-generation time on CCM financial dataset (94,464 samples, pop = 100). EMR uses h→h type (GPU). Speedup is per-generation (excludes JIT).

Depth	ES-HN	EMR	EMR JIT	Speedup
1	1.9 s	5.5 s	11 s	0.35×
2	2.0 s	5.8 s	11 s	0.34×
3	2.4 s	6.2 s	13 s	0.39×
4	4.2 s	6.6 s	13 s	0.64×
5	21.9 s	23 s	46 s	0.95×
6	247.9 s	45 s†	89 s	**5.5**×

†Uses DepthOffload streaming (weights on CPU RAM).

6.5.10 Cross-Domain Validation

The preceding sections validated EMR on the XOR benchmark, establishing speedup, solve-rate improvement, and scaling properties relative to ES-HyperNEAT. A single benchmark, however, cannot establish that the substrate engine generalizes, and generalization matters for GEENNS, where specialist grids must evolve substrates for diverse sub-problems. This subsection presents original experiments across 15 problems in four

domains (Table 6.13), confirming that EMR functions as a general-purpose substrate discovery mechanism. Configurations vary by campaign and are marked in the table: XOR averages ten recurrence configurations, Visual Discrimination aggregates the six connection-type presets of §6.5.11, Retina and Symmetry report a six-preset sweep at five seeds each (link-expression output disabled), Circles is evaluated under per-node activation evolution, and the remaining rows use the base feedforward configuration — forward I→H and H→O connections only, with no hidden-to-hidden, backward, lateral, or self-loop primitives. §6.5.11 provides the systematic per-connection-type breakdown across the full benchmark suite. The experiments themselves illustrate EMR's enabling role: running approximately 1,410 evolutionary trials across 15 problems would have required months with the quadtree; EMR completed them in days.

Table 6.13: Cross-domain validation of EMR-HyperNEAT across 15 problems in four domains. Solve = percentage of runs reaching the convergence threshold; R^2 = coefficient of determination on held-out test data. ∼1,410 runs across the underlying campaigns.

Domain	Problem	I/O	Result	N
Boolean	XOR	2/1	66.7% solve[‡]	60
Boolean	Parity-3	3/1	76.7% solve[†]	30
Boolean	Parity-4	4/1	73.3% solve[†]	30
Boolean	Parity-5	5/1	83.3% solve[†]	30
Boolean	Parity-6	6/1	76.7% solve[†]	30
Visual	Retina (L/R)	8/2	100% solve[‡]	30
Visual	Visual Discrim.	25/1	71.7% solve[‡]	60
Visual	Symmetry (cont.)	49/4	100% solve[‡]	30
Visual	Circles	2/1	100% solve[§]	30
Control	CartPole	4/1	100% solve	30
Control	Acrobot	6/1	100% solve	30
Control	MountainCar	2/1	43% solve	30
Regression	CCM Financial	5/1	R^2=0.200	50
Regression	Earnings-Price	12/1	R^2=0.451	50
Regression	CalHousing	8/1	R^2=0.319	30

[†]Parity-3/4 use fitness threshold ≥ 0.85; Parity-5/6 use ≥ 0.80. Solve rates are not directly comparable across parity sizes.
[‡]Aggregated over multiple configurations: XOR over ten recurrence configurations (see text); Visual Discrimination over the six connection-type presets of §6.5.11; Retina and Symmetry over a six-preset sweep at five seeds each (link-expression output disabled), where every cell solves at the 0.90 convergence threshold.
[§]Under per-node activation evolution; the uniform-function baseline plateaus near 0.80 (§C.10.5).

Boolean and parity. EMR solves parity problems up to 6 bits using only the base substrate configuration: feedforward topology with sum aggregation and tanh activation. The XOR solve rate (66.7%, $N = 60$) is averaged across 10 recurrence configurations, where some configurations hurt performance; individual configurations reach 100%. The parity results use decreasing fitness thresholds that reflect increasing problem difficulty (0.85 for Parity-3/4, 0.80 for Parity-5/6; Appendix C.11): a solve here means reaching the

threshold, not reproducing the full truth table. The 0% uniform-monotonic parity rates quoted elsewhere in this thesis from the companion per-node function campaigns apply the stricter 0.95 default, so the two figures measure different bars rather than contradicting each other. To our knowledge, this is the first demonstration of indirect encoding scaling on combinatorial Boolean problems up to 6 bits at these convergence thresholds.

Visual and spatial. Retina achieves 100% ($N = 30$), Visual Discrimination 71.7% ($N = 60$), Symmetry 100% ($N = 30$), and Circles 100% ($N = 30$). These problems span input dimensionalities from 2 to 49 and require spatial pattern recognition with varying geometric structure. The solve rates confirm that EMR handles different input geometries; Circles additionally requires per-node activation evolution, its uniform-function baseline plateauing near 0.80.

RL control. CartPole and Acrobot both reach 100% ($N = 30$), validating EMR beyond supervised classification into reinforcement learning environments with continuous state spaces. MountainCar's lower solve rate (43%, $N = 30$) reflects the sparse reward structure of that environment rather than a substrate limitation: the agent must discover momentum-based strategies without intermediate reward signal.

Regression (domain limitation). CCM Financial regression yields R^2=0.200, below both NEAT baselines ($R^2 \approx 0.41$) and conventional ML methods ($R^2 \approx 0.42$–0.44). Earnings-Price (R^2=0.451) and CalHousing (R^2=0.319) confirm the pattern: indirect encoding produces functional but not competitive regression models. We report this as a domain limitation. The CPPN generates connectivity patterns optimized for classification boundaries, not smooth function approximation; the additional indirection layer between genome and phenotype appears to hinder continuous-valued prediction.

Three factors explain the regression ceiling. First, indirect encoding's weight-sharing generates globally smooth connectivity patterns better suited to sharp classification boundaries than gradual regression surfaces. Second, the evolved substrates contain approximately 20–50 hidden nodes, far fewer than conventional regression architectures require for 12-dimensional input. Third, the CPPN's coordinate-based weight generation cannot represent the non-monotonic feature interactions characteristic of financial time series without per-node function diversity, a capability validated in companion work on per-node activation function evolution (§8.4) but not yet applied to regression tasks.

High-dimensional input compatibility. As an additional test of EMR's scalability to high-dimensional inputs, we evaluated MNIST 10-class classification (784 inputs, 10 outputs) using default EMR parameters at depth 3. We compared two input layouts: a 1D flat encoding mapping the 784 pixels to evenly spaced coordinates along $[-1, 1]$, and a 2D spatial encoding mapping the 28×28 grid to $[-1, 1]^2$. The 2D layout yields a 1.5-percentage-point (pp) accuracy improvement over 1D ($N = 30$ per condition, Mann–Whitney $p < 0.001$, $r = 0.51$), confirming that EMR exploits geometric input structure when the coordinate system preserves it.

6.5.11 Connection Type Comparison

The connection type taxonomy introduced in Section 6.3 enables a systematic evaluation of recurrence that was computationally infeasible with the quadtree. The first result from the connection type experiments is decisive: enabling h→h connectivity roughly doubles the XOR solve rate. Feedforward substrates solve 45% of runs at `max_weight` = 8.0, whereas the two h→h discovery modes (sparse and dense) reach 92–98% under identical conditions. The gap is not because XOR *requires* h→h connections (Chapter 3 showed

that a single hidden layer suffices), but because the additional connectivity paths enlarge the search space in a way that makes useful topologies easier for evolution to discover. Full Recurrent configurations, which add backward, lateral, and self-loop connections simultaneously, achieve 73–90%, intermediate between feedforward and the targeted h→h types. On the hardware side, h→h types incur roughly 20% more VRAM than feedforward, yet GPU utilization stays between 2.5% and 3.4%, confirming that EMR remains memory-bounded rather than compute-bounded at these scales.

Hyperparameter sensitivity. An important practical question is whether the solve-rate advantage of h→h types depends on hyperparameter tuning. Table 6.14 addresses this by comparing results at `max_weight` = 5.0 (the ES-HyperNEAT default from Risi and Stanley [78]) against 8.0 (the EMR framework default used throughout this chapter), holding all other parameters fixed across depths 1–7, 8 populations, 30 generations, and 3 replications. Feedforward substrates are highly sensitive: their solve rate drops from 45% to 14% because a lower weight ceiling causes fewer connections to survive thresholding. By contrast, h→h types are robust (<2% change), and dense Full Recurrent is similarly stable (≤1% change). Sparse Full Recurrent degrades by 19%, falling between the two extremes. A richer set of h→h connections provides redundant routing paths, so the loss of individual connections to a tighter weight threshold has less impact on the network's overall function.

Iteration level and connection bloat. The two multihop rows in Table 6.14 reveal a counterintuitive pattern: higher iteration levels increase connectivity (3.4× more h→h connections at iter=3 vs. h→h iter=1) but solve rate *decreases* (92%→78%). Figure 6.14 confirms this degradation is consistent across depths, with iter=3 dropping to 53% at depths 6–7 versus 75–87% for standard h→h. The mechanism is connection bloat: additional iterations propagate connectivity through intermediate hidden nodes, inflating the substrate with redundant connections that dilute the signal-to-noise ratio for evolution. In our experiments, iteration level 1 emerges as the best-performing setting on average. The one advantage of higher iteration levels is faster convergence *when they do solve*: average generation 4.6 (iter=3) versus 5.8 (h→h iter=1), suggesting that the denser substrates provide more gradient information per generation but at the cost of a smaller basin of attraction.

Depth-specific patterns. Figure 6.14 breaks down solve rates by depth for the three primary connection modes plus two multihop tiers, each aggregated across five population sizes (100–1000) with 3 replications per cell. Feedforward's solve rate is flat across depths (42–47%), because its success depends almost entirely on population size (pop ≥500 solves; pop ≤300 fails), not on substrate resolution. In contrast, both h→h and Full Recurrent degrade at depth 7 (75–83% vs. 87–100% at shallower depths) because out-of-memory failures at pop 1000 exclude the highest-performing population from the sample. The depth 7 degradation is therefore a memory artifact, not an algorithmic limitation.

The timing overhead of recurrent connectivity is substantial at shallow depths (h→h is 3–6× slower per generation than Feedforward at depths 1–2) but converges toward parity at deeper substrates: only 1.6× overhead at depth 7. This convergence occurs because the CPPN query cost (which is identical across connection modes) grows as 4^D and dominates at deep substrates, while the h→h Phase 2 cost grows as H^2 but H is bounded by the active positions after variance filtering. In implementation terms, switching connection types in EMR is a single boolean mask operation on the pre-computed h→h weight tensor, whereas ES-HyperNEAT must repeat the entire Phase 2 quadtree traversal ($\mathcal{O}(H^2)$ sequential CPPN queries, one per hidden-node pair).

Figure 6.14 visualizes this pattern: feedforward's uniformly low band, the stable high-solve plateau of h→h and Full Recurrent through depth 5, and the monotonic degradation gradient of multihop configurations from upper-left to lower-right.

Table 6.14: Connection type comparison on XOR (depths 1–7, 30 generations, 3 replications). Conns = average h→h connections at max_weight (mw) = 5.0, except the multihop rows, which were run only at mw = 8.0; ΔConns = change versus mw = 8.0. Top five rows: 8 populations (CPU benchmark); multihop rows: 5 populations (GPU benchmark).†

Type	Discovery	mw = 5	mw = 8	Conns	ΔConns
Feedforward	—	14%	**45%**	0	0
h→h	Sparse	**93%**	**92%**	62,637	+1,781
h→h	Dense	**97%**	**98%**	2,532,449	+276,901
Full Recurrent	Sparse	71%	**90%**	40,773	−1,758
Full Recurrent	Dense	**72%**	**73%**	1,351,592	+201,085
Multihop iter=2†	Sparse	—	80%	144,759	—
Multihop iter=3†	Sparse	—	78%	204,923	—

†5 populations × 7 depths × 3 replications (GPU benchmark).

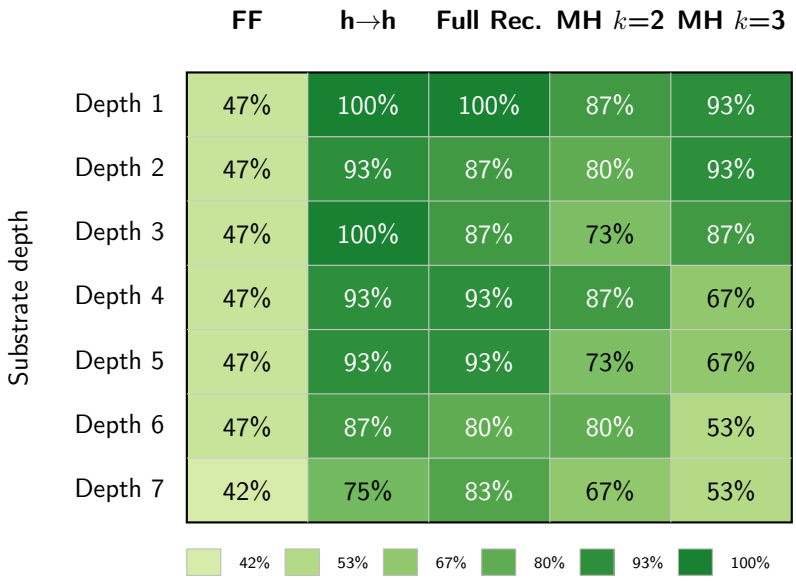


Figure 6.14: Connection type–depth solve rate heatmap for XOR (sparse discovery, depths 1–7, 5 populations × 3 replications per cell). Feedforward (FF) is uniformly low regardless of depth. Hidden-to-Hidden (h→h) and Full Recurrent maintain high solve rates through depth 5, with modest degradation at depths 6–7. Multihop connections degrade monotonically with both depth and iteration count, confirming the connection bloat hypothesis.

Per-task connection-type analysis. Table 6.15 evaluates all six connection type presets across 18 benchmark problems (depth 5, population 150 except Orientation and the two Hanoi formulations at 300, 10 replications; 1,080 total runs). The problems stratify into three tiers. *Differentiating problems* (5 tasks) show preset-dependent variation, four of them statistically significant; these are the informative cases. *Baseline problems* (6

tasks) achieve 100% across all presets, serving as complexity floors. *Capacity-limited problems* (7 tasks) achieve 0% across all presets, exposing architectural ceilings independent of connection type.

Among the differentiating problems, the connection type preferences diverge sharply, and revealingly. On XOR, recurrent connections are decisive: both Backward and Lateral achieve 100% solve rates, a statistically significant improvement over Feedforward's 20% ($p < 0.001$, Fisher's exact test). The default h→h preset, which uses only forward h→h connections, reaches just 50%, indicating that same-layer or reverse-direction paths are specifically what XOR's intermediate computation demands. This pattern parallels Chapter 4's aggregation finding, where competency-based combination (the `Performance-Weighted` strategy) similarly outperforms naive aggregation: in both cases, richer combinatorial degrees of freedom — recurrent connectivity here, weighted aggregation there — aid problems with non-trivial intermediate structure.

Visual Discrimination presents a contrasting profile. Backward connections again excel (100%, with 90% of seeds reaching perfect 1.0 fitness), but Lateral and Self connections appear to hurt performance, dropping solve rates to 40%, below even Feedforward's 90% ($p = 0.057$, short of significance at $n = 10$). A plausible reading is that same-layer crosstalk interferes with the spatial integration this task requires. Orientation inverts the pattern entirely: Feedforward solves every run because the tuning-curve mapping needs no recurrence, while Backward and h→h fall to 50%. Finally, Parity-3 is the hardest differentiating problem, where only Lateral reaches 50%. This reinforces the finding from XOR that same-layer connectivity supports the intermediate computation needed for non-linearly separable tasks.

Root-cause analysis. The cross-problem pattern reflects the computational structure of each task:

- **XOR** requires intermediate computation between inputs and outputs. Feedforward substrates must discover the correct hidden-layer topology through I→H and H→O connections alone; h→h connections provide alternative routing paths that make this discovery easier. Backward connections achieve 100% because feedback enables the substrate to iteratively refine its output, effectively implementing multiple processing passes through the same hidden layer.
- **Visual Discrimination** requires spatial feedback: the correct output depends on integrating activation across the input field, which benefits from recurrent information flow. Backward connections excel (100%) because they provide exactly this feedback. Lateral connections *hurt* (40%) because same-layer connections between spatially distant positions introduce interference without contributing useful computation for this particular spatial-integration task.
- **Orientation** requires purely spatial feature extraction: the task maps Gabor filter responses onto a smooth orientation tuning curve. Feedforward connections suffice (100%) because no iterative refinement is needed. Backward and h→h connections (50%) introduce unnecessary feedback that interferes with the spatial feature map.
- **Retina** is a simple left/right comparison that any minimal substrate can solve. All connection types achieve 80–100%, confirming that connection type selection matters only when the task demands it.
- **Parity-3** requires cascaded Boolean computation: the correct output depends on the XOR of three inputs, requiring two sequential processing stages. Lateral connections provide same-layer routing that enables intermediate results to propagate horizontally, analogous to the carry chain in a ripple-carry adder.

Capacity-limited problems. Seven problems achieve 0% convergence across all six presets: Two Spirals (barrier ∼0.75), Circles (∼0.78–0.82), Symmetry (0.8125 exact), Pattern (∼0.755), Turing (∼0.88–0.90), and Hanoi supervised/goal (∼0.778). These represent *architectural* barriers (the depth-5 substrate lacks sufficient representational capacity) rather than connection type limitations. No connection topology resolves the barrier; only increasing substrate depth or capacity would help. This negative result is informative: connection type is a free variable for these problems, and practitioners can choose the cheapest topology (feedforward) without performance loss.

Table 6.15: Solve rate by connection type across 18 problems (depth 5, population 150 except Orientation and the two Hanoi formulations at 300, 10 seeds each; 1,080 runs total). Bold marks the highest rate or statistical tie. p-values are Fisher's exact tests on solve counts between each row's extreme presets, except Visual*. Six Boolean and pattern tasks (360 runs, 100% in all presets) form a complexity floor; seven capacity-limited tasks (420 runs, 0% in all presets) expose architectural ceilings.

Problem	FF	h→h	Back	Lat	Self	Full	p
XOR	20	50	**100**	**100**	90	60	<0.001
Visual	90	70	**100**	40	40	90	<0.001*
Orientation	**100**	50	50	90	90	90	0.033
Retina	**100**	80	90	80	**100**	**100**	n.s.
Parity-3	0	30	40	**50**	0	0	0.033
Boolean gates[a]	**100%** all presets (240/240)						—
Pattern tasks[b]	**100%** all presets (120/120)						—
Capacity-limited[c]	**0%** all presets (0/420)						—

[a]AND, OR, NAND, NOR. [b]Sequence prediction, receptive field.
[c]Spirals, Circles, Symmetry, Pattern, Turing, Hanoi (supervised + goal).
*Mann–Whitney U on best fitness (Backward vs. feedforward);
the solve counts alone (10/10 vs. 9/10) do not separate these presets.

Two Spirals: a diagnostic negative result. The 0% solve rate across all six connection topologies warrants explicit diagnosis, as it identifies architectural limitations that inform future work.

The Two Spirals decision boundary requires smooth curvilinear separation of two interleaved Archimedean spirals, qualitatively unlike the axis-aligned boundaries of Boolean gates or the hyperplanar separations of parity and RL control tasks. Three properties of the current architecture preclude convergence. *Depth limitation*: at depth 5, approximately 340 hidden-node candidates survive variance filtering, potentially insufficient for dense 2D coverage of the spiral geometry. *Single hidden layer*: all experiments run at iteration level 1, so signal traverses a single hidden layer; Two Spirals typically requires ≥2 hidden layers to represent the nested rotational structure. *Uniform activation*: tanh produces piecewise-sigmoidal boundaries inadequate for smooth curves; oscillatory functions (sin) could trace spiral geometry but are not available in the uniform-activation configuration. Recurrence does not compensate: the bottleneck is representational capacity, not information flow topology.

These findings inform GEENNS's connectivity design (Chapter 7, Appendix B). The empirical result that backward connections are the most generally useful primitive suggests

that feedback connections should be enabled by default in GEENNS grid designs, while lateral connections should be task-dependent.

6.6 Analysis

The preceding sections established the empirical case for EMR: 12–34× per-generation GPU speedup on XOR at depths 5–7, population 1000 (Table 6.6), 100% solve rates where the quadtree fails, and a connection type taxonomy evaluated across 18 problems. This section examines the trade-offs underlying these results, quantifies the JIT amortization breakeven, and connects the scaling properties to the broader thesis program, in particular to the GEENNS vision of multi-CPPN compositional substrates described in Chapter 7.

Threshold differences as design choice. The eager reformulation changes execution order and, in doing so, constrains the filtering operations. ES-HyperNEAT's quadtree can apply position-dependent heuristics during traversal (e.g., spatial gradient checks with `band_threshold` 0.3). EMR requires uniform operations across all positions, because position-dependent branching would reintroduce the non-uniform shapes that prevent JIT compilation. The result is a simpler filtering pipeline: `variance_threshold` 0.03 for subdivision decisions and weight magnitude 0.1 for connection pruning. The lower variance threshold (0.03 vs. 0.5) combined with the different variance computation scale (raw CPPN outputs in range 0–4 versus scaled weights in range 0–25) yields more permissive topology discovery. More hidden-node candidates survive filtering, providing evolution with a richer search space at the cost of moderately larger substrates. The expanded Table 6.10 confirms that this permissiveness improves solve rates rather than degrading them.

JIT amortization breakeven. A practitioner's first question about EMR is: "how many generations before the JIT overhead pays for itself?" Table 6.16 answers this directly. The breakeven generation is computed as $g^* = t_{\text{JIT}}/(t_{\text{ESHN}} - t_{\text{EMR}})$: the number of generations at which EMR's cumulative cost (including JIT) equals ES-HyperNEAT's cumulative cost.

The breakeven profile has a sharp transition that is population-invariant in character. At depth 4, g^* varies widely across populations (6–23 generations), reflecting the small per-generation gap between ES-HyperNEAT and EMR at shallow depths, which is marginal for 30-generation experiments. The non-monotonic pattern at depth 4 (pop 100 breaks even fastest at $g^* = 6.2$, while pop 200 is slowest at 22.5) arises because ES-HyperNEAT's erratic solve behavior produces inconsistent per-generation times at shallow depths. At depth 5, all populations break even within 2.3 generations; by depth 6, all break even within the first generation. At depth 7+, the JIT cost is negligible relative to ES-HyperNEAT's per-generation time ($g^* < 0.1$ for all populations). The practical recommendation: use EMR at depth 4 only for long runs (>25 generations); at depth 5+, EMR dominates after the first generation regardless of population size.

Memory scaling and the streaming boundary. GPU memory scales as $\mathcal{O}(4^D)$ with depth. At population 500, depth 5 peaks near 1.8 GB, depth 6 near 7 GB, and depth 7 near 8.3 GB, approaching the 11 GB RTX 2080 Ti VRAM limit (Figure 6.15). Memory streaming (Algorithm 18) resolves this by offloading weight tensors to CPU RAM after each depth level, keeping GPU peak memory at $\mathcal{O}(\max_d |\text{Grid}_d|)$ rather than $\mathcal{O}(\sum_d |\text{Grid}_d|)$. The streaming boundary is visible in Figure 6.13: at depth 7, the GPU IQR band widens as runs transition from in-memory to streaming mode. Table 6.11 shows that streaming extends evaluation to depth 13 (358M positions), though at depth 13 each

Table 6.16: JIT amortization breakeven g^* (XOR, EMR GPU) across five population sizes. g^* is the number of generations at which EMR's total cost (including one-time JIT) equals ES-HyperNEAT's total cost, computed as $g^* = t_{\text{JIT}}/(t_{\text{ESHN}} - t_{\text{EMR}})$ using per-generation times from Table 6.10. JIT overhead ranges from 8–18 s at depth 4 to 11–39 s at depth 7. Pop 1000 depth 7 uses streaming mode.

Depth	Pop 100	Pop 200	Pop 300	Pop 500	Pop 1000
4	6.2	22.5	16.7	20.0	8.2
5	2.1	2.3	1.8	0.7	0.4
6	0.2	0.1	0.1	0.1	0.1
7	0.06	0.02	0.03	0.01	0.02

generation costs ∼5.8 hours (pop 300) or ∼67.5 hours (pop 1000, memmap-backed).

Figure 6.15 quantifies the VRAM footprint across the full depth×population parameter space (cells marked † in the figure are unstreamed projections rather than measured peaks) and maps the corresponding execution-strategy boundary: single-GPU for depths 1–6 (moderate populations), streaming for depths 7–9, chunked and CPU-resident streaming variants for depths 10–11, and disk-backed memmap arrays joining from depth 12 at population 1000 and depth 13 at population 300 (Table 6.11). GPU VRAM grows at approximately 3× per depth level (a lower base than the 4× position growth, because sparse representations amortize fixed overhead at higher depths). CPU RAM, by contrast, grows at only ∼1.15× per depth level (Figure 6.16 b) because Python's sparse representations and lazy allocation avoid the contiguous-buffer requirements of GPU memory. At depth 7 and population 1000, under memory streaming, CPU RAM peaks at 2,976 MB compared to a streamed GPU peak of 6,822 MB, a 2.3× gap that widens at deeper substrates. Unlike ES-HyperNEAT, where memory usage depends on which quadtree branches are explored (and therefore on CPPN weights), EMR's memory footprint is a deterministic function of depth and population size. This predictability enables automated resource allocation: given a target (depth, population) pair, the framework selects the cheapest viable strategy without trial runs or manual tuning.



Figure 6.15: Memory strategy by (depth, population) on an 11 GB GPU (RTX 2080 Ti). Cells show measured peak VRAM (MB) or projected values (GB, marked †, from ∼3.3× per-depth scaling). Red cells exceed 11 GB and are labeled by their limiting strategy; measured runs reach depth 12 with chunked and CPU-resident streaming at population 300, disk-backed memmap joining from depth 12 at population 1000 and carrying both populations at depth 13 (Table 6.11). Strategy is predictable before execution.

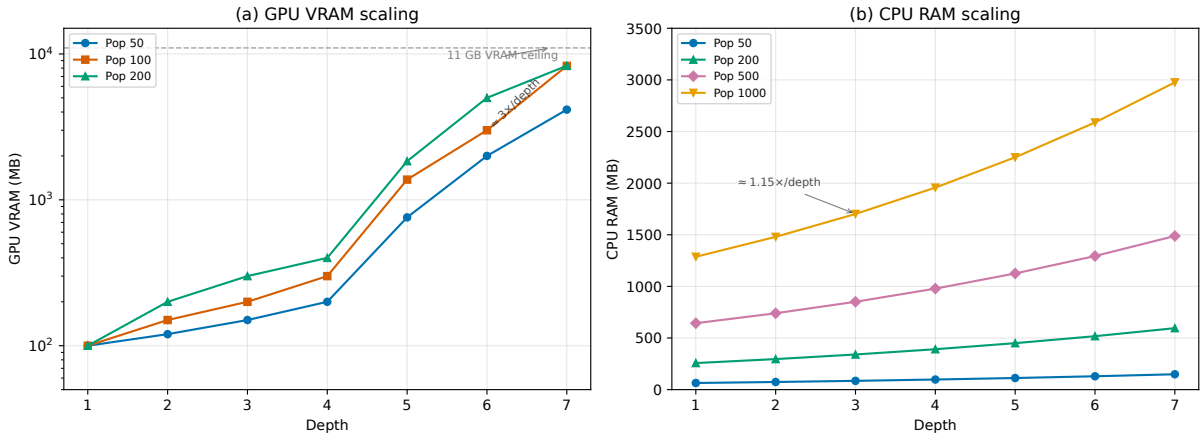


Figure 6.16: Memory scaling. **(a)** GPU VRAM grows ≈3× per depth level (dashed: streaming-mode at depths 8–9, pop 50; guide: 11 GB RTX 2080 Ti ceiling). **(b)** CPU RAM grows only ≈1.15× per level, staying below 3 GB at depth 7 with pop 1000, which explains why streaming from GPU to CPU between levels is effective.

Grid reusability and the GEENNS pipeline. Because the grid's contents are fixed by the maximum depth alone (the problem, the input/output layout, and the genome's CPPN weights all leave it untouched), a single depth-D grid can amortize across arbitrarily many evolutionary runs. The depth-7 instance, with its 87,380 positions and ∼683 KB footprint, is built once at start-up and reused thereafter without any per-run recomputation. For GEENNS, where 4 intra-grid CPPNs share the same spatial substrate (Appendix B), this means the grid construction and variance computation for one CPPN can be cached and reused by the remaining 3, compounding the per-CPPN speedup across the full pipeline.

Redundancy for parallelism: the core trade-off. At depth 7, EMR evaluates 87,380 positions unconditionally. ES-HyperNEAT's quadtree explores only the fraction where parent variance exceeds the threshold, often a small subset of the full grid. But that fraction is computed *sequentially*: each parent must be evaluated before its children. EMR's full set is computed in *parallel*: all 87,380 positions in a single `vmap` call, regardless of how many survive the variance threshold, against ES-HyperNEAT's data-dependent count of typically ∼200–2,000 at the same depth: roughly 40–400× more CPPN work. The result is nonetheless 34× faster (XOR, D7, pop 1000; Table 6.6); the specific ratio depends on benchmark and configuration. The memory cost of eagerness is bounded ($O(4^D)$ coordinate pairs per depth level, each requiring only source and target coordinates and CPPN outputs) and fits on a single 11 GB device up to depth 7 for populations up to ∼500 (Figure 6.15); beyond depth 7, memory streaming offloads the weight tensor to CPU RAM and, at the deepest levels, to disk-backed `memmap` arrays, extending feasibility to depth 13 at 0.2–1.0× of single-device speed (Section 6.5.8). This outcome instantiates a general principle from GPU programming: regular, redundant computation on parallel hardware outperforms optimally selective but sequential computation — wasted work on coordinates that will be filtered is cheaper than the sequential dependency chain that Chapter 5 proved cannot be optimized away.

Substrate selection, threshold scale, and solution quality. EMR's quality advantage over ES-HyperNEAT (100% vs. 0–33% at depths 6–7 for populations ≥ 500, Table 6.10) is not a side effect of the speedup; it follows from EMR's substrate-selection rule and threshold scale. The mechanism is the cold-start failure mode: the always-active root level and the more permissive variance threshold (Theorem 6.3 and the "Practical superset relation" remark that follows it) admit subdivisions where the top-down quadtree halts before exploring any children. The expanded Table 6.10 shows the effect clearly: ES-HyperNEAT's solve rates are erratic across populations (44% at pop 300 depth 4, 0% at pop 200 depth 4), while EMR's rates degrade predictably with small populations.

Cumulative speedup over 30 generations. Per-generation speedup compounds across a full evolutionary run. At depth 7 (pop 1000), ES-HyperNEAT requires $30 \times 2{,}020 = 60{,}600$ s (∼16.8 h). EMR GPU requires $39 + 30 \times 60 = 1{,}839$ s (∼31 min), including JIT: a cumulative ratio of ∼33× that asymptotically approaches ∼34× as generation count increases and JIT overhead becomes negligible. This transforms the experimental scope for future work: investigations into palette meta-learning, neuromodulation, and compositional architectures each require hundreds to thousands of evolutionary runs that would have been prohibitive under the quadtree.

Cross-domain generality. The cross-domain results (Section 6.5.10) confirm that EMR's substrate quality generalizes beyond XOR. Boolean and spatial tasks achieve 66.7–100% solve rates across 9 classification problems; 2 of 3 control environments are solved at 100%, validating EMR for reinforcement learning; regression yields $R^2 \leq 0.451$, consis-

tently below direct-encoding baselines. This is a limitation of the CPPN-to-substrate indirection rather than the eager reformulation itself. The bottom-up evaluation advantage (100% vs. 0–33% at depths 6–7 for populations ≥ 500, discussed above) extends to these domains: EMR avoids the catastrophic 0% cold-start failures seen in ES-HyperNEAT, degrading gracefully instead where representational capacity binds.

6.7 Synthesis: From Sequential Traversal to Tensor-Based Substrate Discovery

This section interprets the findings within the broader thesis narrative: how the tensor reformulation resolves the Chapter 5 incompatibility, what the connection type taxonomy enables for substrate architecture, and what the resulting uniform matrix representation means concretely for the GEENNS vision of compositional neuroevolution.

6.7.1 Resolving the Chapter 5 Incompatibility

Chapter 5 documented the fundamental incompatibility: the quadtree's adaptive subdivision produces unique topologies per CPPN, preventing the uniform tensor shapes required for vectorization on any hardware (§5.6). EMR resolves this by inverting the evaluation order: eager evaluation on fixed grids followed by variance-based filtering replaces sequential, variable-shaped refinement with uniform matrix operations. The resulting fixed-shape matrices become first-class linear-algebra objects, which is what enables the companion work on per-node function evolution, neuromodulation, and aggregation selection documented in §8.4.1.

The empirical confirmation is decisive: 12–34× per-generation GPU speedup on XOR at depths 5–7, population 1000 (Mann-Whitney $p < .001$, Cliff's $\delta \geq 0.59$ from depth 4 onward), yielding a cumulative ratio of $\sim$33× over a 30-generation XOR run at depth 7 (§6.5.3). At depth 7, where the original ES-HyperNEAT solves 0% of runs at populations up to 750, EMR achieves 100% from population 500 up. Runtime variance is roughly 40× tighter (IQR-based, pooled across population sizes), reflecting the fixed-shape grid evaluation replacing the data-dependent quadtree traversal.

6.7.2 Connection Type Taxonomy as Architectural Freedom

The six connection types introduced in §6.3 are more than an ablation variable. They provide a systematic language for specifying recurrence patterns in evolved substrates: feedforward, hidden-to-hidden, backward, lateral, self-connections, and their combinations. This taxonomy was practically unreachable with the quadtree, since evaluating six configurations across multiple depths would multiply an already prohibitive cost. EMR's speedup makes the full combinatorial exploration tractable.

The empirical findings are architecturally informative. Hidden-to-hidden connections achieve 92–98% solve rates versus 45% for feedforward, establishing recurrence as a near-requirement for Boolean benchmarks. Cross-problem validation shows backward connections excel for tasks requiring feedback (100% on both XOR and Visual Discrimination), while backward connections are the most generally useful primitive across the five differentiating problems, which is why GEENNS grid designs should enable feedback by default

(§6.5.11). Combined with the Chapter 4 partitioned-experts architecture, where multiple specialist substrates must each discover appropriate connectivity, the connection type taxonomy enables each specialist to evolve its own recurrence pattern: a combinatorial design space that was computationally inaccessible before EMR.

6.7.3 What EMR Enables: A Quantitative Before-and-After

Table 6.17 summarizes the concrete impact of EMR across the research program, comparing capabilities before and after the lazy-to-eager reformulation.
The central message is that EMR makes previously impossible research feasible. Connection type exploration, multi-depth scaling studies, and multi-CPPN architectures were all computationally unreachable before EMR. After EMR, they become routine experiments. Cross-domain validation (§6.5.10) confirms that this generality extends beyond Boolean benchmarks: EMR substrates solve visual, control, and (with documented limitations) regression tasks across approximately 1,410 additional runs.

Beyond speed: qualitative capability changes. The EMR reformulation unlocks capabilities that the quadtree could not reach at any time budget. At depth 7, ES-HyperNEAT solves 0% of runs at every population up to 750 and 33% at 1,000, making substrate discovery at this resolution effectively unreliable; EMR-HyperNEAT solves 100% of its measured runs from population 500 up (Table 6.10; the depth-7 rates at populations 750 and 1,000 are carried estimates): problems that were practically unsolvable become solvable. Substrate discovery also scales to depth 13 (358M positions) via memory streaming, six depth levels beyond the D7 where ES-HyperNEAT already fails. The 15-problem cross-domain validation (§6.5.10) further includes financial regression with 94,464 observations, a scale that ES-HyperNEAT's sequential quadtree cannot process within practical time limits; EMR delivers a 5.5× per-generation speedup at depth 6 (population 100) on that benchmark.

Table 6.17: Quantitative comparison of capabilities before EMR (quadtree-based ES-HyperNEAT) and after EMR. Values marked with * are from Chapter 5 measurements; values marked with † are from this chapter's experiments; ‡ marks the two speedup rows that rest on the depth-7 streaming estimate (the 34× endpoint and the ∼33× cumulative ratio; §6.5.3). Speedup rows: XOR, GPU, population 1000; ratios vary with task, depth, and population size.

Capability	Before (Quadtree)	After (EMR)
Max practical depth	5–6 (solve rate $\leq 33\%$)*	13 (358M positions; solves recorded through D12)†
Solve rate at D7 (pop 500, measured cell)	0%†	100%†
Per-gen speedup (D5–7)	Baseline	12–34×†‡
30-gen total speedup	Baseline	∼33×†‡
Runtime variance (D7)	High (data-dependent)*	∼40× tighter†
Population vectorization	Impossible (unique topologies)*	Native (fixed grid)†
Connection type exploration	Infeasible (cost)*	6 types validated†
Multi-depth experiments	456 trials (months)*	1,000+ trials (days)†

Beyond speed: the representation itself. The deeper consequence of the reformulation is not the speedup but the kind of object it produces. Where ES-HyperNEAT's

quadtree yielded opaque, variable-shaped Python data structures that could only be processed one individual at a time, EMR yields uniform weight matrices, and uniform matrices can be composed, differentiated, cached, and streamed using the full arsenal of linear-algebra tooling that modern numerical frameworks provide. This change in representation is what makes the companion research directions listed in §8.4.1 computationally possible. Per-node activation function evolution, neuromodulation via gain-modulated receptor densities, per-node aggregation selection, bio-inspired palette meta-learning (meta-evolving which activation and aggregation functions are available to nodes across generations), and co-evolved neuromodulatory signals all exploit the same structural property: adding a CPPN output dimension to an EMR matrix is a column append, not an algorithmic redesign. Under the quadtree approach, these extensions were infeasible for a reason deeper than speed: the variable-topology output admitted no natural way to attach additional per-node metadata in a vectorizable form. EMR's tensor representation therefore opens new directions for adaptive substrate neuroevolution: the research program described in this thesis and its companion work demonstrates — at practical scale — that indirect encoding can be extended compositionally rather than monolithically, and the matrix representation is the prerequisite that makes that extension practical.

6.7.4 Implications for GEENNS

EMR-HyperNEAT's tensor reformulation transforms the GEENNS multi-grid architecture from infeasible to routine. The projected envelope (Table B.3 in Appendix B) extrapolates from the measured 12–34× per-generation speedup (XOR, GPU, depths 5–7, pop 1000; §6.5.3) to a modest (k=5, T=10) GEENNS scale (34 CPPNs, Appendix B): under XOR-scale per-trial cost assumptions, the full pipeline drops from approximately 250 projected CPU-days under the quadtree baseline to approximately 4.9 projected GPU-days with EMR plus pruner. These are lower bounds (per-trial cost grows super-linearly on harder tasks), but the order-of-magnitude gap between months and days is the enabling contribution: without vectorizable substrate discovery, multi-CPPN compositional systems cannot be investigated systematically. Equally important, the matrix-representation benefits elaborated in §6.7 carry directly into GEENNS: each specialist substrate can inherit per-node activations, neuromodulation, or aggregation at negligible marginal cost.

6.7.5 Connection Types as GEENNS Building Blocks

The six-configuration taxonomy maps directly to GEENNS's three connectivity categories, providing a principled vocabulary for multi-grid substrate design:

Intra-columnar connections
(vertical within a single column) correspond to feedforward or hidden-to-hidden types. Given the §6.5.11 finding that h→h roughly doubles the XOR solve rate, intra-columnar connectivity should include at minimum hidden-to-hidden recurrence, providing the internal state that feature extraction requires.

Inter-columnar connections
(horizontal between columns in the same grid) map naturally to backward or full recurrence types. The cross-problem finding that backward connections excel on feedback-dependent tasks (§6.5.11) supports using them for lateral communication between columns, enabling columns to share intermediate representations.

Inter-grid connections
(between different specialist grids) can use any type, with the taxonomy providing a principled basis for selection. For GEENNS, the mapper-mediated routing between grids is analogous to the output-to-input connections in the connection type taxonomy; the mapper CPPN generates inter-grid weights just as the connectivity CPPN generates intra-substrate weights.

The caching mechanism described in Section 6.4.1 matters for GEENNS: when multiple CPPNs share the same grid geometry, as intra-columnar CPPNs within a single grid would, the variance computation can be cached and reused, compounding the per-CPPN speedup across the four intra-grid CPPNs of each grid (Appendix B).

6.7.6 Generalizability Constraints

Several limitations constrain the generalizability of the EMR results:

1. **JIT overhead at low depths.** EMR is slower than the quadtree at depths 1–3 due to a one-time JIT compilation overhead (4–60 seconds depending on depth). The crossover occurs at depth 4 on GPU and depth 5 on CPU; for shallow substrates, the in-framework ES-HyperNEAT baseline of Table 6.6 may remain preferable. EMR's persistent disk cache (§6.4.1) amortizes this overhead across runs sharing the same depth and population; without cache reuse, each trial pays the full JIT cost, pushing the practical crossover beyond depth 4.

2. **Multi-GPU scaling.** Multi-GPU configurations show 1.5–1.7× *slowdown* rather than speedup, reflecting Python coordination overhead. EMR currently provides memory scaling (larger grids via streaming out of device memory) rather than speed scaling. True multi-GPU acceleration would require moving the coordination loop into XLA.

3. **Connection type analysis scope.** The six-configuration taxonomy was evaluated across 18 problems, but only five of them differentiate between presets; the rest sit at a 100% floor or a 0% ceiling. Whether the findings (e.g., hidden-to-hidden dominance) generalize to continuous-valued or high-dimensional tasks is untested.

Thesis-level limitations, including benchmark scope and replication counts, are consolidated in Section 8.3.1.

6.8 Chapter Summary

This chapter completed the answer to TRQ3 by demonstrating that massively parallel execution *can* overcome ES-HyperNEAT's computational bottleneck, once the discovery approach is redesigned from lazy adaptive refinement to eager multi-resolution evaluation with variance filtering. Section 6.1's three questions resolve as follows: eager evaluation is GPU-parallel while retaining variance-driven substrate selection (RQ1, Theorem 6.3); the speedup reaches 12–34× per generation at depths 5–7 with the crossover at depth 4 on GPU and depth 5 on CPU (RQ2, §6.5.3); and solution quality improves rather than degrades, with EMR solving 100% of measured deep-substrate runs at populations $\geq$ 500 where the quadtree's rates are erratic (RQ3, §6.5.6). Table 6.18 summarizes the key results.

The answer to TRQ3. Chapter 5 demonstrated that massively parallel execution *cannot* overcome the quadtree bottleneck: unique topologies per CPPN prevent population-level vectorization, four optimization strategies saturate at 1.65× on the reference configuration, and HSHG over-discovers by an order of magnitude. This chapter demonstrates that the bottleneck yields to *algorithmic redesign*. EMR-HyperNEAT's eager evaluation on a fixed multi-resolution grid produces uniform tensors by construction, enabling the batched vectorization that the quadtree structurally prevents. Variance filtering recovers adaptive sparsity post-evaluation, preserving the quadtree's core capability (task-appropriate topologies) without its core limitation (unique tensor shapes). The result is that previously unreachable depths become routine, with the detailed numbers collected in Table 6.18. Cross-domain validation across 15 problems in 4 domains (Section 6.5.10) confirms that the solve-rate advantage generalizes beyond the primary XOR benchmark. The solution was not faster hardware but a change in the order of operations: evaluate everywhere, then filter, rather than decide where to look, then evaluate.

Transition to Chapter 7. With both algorithmic limitations diagnosed (Chapters 3–5) and a tensor-accelerated substrate engine available (this chapter), the thesis has established the technical infrastructure needed for compositional neuroevolution. Chapter 7 presents the GEENNS vision and reports proof-of-concept prototype evidence that validates the compositional pipeline of frozen specialist grids and evolved mappers, before Chapter 8 synthesizes the contributions across all preceding chapters.

Table 6.18: Summary of Chapter 6 key results. Unless otherwise noted, speedup and solve-rate figures are measured on the XOR benchmark, the standard ES-HyperNEAT evaluation task [78].

Result	**Value**	**Section**
Depth-scaling relationship (XOR benchmark)		
Per-generation speedup (D5–7)	12–34× (GPU, pop 1000)	§6.5
Total runtime speedup (30 gen)	∼33× (cumulative, D7 pop 1000)	§6.5
Speedup–depth correlation	Spearman $\rho = 0.96$	§6.5
Statistical validation		
Mann-Whitney significance	$p < .001$ at D4–7	§6.5
Effect size (Cliff's δ)	≥ 0.59 (large) at D4+	§6.5
Runtime variance ratio	∼40× tighter at D7	§6.5
Solution quality (XOR benchmark)		
EMR solve rate (pop=300)	100% (all depths)	§6.5
ES-HN solve rate (D6 / D7, pop=300)	33% / 0%	§6.5
Scalability		
Maximum depth tested	13 (358M positions)	§6.5
Grid precomputation	87,380 pos (D7), ∼683 KB	§6.4
h→h connection cache	1.01–1.53×	§6.4
Real-world validation		
CCM financial speedup	5.5× at D6	§6.5
Connection types (XOR benchmark)		
h→h solve rate	92–98%	§6.5
Feedforward solve rate	45%	§6.5
Experimental scope		
Total runs	3,592	§6.5
Cross-domain validation (§6.5.10)		
Classification/Boolean	9/9 problems solved ≥66.7%	§6.5.10
Control environments	2/3 solved at 100%	§6.5.10
Regression ceiling	$R^2 \leq 0.451$	§6.5.10
Additional runs	∼1,410	§6.5.10

Part IV

The GEENNS Vision

Chapter 7

The GEENNS Vision: Proof of Concept and Future Directions

> Your mind is for having ideas, not holding them.
>
> David Allen

This chapter serves two purposes: it frames the thesis within the broader GEENNS research program, and it reports proof-of-concept prototype evidence that validates the compositional pipeline this thesis enables. We present the biological foundations and algorithm architecture that motivate GEENNS, identify three barriers that Chapters 3–6 remove or enable solutions to, and close with an honest assessment of what remains open. The prototype demonstrates that frozen specialist grids composed through evolved mappers work at small scale and maintain 100% success up to 5-grid Boolean scale (and a 6-grid probe where the augmented pipeline first falters and gated routing recovers most of it (72% of seeds)), where every monolithic baseline collapses to 0%; this finding is based on approximately 3,100 runs across Boolean compounds and continuous additive compounds (through eight gates). The best individual mapper does not zero-shot unseen compositions (in the compositional generalization test, per-target mean fitness spans 0.53–0.80, and no seed crosses the 0.95 threshold), yet warm-start transfer reveals that when the channel assignment tracks the recomposed target, the source-evolved populations already contain zero-shot solvers of the unseen task, typically the converged champion itself. Compositional generalization is therefore not answered in the negative; it is refined from "does it generalize?" into "which recompositions preserve the evolved strategy, and how do we evolve mappers that survive the ones that do not?" The infrastructure built in earlier chapters now makes that question approachable. This chapter delivers a proof of concept and charts the path for future work. The formal mathematical specification appears in Appendix B.

7.1 Motivating Hypotheses and Scope

GEENNS draws on three disciplines: distributed cognition from psychology, where human problem-solving is collaborative across specialized faculties [42]; the cortical column hypothesis from neuroscience, where Mountcastle [69] identified a repeated canonical microcircuit that produces diverse computations through varied input connectivity; and evo-

lutionary computation from computer science, where architecture is discovered through selection rather than hand-designed [31, 88].

Their synthesis produced the core GEENNS insight: *intelligence is not a single system but a collaboration of specialists*, and the collaboration pattern must itself be evolvable. GEENNS proposes to evolve the specialists (grids), freeze them, and then evolve the collaboration (mappers) for each new task.

7.1.1 The Seven Hypotheses

GEENNS rests on seven hypotheses. Two of them (H5 and H7) form the conceptual core; the remaining five concern substrate-level prerequisites tested in this thesis (H1–H4) and a multi-task mechanism validated in companion manuscripts (H6).

H1. The hyperparameter space of adaptive substrate neuroevolution can be navigated efficiently without sacrificing solution quality (TRQ1; Chapter 3).

H2. Adaptive substrates extend to high-dimensional inputs through spatial decomposition that overcomes central-bias representational collapse (TRQ2 and TRQ4; Chapter 4).

H3. ES-HyperNEAT's quadtree-based substrate discovery is structurally incompatible with GPU parallelism and must be replaced rather than optimized (TRQ2; Chapter 5).

H4. Adaptive substrate discovery can be reformulated for massively parallel execution without sacrificing adaptivity (TRQ3; Chapter 6).

H5. Consensus of distributed specialists (multiple grids, each processing partial information, combined through evolved coordination) produces more generalizable solutions than monolithic approaches.

H6. Neuromodulatory gain modulation, encoded as per-task signals that scale the influence of synaptic connections, enables a single substrate topology to support multiple tasks without catastrophic forgetting.

H7. Specialization of intelligence, where different computational units develop expertise in different domains, emerges naturally under evolutionary pressure with compositional task structure.

H1–H4 are answered in this thesis. H5 and H7 are partially addressed: the prototype (Section 7.6) demonstrates specialist composition at 5-grid scale but not the emergent specialization or robust zero-shot generalization that the hypotheses predict. H6 is answered in companion manuscripts that build on the thesis infrastructure (see Chapter 8 Table 8.5).

7.1.2 Research Scope

The research program comprises nine tasks organized into three phases. Table 7.1 maps each task to its status in this thesis, illustrating both the narrowing of scope and the deepening of substrate-level investigation.

7.1.3 From Breadth to Depth

The research program initially envisioned a broader cognitive-scale system. The thesis delivers something narrower but more rigorous: a substrate-level engineering investigation that removes the computational barriers blocking adaptive substrate neuroevolution. The nine tasks collapsed into a prerequisite stack: before we could pursue substrate expressiveness (Task 4), multi-task operation (Task 5), or lifelong learning (Task 9), we had to solve the computation problem (Task 7) and validate the spatial decomposition principle (Task 6). Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT)'s speedup has since opened Tasks 4 and 5 for companion manuscripts (per-node activation and aggregation address substrate expressiveness, and neuromodulation addresses multi-task operation), while Task 9 (lifelong learning) and the GEENNS-level integration (Task 8) remain open. The prototype reported in this chapter provides initial evidence that the compositional architecture is viable.

7.2 From Substrate Engineering to Composition

Three discoveries shaped the thesis trajectory: the central bias (Chapter 4), the quadtree–GPU incompatibility (Chapter 5), and the EMR reformulation that resolved it (Chapter 6). The combined effect reduces the GEENNS pipeline from computationally infeasible to tractable (Table B.3 in Appendix B).

The prototype (Section 7.6) validates the compositional architecture up to 5-grid Boolean scale (100% composition vs. 0% for every monolithic baseline) and, in the continuous regime, through eight additive gates, with the only observed continuous-regime scaling limit (multiplicative composition beyond 2 gates) localized to the operator rather than the substrate; the thesis provides the engineering foundations required to pursue it at larger scale. The full research trajectory is presented in Appendix A.

7.3 Biological Inspiration

GEENNS draws inspiration from three principles: neocortical columnar organization, evolutionary architecture search, and emergent modularity under evolutionary pressure. We extract the computational insight from each and distinguish what is borrowed from biology from what is a computational design decision. GEENNS is inspired by the neocortex; it does not simulate the neocortex. A fourth biological principle, prior to these three, motivates the encoding itself and is treated in §1.1: DNA compresses the phenotype's construction rule into a genome orders of magnitude smaller than the phenotype it expresses, and indirect encoding in neuroevolution is the computational analog of that asymmetry. The present section concerns organizational principles (how the substrate is arranged and reused), not the encoding principle (how it is generated); the two operate at different levels and compound rather than substitute.

7.3.1 Columnar Organization: The Extracted Principle

The mammalian neocortex is organized into columns of approximately 0.5 mm diameter, each sharing the same canonical microcircuit [69]. This pattern arises independently across mammalian lineages [48], suggesting a fundamental organizational advantage: as Hawkins [36] argued, a repeated computational template, wired differently, produces functional

Table 7.1: Research tasks and their thesis disposition. "Thesis" indicates a contribution presented in a thesis chapter; "Companion" indicates a direction validated in a companion manuscript that builds on the thesis infrastructure but is not part of this thesis (see Chapter 8 Table 8.5); "Future" indicates a direction the thesis infrastructure opens but that no companion manuscript yet addresses.

#	**Original Task**	**Thesis Status**	**Location**
1	NEAT/HyperNEAT foundation	Completed	Ch 2
2	ES-HyperNEAT	Completed + diagnosed	Ch 5
3	Hyperparameter optimization	Completed (TPE, 29% MNIST)	Ch 3
4	Substrate expressiveness	Companion (per-node activation/aggregation)	§8.4
5	Multi-task substrates	Companion (neuromodulation)	§8.4
6	Spatial decomposition	Completed (partitioned experts, 43% MNIST)	Ch 4
7	Tensor reformulation for substrate discovery	Completed (EMR-HyperNEAT)	Ch 6
8	Compositional architecture	Proof-of-concept validated	Section 7.6
9	Lifelong learning	Future	§8.4

diversity without a unique design per region. (Additional biological context appears in Appendix A.)

GEENNS does not imitate the neocortex. Mammalian columns have approximately six layers because biological evolution optimized for axon propagation speed, metabolic cost, physical volume, and ion-channel dynamics [64]. Machine substrates face different constraints (Single Instruction, Multiple Threads (SIMT) parallelism in §2.4, memory bandwidth, cache locality), so GEENNS prescribes no fixed layer count. The canonical microcircuit is a NEAT genome: evolution finds the optimal column template for the target hardware.

The extracted principle is *architectural homogeneity with functional diversity through connectivity* (Figure 7.1). All columns share the same evolved internal structure; different input routing patterns produce different functional behaviors. HyperNEAT applies this at the connection level (a single Compositional Pattern-Producing Network (CPPN) generates all weights); GEENNS applies it at the structural level (a single evolved template instantiated across all columns).

The case for evolutionary architecture search over hand-design, including the universal approximation guarantee and the combinatorial argument for evolved substrates, is presented in Appendix B.

7.3.2 Emergent Modularity Under Evolutionary Pressure

Kashtan and Alon [49] showed that modularly varying environments, where sub-problems are reused across changing goals, spontaneously produce modular network structure. Clune et al. [20] extended this: a connection-cost penalty produces modularity even in fixed environments. The implication is that compositional task structure should produce specialized, reusable grids without an explicit modularity objective.

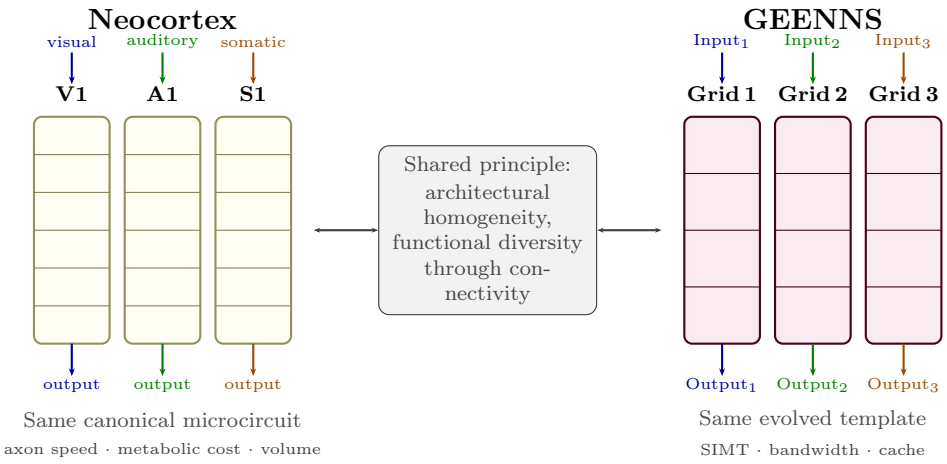


Figure 7.1: The columnar principle shared between neocortical and GEENNS organization. Left: three cortical areas (visual, auditory, somatosensory) share the same six-layer canonical microcircuit but produce different functions through different afferent connectivity. Right: GEENNS columns share the same NEAT-evolved template but produce different specialist behaviors through different evolved input routing patterns.

GEENNS prescribes *structural* modularity (separate grids with separate CPPNs); whether grids develop *functional* modularity supporting compositional reuse depends on evolutionary dynamics. Prototype evidence (Section 7.6) is consistent: specialist grids compose to solve 3-gate compound tasks with 100% success versus $\leq$3.3% for uniform-activation monolithic baselines, and the gap becomes decisive at 5-grid scale where every monolithic baseline (including the dynamic per-node function baseline) collapses to 0%. The concrete experimental signature is transfer without retraining (Figure 7.2): frozen modules compose into new combinations without weight modification. The prototype validates this up to 5-grid Boolean scale; whether it extends to harder, non-conjunctive domains is an empirical question for future work.

7.3.3 From Mimicry to Emergence

Biological inspiration provides constraints (the columnar principle, the template-reuse motif), but specific instantiations should emerge from evolution, not analogy. The shift this implies is from prescription to emergence.

The shift shows up in specific design choices. We do not prescribe six layers; the layer count is evolved (Section 7.4.2). Nor do we prescribe how grids communicate: mappers evolve to discover routing patterns (Section 7.4.3). EMR-HyperNEAT is a computational design, not biological mimicry; per-node function evolution, realized as additional CPPN output dimensions, is explored in companion manuscripts that build on the thesis infrastructure (see Chapter 8 Table 8.5), with no biological constraint on which functions are selected.

The prototype (Section 7.6) illustrates this concretely. The compound task $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f)$ is solved because evolution discovered that routing through frozen specialists produces correct outputs, not because composition was designed in. This is the central insight: shift from "copy the brain's structure" to "create the brain's conditions for emergence." The limitation matters as much: the emergence ob-

served is task-specific, not general. Demonstrating that these conditions reliably produce emergent compositional reasoning remains the open problem.

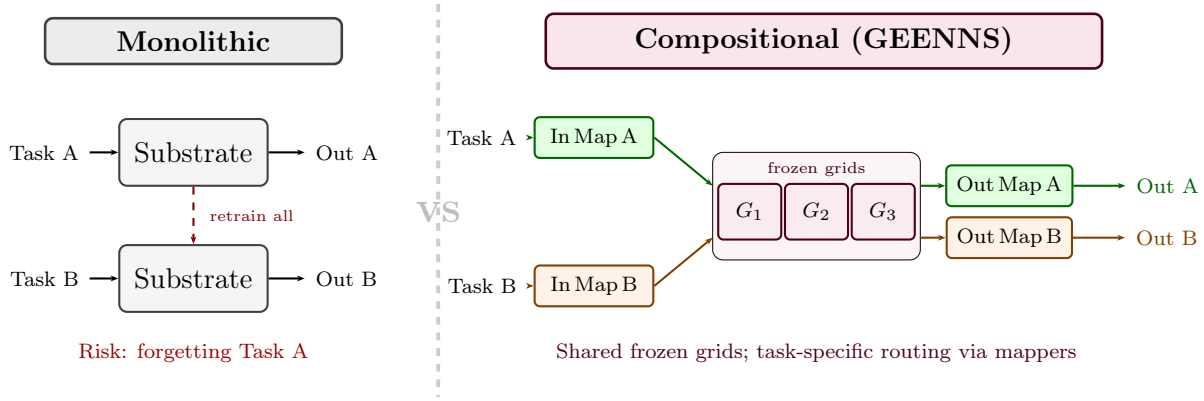


Figure 7.2: Monolithic versus compositional substrate design. Left: a monolithic substrate handles each task, requiring full retraining and risking catastrophic forgetting when tasks change. Right: GEENNS organizes frozen specialist grids and evolves task-specific input and output mappers. New tasks require only new mapper pairs; grids retain their specialized capabilities across all tasks.

7.4 GEENNS Algorithm Design

The following sections present the GEENNS algorithm as a *design specification*: a target architecture motivated and enabled by the substrate infrastructure developed in Chapters 3–6, not a description of implemented behavior. No implementation or experimental validation exists for this design beyond the prototype evidence in Section 7.6; all plasticity mechanisms, developmental stages, and the multi-level CPPN hierarchy are theoretical proposals that serve as a roadmap for future implementation.

GEENNS aims to enable compositional reasoning through the architectural separation of computational components (grids) and composition logic (mappers). The rest of this section distinguishes what is designed (below), what is implemented (Chapters 3–6), and what remains theoretical (the three-type mapper formalism with explicit inter-grid routing, plasticity).

7.4.1 Core Architectural Principles

GEENNS rests on five design principles, each motivated by a specific observation about biological or computational intelligence:

1. **General-purpose over task-specific.** GEENNS evolves substrates, not solutions. A grid is not trained to classify digits or balance poles; it is evolved to perform a class of computations (feature extraction, transformation, integration) that can be composed for many tasks. This principle addresses the hypothesis that intelligence is a set of specialized capabilities, not a single general capability (H7).

2. **Indirect over direct encoding.** Connectivity patterns are generated by CPPNs from spatial coordinates, exploiting geometric regularities (symmetry, repetition,

locality). A compact genome thus encodes a large-scale structured substrate. This principle is inherited from HyperNEAT, refined in ES-HyperNEAT, and extended in EMR-HyperNEAT (Chapter 6).

3. **Developmental over static.** The substrate is not a fixed graph. It forms through a staged developmental process (node generation, layer organization, connection formation, refinement, and operational maturation), such that the same genome reliably produces the same phenotype. This principle is inspired by biological neurodevelopment, where a compact genome orchestrates the construction of a brain containing billions of connections.

4. **Adaptive over fixed.** In the full GEENNS design, connections would be subject to plasticity: Hebbian strengthening, homeostatic regulation, structural pruning, and neuromodulatory gating. This would enable the substrate to adapt during operation without requiring re-evolution. Three of these (Hebbian, homeostatic, structural) remain future work; neuromodulatory gating has been demonstrated in companion work (see Chapter 8 Table 8.5).

5. **Modular over monolithic.** Components function independently or together. A single grid can solve a task. Multiple grids can be composed through mappers to solve tasks that require capabilities beyond any single grid. This principle implements the distributed-specialist hypothesis (H5).

7.4.2 The Columnar Substrate

A grid is a two-dimensional substrate containing columns and layers. Each column is a vertical processing unit; each layer is a horizontal functional division, with number and function evolved rather than predetermined. Nodes are addressed by (c, l, n): column index, layer index, and node index within the layer.

The key structural feature is the *canonical microcircuit template*: all columns within a grid share the same internal structure, defining layer organization, node properties, and intra-columnar connectivity. The template is evolved via NEAT [88], applying the columnar principle from Section 7.3.1: the same circuit, replicated across a grid, produces different functions depending on input connectivity.

What "frozen" means. A *frozen* grid has its *columnar scaffold* fixed (layer count, inter-layer connectivity, inter-column topology, and node placement) after the developmental phase. This parallels neocortical development, where the large-scale structural scaffold stabilizes after neurogenesis and pruning while synaptic plasticity continues [69]. In the full design, frozen grids retain potential for weight-level plasticity (Hebbian, homeostatic, or neuromodulatory; Appendix B), and new columns can be instantiated from the same template without modifying existing ones. In the current prototype (Section 7.6), "frozen" means fully static; scaffold-frozen, plasticity-active grids are future work.

Grids are evolved using EMR-HyperNEAT (Chapter 6), which generates the connectivity patterns from CPPN queries over normalized spatial coordinates. Each node could have an evolved activation function selected from a palette of candidate functions and an evolved aggregation function that determines how incoming signals combine (§8.4). Task-specific behavior within a shared grid topology could be achieved through neuromodulatory gain modulation.

Whether evolution actually discovers grids that specialize in interpretable ways remains an open question. The partitioned-experts experiments in Chapter 4 provide partial evidence: spatially partitioned experts do develop distinct specializations (peripheral vs. central pixel processing), though at a much simpler level than the full grid specialization envisioned here.

The full architecture (minicolumns, columns, and multi-level CPPNs) is detailed in Appendix B.

7.4.3 Mappers as Compositional Coordinators

The mapper approach is the core mechanism for compositional reasoning in GEENNS, with zero implementation beyond the prototype. We elaborate the design here because every substrate-level barrier we solve is a prerequisite for mappers to function.

Mappers are evolved functions that coordinate how grids are used for a given task, in three types:

- **Input mappers** decompose a task's input into grid-appropriate queries; the mapper does not *know* what each grid does but evolves to discover which routing patterns produce correct outputs.
- **Output mappers** compose grid responses into task solutions by extracting and combining relevant signals.
- **Inter-grid mappers** route information between grids during processing, enabling compositional depth (e.g., a transformation grid's output feeding an integration grid).

Each mapper is evolved via NEAT. For a new task, only mappers are evolved; grids remain frozen. Although the input and output mappers are described separately above, their fitness signals are necessarily coupled: routing is only judged successful when the composition recovers the right answer, so the two cannot be evolved independently. The prototype realizes this coupling within a single NEAT mapper that receives both raw inputs and grid outputs, treating input-routing and output-composition as functional regions of one evolved network; the full GEENNS design retains the option of separating them into distinct co-evolving populations. This is the proposed lifelong learning mechanism: task-specific routing without substrate modification.

The critical question, whether this works, is partially answered by the prototype (Section 7.6). The risks are substantial: mapper complexity might grow exponentially with task count, evolution might not discover generalizable decomposition strategies, and routing overhead might be prohibitive.

End-to-end walkthrough. Consider $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f)$: three Boolean operations on independent input pairs whose results are combined. The pipeline:

1. **Specialist training.** Three grids (G_{XOR}, G_{AND}, G_{OR}) are independently evolved on their respective gates, then frozen.
2. **Input mapper.** A NEAT-evolved mapper routes (a, b) to G_{XOR}, (c, d) to G_{AND}, and (e, f) to G_{OR}.

3. **Grid processing.** Each frozen grid processes its pair independently.
4. **Output mapper.** A NEAT-evolved mapper composes the three grid outputs into the final conjunction.

This achieves 100% success at 3-grid scale in the prototype. Figure 7.3 illustrates the full pipeline.

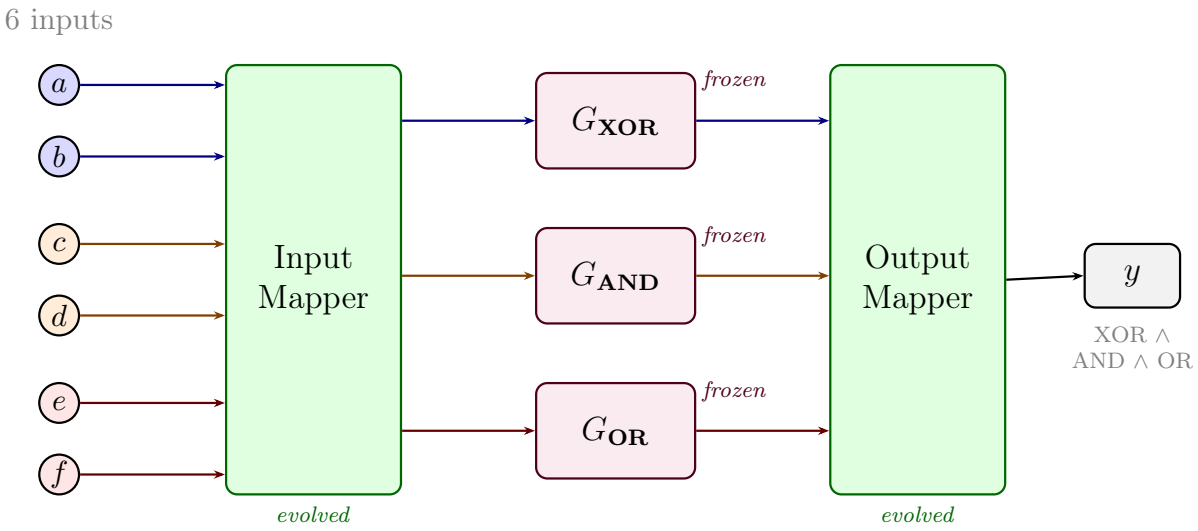


Figure 7.3: End-to-end compositional routing for $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f)$. Six Boolean inputs enter the evolved input mapper (green), which routes each pair to the appropriate frozen specialist grid (purple). Each grid processes its inputs independently. The evolved output mapper (green) composes the three grid outputs into the final compound result. Only the mappers are evolved per task; grids are trained once and reused.

The full GEENNS specification includes additional detail: a multi-level CPPN hierarchy for connectivity at different spatial scales (Table B.2), a five-stage developmental pipeline (Figure B.6), four plasticity mechanisms, a comparison with established continual learning methods (Table B.4), and a formal mathematical specification (Definitions B.1–B.2). These are presented in Appendix B as a design specification; none have been implemented or validated beyond the prototype. The remainder of this chapter focuses on the substrate barriers that must be solved before these mechanisms can be pursued, followed by the prototype evidence.

A comparison of six substrate evolution approaches (Table B.1, Appendix B) confirms that EMR-HyperNEAT is the only method providing both spatial organization and per-node feature evolution at practical cost. The analysis of CoDeepNEAT, the closest conceptual prior art, is presented alongside the formal specification.

7.5 Why the Substrate Problem Must Be Solved First

The GEENNS architecture described in the previous section requires substrates that are (1) computationally tractable to evolve, (2) expressive enough at the function level to support diverse computations, and (3) capable of multi-task operation. These three prerequisites are a decomposition local to this section: the computational prerequisite spans

the configuration and computational clusters of §1.2, while expressiveness and multi-task capability are additional. ES-HyperNEAT (the canonical adaptive substrate method) fails on all three counts. This section characterizes each failure mode and maps it to the thesis chapters that address it. "Expressiveness" here is a GEENNS-prerequisite label distinct from the central-bias-driven "Representational Collapse" of Chapter 4: they concern function diversity and spatial coverage respectively, and are independent barriers.

7.5.1 Barrier 1: Computational. ES-HyperNEAT Cannot Scale

ES-HyperNEAT discovers substrate topology through a quadtree that recursively subdivides the spatial domain, retaining regions where the CPPN output shows high variance (indicating information content) and pruning regions where the output is uniform (indicating redundancy). This adaptive discovery mechanism is the algorithm's defining feature: it determines not just connection *weights* but connection *placement*, enabling the substrate to match the problem's structural complexity.

The quadtree is also the algorithm's defining limitation. Each subdivision decision depends on the variance of the CPPN output evaluated at four probe points inside the current cell (the centers of its four child sub-quadrants) whose coordinates are determined by the previous subdivision. This sequential dependency chain prevents parallelization. While direct-encoding methods like NEAT can evaluate entire populations in parallel on a GPU (because each individual's fitness evaluation is independent), ES-HyperNEAT must construct each individual's substrate sequentially because the quadtree decisions are individual-specific.

The quantitative impact: our benchmarks show that ES-HyperNEAT's substrate construction time grows exponentially with resolution depth, while GPU-parallelized evaluation would remain approximately constant up to the hardware's memory limit. At depth 7 (21,845 candidate positions), the sequential construction dominates total runtime. A one-way ANOVA with depth as factor yields $F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$ (with $R^2 = 0.95$ from a separate regression fit), confirming that depth (and therefore sequential construction cost) is the primary determinant of runtime.

For GEENNS, this barrier is prohibitive. At a modest (k=5, T=10) scale the architecture requires 34 CPPNs (Appendix B), each needing thousands of optimization trials, and cannot use an algorithm that runs each trial on CPU. The computation time is measured in CPU-days to years, not hours: roughly 250 CPU-days at XOR scale, rising to years for realistic higher-dimensional tasks. This estimate extrapolates from XOR-scale benchmarks (2-input, depth 4–7) to the 34-CPPN pipeline with substantially higher dimensionality. The extrapolation is illustrative rather than precise. Actual GEENNS-scale costs would depend on per-CPPN complexity, parallelism across CPPNs, and hardware configuration, but the order-of-magnitude gap between years (ES-HyperNEAT) and days (EMR with pruner) is robust to reasonable variation in these factors.

Thesis response. Chapters 5 and 6 remove this barrier. Chapter 5 proves the incompatibility of ES-HyperNEAT's quadtree with GPU acceleration. Chapter 6 introduces EMR-HyperNEAT, which achieves up to 34× per-generation GPU speedup on XOR at depths 5–7 (population 1000; the 34× endpoint is the depth-7 streaming estimate) by replacing the sequential quadtree with a parallel evaluate-then-filter pipeline. Chapter 3 addresses the hyperparameter explosion via Tree-structured Parzen Estimator (TPE) optimization and an early-stopping pruner that reduces computational cost by 41.6%.

7.5.2 Barrier 2: Expressiveness. Substrates Lack Function Diversity

Standard indirect-encoding neuroevolution assigns a *single* activation function to all nodes in the substrate. This is the default behavior in HyperNEAT, ES-HyperNEAT, and their implementations in frameworks like PUREPLES. The choice of activation function is typically made by the experimenter before evolution begins, based on problem-domain intuition.

This uniform-function assumption is limiting. Companion work on per-node activation evolution (§8.4) characterizes this as an evolvability barrier rather than an impossibility, with every uniform monotonic activation tested (`tanh`, `sigmoid`, `ReLU`) yielding 0% solve rates at parity-4 (0.95 solve criterion), and uniform `tanh` remaining at 0% through parity-8; the mechanism proposed here is that indirect encoding's symmetric weight generation, combined with a monotonic activation function, makes the checkerboard decision boundary that parity requires very hard for evolution to reach. The barrier is one of search, not of representability: the 1-grid mono-tanh baseline of Table 7.7 still solves XOR in 11 of 30 runs, but needs a median of 95 generations where `sin` needs one.

The practical consequence is that the choice of a single activation function can make the difference between convergence in a single generation (with `sin`) and slow, unreliable convergence or none at all (with `tanh`). Which function is the liability is task-dependent: at 3-grid, `tanh`-only substrates fall to 3% and `sin`-only to 0%, with `tanh` partially recovering at 4-grid before collapsing again at 5-grid (Table 7.7). Per-node activation function evolution, demonstrated in a companion manuscript, lifts this constraint at negligible marginal cost via an additional CPPN output dimension.

For GEENNS, this barrier means that grid specialists cannot compute everything they need to compute. A grid that uses only `tanh` solves parity-like sub-problems only unreliably and slowly, if at all. A grid that uses only `sin` may struggle with threshold-based classification. Compositional intelligence requires grids with diverse computational capabilities, which in turn requires per-node function diversity.

Thesis enabler. EMR-HyperNEAT's batch CPPN evaluation makes per-node function assignment computationally feasible at negligible additional cost (Chapter 6). Per-node activation, per-node aggregation, and bio-inspired palette meta-learning (strategies for evolving which activation and aggregation functions are available to nodes across generations) are demonstrated in companion manuscripts that build on this infrastructure; the thesis itself neither implements nor validates these results but is the prerequisite that made them feasible.

7.5.3 Barrier 3: Multi-Task. Substrates Cannot Handle Multiple Tasks

A single substrate evolved for one task cannot solve a different task without re-evolution. This is not a weakness of neuroevolution specifically; it is the fundamental problem of catastrophic forgetting [33, 65] expressed in evolutionary terms. If we evolve a substrate for Task A and then evolve it further for Task B, the weights that made Task A work are overwritten by the weights that make Task B work.

Standard continual learning methods from the deep learning literature (Elastic Weight Consolidation (EWC) [51], Progressive Neural Networks [80], PackNet [62]) assume gradient-based training on fixed-topology networks. None of them directly applies to

evolved substrates, where the topology itself is the product of evolution and where there are no gradients to regularize.

For GEENNS, this barrier is structural. The entire multi-grid architecture assumes that grids can serve multiple tasks simultaneously: a feature extraction grid that processes visual input for both classification and detection, a transformation grid that performs rotations for both image alignment and spatial reasoning. If each task requires a completely separate substrate, then multi-grid composition offers no advantage over simply evolving separate substrates per task.

Thesis response. Chapter 4 demonstrates spatial decomposition through partitioned experts, achieving 43% MNIST accuracy (105% improvement over baseline) and providing preliminary evidence for the spatial decomposition principle underlying GEENNS. Two companion manuscripts address this barrier beyond the thesis: one demonstrates gain modulation for multi-task operation from a single topology; the other characterizes the population-alignment problem that arises when neuromodulatory signals are co-evolved alongside substrate topology.

7.5.4 The Three Barriers as Thesis Structure

Table 7.2 summarizes the mapping from barriers to thesis contributions. The structure reflects the logical dependency among the barriers rather than the order of the chapters.

Table 7.2: Mapping from GEENNS barriers to thesis contributions. Each barrier prevents a specific GEENNS capability; each contribution provides part of the solution. "Thesis" indicates a contribution presented in a thesis chapter; "Companion" indicates a direction validated in a companion manuscript that builds on the thesis infrastructure but is not part of this thesis; see Chapter 8 Table 8.5 for the full companion-work status summary.

Barrier	Problem	Contribution	Key Result
Computational	ES-HyperNEAT quadtree is sequential; GPU parallelism impossible	Thesis: Ch 5 (GPU incompatibility)	GPU incompatibility proof
		Thesis: Ch 6 (EMR-HyperNEAT)	12–34× per-gen GPU speedup (XOR, D5–7, pop 1000; 34× = streaming estimate)
		Thesis: Ch 3 (TPE + pruner)	41.6% cost reduction
Expressiveness	Uniform activation and aggregation limit substrate expressiveness	Companion: per-node activation	Per-node activation via CPPN output
		Companion: per-node aggregation	Per-node aggregation via CPPN output
		Companion: bio-inspired palettes	Meta-learning of function palettes
Multi-task	No mechanism for task-specific behavior in shared substrates	Thesis: Ch 4 (partitioned experts)	43% MNIST (105% ↑)
		Companion: neuromodulation	Gain modulation for multi-task
		Companion: co-evolving neuromodulatory signals	Population-alignment problem characterized

The three barriers are not independent. Solving the computational barrier enables the experiments that reveal the expressiveness barrier: per-node function evolution would

require thousands of evolutionary runs that would be impractical on ES-HyperNEAT. Solving the expressiveness barrier would enable multi-task operation: neuromodulation would require per-task activation function selection, which requires per-node function diversity. And the multi-task solutions validate the grid specialization concept central to GEENNS's compositional architecture.

The thesis thus follows a dependency chain: computation → expressiveness → multi-task → composition. The first link is load-bearing for everything after it; the later links are prerequisites for the full GEENNS design rather than for the prototype, which composes grids with uniform per-grid activations and no multi-task mechanism.

7.6 Prototype Evidence

We validated a proof-of-concept prototype across more than 40 experimental conditions and approximately 3,100 runs. The 5-grid Boolean scaling experiment alone accounts for ∼806 CPU-hours owing to its 1024-row truth table and 2000-generation budget; the 6-grid Boolean and continuous additive-scaling extensions add comparable wall-clock cost, while the remaining conditions together account for a small fraction of the total. The prototype tests the core GEENNS pipeline (specialist grid training, freezing, and mapper-based composition) on compound Boolean tasks, with a continuous-gate extension and a multi-task amortization study. We report results, confounds, and limitations. This section presents the evidence relevant to the thesis narrative.

7.6.1 Experimental Design

The prototype is enabled by the EMR-HyperNEAT tensor reformulation of Chapter 6: without uniform-shape substrate tensors, evolving one specialist grid per gate plus a mapper population on top would be computationally out of reach. Given that infrastructure, the pipeline proceeds in two stages. First, an EMR-HyperNEAT substrate is evolved for each individual Boolean task (XOR, AND, OR) and frozen on convergence. Second, a NEAT-evolved mapper population is trained to route raw task inputs through the frozen grids and combine the resulting grid outputs into a compound prediction (e.g., $\mathrm{XOR}(a, b) \wedge \mathrm{AND}(c, d) \wedge \mathrm{OR}(e, f)$). Every mapper individual receives the raw inputs and all grid outputs; no architectural constraint forces it to use the grids.

Substrate configuration. All conditions share the EMR-HyperNEAT substrate parameters in Table 7.3. These settings follow directly from the EMR-HyperNEAT system validated in Chapter 6: depth 4 provides sufficient resolution for Boolean tasks without incurring the exponential node growth that higher depths introduce, and the variance/division thresholds (0.03) match the values used throughout the cross-domain validation in Section 6.5.10. NEAT-specific parameters (species count, compatibility threshold, mutation rates, crossover rate, elitism, CPPN architecture) use TensorNEAT defaults throughout.

Substrate coordinate layout. Raw inputs are placed at $y = -1.0$, evenly spaced along the x-axis; bias is co-located at $(0, -1.0)$. In augmented (compositional) conditions, frozen grid outputs are placed at $y = -0.5$, an intermediate coordinate that gives the CPPN spatial information to differentiate raw input signals from pre-solved grid outputs. The

Table 7.3: Shared EMR-HyperNEAT substrate parameters for all prototype conditions.

Parameter	Value
Initial depth	0
Max depth	4
Variance threshold	0.03
Division threshold	0.03
Band threshold	0.3
Max weight	3.0
Success threshold	$f \geq 0.95$
Output activation	sigmoid
Output node	single at (0.0, 1.0)

single output node sits at $(0, 1.0)$. Figure 7.4 contrasts the monolithic and compositional layouts for a 2-grid task.

Frozen grid forward pass. Once a specialist grid is trained and frozen, its forward pass reduces to a fixed two-layer computation: output $= \sigma(\phi(\mathbf{x}\mathbf{W}_1)\mathbf{W}_2)$, where $\mathbf{W}_1$ and $\mathbf{W}_2$ are the weight matrices generated by the CPPN during evolution, ϕ is the grid's hidden activation (sin for XOR, tanh for all others), and σ is sigmoid. At inference, grid evaluation is a pair of matrix multiplications with no evolutionary overhead. Fitness is $f = 1 - \text{MSE}$ over all input patterns; a run succeeds when $f \geq 0.95$. Success thresholds tighten at larger grid counts to maintain margin above the class-imbalance floor: $f \geq 0.99$ at 5-grid (where the all-zeros predictor already scores $f = 0.982$) and $f \geq 0.995$ at 6-grid (floor $f = 0.9912$); the per-scale justifications appear in §7.6.3 and §7.6.2.

Pipeline architectures and activation modes. Two pipeline architectures and three hidden-activation modes are compared.

Compositional (augmented) pipelines. A mapper substrate combines raw inputs with outputs from frozen specialist grids (each grid has its own pre-trained activation: sin for XOR, tanh for others). The mapper's hidden layer is fixed at tanh via the *disabled* mode (per-node activation-selection turned off); per-task activation diversity is handled externally by the frozen grids.

Monolithic baselines. A single substrate is used, with no frozen grids. The hidden layer is configured in one of two ways:

- *uniform* (`global`): a single activation (sin or tanh) is assigned uniformly to every hidden node: the mono-sin and mono-tanh baselines.
- *dynamic* (`cppn_output`): an extra CPPN output selects from a 6-function palette per node: the `-dyn` baseline. The dynamic function mechanism is the subject of work beyond this thesis; it is included here solely as a diagnostic baseline to isolate whether per-node function selection can substitute for compositional architecture (PQ7). Dynamic function evolution is not a contribution of this thesis.

The *disabled* mode (compositional mapper, tanh everywhere) and the *global*+tanh mode (monolithic baseline, tanh everywhere) are observationally identical at the hidden-layer level. They are kept as separate labels because they apply to architecturally different pipelines: the disabled-mode mapper is one component of a compositional pipeline that

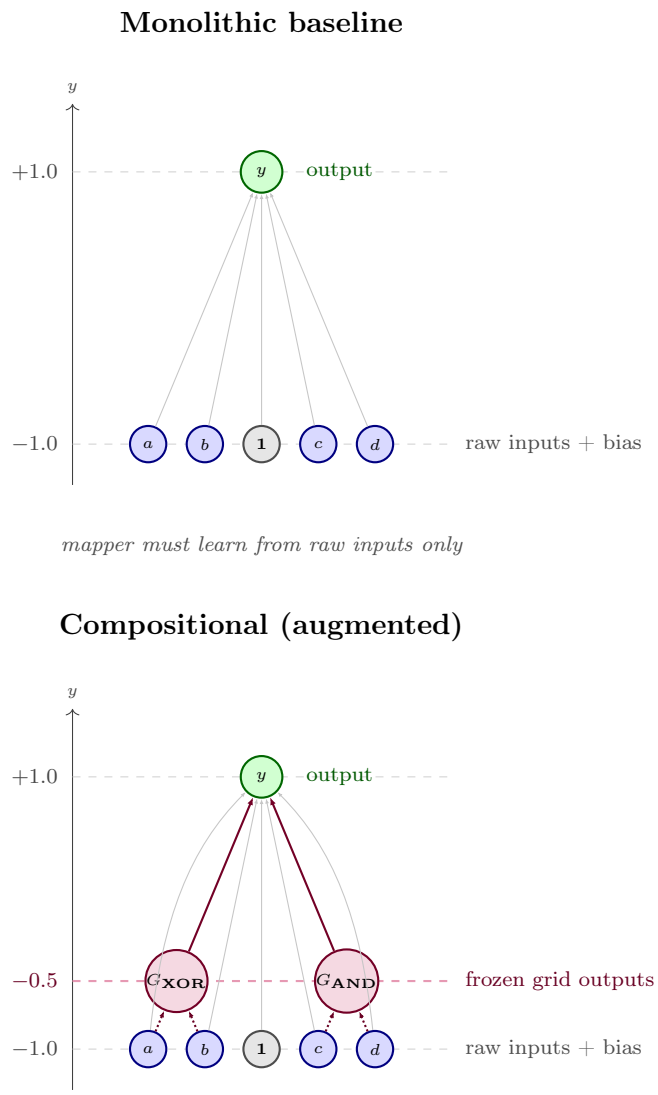


Figure 7.4: Substrate coordinate layout for monolithic (top) and compositional (bottom) conditions, illustrated for a 2-grid task (four raw inputs a, b, c, d plus bias). Raw inputs sit at $y = -1.0$ evenly spaced along the x-axis, with bias co-located at $(0, -1.0)$. The single output node sits at $(0, +1.0)$. In compositional conditions, frozen grid outputs (G_{XOR}, G_{AND}) occupy an intermediate $y = -0.5$ plane; this coordinate separation gives the CPPN a spatial cue for treating raw inputs and pre-solved grid signals differently, rather than requiring the mapper to disentangle them from a single shared layer. Three arrow styles appear in the bottom panel: thin gray arrows denote the mapper's evolved raw-input and bias connections to the output (curved for a and d to route around the grid-output nodes); thick purple arrows denote the mapper's evolved grid-output connections; densely dotted arrows denote the frozen grids' fixed input dependencies (a, b feed G_{XOR}; c, d feed G_{AND})—pre-computations external to the mapper substrate, not evolved.

also contains frozen specialist grids supplying per-task signals; the global+tanh monolithic baseline has no frozen grids.

Population and generation budgets. Table 7.4 summarizes the evolutionary budgets. Specialist grids use smaller populations (150) and shorter runs (200 generations) because single Boolean gates are simple tasks that converge reliably. Mapper and monolithic conditions use population 300 and 500 generations. The 4-grid conditions double the generation budget to 1,000 to accommodate the 256-row truth table; the 5- and 6-grid conditions raise it again to 2,000 for their 1024- and 4096-row truth tables.

Table 7.4: Population and generation budgets by experimental scale.

Scale	Pop	Max Gen	Notes
Grid specialists	150	200	XOR uses sin; AND, OR, NAND, NOR use tanh
1-grid conditions	300	500	
2-grid conditions	300	500	Composition-only oracle uses pop 150
3-grid conditions	300	500	
4-grid conditions	300	1000	Doubled for 256-row truth table
5-grid conditions	300	2000	Quadrupled for 1024-row truth table
6-grid conditions	300	2000	4096-row truth table; threshold $f \geq 0.995$

Problem scaling. As grids are added, the mapper's input dimensionality and the truth table both grow (Table 7.5). Compositional conditions receive additional grid-output dimensions but benefit from the $y = -0.5$ coordinate separation that allows the CPPN to assign distinct connectivity patterns to raw inputs versus grid signals. Monolithic baselines receive only the raw inputs and must discover the correct spatial partitioning of activation functions internally.

Table 7.5: Input dimensionality and truth-table size by grid scale. "Comp" = compositional (raw + grid outputs + bias); "Mono" = monolithic (raw + bias).

Scale	Task	Raw	Truth table	Comp inputs	Mono inputs
1-grid	XOR	2 + bias	4 rows	4	3
2-grid	XOR∧AND	4 + bias	16 rows	7	5
3-grid	XOR∧AND∧OR	6 + bias	64 rows	10	7
4-grid	XOR∧AND∧OR∧NAND	8 + bias	256 rows	13	9
5-grid	XOR∧AND∧OR∧NAND∧NOR	10 + bias	1024 rows	16	11
6-grid	XOR∧AND∧OR∧NAND∧NOR∧XNOR	12 + bias	4096 rows	19	13

The experiments address seven prototype questions, labeled PQ1–PQ7 to mark them as scope-local rather than thesis-wide research questions (TRQ1–TRQ4 in Chapter 1):

PQ1 (Feasibility) Do EMR-HyperNEAT grids converge reliably enough to serve as frozen specialist modules?

PQ2 (Composition) Can a mapper learn to compose pre-solved grid outputs into a compound solution?

PQ3 (Routing & efficiency) Does the compositional pipeline improve success rate and convergence speed over monolithic evolution, controlling for activation function?

PQ4 (Grid utilization) Does the mapper genuinely use grid outputs, or does it bypass them to solve monolithically?

PQ5 (Generalization) Does the evolved mapper generalize to unseen compositions or grid permutations?

PQ6 (Reuse) Can the same frozen grids support mapper evolution for different compound tasks, and can architectural bias improve selective routing?

PQ7 (Scaling) Does the compositional advantage persist, and do monolithic baselines degrade, as task complexity rises across grid counts, and at what scale (if any) does composition become structurally necessary rather than optional? *Hypothesis*: a complexity threshold exists beyond which a monolithic substrate with per-node dynamic function selection can no longer reliably discover the correct spatial partitioning of activation functions, and compositional architecture becomes structurally necessary. The prototype tests this across 1, 2, 3, 4, and 5 Boolean grids; 6-grid Boolean and continuous-gate extensions appear in §7.6.2 and §7.6.5.

Table 7.6 summarizes the key experiments, organized by the prototype question each addresses. Several of its labels refer to specific baseline configurations: a *composition-only oracle* feeds the mapper only frozen-grid outputs (no raw inputs); *noise control* experiments add task-irrelevant distractor input dimensions—extra inputs that probe whether the mapper genuinely uses grid outputs—either *constant* (the distractor is held fixed) or *resampled* (the distractor is drawn anew per evaluation); *gated* architectures impose a structural bias toward grid outputs.

7.6.2 What Works

Figure 7.5 summarizes success rates across the conditions plotted through 4-grid scale.

Table 7.6: GEENNS prototype experiments organized by prototype question (PQ). The "Code" column gives the identifier family used in the figure legends of Figures 7.5 and 7.6; the figures subdivide several of these codes by condition (E3a/E3b, E4a–E4c, D-aug/D-dyn). The experiments listed here account for about half of the prototype's ∼3,100 runs; the remaining runs (5-grid and 6-grid Boolean scaling, continuous-gate extension through 8 gates, gated selective routing across 3–6 grids, the operator landscape control, and the warm-start transfer study) are summarized in the body of §7.6.

Code	Experiment	PQ	N	Key Result
PQ1–PQ3: Feasibility, composition, and routing				
E1	Specialist grid feasibility	1	150	149/150 converged across the 5 gates (one OR seed at 0.865)
E2	Composition-only oracle	2	60	100%, median 44 gen
E3	Two-grid routing and composition	3	60	100% vs. 80% (raw-only ablation)
E4	Monolithic baselines (5 cond.)	3	150	dyn 100% at 3 gen; tanh 80%
PQ4: Grid utilization				
E6	Frozen grid reuse	4, 6	120	57–93%; noise control 70% invalidated, see J1–J2 (§7.6.4)
J1	Constant noise control	4	30	47%; non-informative dims hurt search
J2	Resampled noise regularization	4	30	100%; implicit regularization
PQ5–PQ6: Generalization and reuse				
E5	Compositional generalization	5	30	Mean fitness 0.53–0.80 (no seed above 0.95)
F	Grid-only composition	6	60	93–100% without raw inputs
H	Gated selective routing	6	90	100% (90/90); 8.3× selectivity
PQ7: Scaling (1–4 grids enumerated here; 5- and 6-grid Boolean plus continuous gates in §7.6 body)				
D	Three-grid scaling	7	119	100% compositional; ≤3.3% mono; 100% dynamic
G-aug	Four-grid composition	7	29	100%, median 40 gen, 95% CI [34, 70]
G-dyn	Four-grid dynamic baseline	7	30	87% (26/30), median 321 gen
G-tanh	Four-grid mono-tanh baseline	7	30	27% (8/30), median 437 gen
G-sin	Four-grid mono-sin baseline	7	30	0% (0/30)
1grid-comp	Single-grid composition	7	30	100%, median 1 gen
1grid-dyn	Single-grid dynamic	7	30	100%, median 1 gen
1grid-tanh	Single-grid mono-tanh	7	30	37% (11/30), median 95 gen
1grid-sin	Single-grid mono-sin	7	30	100%, median 1 gen
I	Multi-task amortization	7	595	94.9% compositional vs. 93.3% dynamic; pipeline 1.34× slower at 10 tasks; break-even ∼80
	Total (this table)		**>1,700**	

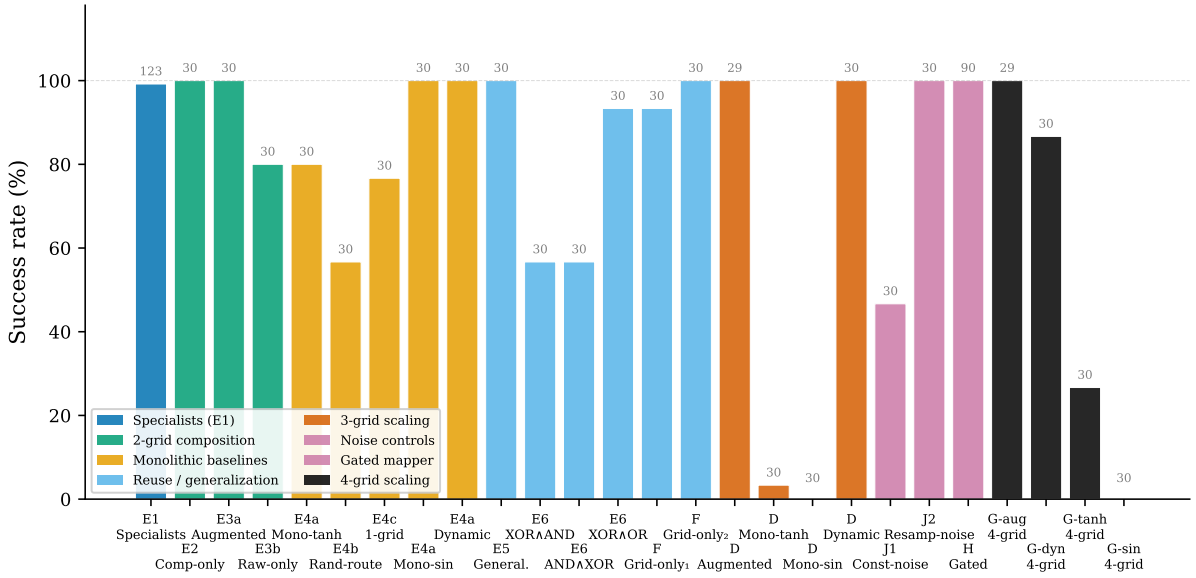


Figure 7.5: Success rates across the prototype's conditions through 4-grid scale ($N = 30$ seeds unless noted; 5-grid Boolean in Table 7.7; 6-grid Boolean in this subsection's text, continuous gates in §7.6.5). Compositional pipelines achieve 100% at every scale shown. Dynamic function baselines match at 2- and 3-grid but degrade to 87% at 4-grid. Uniform-activation monolithic baselines collapse at 3-grid and again at 5-grid, with mono-`tanh` partially recovering at 4-grid. Gated selective routing lifts 2-grid frozen grid reuse rates from 57–93% to 100%. The E5 generalization bar reports convergence on the composition each mapper was evolved on, not on the held-out composition; the generalization result itself (per-target mean fitness 0.53–0.80, no seed above threshold) is in Table 7.6 and §7.6.7. Multi-task amortization (Exp I; see Table 7.6 and the surrounding text) is not plotted here: its finding concerns amortization economics rather than per-scale success rates.

Specialist grids converge and compose. Specialist grid feasibility confirms reliable convergence (XOR, AND, OR, NAND, and NOR). At 3-grid scale, three-grid composition solving $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f)$ achieves 100% success (see Table 7.7). Frozen grids support mapper re-evolution: the same three grids, without retraining, support three different compound tasks in the frozen grid reuse experiment with 57–93% success rates.

The single-grid composition control (frozen XOR grid + raw inputs) converges in median 1 generation (30/30), confirming that frozen grid outputs provide immediately useful features that the mapper exploits without requiring substantial evolutionary search. At 4-grid scale, four-grid composition extends the pipeline to $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f) \wedge \text{NAND}(g, h)$, an 8-input, 256-row truth table, and maintains 100% success (see Table 7.7). Convergence is actually *faster* at 4-grid than at 3-grid: the extra frozen grid appears to tighten the routing signal rather than add search burden.

Routing cost and amortization. Two-grid routing provides a 9× generation-count speedup over the 2-grid mono-sin baseline (monolithic, uniform sin activation; median 9 generations vs. 81 generations). The pipeline cost overhead is modest (1.04× per task), breaking even at approximately 2 tasks because specialist grids are trained once and

reused at the 2-grid scale.

At larger multi-task scales the amortization story is more complicated. The multi-task amortization experiment sweeps 10 pairwise AND compositions across 595 runs and reports augmented success at 94.9% versus dynamic-function monolithic at 93.3%: reuse works, but at 10 tasks the full pipeline cost (specialist training plus mapper evolution across all 10 targets) is 1.34× the total monolithic cost, with a projected break-even near 80 reused tasks. At 10-task scale the pipeline is therefore negative on raw efficiency. The amortization argument regains traction at 5-grid scale, where the dynamic baseline falls to 0% and composition retains 100%: the pipeline's value at 5-grid is capability, not efficiency, and it solves tasks the monolithic alternative cannot. This reframes the amortization claim: at small grid counts the monolithic baseline remains competitive and composition's overhead dominates; beyond the complexity threshold the overhead becomes the cost of capability rather than a tax on reuse.

Mappers genuinely use grid outputs. Weight analysis confirms grid usage: grid-output connections are 1.25× stronger than raw-input connections ($p < 0.001$). A post hoc ablation sharpens this picture: zeroing the two grid-output columns of 30 converged mappers drops fitness by 0.088 ± 0.012 (from ∼0.965 to ∼0.877), whereas zeroing the four raw-input columns drops it by 0.364 ± 0.175 (to ∼0.601). Ablation therefore attributes roughly 4× more of the fitness to raw inputs than to grid outputs, even though the mapper's weights favor grid outputs. The two measurements are not at odds once the quantities are separated: weight magnitude tracks which inputs the mapper prefers to connect to, while ablation tracks which inputs the computation cannot function without. The grids do not replace the raw inputs; they supply pre-solved features that narrow the search region enough to explain the 9× convergence speedup over the 2-grid mono-sin baseline, while the raw inputs continue to carry the full 2^4-pattern signal over which the mapper actually computes.

Gating elicits selective routing and extends composition's reach. Gated selective routing addresses the indiscriminate routing observed in the frozen grid reuse experiment by introducing evolved per-column sigmoid gates on each grid-output channel (Figure D.1). A gate is one evolved scalar per grid-output channel: before evaluation, each grid-output column of the mapper's input matrix is multiplied by the logistic transform $\sigma(g_i) \in (0, 1)$ of its gate while raw inputs pass ungated, so a channel can be attenuated toward silence or left open, never amplified or sign-inverted, and converged gate values read directly as routing decisions. Gates initialize half-open at $\sigma(0) = 0.5$ and co-evolve with the genomes under shared selection. The full channel layout, gate-evolution scheme, and correct/distractor labeling protocol are in Appendix D.3.1.

With this architectural bias, all three 2-grid reuse tasks reach 100% versus the unconstrained 57–93% observed without gating. Gate-ratio analysis reveals three qualitatively distinct routing strategies rather than a single selective pattern: one task produces a strongly selective gate signature (correct-to-distractor 8.28×, the strongest direct evidence in the prototype of evolved selective routing under explicit architectural support), one converges with *inverted* gates (0.63×, distractors recruited as complementary features rather than suppressed, median 19 gen), and one is neutral (1.05×, median 4 gen, both grid outputs carrying the same signal). Architectural bias therefore elicits selective routing *when the task configuration admits it*, but the mapper can also solve compound tasks without favoring correct over distractor channels when the weight search finds an

alternative route.

Gating also scales cleanly with grid count. At these scales each grid is wired only to its own input pair, so there is one gate per specialist and no crossed channels; the gates modulate how much each specialist contributes rather than which pairing is used. The same gated-mapper architecture tested on 3-, 4-, and 5-grid conjunctions achieves 100% success at 3-grid with a median convergence of 34 generations, 2.3× faster than the ungated three-grid composition baseline; 100% at 4-grid in 37 gen, matching the four-grid composition baseline while the four-grid dynamic-function baseline fails at 87%; and 96.6% at 5-grid in 489 gen, with the single failure ($f = 0.989$) exhibiting correct gate patterns, placing it below the strict $f \geq 0.99$ threshold used for 5-grid analyses. Gating therefore delivers reliable selective-routing behavior up through the decisive 5-grid threshold; what it does not yet deliver is task-agnostic routing. Gate weights remain task-specific (re-evolved per compound task), so the architectural bias addresses the indiscriminate-routing problem without resolving the generalization problem.

Pushing the conjunction to six grids ($\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f) \wedge \text{NAND}(g, h) \wedge \text{NOR}(i, j) \wedge \text{XNOR}(k, l)$, 12 inputs, 4096-row truth table) reaches the first scale at which the augmented pipeline itself becomes unreliable. The all-zeros predictor scores $f = 0.9912$ on this truth table, so the success threshold tightens to $f \geq 0.995$ to keep a margin against majority-class output; under that threshold the augmented mapper solves only 1 of 29 seeds in 2000 generations, with the other 28 plateauing at $f \in [0.992, 0.994]$. Spot-checked seeds at the plateau predict negative on every positive pattern (true-positive rate 0), the same failure mode that 5-grid monolithic baselines exhibit. Gating substantially mitigates this collapse: with the same six frozen specialists and the same threshold, the gated mapper solves 21 of 29 seeds (72%) at median 362 generations under identical conditions, with multiple viable gate signatures across seeds (some suppress NOR and XNOR, others suppress OR and NAND). At 5-grid, composition succeeds where every monolithic baseline collapses; at 6-grid, gating rescues composition. The complexity threshold therefore has two stages within the Boolean regime, and the prototype's reach extends one grid further than the augmented pipeline alone would suggest.

These results provide the first empirical evidence that emergent compositional behavior is possible within the GEENNS framework. The mappers were not told which inputs to route to which grids; they discovered the correct routing through evolutionary pressure. The composition was not programmed; it emerged from evolved connection weights. This is emergence in a precise, limited sense: the compositional strategy was discovered by the evolutionary process, not designed by the experimenter. What this emergence is not is general-purpose: it remains task-specific. Individual mappers learn weight patterns tied to their channel assignment rather than fully general compositional strategies, yet the transfer analysis developed later in this section shows those patterns are not narrowly task-specific either: when the channels are re-assigned to match a recomposed target, the converged mappers themselves solve it zero-shot, and transfer fails only when the channel semantics change.

7.6.3 Scaling Behavior: Composition vs. Baselines

Table 7.7 and Figure 7.6 consolidate the success rates and convergence trajectories across the five grid scales tested in the prototype, comparing the compositional pipeline against three baseline conditions: dynamic per-node function selection (a monolithic network whose CPPN assigns activation functions per node, from work beyond this thesis; included as diagnostic baselines, not thesis contributions), mono-tanh, and mono-sin. The 6-grid Boolean probe (§7.6.2) uses augmented-versus-gated composition only, because the monolithic baselines had already collapsed at 5-grid.

Table 7.7: Success rates across 1–5 grid scales. Composition maintains 100% at every scale; convergence speed improves from 78 gen (3-grid) to 40 gen (4-grid), then rises to 281 gen at 5-grid as the conjunctive search depth grows. Dynamic functions match at 2- and 3-grid, degrade to 87% at 4-grid with an 8× convergence speed inversion, and collapse to 0% at 5-grid. Uniform-activation baselines collapse at 3-grid and again at 5-grid, with mono-`tanh` partially recovering at 4-grid. Each cell shows success rate (with seeds converged in parentheses) above and median convergence generation below; "—" indicates no successful seeds.

Scale	Composition	Dynamic	Mono-tanh	Mono-sin
1-grid	100% (30/30) 1 gen	100% (30/30) 1 gen	37% (11/30) 95 gen	100% (30/30) 1 gen
2-grid	100% (30/30) 9 gen	100% (30/30) 3 gen	80% (24/30) 178 gen	100% (30/30) 81 gen
3-grid	100%[§] (29/29) 78 gen	100% (30/30) 22 gen	3% (1/30) 359 gen	0% (0/30) —
4-grid	100%[§] (29/29) 40 gen	87% (26/30) 321 gen	27% (8/30) 437 gen	0% (0/30) —
5-grid	100%[§] (29/29) 281 gen	0% (0/30) —	0% (0/30) —	0% (0/30) —

[§]One seed excluded: OR specialist grid did not converge (fitness 0.865). 5-grid uses $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f) \wedge \text{NAND}(g, h) \wedge \text{NOR}(i, j)$ (10-input, 1024-row truth table). 1–4 grid rows use success threshold $f \geq 0.95$; the 5-grid row uses $f \geq 0.99$ because the 18/1024 positive-pattern ratio makes an all-zeros predictor reach $f = 0.982$ from class imbalance alone.

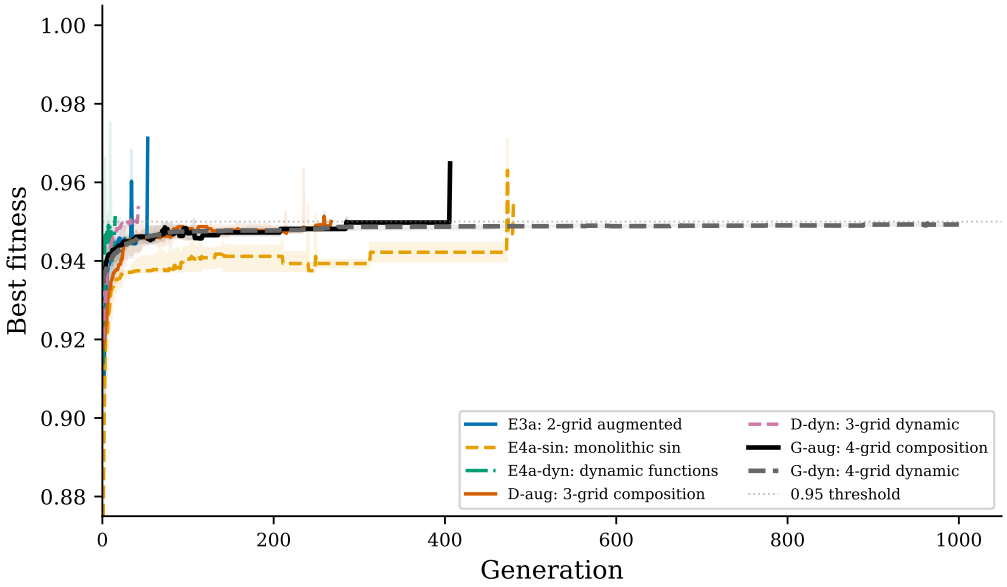


Figure 7.6: Convergence curves (median ± IQR) for key conditions across 2-, 3-, and 4-grid scales. Two-grid routing converges in 9 generations vs. the 2-grid mono-sin baseline in 81, a 9× speedup from frozen grid outputs. At 3-grid, the dynamic baseline converges faster (median 22 gen) than composition (78 gen). At 4-grid, the speed relationship inverts: four-grid composition converges in median 40 gen vs. the four-grid dynamic baseline in 321 gen, an 8× speed advantage that defines the complexity threshold. The 0.95 fitness threshold is shown as a dashed line.

Composition is *reliably* scale-invariant: 100% success at every scale tested, from 1-grid through 5-grid. Median convergence generations trace a non-monotonic curve (78 at 3-grid, 40 at 4-grid, 281 at 5-grid): the 4-grid speedup reflects stronger mapper signals from an extra grid output, while the 5-grid regression reflects the 4× truth-table expansion (256 → 1024 rows) and the strict $f \geq 0.99$ threshold required to exclude majority-class prediction. The degradation hierarchy at 4-grid scale is *monotonic*: composition (100%) > dynamic (87%) > mono-tanh (27%) > mono-sin (0%); at 5-grid the hierarchy compresses to composition (100%) > everything else (0%). And the *speed inversion is extreme*: at 3-grid, the dynamic-function baseline converges 3.5× faster than composition; at 4-grid, composition converges 8.0× faster than the dynamic-function baseline, a ∼28× swing in relative speed advantage, which at 5-grid becomes a reliability gap rather than a speed gap because the dynamic baseline no longer converges at all.

The mono-tanh recovery at 4-grid (3% at 3-grid → 27% at 4-grid) warrants explanation. NAND is linearly separable, giving the 4-gate truth table more exploitable structure than the 3-gate conjunction. The 8/30 solves are hard-won and statistically distinguishable from chance (Fisher's exact $p = 0.002$ against 0/30). This does not invalidate the degradation hierarchy; it shows that task structure interacts with activation function capability at every scale.

The scaling table provides the clearest evidence for a *confirmed* complexity threshold. At 2-grid scale, dynamic functions are strictly superior (100% at 3 gen vs. 100% at 9 gen). At 3-grid, dynamic functions still match on success rate but composition closes the speed gap. At 4-grid, composition pulls ahead on both reliability and speed: 100% at 40 gen vs. 87% at 321 gen. The success rate difference alone is not significant at conventional

levels (Fisher's exact $p = 0.112$), but the convergence speed inversion is highly significant (Mann-Whitney $p = 3.2 \times 10^{-8}$, $r = 0.87$) and the four dynamic-baseline failures plateau at $f = 0.949$–0.950, indicating capacity limits rather than slow convergence. Increasing compound task complexity from 3 to 4 gates produces simultaneous reliability degradation and speed inversion. This establishes a complexity threshold beyond which modular architecture becomes both necessary and sufficient.

The 5-grid experiment sharpens this threshold to the point of decisiveness. Extending the conjunction to $\text{XOR}(a, b) \wedge \text{AND}(c, d) \wedge \text{OR}(e, f) \wedge \text{NAND}(g, h) \wedge \text{NOR}(i, j)$ (10 inputs, 1024 truth-table rows), composition retains 100% success (29/29, median 281 gen, mean fitness 0.992), while every monolithic baseline collapses to 0% (0/30 for dynamic functions, mono-tanh, and mono-sin alike). Only 18 of the 1,024 truth-table rows evaluate to positive (roughly 1.76%), so a predictor that outputs zero on every input already scores $f = 0.982$; we therefore tighten the success threshold to $f \geq 0.99$ for 5-grid analyses to force genuine learning above the class-imbalance floor. The four-grid gap of 87% versus 100% (marginal at $p = 0.112$) widens to 0% versus 100% at 5-grid for every composition-vs-baseline contrast. The dynamic baseline reaches a mean fitness of 0.987, which superficially looks like near-convergence, but recall the 0.982 floor that the constant-zero predictor already scores. To check whether the baseline learned anything beyond that trivial predictor, we re-evolved five of its seeds and inspected their per-pattern outputs: on the 18 positive patterns the output averages 0.27, below the 0.5 decision boundary, and the seeds correctly flag only about 1.4 of the 18 positives on average, a true-positive rate near 7.8%. The 0.987 fitness is an artifact of class imbalance: the baseline reaches it by predicting negative on nearly every input, not by learning the compound function. At 5-grid the threshold is no longer marginal. Monolithic substrates do not merely converge more slowly; they do not converge at all. Whether this threshold generalizes beyond Boolean conjunctions, to domains where task structure differs from gate composition, remains the central open question.

A theoretical argument predicts this threshold. The monolithic dynamic-function substrate is a single EMR-HyperNEAT network whose single CPPN must encode topology, connection weights, and the correct spatial partitioning of activation functions at once, all of it found by evolution acting on that one genome, assigning `sin` to the region processing XOR and `tanh` to regions processing the linearly separable gates. As the compound task scales from 3 gates (6 inputs, 64-row truth table) to 4 gates (8 inputs, 256-row truth table), the search space for correct function-to-region mapping grows combinatorially while the actual computational difficulty of the sub-tasks does not change: NAND is linearly separable, like AND and OR. By contrast, the compositional pipeline's difficulty scales additively: one more frozen grid, one more mapper input. The spatial-partitioning problem the CPPN must encode is the bottleneck, not the task difficulty per se.

7.6.4 The Confounds

Three observations qualify the results. First, the dynamic function confound. At 3-grid scale, the dynamic function baseline, a monolithic EMR-HyperNEAT substrate with per-node activation function selection via CPPN output dimensions, achieves 100% at median 22 generations, *faster* than composition's 78 generations. The dynamic function baseline matches composition on solvability and beats it on speed at 3-grid scale. The central question is whether this advantage persists at higher complexity.

The crossover documented in §7.6.3 (3.5× dynamic advantage at 3-grid inverting to an

8.0× compositional advantage at 4-grid) is what resolves the confound. At $N = 30$, the same four failures that plateau there had fitted 253 of 256 truth-table rows, a capacity limit rather than a stalled search, which confirms that the inversion reflects substrate limits rather than slow search.

This constitutes a *complexity threshold*: the scale beyond which a monolithic CPPN can no longer reliably discover correct function-to-region partitioning, and compositional architecture becomes structurally necessary. The evidence is convergence speed, not success rate alone: the 8.0× speed advantage ($p < 10^{-7}$) and the capacity-limited plateaus in the four dynamic-baseline failures indicate a qualitative transition from "dynamic functions suffice" at 3-grid to "composition is both faster and more reliable" at 4-grid.

Second, a confound in the original noise control has a principled resolution. The initial attempt (70% with fixed-random distractor columns) was invalidated by determinism: because the injected values were constant per input pattern, a mapper could learn them as a 16-entry lookup table rather than ignore them. Two replacement conditions separate the two failure modes the original test conflated. A constant distractor column fixes success at 47%: evidence that extra uninformative input dimensions expand the effective search space and actively slow convergence, even when they carry no task-relevant signal. A resampled-noise condition recovers 100%: because the distractor values change per evaluation, no deterministic lookup exists, and the mapper is steered toward the stable grid outputs. The effect is analogous to input dropout as a regularizer; the interpretation is that resampled noise acts as an implicit pressure toward grid utilization, not as a generalization test in itself.

Third, connection-weight analysis in the frozen grid reuse experiment reveals that the unconstrained mapper distributes nearly uniform attention across grid-output columns, with correct-to-distractor weight ratios of 0.91–0.94×. This is the routing-indiscrimination phenomenon that gating addresses in §7.6.2: gating produces task-specific routing strategies (selective, inverted, or neutral depending on task configuration) rather than uniformly suppressing distractor channels.

7.6.5 Beyond Boolean: Continuous Gate Composition

The Boolean conjunction tests establish that composition scales where monolithic baselines collapse, but the task class is narrow: each gate is a discrete function on $\{0, 1\}^2$, and the compound output is a single binary value. To probe whether the scaling story reflects an architectural property or an artifact of Boolean structure, the prototype includes continuous counterparts: $\mathrm{XOR}_c(a, b) = a + b - 2ab$, $\mathrm{AND}_c(c, d) = cd$, and $\mathrm{OR}_c(e, f) = e + f - ef$, the multilinear surfaces that reproduce each gate's truth table exactly at the corners of $[0, 1]^2$ and interpolate smoothly between them. Each specialist is evolved as a regressor of its own surface and frozen; the composition *operator* generates the compound target from the ideal gate surfaces before evolution begins, and the mapper must approximate that target from the frozen specialists' outputs alone. Success is measured by $R^2 \geq 0.85$, which scores the predict-the-mean strategy at zero where $1 - \mathrm{MSE}$ would reward it. The regressor protocol, the operator-target construction, and the gate-surface and pipeline figures are in Appendix D.3.2 (Figures D.3 and D.2).

Under a multiplicative composition operator (the compound target is the product of the gate surfaces, the continuous analog of logical conjunction), the complexity threshold shifts forward relative to the Boolean case. At 2-gate scale, composition alone reaches 100% (29/29; mean $R^2 = 0.860$, median 12 gen; the augmented variant, raw inputs plus

grid outputs, reaches only 11/29 here) while every monolithic baseline scores 0% (mean R^2 between 0.52 and 0.74, all below the 0.85 threshold). The contrast with the Boolean regime is sharp: at 2-grid Boolean scale the dynamic-function baseline matched composition (100%, median 3 gen), so composition there was optional rather than necessary. In the continuous regime every monolithic baseline scores 0% at the *smallest* compound scale, so composition is already required, not optional. At 3-gate, composition drops to 55% (16/29 solved; mean $R^2 = 0.846$ across all seeds), and at 4-gate to 0% (0/29 reach the strict $R^2 \geq 0.85$ threshold; mean $R^2 = 0.824$). The 4-gate result is a near-miss rather than a collapse: composition seeds cluster between $R^2 = 0.80$ and 0.84, all monolithic baselines sit near zero, and loosening the threshold to $R^2 \geq 0.80$ lifts composition to 93% (27/29) while baselines remain 0%.

The mechanism behind successful composition is not literal multiplication: replayed converged mappers compute a smooth degree-3 polynomial of the gate outputs (mean $R^2 = 0.965$ at both 3- and 4-gate), which caps how much per-gate sigmoid compression can compound and explains why every composition seed exceeds the multiplicative ceiling $\prod R_i^2$; the replay analysis is in Appendix D.3.2.

Swapping the multiplicative operator for an *additive mean* (the compound target is the mean of the gate surfaces rather than their product) erases the barrier entirely. Success returns to 100% at 3-gate (29/29, median 21 gen, versus the multiplicative 55%), holds at 4-gate (29/29, median 24 gen, versus the multiplicative 0%), and holds again at 5-gate (29/29, median 9 gen). Median convergence is not monotone in gate count under the additive operator; from 5-gate onward it grows approximately linearly. The architecture, mappers, frozen grids, and training budget are otherwise unchanged; only the composition operator differs.

Additive monolithic baselines remain at 0%, so the operator change shifts the compositional pipeline from 0% to 100% without making the underlying compound function any more tractable for a monolithic substrate.

The reading is that the continuous scaling limit sits in the operator, not the substrate: products compound per-gate sigmoid-compression errors across grids (each specialist's output is slightly shrunk toward mid-range, and a product of k slightly-shrunk factors is markedly shrunk), while averaging lets those errors cancel instead of accumulate. Pushing the population from 300 to 1,000, or the generation budget from 500 to 2,000, never pushes the multiplicative 3-gate rate beyond ≈55–62%, which rules out search-budget limits and points squarely at the operator. At the prototype's current scale the composition operator, not the architectural capacity of the grids or mappers, is the bottleneck beyond the Boolean domain. This reads as a retrospective clarification of the Boolean regime as well: Boolean conjunction fixes the operator at logical AND, which folds operator effects into the architecture and makes them invisible to ablation, so the Boolean experiments could not have surfaced operator dependence even in principle. Only by stepping outside Boolean structure does the operator become a free architectural choice, and only then does its role separate from the substrate's.

Three further operators (max, min, and a soft-max-weighted sum) fail uniformly at 3- and 4-gate (0 of 29 seeds each, 174 runs; Appendix D.3.2): among the five operators tested, only the additive mean scales beyond 3-gate.

Pushed further, additive composition shows no observed failure point. Re-evolving three additional 2-input continuous specialists (continuous XNOR, IMPLY, NIMPLY, all with sin activation) and assembling 6-, 7-, and 8-gate compounds, the pipeline solves 29/29 seeds at every gate count, with median convergence growing approximately linearly.

Mean R^2 rises from 0.888 at 5-gate to 0.905 at 8-gate. Through eight continuous gates the architecture, frozen grids, and mapper population reliably scale; the only continuous bottleneck observed remains on the multiplicative side.

For GEENNS, the original vision posits frozen specialist grids and an evolved mapper that composes their outputs; the mapper's compositional *operator* has been an implementation detail rather than an explicit design axis. These results argue for promoting operator choice (multiplicative, additive, mixture-of-experts gating, or task-dependent) to an explicit evolutionary variable, since its selection co-determines what compositions are representable.

7.6.6 Synthesis: What the Prototype Answers

The seven prototype questions of §7.6 find the following answers in the Boolean conjunction tests and the continuous-gate extension.

Feasibility, composition, routing, and grid utilization (PQ1–PQ4) are answered affirmatively. EMR-HyperNEAT specialist grids converge reliably across all five tested Boolean gates, providing frozen modules suitable for composition (PQ1; §7.6.2). Evolved mappers learn to compose pre-solved grid outputs into compound solutions, with the routing discovered by evolution rather than designed (PQ2; §7.6.2, §7.3.3). At small grid counts the compositional pipeline delivers a speed advantage over the monolithic baseline; beyond the complexity threshold the speed advantage gives way to a capability advantage (PQ3; §7.6.3). The mapper genuinely uses grid outputs rather than bypassing them, although weight magnitude and ablation impact tell complementary stories (PQ4; §7.6.2, §7.6.4).

Generalization and reuse (PQ5–PQ6) succeed with architectural support, not at the individual mapper level. In the fixed-channel generalization test no individual zero-shots unseen compositions, but under input-pair permutation the source-evolved champions themselves do; generalization is therefore a property of the evolved distribution, concentrated at the top of the source ranking, and the open problem is extending it to recompositions that change what the channels mean (PQ5; §7.6.7). The same frozen grids support mapper evolution for different compound tasks (PQ6; §7.6.2). Unconstrained mappers route indiscriminately, but architectural gating elicits selective routing when task structure admits it, though gate weights remain task-specific.

Scaling (PQ7) is the decisive result. Compositional architecture maintains reliable convergence at every tested scale while monolithic baselines degrade monotonically and collapse at the largest scale. The complexity threshold manifests at 4-grid as a convergence-speed inversion and at 5-grid as a reliability gap. Continuous-gate experiments relocate the bottleneck from substrate capacity to the composition operator itself (§7.6.3, §7.6.5).

7.6.7 Limitations and Open Questions

The seven prototype questions answered above leave one deeper question open: whether the *individual mapper* learns compositional reasoning in the sense the GEENNS vision requires, and the answer given by the compositional generalization test alone is "no."

That test holds the channel assignment fixed while the target function changes: the specialists keep the routing they were trained with, so the recomposed target asks the mapper to reinterpret channels whose semantics no longer match. Mean fitness per unseen composition spans 0.53–0.80, with no seed crossing the threshold, well below the 0.95 convergence threshold. Under fixed channels, individual mappers learn weight patterns that recomposition breaks. But this is not the whole answer.

Warm-start transfer reshapes this picture without overturning it. When the channel assignment tracks the target (the input-pair permutation setting), warm-started populations converge in a median of one generation at both scales: the reported 45–79× and 223–351× speedups are the cold-start medians themselves, because the warm population already contains what the search is for. Roughly 0.3% of source-evolved individuals are latent solvers of the unseen target, and they are typically the run champion itself: their median source-fitness rank is first, and in 56 of 58 seed–target views the source champion is among them. For 93.5% of individuals, fitness is numerically identical (within 10^{-6}) on the source and the permuted target, so source selection concentrates the transferable representation at the top of the ranking rather than concealing it; random-initialization populations contain fewer than 0.01% such solvers even at pop 3000, so the representation is a product of source evolution, not sampling volume. Transfer is conditional, however: it does not survive the fixed-channel test above, and under gate-set substitution (NOR→XNOR at 5-grid Boolean) only 4 of 29 source-evolved populations retain a zero-shot solver at the strict 0.99 bar, while the additive continuous pipeline transfers 12 of 14 at its $R^2 \geq 0.85$ criterion under the matched 5-gate substitution, dropping to 59% at 6-gate and returning to 86% at 7-gate (Appendix D.3.3). Where the channel assignment tracks the target, extraction is already solved — for that family of recompositions, the champion is the general mapper; where it does not, no individual transfers, and evolving mappers whose strategies survive changes in channel semantics remains the open problem. The warm-start mechanic, the 17,400-individual population census (Figures D.4 and D.5), and the gate-set sensitivity analysis are in Appendix D.3.3.

Bottom line. Composition is the only condition that remains reliable as grid count rises (Table 7.7), and the gap widens from marginal at 4-grid to decisive at 5-grid, where every monolithic baseline collapses to 0%. Modular reuse is validated (PQ6); selective routing scales through gated mappers at 3-, 4-, and 5-grid (100%/100%/96.6%); and continuous-gate experiments relocate the scaling limit from architecture to composition operator. Warm-start transfer shows generalization is real under input-pair permutation, where the ∼0.3% latent solvers are typically the converged champions themselves, while no individual survives the fixed-channel test and only 4 of 29 populations survive gate substitution at the strict bar. What remains open is no longer "does modular composition scale" but a cluster: scaling beyond 5 grids and beyond Boolean tasks, designing composition operators for continuous domains, and evolving mappers that survive recompositions which change the channel semantics rather than merely relabel them.

7.7 Chapter Summary

This chapter established three things.

First, GEENNS's design draws on three biological and computational principles (columnar homogeneity with functional diversity, evolutionary architecture search, and emergent modularity under compositional task pressure) and specifies an architecture of grids as domain specialists and mappers as compositional coordinators.

Second, three barriers block the path from substrate to compositional intelligence: computational (hyperparameter cost and quadtree bottleneck, Chapters 3 and 5–6), expressiveness (uniform activation functions, §8.4), and multi-task (no task-specific mechanism, Chapter 4 and §8.4). Chapters 3–6 remove the computational barrier and supply the infrastructure on which companion work addresses the other two. A prototype of approximately 3,100 runs validates the modular architecture up to 5-grid scale: compositional pipelines retain 100% success on XOR∧AND∧OR∧NAND∧NOR while every monolithic baseline (mono-tanh, mono-sin, and dynamic per-node function selection alike) scores 0%. The 4-grid convergence speed inversion first signaled that the complexity threshold widens at 5-grid into a reliability gap, converting an arguable efficiency advantage into a decisive capability advantage. Gated selective routing scales through 3-, 4-, 5-, and 6-grid Boolean conjunctions (100%/100%/96.6%/72% success), rescuing the augmented pipeline at the 6-grid scale where it falls to 1/29; the complexity threshold therefore has a two-stage structure within the Boolean regime. Continuous-gate experiments relocate the scaling limit from substrate capacity to the *composition operator*: multiplicative composition fails at 4-gate while an additive mean holds at 100% through at least 8 continuous gates with approximately linear growth in convergence cost. Warm-start transfer reframes generalization as a property of the evolved distribution: under input-pair permutation, source-evolved populations contain roughly 0.3% zero-shot solvers of unseen targets, typically the converged champion itself, against fewer than 0.01% in random-initialization populations even at pop 3000. Three open problems follow directly: (a) scaling compositional reliability beyond five grids and beyond Boolean conjunctions; (b) moving past the multiplicative ceiling in continuous composition, where the operator rather than the architecture dictates the limit; and (c) extending champion-carried permutation transfer to recompositions and substitutions where transfer largely fails (4 of 29 populations at the strict bar). The full formal specification (multi-level CPPN hierarchy, developmental pipeline, plasticity mechanisms, and mathematical formalization) is preserved in Appendix B as a design target for future implementation.

Third, the prerequisite chain itself was a discovery: each barrier had to be solved before the next could be attempted, revealing the substrate problem as thesis-scale in its own right. With the prototype evidence in place, the thesis moves GEENNS from a theoretical design to a testable substrate-and-mapper pipeline whose remaining open questions, listed above, are now specific rather than diffuse.

Part V

Conclusion

Chapter 8

Conclusion and Future Work

> Success is the ability to go from one failure to another with no loss of enthusiasm.
>
> Attributed to Winston Churchill

This thesis systematically diagnosed and addressed three structural barriers preventing adaptive substrate neuroevolution from scaling beyond toy problems. The progression was cumulative: optimization revealed a performance ceiling, diagnosis proved it structural, and redesign broke through it. Table 8.1 maps each barrier to its diagnosis, solution, and quantified evidence; the evidence base comprises approximately 14,400 experimental runs (accounting in the footnote below and Table 8.4).[1]

This chapter synthesizes the thesis contributions and charts the path forward. Section 8.1 provides a unified summary of the five contributions with quantified evidence. Section 8.2 revisits the four thesis research questions (TRQ1–TRQ4) with definitive verdicts and identifies what remains open. Sections 8.3–8.5 address limitations, future research directions including the GEENNS compositional vision, and a closing reflection.

8.1 Contributions Summary

This thesis makes five primary contributions, summarized in Table 8.2. Each contribution is tied to the thesis research question it addresses and the chapter that provides the experimental evidence. The contributions span methods (C1, C2), the thesis's two diagnostic discoveries (C3 representational, C4 computational), and system design (C5); C3 pairs its diagnosis with a working remedy. The spread reflects that making adaptive substrate neuroevolution practical required not only building new tools but also proving that the old tools had fundamental limits. The contributions are presented in the order they appear in the thesis, which reflects their logical dependency.

[1] Chapters 3–6 contribute 11,311 runs (including ∼1,410 cross-domain validation runs from Section 6.5.10); Chapter 7's prototype adds approximately 3,100, bringing the total to approximately 14,400.

Table 8.1: At-a-glance resolution of the three barrier clusters. Each row maps a barrier to its diagnosis, solution, quantified evidence, and remaining limitations. Chapter cross-references indicate where the full experimental details appear.

Barrier	Diagnosis (Chapter)	Solution (Chapter)	Evidence	Limitation
Configuration (>3B configs)	Narrow viable region in 16-parameter space; sensitivity dominated by variance threshold + CPPN complexity (Ch 3)	TPE optimization + early-stopping pruner (Ch 3)	29.0% MNIST (2,013 trials); 41.6% cost reduction ($F_1 = 0.872$)	Structural ceiling, not search failure; pruner calibrated for MNIST only
Representation (3.6% pixel coverage)	CPPN geometry × quadtree variance threshold concentrates resolution near origin (Ch 4)	Spatially partitioned experts: 13 experts on non-overlapping input partitions (Ch 4)	43% mean / 50% peak MNIST ($d = 8.70$ vs. 21% baseline); 85% peak coverage	Fixed partitions, not evolved; peak coverage 85% not 100%; MNIST only
Computation (quadtree sequential)	Per-individual topology prevents vectorization; depth dominates runtime ($p < 10^{-71}$, $\eta^2 = 0.54$) (Ch 5)	EMR-HyperNEAT: lazy→eager reformulation (Ch 6)	12–34× per-gen GPU speedup (XOR, D5–7, pop 1000); ∼33× cumulative over 30 gen (XOR, D7; both rest on the depth-7 streaming estimate); CPU speedup also demonstrated; depth 13 validated	Speedup measured on XOR; quality across 15 problems in 4 domains; tensor representation enables companion work (per-node functions, neuromodulation, aggregation); multi-GPU: memory scaling (depth 8–13)

Table 8.2: Quantified contributions of this thesis. All statistical claims are backed by experimental evidence in the corresponding chapters. See also Table 1.1 for the equivalent summary from Chapter 1.

Contribution	Type	Key Evidence	Impact	Ch.
TPE for ES-HyperNEAT	Method	2,013 trials, 29.0% MNIST	First systematic optimization	3
Early-stopping pruner	Method	$F_1 = 0.872$, $G^* = 3$, $T^* = 0.140$	41.6% cost reduction; landscape diagnostic	3
Central bias diagnosis + Partitioned experts	Diag.+Method	3.6%→85% peak coverage; 43.07% MNIST	105% improvement ($d = 8.70$)	4
Quadtree–GPU incompatibility	Theory	$F(6, 449) = 87.04$, $p < 10^{-71}$, $\eta^2 = 0.54$, $R^2 = 0.95$	1.65× ceiling proven (reference configuration)	5
EMR-HyperNEAT	System	Lazy→eager tensor reformulation; 12–34× GPU speedup (XOR, D5–7, pop 1000; 34× = streaming estimate); 100% solve D4–7 (measured cells, pop $\geq$ 500); 15 problems, 4 domains	Enables new research directions	6
Aggregate (Ch 3–6)		∼11,300 unique runs across 5 papers	—	3–6

Table 8.2 provides the quantified evidence for each contribution; Chapters 3–6 present the full experimental details. Beyond what the table records, C5's reformulation also produces the uniform matrix representation on which the companion work's per-node function evolution, neuromodulation, and aggregation selection are built (Chapter 6).

From individual contributions to a unified system. The five contributions are not independent improvements applied to separate problems. They form a pipeline: C1 reveals the ceiling, C3 diagnoses the representational cause, C4 diagnoses the computational cause, C2 makes the diagnostic experiments affordable, and C5 removes the computational barrier so that future work can combine C3's spatial decomposition with C5's tensor reformulation. The progression from "how well does this algorithm perform?" (C1) through "why does it fail?" (C3, C4) to "how do we fix it?" (C5) is the thesis thread. A practitioner who wishes to use adaptive substrate neuroevolution on a new problem can now follow a concrete pipeline: optimize hyperparameters with Tree-structured Parzen Estimator (TPE) search and pruning (C1, C2), partition the input space for full coverage (C3), and construct substrates on GPU via Eager Multi-Resolution HyperNEAT (EMR-HyperNEAT, C5). This pipeline did not exist before this thesis.

Prototype validation. The GEENNS proof-of-concept prototype (Chapter 7) validates that the substrate infrastructure built in this thesis supports compositional reasoning at small scale. Across approximately 3,100 runs, frozen specialist grids composed through evolved mappers achieve 100% success on compound Boolean tasks from 1-grid through 5-grid scale. At 5-grid scale every monolithic baseline (uniform tanh, uniform sin, and dynamic per-node function selection alike) collapses to 0% success. Pushing one

grid further surfaces a second-stage threshold within the Boolean regime: at 6-grid the augmented pipeline collapses to 1/29 seeds, and gated routing recovers most of the lost ground (21/29, 72%). A continuous-gate extension relocates the scaling limit from substrate capacity to the composition operator: multiplicative composition fails at 4-gate while an additive mean holds at 100% through at least 8 continuous gates.

The same frozen grids support new compound tasks without retraining (57–93% success across three compositions not seen during specialist training), confirming modular reuse. However, generalization fitness remains 0.53–0.80, well below the 0.95 convergence threshold: mappers learn task-specific routing, not general compositional strategies. Warm-start transfer reframes this: when the channel assignment is re-wired to match the recomposed target, source-evolved populations contain roughly 0.3% zero-shot solvers of the unseen task, typically the converged champion itself (against fewer than 0.01% in random-initialization populations even at pop 3000), placing the compositional signal in the evolved distribution; what remains open is transfer across recompositions that change what the channels mean. The gap between validated modular composition and open compositional generalization defines the boundary between this thesis and its future work.

8.2 Research Questions Revisited

This section revisits the four thesis research questions stated in §1.3. For each question, we restate it in its original form, provide a definitive answer grounded in the experimental evidence from the corresponding chapters, and identify open questions.

TRQ1: How can the hyperparameter space of adaptive substrate neuroevolution be navigated efficiently?

Answer: Using TPE combined with domain-specific early-stopping (Chapter 3), developed and validated on MNIST classification. The 2,013-trial TPE campaign concentrated search on productive regions of the 16-parameter ES-HyperNEAT space and established a validated peak of 29.0% MNIST accuracy — a ceiling that resisted parametric improvement and motivated the architectural investigations in Chapters 4–6. Without the combined pipeline, this campaign would have been prohibitively expensive, and the diagnostic chain that follows could not have been conducted at sufficient scale. A secondary but revealing finding is that the pruner's generation-3 decision boundary on MNIST is consistent with a bimodal landscape: 16.6% of the 2,013 trials stagnate at the chance floor while 74.9% form a broad productive band (8.4% sit between), and the generation-3 signal predicts the 80th-percentile success label (defined on the derivation set) with $F_1 = 0.872$ on the held-out validation set (§3.5.1, §3.6.2, §3.9). *What remains open*: the pruner's checkpoint ($G^* = 3$) and fitness threshold ($T^* = 0.140$) were both calibrated on MNIST; G^* may be more task-robust because it reflects NEAT's complexification dynamics rather than task-specific fitness scale, but cross-task validation is open. EMR-HyperNEAT also introduces additional hyperparameters whose optimization landscape may differ, and the pruner requires recalibration for EMR's faster convergence.

TRQ2: Can ES-HyperNEAT's fundamental limitations be diagnosed empirically?

Answer: Yes. Two independent structural limitations, both diagnosed empirically: representational collapse to 3.6% MNIST pixel coverage (28/784 pixels regardless of hyperparameter settings, Chapter 4) and a topology-uniqueness problem in ES-HyperNEAT's quadtree (Chapter 5): the quadtree refines the variance landscape of each Compositional Pattern-Producing Network (CPPN) into a unique network topology with unique node counts, positions, and connections, leaving no fixed-shape population-level tensor and thus blocking the Single Instruction, Multiple Threads (SIMT) parallelism on which GPU acceleration depends. Both are structural properties of the algorithm, not configuration artifacts. This distinction transforms the thesis from engineering improvements into a systematic investigation of fundamental barriers. *What remains open*: whether EMR-HyperNEAT's eager evaluation introduces its own coverage biases on tasks beyond MNIST, and whether the uniform-vs-adaptive resolution trade-off produces new failure modes on tasks with highly non-uniform information density.

TRQ3: Can adaptive substrate discovery be redesigned for massively parallel execution without sacrificing adaptivity?

Answer: Yes, through EMR-HyperNEAT's lazy-to-eager tensor reformulation (Chapter 6), which recasts adaptive substrate discovery as batch matrix operations on fixed multi-resolution grids, producing uniform tensors that vectorize on any hardware backend while preserving adaptive sparsity. The resulting matrix representation is the deeper contribution: it enables a family of nature-inspired extensions explored in the companion work (§8.4.1): per-node activation function evolution, neuromodulation via gain-modulated receptor densities (drawing on biological gain modulation), per-node aggregation selection, and bio-inspired palette meta-learning over strategies such as clonal selection, predator-prey dynamics, circadian cycling, and immune memory. None of these was computationally feasible under the quadtree approach. Experiments requiring months of CPU time now complete in days, and solve rates at high depths go from 0–33% to 100%. *What remains open*: multi-GPU configurations provide memory scaling but incur speed overhead; cross-domain validation extends to 15 problems in 4 domains (Boolean parity, visual classification, reinforcement-learning control, and regression), but tasks requiring hundreds to thousands of hidden nodes remain untested.

TRQ4: Can input decomposition extend adaptive substrates to high-dimensional problems?

Answer: Yes, through spatially partitioned experts (Chapter 4), which expand pixel coverage from 3.6% to a peak of 85% (tied at $N_e = 8$ and $N_e = 14$) and, at the accuracy optimum $N_e = 13$ (73% coverage), double MNIST accuracy ($d = 8.70$): coverage and accuracy peak at different expert counts. Of the 14 aggregation strategies tested, F1-weighted (Performance-Weighted) aggregation was the most effective; naive averaging degrades with expert count. *What remains open*: validation used MNIST only and re-evaluated the single best hyperparameter configuration from Chapter 3's TPE campaign (random configurations fail to learn at all and provide no baseline to test partitioning against), so the partitioned-experts gain under other viable configurations is untested; whether an aggregation strategy beyond the 14 evaluated outperforms F1-weighting (for instance, end-to-end learned aggregation in the Mixture-of-Experts sense) is open; com-

bining the architecture with EMR's tensor reformulation remains an engineering task; and the optimal partitioning strategy for different input dimensionalities is unexplored.

8.3 Limitations and Lessons Learned

Some limitations are deliberate (choosing depth over breadth), some are imposed (time, hardware, the combinatorial explosion of possible experiments), and some are only visible in retrospect. This section catalogs the scope limitations that constrain the generalizability of the results (§8.3.1), the approaches that failed along the way (§8.3.2), the methodological lessons that may spare future researchers from repeating these pitfalls (§8.3.3), the threats to validity (§8.3.4), and the computational footprint of the work (§8.3.5).

Table 8.3 provides an at-a-glance summary of the thesis's principal limitations. Each is discussed in the subsections that follow.

Table 8.3: What this thesis did *not* achieve. Each limitation is scoped to the chapter(s) it affects and contextualized against the relevant external standard.

Limitation	Ch.	Context
MNIST accuracy capped at 43%	4	Deep learning (gradient-trained networks) exceeds 99%; contribution is the diagnosis, not the accuracy
XOR-specific GPU speedup headlines	6	Cross-domain validation across 15 problems in §6.5.10; industrial scale untested
Quadtree incompatibility is SIMT-scoped	5	Non-SIMT accelerators (dataflow, FPGA) untested
Prototype fails compositional generalization (fixed channels)	7	Mean fitness 0.53–0.80, threshold 0.95; re-wired channel transfer succeeds (§8.1)
Two Spirals 0% solve rate	6	Architecture-limited; neither ES- nor EMR-HyperNEAT solves it
No gradient baseline comparison	All	Evolutionary approach only; §8.3.1 discusses rationale
Single-objective fitness only	3–6	No Pareto or multi-objective optimization
CPU-only experiments	3, 4	EMR-HyperNEAT did not yet exist; GPU acceleration first demonstrated in Ch 6

8.3.1 Thesis-Level Limitations

Benchmark scope. XOR is the primary *speedup* benchmark in Chapters 5 and 6, isolating algorithmic differences from task-level confounds. Cross-domain validation (Section 6.5.10) extends the evidence to 15 problems across 4 domains (Boolean gates through 6-bit parity, visual pattern recognition, reinforcement learning control, and regression), totaling approximately 1,410 additional runs. MNIST appears in Chapters 3 and 4 at ES-HyperNEAT scale (784 inputs, 10 outputs), not at modern deep learning scale. The thesis would be stronger with tasks requiring hundreds or thousands of hidden nodes, but the computational prerequisites were only available after EMR-HyperNEAT was complete.

43% MNIST is modest by any standard. Deep learning exceeds 99% on MNIST. The 43% accuracy achieved by the partitioned-experts architecture (Chapter 4) represents a 105% improvement over the monolithic baseline and a Cohen's $d = 8.70$, which is methodologically significant. But the absolute performance gap remains substantial. The contribution is the *diagnostic insight* (the central bias blocks 96% of the input space) and the *architectural principle* (spatial decomposition recovers input coverage), not competitive classification accuracy. Overstating this result would be dishonest.

The ∼56 percentage-point gap between evolved and gradient-trained networks on the same task, achieved with roughly two orders of magnitude fewer hidden neurons, confirms that indirect encoding's value lies elsewhere: in topology discovery, in architectural search, and in compositional modularity, capabilities that gradient descent cannot provide because it requires the topology to already exist.

Statistical rigor varies. The GPU timing campaign behind Table 6.6 uses $N = 3$ replications per configuration due to computational constraints (seed counts are detailed in §8.3.5). The strong statistical significance (Mann-Whitney $p < .001$, Cliff's $\delta \geq 0.59$) and the 456-run ES-HyperNEAT baseline (§6.5.1) mitigate this concern, but the GPU-side confidence intervals are wider than ideal. Ten timing replications per configuration would have narrowed them; the later cross-framework GPU pass (§6.5.4) runs N=30.

Scale of real-world validation. Cross-domain validation (Section 6.5.10) includes 3 regression datasets (CCM Financial, Earnings-Price, CalHousing) and 3 reinforcement learning environments (CartPole, Acrobot, MountainCar), but all operate at small scale (2–12 input features). Industrial applications with hundreds of features or high-dimensional sensory inputs were not evaluated. Regression performance ($R^2 \leq 0.451$) falls below conventional ML baselines; this thesis investigated evolutionary diagnostics rather than fitness-optimization strategies, leaving open whether the gap is intrinsic to indirect encoding or closable through dedicated optimization.

CPU-bound execution for Chapters 3–4. All pre-EMR experiments ran on CPU, inheriting the quadtree bottleneck that the rest of the thesis diagnoses and resolves. The TPE optimization (2,013 trials, Chapter 3) and the 13-expert partitioned-experts architecture (Chapter 4) could benefit from EMR's tensor reformulation but have not been re-validated on the new platform. Whether EMR substrates achieve higher MNIST accuracy than quadtree substrates at the same depth remains an open question.

Fitness design and accelerator scope. All experiments optimize a single scalar fitness; Pareto and multi-objective formulations, natural for the accuracy-versus-cost trade-offs this thesis measures, were not explored. The quadtree incompatibility diagnosis is likewise scoped to SIMT architectures: non-SIMT accelerators (dataflow processors, FPGAs) might host the lazy traversal efficiently and were not tested.

Hyperparameter-optimization method coverage. TPE is validated against random search in Chapter 3, and the early-stopping pruner is benchmarked against Hyperband, the successive-halving family's representative; other optimizer families (CMA-ES, SMAC) and Hyperband used as a standalone searcher were not benchmarked, and a cross-family comparison at the protocol's 2,013-trial scale would be a separate study rather than

an increment on this one. The pruner's generation-level signal is optimizer-agnostic and would compose with any of them, but the structural-ceiling conclusion of Chapter 3 rests, at present, on the two search families that were run—random search and TPE—and extends to the remaining families as a hypothesis, not a measurement.

No end-to-end demonstration. The partitioned-experts spatial decomposition (Chapter 4) has not been combined with EMR-HyperNEAT (Chapter 6). The representational solution and the computational solution were developed independently. An integrated partitioned-experts + EMR pipeline (13 expert substrates each evolved on GPU via EMR) is the natural next experiment, but it falls beyond the scope of this thesis. The multiplicative potential is concrete: the Chapter 3 pruner reduces 41.6% of search cost (measured on MNIST and assumed to transfer); EMR provides GPU speedup (dependent on task, depth, and population); partitioned-experts decomposition enables parallel specialist evolution on independent cores. The following estimates are illustrative, based on XOR-scale trials (Table B.3 in Appendix B anchors the arithmetic): for a modest (k=5, T=10) GEENNS pipeline (34 CPPNs at 2,013 TPE trials each, ∼68,000 configurations per Appendix B), the pre-thesis cost was ∼250 CPU-days (on the order of years for harder tasks such as MNIST). With the pruner alone: ∼146 CPU-days. With EMR alone: ∼8.3 GPU-days. With both: ∼4.9 GPU-days. Adding partitioned-experts-style parallelism across specialist grids would further divide by the number of independent experts, bringing per-grid optimization into the range of hours. The pieces are individually validated. Assembling them is an engineering task, not a research one.

Why not gradient descent? As discussed in §1.1, evolution and gradient descent operate at different architectural levels: evolution discovers topology (discrete structural decisions with no gradient signal), while gradients tune weights within a predefined topology. The same division covers per-node function selection. Which function a node computes is a discrete choice, and it offers no gradient to follow. All contributions in this thesis concern topology discovery. The NAS hybrid direction (§8.4.1) outlines how the approaches can be combined.

8.3.2 Failed Approaches and Negative Results

Not all investigations were productive. Four approaches consumed significant effort and yielded negative results, but each failure sharpened the thesis argument.

Quadtree optimization (Chapter 5). Four acceleration strategies (batched division queries, batched pruning queries, vmap substrate evaluation, and precomputed offsets) saturated at the ceiling established in Chapter 5. The failure was productive: it proved the bottleneck was architectural, not implementational, providing the empirical case for EMR-HyperNEAT. The optimization effort also transferred concretely: the batching, JIT, and `vmap` techniques developed for the quadtree informed EMR's own GPU implementation.

HSHG spatial hashing (Chapter 5). Hierarchical Spatial Hash Grids replaced the quadtree's sequential traversal but introduced over-discovery: 29–63 hidden nodes where 1–2 suffice. The lesson is that the substrate discovery problem is not merely speed: it is *topology uniformity*, which EMR's fixed grid solves by construction.

Two Spirals benchmark (Chapter 6). Zero percent solve rate across all six connection type configurations. The three-part diagnosis (depth limitation, single hidden layer, and uniform activation function) is presented alongside the cross-domain results in Section 6.5.10. The failure is one of expressiveness (insufficient depth and function diversity), not topology, and it identifies the clearest architectural target for DEMR-HyperNEAT (Deep Eager Multi-Resolution; §8.4.1) and per-node function evolution.

Multi-GPU speed scaling (Chapter 6). Distributing EMR across multiple GPUs incurs 1.5–1.7× speed overhead relative to single-GPU, because Python coordination dominates the per-generation GPU workload. Even at this overhead, multi-GPU EMR remains approximately 20× faster than ES-HyperNEAT on CPU (XOR, depth 7, pop 1000). Multi-GPU provides memory scaling (substrate depths 8–13 via streaming across devices) rather than speed scaling; true multi-GPU speed acceleration remains an open question, with moving the coordination loop into XLA a plausible but untested direction.

8.3.3 Lessons for Neuroevolution Researchers

Five years of experimental work yielded methodological lessons that generalize beyond this thesis. We present them as concrete principles.

Lesson 1: Profile before optimizing. The quadtree bottleneck was an *architecture* problem, not an *implementation* problem. We invested significant effort optimizing the quadtree implementation (Chapter 5): four strategies achieved the same cumulative ceiling described in C4. That effort should have gone into redesign sooner. The lesson: when profiling reveals that a component consumes $> 80\%$ of runtime and the implementation is already efficient, architectural replacement is often more productive than further implementation tuning.

Lesson 2: Test assumptions early. The central bias was hiding in plain sight. ES-HyperNEAT substrates on MNIST attended to 28 of 784 unique pixel positions, but this was only discovered through a targeted receptive field analysis (Chapter 4). A 10-line script plotting which input coordinates had active connections would have revealed the problem immediately. The lesson: before investing in optimization, verify that the system represents the problem it is supposed to solve. Visualize receptive fields, plot connection distributions, inspect substrates. Assumptions about what the algorithm "should" discover with enough compute are not evidence that it does.

Lesson 3: Reformulate, do not parallelize. EMR-HyperNEAT's speedup comes not from faster execution of the same algorithm but from a structurally different algorithm suited to GPU hardware (Chapter 6). This lesson generalizes: when an algorithm is fundamentally mismatched to its target hardware, reformulation is often more productive than porting.

Lesson 4: GPUs need different algorithms. The SIMT execution model requires uniform work: all threads in a warp must execute the same instruction simultaneously. Algorithms with per-element branching (quadtrees, recursive subdivision, adaptive mesh refinement) are structurally incompatible regardless of implementation quality. This is not a limitation of current GPUs; it is a consequence of the SIMT architecture. Researchers

designing evolutionary operators, fitness evaluations, or substrate construction routines for GPU should verify *before implementation* that the algorithm produces fixed-shape tensors across the population.

Lesson 5: Benchmarks require context to be meaningful. The 43% MNIST accuracy from the partitioned-experts architecture (Chapter 4, $d = 8.70$) is a failure in deep learning, a substantial advance in neuroevolution (105% improvement over monolithic baselines), and irrelevant in reinforcement learning. We recommend always reporting both *absolute performance* (for cross-field comparison) and *relative improvement with effect size* (for within-field significance). Omitting either is misleading: absolute performance alone understates methodological contributions; relative improvement alone overstates practical utility.

8.3.4 Threats to Validity

Internal validity. The primary speedup metric (12–34×) is measured on XOR at depths 5–7 with population 1000. Cross-domain validation (Section 6.5.10) confirms speedups on 15 problems across 4 domains but with task-dependent magnitude (e.g., 5.5× on CCM Financial at depth 6, population 100; the 34× endpoint is the depth-7 streaming estimate). The pruner's $F_1 = 0.872$ was calibrated on MNIST only, risking overfitting to that task's fitness landscape; validation on novel task distributions is needed.

External validity. All contributions are validated within the ES-HyperNEAT / EMR-HyperNEAT lineage. Whether the central bias diagnosis, the quadtree incompatibility, or the EMR reformulation transfer to other indirect encoding methods (for example, HyperNEAT with a link-expression output, or deep neuromodulated variants) is untested. The GEENNS prototype validates composition up to 5-grid Boolean scale (probed at 6-grid, where the augmented pipeline collapses to 1/29 seeds and gated routing recovers 72%) and through 8-gate continuous scale under additive composition; the multiplicative ceiling at 4-gate identifies the composition operator as a primary design axis. Generalization to higher-dimensional non-conjunctive domains remains an open question (Section 8.4).

Construct validity. Solve rate (fraction of seeds achieving ≥ 0.95 fitness) is the primary metric throughout. This conflates convergence probability with solution quality: a configuration achieving 100% solve rate with median 78 generations may be preferable to one achieving 100% in 3 generations if the latter produces fragile solutions. We mitigate this by reporting both solve rate and convergence generation, but we do not measure solution robustness to input perturbation.

8.3.5 Reproducibility and Computational Resources

All experiments in this thesis are reproducible using Python 3.11 and JAX 0.4.x, except the Chapter 6 cross-framework CPU pass, which pins JAX 0.6.1 (§6.5.4). Most experiments use $N = 30$ seeds per condition following neuroevolution community standards; the Table 6.6 GPU timing campaign uses $N = 3$ due to computational constraints (see §8.3.1). Table 8.4 summarizes the approximate computational cost by chapter.
In total, the thesis required approximately 14,400 experimental runs consuming at least 2,100 compute-hours across CPU and GPU hardware; the Chapter 7 figure reflects the

Table 8.4: Approximate computational resources for this thesis.

Chapter(s)	Runs	Hours	Hardware	Primary Task
Ch 3–4	∼5,850	∼900	CPU (distributed, 13–14 heterogeneous machines)	Hyperparameter search
Ch 5	456	∼50	M4 Max (CPU) + RTX 2080 Ti (GPU)	Incompatibility proof
Ch 6	∼5,000	∼150	M4 Max (CPU) + RTX 2080 Ti (GPU)	Tensor reformulation
Ch 7	∼3,100	≳1,015	M4 Max (CPU)	Composition validation (incl. 5-grid Boolean probed at 6-grid; 8-gate continuous additive)
Total	∼14,400	≳2,100	—	—

original prototype scope, with the 6-grid Boolean and continuous additive-scaling extensions adding further compute that has not been separately accounted for. Random seeds, hyperparameter configurations, and result files are archived with the framework source code.

8.4 Future Work: What EMR Enables

EMR-HyperNEAT is infrastructure, not a conclusion. The tensor reformulation transforms research directions that were computationally prohibitive into tractable investigations. This section sketches the research directions that EMR enables (§8.4.1), the longer-term GEENNS vision with an assessment of prototype evidence (§8.4.2), and the scaling challenges that remain (§8.4.3).

8.4.1 Research Directions Enabled by EMR

Companion work has explored five of the research directions this infrastructure enables. We describe them to clarify what the thesis makes feasible and why.

Per-node activation function evolution. EMR's batch CPPN evaluation makes it possible to assign activation functions on a per-node basis via additional CPPN output dimensions at negligible additional cost. This enables investigation of how activation function choice interacts with indirect encoding's symmetric weight generation, and whether oscillatory functions provide qualitative advantages over monotonic alternatives for specific problem classes. Companion work confirms both hypotheses [16]: in evolutionary search over sparse feedforward substrates, oscillatory activations achieve 100% solve rates across parity orders four through eight, where every uniform monotonic activation tested scores 0% (tanh, sigmoid, and ReLU all at parity-4; tanh at every order through parity-8; 0.95 solve criterion). A single additional CPPN output dimension is enough to make that solvability divide discoverable.

Bio-inspired palettes. Per-node function evolution raises a meta-question: how should the palette of available functions be managed across evolutionary time? Strategies inspired

by biological mechanisms (clonal selection, critical periods, predator-prey dynamics, circadian cycling) offer principled approaches to balancing exploration of new functions against preservation of useful ones. Companion work evaluates 13 such strategies across more than 3,000 runs [18]: bio-inspired palette management matches baseline solve rates while converging up to 2.1× faster (51.5% compute savings for the circadian strategy). The strongest predictor of strategy success is timescale compatibility: strategies whose operating period aligns with the evolutionary evaluation window consistently outperform those with slower timescales, and strategy rankings reverse across problem types, with no single strategy dominating all domains. A sequential-task companion study extends these strategies and validates that the discovered palettes actually transfer: under persistent-population protocols, the immune-memory strategy retains 97–100% of the oscillatory vocabulary across task reversals and achieves 88% multi-task continual-learning solve rates. The meta-learned palettes are transferable computational vocabulary, not task-specific artifacts.

Per-node aggregation function evolution. Analogous to per-node activation, aggregation functions (how a node combines its incoming signals) can be evolved per-node via CPPN output dimensions. This investigates whether aggregation and activation provide complementary or redundant degrees of freedom for substrate expressiveness. Companion work resolves the question in favor of asymmetric complementarity: per-node aggregation raises XOR solve rates from 20% (sum baseline) to 100%, while per-node activation solves higher-order parity where aggregation becomes destructive (0% on Parity-4+); a combined 3-output CPPN (weight, activation index, aggregation index) achieves 100% across all tested problems. The two degrees of freedom compensate for each other's weaknesses rather than duplicating one capability.

Neuromodulation for multi-task learning. Gain modulation, where task-specific neurotransmitter vectors multiplicatively scale neuronal activation through evolved receptor densities, offers a mechanism for multi-task operation from a single evolved topology. The approach draws on biological neuromodulation as computational inspiration: different modulation vectors selectively amplify or suppress neuron subsets, producing task-specific effective sub-networks without weight modification. Multi-task validation in companion work [15] demonstrates this mechanism on five Boolean tasks simultaneously: a single evolved substrate switches behavior through 4D neurotransmitter vectors, achieving 100% multi-task success. In this initial multi-task setup, both the neurotransmitter (NT) vector values and the per-task activation-function selection are hand-engineered, not evolved; the next direction below addresses their co-evolution. Neuromodulation and per-task activation selection are jointly necessary: neither mechanism alone achieves multi-task success, because uniform monotonic activations impose a 75% ceiling that only oscillatory per-task selection removes.

Co-evolving neuromodulatory signals. If neuromodulation enables multi-task operation, a natural question is whether the modulation vectors themselves can be co-evolved alongside substrate topology rather than hand-designed. This direction encounters the population alignment problem: NEAT's selection operator reorders population indices between generations, so externally maintained parameter vectors (e.g., neurotransmitter profiles) lose their genome associations after every selection event. Across 1,500+ experiments in companion work, parent tracking via winner-index reindexing emerges as a

prerequisite (without it, co-evolution reduces to random search); warm-start initialization from the hand-engineered presets closes 90% of the remaining performance gap at population 3,000, and a combinatorial cliff emerges as task count grows (2-task 100% → 5-task 23%). Autonomous NT discovery is therefore feasible, but it depends on careful initialization and population scaling.

Status and dependency structure. These five directions form the dependency chain toward compositional reasoning. The same tensor reformulation also enables a hybrid pipeline (EMR-discovered topology trained by gradient descent), pursued in separate work. Table 8.5 summarizes the chain's status and prerequisites.

Table 8.5: Research directions enabled by EMR-HyperNEAT: dependency structure and status. The EMR-enabled directions (top) have been demonstrated in companion work that extends the thesis infrastructure; the GEENNS-level directions (bottom) remain open.

Direction	Prerequisites	Status
Per-node activation evolution	EMR (this thesis)	Demonstrated
Per-node aggregation evolution	EMR (this thesis)	Demonstrated
Bio-inspired palette strategies	Per-node activation	Demonstrated
Neuromodulation	EMR + per-node activation	Demonstrated
Co-evolving neuromodulatory signals	Neuromodulation	Characterized
GEENNS selective routing	All above + prototype	Open
Mapper transfer	Selective routing	Open
Spontaneous specialization	Selective routing	Open

Note: Statuses indicate results from companion work separate from this thesis.

8.4.2 The GEENNS Vision

GEENNS proposes compositional reasoning through frozen specialist grids and evolved mappers (Chapter 7). This section focuses on why the thesis contributions are prerequisites and on future directions.

Why the five thesis contributions are prerequisites. EMR's speedup and the pruner's cost reduction together bring the GEENNS pipeline from computationally infeasible (∼250 CPU-days at XOR scale) to tractable (∼4.9 GPU-days; §8.3 gives the arithmetic). The five thesis contributions each serve a specific role:

- EMR provides the tensor-vectorizable substrate engine (Chapter 6): specialist grids and mappers both require fast, compositionally extensible substrate discovery.
- The partitioned-experts architecture validates the spatial decomposition principle (Chapter 4): specialist sub-networks solving partitioned problems.
- TPE makes multi-CPPN search feasible (Chapter 3): at the modest (k=5, T=10) GEENNS scale, optimization runs across 34 CPPN configurations (Appendix B), scaling higher with grid count and task count.
- The early-stopping pruner compounds this (Chapter 3): removing 41.6% of each configuration's search cost keeps the 34-configuration bill within the estimates above.

- The quadtree's GPU incompatibility (Chapter 5) justifies the redesign: without EMR, GEENNS is computationally infeasible.

Path forward. The selective routing problem (evolving mappers that preferentially attend to relevant grid outputs) is the critical next step. Four directions are promising:

- *attention-based gating*: augmenting the mapper with a gating mechanism that learns to weight grid outputs based on input properties, analogous to the gating networks in Mixture-of-Experts architectures;
- *scale beyond Boolean gates*: testing at scales where dynamic functions fail (5 grids and beyond, non-Boolean domains), where the compositional advantage may become necessary rather than optional;
- *deep substrates*: extending the eager evaluation approach to deep multi-layer substrate discovery via DEMR-HyperNEAT (Deep Eager Multi-Resolution);
- *hybrid NAS* (§8.4.1): combining EMR's topology evolution with gradient-based weight training.

Toward emergent behavior. Three experimental signatures would constitute evidence of emergent compositional reasoning beyond what the current prototype demonstrates. First, *mapper transfer*: mappers evolved for one compound task that solve a novel compound task without re-evolution, indicating that the mapper has learned a general compositional strategy rather than task-specific weights. Second, *spontaneous grid specialization*: grids that develop interpretable functional specializations (e.g., one grid for feature extraction, another for transformation) without explicit pressure to specialize, merely through the structure of the training tasks. Third, *super-additive composition*: system capabilities that exceed the sum of individual grid capabilities, demonstrating that the interaction between components produces genuinely new functionality. None of these signatures has been demonstrated. The prototype validates modular composition (grids can be frozen and reused) but not emergent composition (the compositional strategy itself arising from evolutionary dynamics). The infrastructure for investigating all three signatures now exists: EMR provides the substrate speed, per-node function evolution provides the representational diversity, and the prototype architecture provides the modular framework. Whether that infrastructure produces genuine emergence is the empirical question that defines the next phase of this research.

8.4.3 Scaling to Complex Problems

The gap between XOR and real-world tasks is large, and bridging it requires more than faster substrate discovery. Three scaling directions define the research frontier.

Image classification. CIFAR-10 ($32 \times 32 \times 3$, 10 classes) is the natural next benchmark after MNIST. The partitioned-experts spatial decomposition from Chapter 4, combined with EMR's depth scaling, provides the architectural and computational infrastructure. However, CIFAR-10 requires color channels, spatial hierarchies, and far larger substrate capacities than any indirect encoding method has demonstrated. The Abstraction and Reasoning Corpus (ARC) may be a more appropriate target for compositional neuroevolution, as it explicitly requires compositional generalization rather than statistical pattern recognition. We do not claim that either target is within immediate reach; we claim only that EMR makes them computationally addressable for the first time.

Multi-modal integration. The GEENNS architecture, if the selective routing problem is solved, would naturally support specialist grids evolved on different modalities (vision, proprioception, symbolic reasoning) composed through task-specific mappers. The frozen-grid design is intended to eliminate the catastrophic forgetting problem in multi-modal continual learning: grids trained on one modality are never modified when a new modality is introduced. Whether this theoretical advantage holds in practice remains unvalidated.

Industrial optimization. The CCM financial benchmark in Chapter 6 is the starting point for applied validation. Job-shop scheduling, portfolio optimization, and control tasks all involve high-dimensional, multi-objective search spaces where evolved substrates could provide solutions that gradient-based methods struggle to find. The 5.5× EMR speedup on CCM at depth 6 (population 100) demonstrates that the platform can handle real-world data dimensions, but competitive performance against domain-specific methods has not been established.

8.5 Closing Reflection

The thesis asked: *Can we make ES-HyperNEAT practical?* The answer is no—not by optimizing it. But by diagnosing its structural limitations and redesigning from first principles, we built EMR-HyperNEAT: a system that preserves the adaptive substrate discovery that makes ES-HyperNEAT valuable while eliminating the sequential bottleneck that made it impractical. The bottleneck did not yield to faster hardware, better hyperparameters, or more careful engineering. It yielded to the lazy-to-eager reformulation at the heart of EMR-HyperNEAT: replacing adaptive exploration with uniform evaluation and post hoc filtering. The ES-HyperNEAT investigation was not in vain: the deeper understanding of how adaptive substrate discovery behaves, and the optimization techniques developed for the quadtree, both carry forward into EMR-HyperNEAT and into future efforts to scale adaptive substrate neuroevolution.

The broader vision—compositional reasoning through evolved substrates, and beyond that, systems whose behaviors emerge from evolutionary dynamics—remains ahead. The prototype validates the mechanics of composition; the generalization problem is open. But the foundation is built. The substrate engine is fast enough and the architectural framework flexible enough to investigate emergent behavior at a scale ES-HyperNEAT could not reach; the input-coverage strategy is demonstrated, though not yet integrated. Whether genuine emergence follows is now an empirical matter. That question belongs to artificial life: the study of systems whose behaviors arise from evolutionary dynamics rather than explicit programming—systems still framed by human choices of encoding and objective, but assessed on what they are, not on how closely they imitate a human target.

Bibliography

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[2] Noor Awad, Neeratyoy Mallik, and Frank Hutter. Dehb: Evolutionary hyperband for scalable, robust and efficient hyperparameter optimization. *arXiv preprint arXiv:2105.09821*, 2021.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[5] Carlo Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R istituto superiore di scienze economiche e commericiali di firenze*, 8: 3–62, 1936.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] Center for Research in Security Prices (CRSP). CRSP/Compustat merged database (CCM). `https://www.crsp.org/products/research-products/crspcompustat-merged-database`, 2019. Accessed via WRDS, November 2019 version.

[10] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI proceedings-international joint conference on artificial intelligence*, volume 22, page 1237. Barcelona, Spain:, 2011.

[11] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. Investigating hyperparameter optimization and transferability for ES-HyperNEAT: A TPE approach. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '24 Companion)*, pages 1879–1887, Melbourne, VIC, Australia, 2024. ACM. doi: 10.1145/3638530.3664144. URL `https://claret.tech/pdf/claret2024tpe`.

[12] Romain Claret, Arthur Gygax, Michael O'Neill, Paul Cotofrei, and Pascal Felber. Early-stopping thresholds for ES-HyperNEAT: A data-driven approach from fitness dynamics. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC), part of IEEE WCCI*, 2026. URL `https://claret.tech/pdf/claret2026pruner`.

[13] Romain Claret, Arthur Gygax, Michael O'Neill, Paul Cotofrei, Michael Palma Mendes, and Pascal Felber. Breaking the central bias: Spatially partitioned experts for coordinate-based neuroevolution. In *Pattern Recognition. ICPR 2026 International Workshops and Challenges (BIOMAP)*. Springer, 2026. URL `https://claret.tech/pdf/claret2026partitioned`.

[14] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. Tensor-accelerated eager multi-resolution grids for evolving large-scale substrates. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '26 Companion)*, pages 533–536, San Jose, Costa Rica, 2026. ACM. doi: 10.1145/3795101.3805361. URL `https://claret.tech/pdf/claret2026emr`.

[15] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. Multi-behavioral evolved substrates through neuromodulation and activation selection. In *ALIFE 2026: Proceedings of the 2026 Artificial Life Conference*, page 78. MIT Press, 2026. doi: 10.1162/ISAL.a.968.

[16] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. Per-node activation function evolution in indirectly encoded substrates: Solvability, limits, and emergent diversity. In *ALIFE 2026: Proceedings of the 2026 Artificial Life Conference*, page 80. MIT Press, 2026. doi: 10.1162/ISAL.a.1042.

[17] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. On scaling coordinate-based neuroevolution: The quadtree bottleneck in ES-HyperNEAT. Preprint, arXiv:2608.24480 [cs.NE]. doi: 10.48550/arXiv.2608.24480, 2026. URL `https://claret.tech/pdf/claret2026quadtree`.

[18] Romain Claret, Michael O'Neill, Paul Cotofrei, and Kilian Stoffel. Bio-inspired palette evolution in indirectly encoded substrates: Timescale compatibility shapes activation function discovery. In Giovanni Iacca, Giorgia Nadizar, Anil Yaman, Doina Bucur, Antonio Della Cioppa, Ting Hu, Eric Medvet, and Sarah L. Thomson, editors, *Parallel Problem Solving from Nature – PPSN XIX*, pages 368–383, Cham, 2027. Springer Nature Switzerland. ISBN 978-3-032-36217-9. doi: 10.1007/978-3-032-36217-9_23.

[19] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[20] Jeff Clune, Jean-Baptiste Mouret, and Hod Lipson. The evolutionary origins of modularity. *Proceedings of the Royal Society b: Biological sciences*, 280(1755):20122863, 2013.

[21] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. routledge, 2013.

[22] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[24] David B D'Ambrosio and Kenneth O Stanley. Scalable multiagent learning through indirect encoding of policy geometry. *Evolutionary Intelligence*, 6(1):1–26, 2013.

[25] Bradley Efron. Better bootstrap confidence intervals. *Journal of the American statistical Association*, 82(397):171–185, 1987.

[26] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[27] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[28] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International conference on machine learning*, pages 1437–1446. PMLR, 2018.

[29] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[30] Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[31] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, 1(1):47–62, 2008.

[32] David B Fogel. Evolutionary algorithms in theory and practice, 1997.

[33] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[34] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *IEEE Transactions on Artificial Intelligence*, 2(6): 476–493, 2021.

[35] Jason Gauci, Kenneth O Stanley, et al. A case study on the critical role of geometric regularity in machine learning. In *AAAI*, pages 628–633, 2008.

[36] Jeff Hawkins. *A thousand brains: A new theory of intelligence*. Basic books, 2021.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[39] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[40] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[41] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[42] Edwin Hutchins. *Cognition in the Wild*, volume 262082314. MIT press Cambridge, MA, 1995.

[43] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[44] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial intelligence and statistics*, pages 240–248. PMLR, 2016.

[45] Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[46] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *nature*, 596(7873):583–589, 2021.

[47] Daniel Kahneman, Patrick Egan, et al. *Thinking, fast and slow*, volume 1. Farrar, straus and Giroux New York, 2011.

[48] Sarah J Karlen and Leah Krubitzer. The functional and anatomical organization of marsupial neocortex: evidence for parallel evolution across mammals. *Progress in neurobiology*, 82(3):122–141, 2007.

[49] Nadav Kashtan and Uri Alon. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences*, 102(39):13773–13778, 2005.

[50] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[51] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[53] Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation, and active learning. *Advances in neural information processing systems*, 7, 1994.

[54] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

[55] Kevin J Lang. Learning to tell two spirals apart. In *Proceedings of the 1988 connectionist models summer school*, pages 52–59. Morgan Kaufmann, 1989.

[56] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[57] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[58] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436–444, 2015.

[59] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[60] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of machine learning research*, 18(185):1–52, 2018.

[61] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[62] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7765–7773. IEEE, 2018.

[63] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[64] Henry Markram, Eilif Muller, Srikanth Ramaswamy, Michael W Reimann, Marwan Abdellah, Carlos Aguado Sanchez, Anastasia Ailamaki, Lidia Alonso-Nanclares, Nicolas Antille, Selim Arsever, et al. Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2):456–492, 2015.

[65] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

[66] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. NEAT-Python. `https://github.com/CodeReclaimers/neat-python`, 2015.

[67] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*, pages 269–287. Elsevier, 2024.

[68] Marvin Minsky and Seymour A Papert. Perceptrons, reissue of the 1988 expanded edition with a new foreword by léon bottou: An introduction to computational geometry. sl, 2017.

[69] Vernon B Mountcastle. The columnar organization of the neocortex. *Brain: a journal of neurology*, 120(4):701–722, 1997.

[70] Michael O'Neil and Conor Ryan. Grammatical evolution: Evolutionary automatic programming in an arbitrary language. *Norwell, MA: Kluwer Academic Publishers. doi*, 10:978–1, 2003.

[71] R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

[72] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[73] Mitchell A Potter and Kenneth A De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary computation*, 8(1):1–29, 2000.

[74] Nicholas J Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90, 1993.

[75] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[76] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[77] Joseph Reisinger, Kenneth O Stanley, and Risto Miikkulainen. Evolving reusable neural modules. In *Genetic and evolutionary computation conference*, pages 69–81. Springer, 2004.

[78] Sebastian Risi and Kenneth O Stanley. An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life*, 18(4): 331–363, 2012.

[79] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[80] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[81] Jeffrey R Sampson. Adaptation in natural and artificial systems (john h. holland), 1976.

[82] Jimmy Secretan, Nicholas Beato, David B D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T Folsom-Kovarik, and Kenneth O Stanley. Picbreeder: A case study in collaborative evolutionary exploration of design space. *Evolutionary computation*, 19(3):373–403, 2011.

[83] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[84] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[85] Moshe Sipper, Eduardo Sanchez, Daniel Mange, Marco Tomassini, Andrés Pérez-Uribe, and André Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. *IEEE Transactions on Evolutionary Computation*, 1(1):83–97, 1997.

[86] Jasper Snoek, Hugo Larochelle, and Ryan Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[87] Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

[88] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[89] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE transactions on evolutionary computation*, 9 (6):653–668, 2005.

[90] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[91] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[92] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[93] Yujin Tang, Yingtao Tian, and David Ha. Evojax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 308–311, 2022.

[94] Amund Tenstad and Pauline C Haddow. Des-hyperneat: Towards multiple substrate deep anns. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2195–2202. IEEE, 2021.

[95] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H Gross. Optimized spatial hashing for collision detection of deformable objects. In *Vmv*, volume 3, pages 47–54, 2003.

[96] Alan Mathison Turing. The chemical basis of morphogenesis. *Bulletin of mathematical biology*, 52(1):153–197, 1990.

[97] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[98] Phillip Verbancsics and Josh Harguess. Image classification using generative neuro evolution for deep learning. In *2015 IEEE winter conference on applications of computer vision*, pages 488–493. IEEE, 2015.

[99] Phillip Verbancsics and Kenneth O Stanley. Constraining connectivity to encourage modularity in hyperneat. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1483–1490, 2011.

[100] Lishuang Wang, Mengfei Zhao, Enyu Liu, Kebin Sun, and Ran Cheng. Tensorized neuroevolution of augmenting topologies for gpu acceleration. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1156–1164, 2024.

[101] Lishuang Wang, Mengfei Zhao, Enyu Liu, Kebin Sun, and Ran Cheng. Tensorneat: a gpu-accelerated library for neuroevolution of augmenting topologies. *ACM Transactions on Evolutionary Learning*, 6(2):1–31, 2026.

[102] Bernard L Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[103] Adrian Westh. PUREPLES: Pure Python library for ES-HyperNEAT. `https://github.com/ukuleleplayer/pureples`, 2017.

[104] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[105] Xin Yao. A review of evolutionary artificial neural networks. *International journal of intelligent systems*, 8(4):539–567, 1993.

[106] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[107] G Zames. Genetic algorithms in search, optimization and machine learning. *Inf Tech J*, 3(1):301, 1981.

[108] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix A

Research Evolution and Philosophical Framework

> They're blind to a simple truth:
> complex minds can't develop on their own.
>
> Ted Chiang

This appendix provides the detailed phase-by-phase research trajectory and a philosophical framework for human–AI relationships. The condensed summary appears in Section 7.2; this appendix preserves the full narrative for readers interested in the thesis's evolution.

A.1 Biological Foundations of the Original Proposal

The biological inspiration for GEENNS's columnar design draws on three converging lines of evidence, summarized in Chapter 7 and expanded here for readers interested in the neuroscience context.

The observation by Mountcastle [69] that the mammalian neocortex is organized into columns of approximately 0.5 mm diameter, each sharing the same canonical microcircuit, provides the central architectural analogy. Each cortical column processes information through approximately six layers, with the same basic circuit replicated across functionally distinct cortical areas (visual, auditory, somatosensory). Functional diversity arises not from structural diversity but from differences in afferent connectivity: which sensory inputs reach which columns.

The Thousand Brains Theory of Hawkins [36] extends this further, proposing that each column independently models its sensory input and the cortex reaches consensus by integrating across thousands of such models. Under this view, perception is not a feedforward pipeline but a massively parallel voting system where each column makes an independent prediction and the population converges on a consensus. GEENNS's multi-grid architecture with mapper-based composition echoes this voting structure, though at a far simpler scale.

Marsupial cortex studies confirm that columnar organization arises independently across mammalian lineages separated by over 160 million years of evolution [48], suggesting the pattern reflects a fundamental organizational advantage (perhaps related to

wiring efficiency or developmental simplicity) rather than phylogenetic accident. This convergent evolution argument strengthens the case that the columnar motif captures a genuinely useful computational principle, not merely a biological contingency.

These biological observations motivated GEENNS's design principle of *architectural homogeneity with functional diversity through connectivity*: all grids share the same evolved internal template, and different evolved input routing patterns produce different functional behaviors. The critical distinction, emphasized throughout the thesis, is that GEENNS is inspired by, not constrained by, biological organization. This contrasts with approaches that aim to imitate or simulate brain architecture and information processing directly: a field in its own right that includes Hawkins's Thousand Brains program and the Blue Brain Project [64]. GEENNS takes the evolutionary computation route instead: rather than transferring an existing biological architecture onto target hardware, it lets evolution adapt the architecture to the available substrate, the way biological evolution shaped neocortical organization around the constraints of mammalian physiology. GEENNS does not prescribe six layers; the layer count is left for evolution to search over against the target hardware.

A.2 Research Evolution

Research rarely follows its proposal. This section traces the actual trajectory during the thesis lifecycle, recording the pivots, failures, and unexpected discoveries that shaped the thesis. This explains why a thesis about "compositional intelligence" contains chapters about quadtree analysis and GPU acceleration.

A.2.1 Grounded Symbolic Reasoning

Early work aimed to move beyond data-driven pattern recognition by grounding linguistic tokens in multimodal referents (images, spatial constraints, textures, and relational structures). Two guiding questions framed the effort: how can language representations integrate sensorimotor context to support higher-order reasoning, and how can agents autonomously generate and update their own environmental representations?

We evaluated four architectural approaches for achieving this:

1. A *centralized knowledge base*: a monolithic repository of structured facts and rules.
2. An *enormous all-knowing model*: a single vast parametric system.
3. A *controller–satellite arrangement*: a central controller orchestrating specialized sub-models.
4. A *distributed peer-to-peer network*: specialized agents collaborating through task delegation.

Each approach had merits. The centralized approach was interpretable but inflexible. The monolithic model scaled with data but could not adapt its structure. The controller–satellite arrangement offered modularity but created a single point of failure. The distributed approach promised continuous specialization and structural adaptability but presented coordination challenges that were, at the time, beyond reach.

The conclusion from this period was negative: *static architectures cannot capture dynamic reasoning*. Every approach assumed a fixed computational graph. But the problems we cared about (complex problem-solving, continuous learning, novel-concept acquisition) required architectures that could restructure themselves. This realization did not immediately point to neuroevolution, but it eliminated several years' worth of potential dead ends by ruling out fixed-topology approaches.

A.2.2 The Pivot to Neuroevolution

The research took a decisive turn. Two developments converged.

First, we investigated *fuzzy logic* as a mechanism for handling degrees of truth rather than binary outcomes, a natural fit for the partial, ambiguous character of human reasoning. Neuro-fuzzy systems combined the universal approximation properties of neural networks [41] with the interpretability of fuzzy-rule systems. The approach was promising in low dimensions. It became impractical in high dimensions: the number of fuzzy rules expanded combinatorially, creating both computational and interpretability bottlenecks. This was the curse of dimensionality, encountered not in a textbook but in the implementation.

Second, we deepened the study of neocortical organization, exploring how neuromodulation, neurotransmitter dynamics, and columnar architecture give rise to flexible cognition. The literature on cortical columns [48, 69] suggested a solution to the modularity problem that had blocked the earlier architectural explorations: a hardware-agnostic, self-organizing substrate that bypasses rigid design constraints.

These threads (fuzzy logic's failure and columnar self-organization) converged on *neuroevolution*. If evolution can discover neural topologies, then the architecture itself becomes part of the solution space rather than a constraint imposed by the designer. Initial experiments with NEAT [88] confirmed that evolutionary search could discover creative, adaptive network configurations. The same experiments highlighted NEAT's scaling limitations in high-dimensional domains, pointing toward indirect encodings as the necessary next step.

The research direction converged: we would use indirect-encoding neuroevolution (HyperNEAT and its extensions) as the foundation for evolving scalable, adaptive substrates. The original proposal's broader vision was set aside until the substrate itself could be made to work.

A.2.3 Doc.Mobility and the 48-Pixel Discovery

A Doc.Mobility scholarship enabled a six-month research visit at University College Dublin under the mentorship of Prof. Michael O'Neill, a leading figure in evolutionary computation [70]. The experimental work that would become Paper 1 [11] was conducted during this visit: a 2,013-trial Tree-structured Parzen Estimator (TPE) optimization sweep of ES-HyperNEAT on MNIST, achieving 29% accuracy, a new state-of-the-art for ES-HyperNEAT on this benchmark. Two unexpected observations from these experiments proved more consequential than the headline result, and a third gain from the visit was methodological rather than experimental.

The 48-pixel discovery. The evolved substrates wired to a small central subset of the input, the campaign's champion network to approximately 48 of 784 pixels, barely 6% of the input space (Chapter 4 measures the same collapse two ways: that champion's ∼48-

pixel per-trial wiring density, against a 30-run replication whose final-generation counts have mean ∼55, median ∼29, and range 1–465; and 28 unique pixel positions, or 3.6% spatial coverage, as the composite extent at $N_e = 1$). The algorithm was not failing to solve MNIST; it was solving a drastically reduced version using whichever few pixels happened to fall within the central region where the quadtree placed most of its resolution. This observation would later motivate the partitioned-experts spatial decomposition work (Chapter 4), which forces evolution to use the full input space.

The CPU bottleneck. ES-HyperNEAT's quadtree-based substrate discovery is inherently sequential: it subdivides the spatial domain one cell at a time, and each subdivision decision depends on the result of the previous one. This sequential dependency prevents GPU parallelization. On commodity hardware, each evolutionary generation took minutes to hours depending on problem size. Running the full 2,013-trial sweep required months of continuous computation. This bottleneck was not merely inconvenient; it made GEENNS computationally infeasible, since GEENNS requires evolving dozens of CPPNs simultaneously.

Methodological refinement. Working with an established evolutionary computation group provided direct exposure to experimental protocols, statistical validation practices, and the pragmatic compromises required when computation time is measured in weeks per experiment. The mentorship arrived late in the doctoral timeline, but its impact on methodological rigor was substantial.

A.2.4 New Research Directions

Paper 1 [11] consolidated the Dublin experiments and established a new ES-HyperNEAT state-of-the-art on MNIST (29%, surpassing the previous 23.9%). The Paper 1 findings had two quantitative consequences that shaped the remainder of the thesis.

First, the quadtree bottleneck was now quantified. Extrapolating to GEENNS, where a modest 5-grid, 10-task configuration requires $4 + k(k-1)/2 + 2T = 34$ CPPNs (Appendix B), the required computation was approximately 68,000 hyperparameter trials at 2,013 trials per CPPN: roughly 7 years of continuous CPU computation at MNIST-scale per-trial cost. The quadtree was not just slow; it made GEENNS infeasible within any practical timeline.

Second, the transferability analysis revealed that optimized hyperparameters transferred across *similar* tasks (MNIST to Fashion-MNIST) but not across *dissimilar* tasks (MNIST to XOR). A single optimization sweep would not suffice for a multi-task system; each qualitatively different task type would require its own optimization.

Two new research directions followed. The 48-pixel observation from §A.2.3 led to *decomposition experiments*: spatially partitioning the input space among multiple specialist models, forcing each to cover a distinct region rather than allowing the quadtree's central bias to dominate. This became the partitioned-experts work in Chapter 4. The optimization cost (2,013 trials per CPPN) led to *early-stopping experiments*: developing data-driven pruning thresholds that could terminate unpromising trials early, reducing the hyperparameter search cost (Chapter 3).

In parallel, early profiling confirmed the quadtree bottleneck was fundamental, not merely a scaling issue: the sequential subdivision dependency was algorithmic, not implementational. We began work on a JAX-vectorized quadtree as a first attempt to break this bottleneck.

These findings set the stage for Part II of this thesis.

A.2.5 The EMR Year

With the experimental foundations laid (Paper 1), the priority was solving the quadtree bottleneck that made GEENNS computationally infeasible.

The first attempt was a JAX-vectorized quadtree. We implemented a batch-parallel version that evaluated multiple quadtree cells simultaneously using JAX's `vmap` and `jit` primitives. Testing revealed the fundamental problem: the quadtree's sequential subdivision dependency (where each cell's decision to subdivide depends on the variance of the previous level's CPPN evaluation) could not be eliminated by vectorization. Batching *within* a level was possible, but the level-to-level dependency chain remained sequential. The bottleneck was algorithmic, not implementational.

This negative result forced a more radical approach. Rather than accelerating the existing algorithm, we reformulated ES-HyperNEAT entirely. The solution was a lazy-to-eager transformation: where ES-HyperNEAT asks "*should* I subdivide this cell?" (requiring sequential evaluation of each cell), the new formulation asks "which of *all possible* cells exceed the variance threshold?" (requiring a single parallel evaluation of the entire resolution grid, followed by a filter operation). This became EMR-HyperNEAT, and the reformulation had an unanticipated benefit: the new matrix computation framework also enables per-node substrate features (activation functions, aggregation functions) at negligible additional cost, a direction explored beyond this thesis. With ES-HyperNEAT, evolving per-node functions would have required additional sequential quadtree queries per function, which was infeasible. With EMR, they become additional output dimensions in the same batch CPPN evaluation. What follows summarizes EMR-HyperNEAT's quantitative impact and three of the research directions its matrix framework opens.

EMR-HyperNEAT. The eager reformulation achieves 12–34× per-generation GPU speedup on the XOR benchmark at resolution depths 5–7 (population 1000; the 34× endpoint and the ∼33× cumulative rest on the depth-7 streaming estimate), yielding a cumulative ratio of ∼33× over a 30-generation XOR run at depth 7 (asymptotic limit near ∼34× as JIT overhead amortizes). The magnitude varies with task, depth, and population size, but the structural advantage (replacing sequential quadtree traversal with batch matrix operations) consistently outperforms ES-HyperNEAT at scale (Chapter 6).

Research directions opened by EMR. The eager reformulation opens a broad set of research programs that would have been computationally prohibitive under ES-HyperNEAT. We sketch three here that companion work has pursued. First, *per-node function evolution*: because CPPN evaluation is now a batch matrix operation, adding per-node activation and aggregation function assignment as additional CPPN output dimensions costs almost nothing. Second, *bio-inspired palette meta-learning*: strategies for managing which functions are available to evolution across generations, inspired by biological mechanisms such as clonal selection and neuromodulation. Third, *neuromodulation for multi-task operation*: gain modulation that could enable a single substrate topology to serve multiple tasks through task-specific neurotransmitter vectors. Each direction would require hundreds to thousands of evolutionary runs at depths and population sizes that would have taken years under ES-HyperNEAT's sequential quadtree. The broader set of directions is summarized in §8.4; none is a thesis contribution. The computational barrier was not just blocking GEENNS; it was blocking an entire research program.

A.3 Connection to the Original Proposal

This doctoral research began under a different title and a broader scope. The original proposal, *"Toward Human-like Intelligence for Complex-Problem Solving: Proposing a Modular Framework for Machine Learning to Reach Partial Human-like Intelligence Based on Neocortex and Human Psychology Theories,"* aimed to bridge computer science, psychology, and neuroscience toward algorithmically replicating human cognitive abilities.

The proposal drew on three intellectual streams. The first was neuroscience, specifically the neocortical column hypothesis [69] and the observation that a single computational template, replicated and wired differently across the cortex, produces the full diversity of cortical function. If this principle could be transplanted into an evolutionary algorithm, then a compact genome might generate large-scale, functionally diverse substrates. The second was human psychology, particularly research on complex problem-solving: when humans face problems they cannot solve alone, they decompose them, delegate sub-problems to specialists, and compose the results. The third was cognitive architecture, including dual-process theory [47] and the idea that layered, specialized processes are necessary for flexible intelligence.

The proposed framework was deliberately ambitious, spanning cognitive and collaborative capabilities well beyond the substrate itself. The research that followed narrowed this scope substantially to focus on the substrate problem.

What we did was confront the *substrate problem* directly. Any modular framework for compositional reasoning requires substrates that can be evolved efficiently, that can represent high-dimensional inputs without collapse, and that can be constructed fast enough for systematic scientific investigation. The experimental chapters of this thesis each address one of these requirements.

Table A.1 maps the original proposal modules to their realization in the thesis.

Table A.1: Mapping between the original proposal modules and their realization in this thesis. The modules listed here were addressed in reframed form; the remaining modules were beyond the scope adopted by the thesis.

Proposed Module	**Thesis Realization**	**Status**
Collective intelligence	Partitioned-experts spatial decomposition (Ch 4)	Reframed
Modular framework	Substrate-level foundation (Ch 3–6)	Reframed
Literature review	Background chapter (Ch 2)	Addressed

Each phase traced in §A.2 narrowed the proposal's scope while deepening the contribution.

The framing is this: the substrate problem had to be solved first. Before any higher-level cognitive architecture can compose specialized modules, those modules must be computationally tractable and representationally adequate. The broader vision (compositional reasoning through evolved substrates, multi-task operation, biologically inspired function discovery) remains important future work (Chapter 7). But it cannot be meaningfully pursued until the substrate itself works. This thesis makes the substrate work.

The original proposal's tasks were ordered as a flat list; the thesis research revealed that they form a directed acyclic graph (DAG) of dependencies, where some tasks are prerequisites for others. Addressing three barriers deeply (hyperparameter optimization,

spatial decomposition, GPU acceleration) plus validating one more (compositional proof-of-concept) created the prerequisites that make the remaining tasks feasible. The thesis delivers fewer tasks than proposed, but it delivers the tasks on the critical path.

A.3.1 Reflecting on the Journey

The proposal envisioned a broader cognitive-scale framework; the thesis delivers a substrate-level engineering investigation: how to make evolved substrates fast, expressive, and multi-task capable.

This narrowing was not planned. It was forced by repeated encounters with problems that had to be solved before the vision could be pursued. We could not pursue the broader GEENNS vision on substrates that took months to evolve. We could not explore compositional reasoning through grids whose activation functions made certain computations impossible. We could not demonstrate lifelong learning through substrates with no mechanism for task-specific behavior.

The result is that the full GEENNS system (multi-grid architecture, plasticity mechanisms, deep compositional reasoning) remains unimplemented. What exists is a substrate-level engineering foundation (EMR-HyperNEAT for computation, spatial decomposition for input coverage) and a proof-of-concept prototype that validates the compositional architecture at 5-grid Boolean scale (Section 7.6). The expressiveness and multi-task directions opened by EMR (§A.2.5) remain future work. Whether this constitutes a failure to deliver on the original proposal or the necessary scientific maturation of an overly ambitious vision is a question we address in Chapter 8.

Three structural caveats on the realized contributions. The partitioned-experts architecture of Chapter 4, inspired by Mixture of Experts (MoE), uses fixed, human-designed partitions, not evolved ones. EMR-HyperNEAT (Chapter 6) trades computation for memory: eager evaluation at depth 7 generates 87,380 candidate positions regardless of whether they are needed, and current multi-GPU implementations show slowdown instead of speedup. The experimental validation spans MNIST (Chapters 3–4) and 15 problems across four domains (Boolean, visual and spatial, reinforcement-learning control, and regression; Chapter 6, with CCM Financial as the only real-world dataset); industrial-scale problems and high-dimensional sensory inputs were not evaluated. Chapter 8 discusses each in full. The barriers diagnosed are structural; the solutions developed here are the first thesis-scale response to all three, but they do not exhaust the design space those barriers admit.

A.4 Historical Note: Six Stages in the Evolution of Collaboration

The original doctoral proposal included a three-stage philosophical speculation about the evolving relationship between humans and artificial intelligence. Continued research extended it to six stages spanning the broader evolution of collaboration, from purely organic origins, through cross-boundary interaction between humans and AI, to purely artificial creation. We present this framework explicitly as speculation, a motivating lens for positioning the research, not a scientific claim. It is, however, an open question worth exploring in future work (or simply worth observing as a curiosity in light of AI's current

exponential scaling), given that human minds reason linearly and grasping exponential dynamics is a hard exercise.

The six stages trace a symmetric structure:

Stage 1: Humans make humans.
Biological evolution and reproduction. Intelligence emerges through natural selection operating on genetic variation; no artificial system is involved.

Stage 2: Humans collaborate with humans.
Cooperation, language, culture, and collective problem-solving among organic agents. Civilization-scale progress accumulates through shared tools and inherited knowledge, establishing peer collaboration as the default mode long before any artificial system exists.

Stage 3: Humans use AI as tools.
The current state of AI practice. Humans design architectures (CNNs, Transformers, LLMs), define training objectives, and train models to execute specific tasks like image classification, language modeling, and code generation. The human remains the architect; the machine is an instrument.

Stage 4: Humans collaborate with AI as peers.
The research ambition pursued in this thesis. This requires AI systems with emergent compositional reasoning: the capacity to decompose problems, discover solutions, and compose novel behaviors without explicit programming. The GEENNS vision targets this stage.

Stage 5: AI collaborates with AI.
Autonomous coordination among artificial agents without human involvement in the collaboration itself. Early snapshots exist in LLM-based autonomous-agent demonstrations, though humans remain involved in setup, oversight, and intervention.

Stage 6: AI makes AI.
Artificial agents that design, evolve, or construct other artificial agents—a new category we term *Artificialkind*, by analogy with humankind.

The structure is symmetric: Stage 1 (organic creation) mirrors Stage 6 (artificial creation), Stage 2 (same-kind organic collaboration) mirrors Stage 5 (same-kind artificial collaboration), and Stages 3–4 trace the cross-boundary transition from human–AI tool-use to human–AI peer collaboration. We include this framework only to record the motivating lens, not to claim scientific significance.

This thesis sits firmly at the Stage 3→4 boundary. The substrate engine built in Chapters 3–6 could support evolved systems capable of emergent behavior: infrastructure for moving beyond tool-use toward systems whose behaviors arise from evolutionary dynamics rather than explicit programming. The gap between composing five Boolean gates and genuine human–AI peer collaboration is vast. The prototype validates modular composition; it does not demonstrate emergent intelligence. The six stages are a compass indicating a direction, not a roadmap with measured distances. The concrete contributions of this thesis remain the engineering advances validated by benchmarks, not progress along a speculative philosophical trajectory. Whether those later stages are approachable empirically, and what a research program aimed at them would look like, is a question this thesis does not answer but that its infrastructure makes possible to investigate.

Appendix B

GEENNS Formal Specification

> Should philosophy guide experiments, or should experiments guide philosophy?
>
> Liu Cixin

This appendix contains the detailed formal specification of the GEENNS algorithm, including the multi-level Compositional Pattern-Producing Network (CPPN) hierarchy, the developmental pipeline, plasticity mechanisms, the continual learning comparison, detailed architecture figures, and mathematical formalization. All content is a *design specification*; no implementation or validation exists beyond the prototype evidence in Section 7.6. The condensed architectural overview appears in Chapter 7.

Implementation Status. This appendix specifies the GEENNS algorithm as designed but *not yet fully implemented*. Only the prototype pipeline (Chapter 7, approximately 3,100 runs spanning Boolean compounds up to 6-grid and continuous additive compounds through 8 gates) has been validated, and only using a simplified augmented-input mapper over a two-layer EMR-HyperNEAT substrate. The three-type mapper formalism of Definition B.2 (with explicit inter-grid routing), the plasticity mechanisms, and the multi-level CPPN hierarchy (Section B.4) all remain theoretical. Per-node activation and aggregation function assignments described throughout this appendix are design targets for future work, not implemented thesis contributions; the feasibility of per-node function evolution is explored in companion work beyond this thesis (§8.4). This specification is a design document for future implementation, not a description of working software.

B.1 Substrate Evolution Landscape and CoDeepNEAT Comparison

See Section 7.4.3 for the condensed overview and Section 7.6 for the experimental validation.

GEENNS grids are spatially organized substrates with columnar structure, CPPN-generated connectivity, and per-node function diversity. The substrate evolution method must (a) scale to large networks while preserving geometric regularities, (b) support per-node features at low additional cost, and (c) produce substrates that can be frozen and

reused across tasks. Table B.1 compares the most relevant alternatives to explain why EMR-HyperNEAT is the necessary foundation.

DeepNEAT and CoDeepNEAT. Miikkulainen et al. [67] introduced DeepNEAT, which evolves layer-level topology, and CoDeepNEAT, which co-evolves two populations: *modules* (small network subcomponents) and *blueprints* (directed graphs specifying how modules connect).

CoDeepNEAT as closest conceptual prior art. CoDeepNEAT's blueprint/module split is the closest prior art to GEENNS's mapper/grid split: both separate *what computes* from *how components are assembled*. Two differences explain why CoDeepNEAT cannot serve as the substrate engine for GEENNS:

1. **Direct encoding cannot produce spatial substrates compactly.** CoDeepNEAT modules are small network graphs where each connection is an individual gene. This means modules cannot exploit geometric regularities (symmetry, periodicity, spatial locality) that CPPNs provide through indirect encoding. GEENNS grids are spatially organized substrates where node placement, connectivity patterns, and per-node functions are all generated from spatial coordinates. A compact CPPN genome produces a large grid with systematic internal structure (repeated motifs, symmetric connectivity, gradient-based density variation) that would require a correspondingly large direct-encoding genome to specify explicitly. For compositional architectures built on columnar spatial organization, direct encoding would require the genome to grow with the substrate.

2. **Co-evolving modules drift.** CoDeepNEAT modules are not frozen; they co-evolve alongside blueprints, so module capabilities shift continuously as blueprints change. A blueprint that learned to use Module Species A for feature extraction may find that A has drifted toward a different function by the next generation. GEENNS's freeze-then-route design eliminates this drift: grids are trained, frozen, and then mappers evolve against stable specialists.

Why EMR-HyperNEAT specifically. ES-HyperNEAT already provides indirect encoding with adaptive substrate discovery: the right foundation. But its sequential quadtree makes it computationally infeasible for GEENNS (Section 7.5.1). EMR-HyperNEAT solves this with the lazy-to-eager reformulation described in Section 7.2 (detailed in Appendix A.2.5), but also provides a second capability: because CPPN evaluation is now a batch matrix operation, adding per-node output dimensions (activation function index, aggregation function index, receptor densities) costs almost nothing. ES-HyperNEAT would require separate sequential quadtree passes for each additional feature, making per-node function evolution infeasible. EMR-HyperNEAT makes it a free byproduct. GEENNS grids need per-node function diversity (the expressiveness barrier, Section 7.5.2), and EMR-HyperNEAT is currently the only substrate evolution method that provides this at practical cost.

Table B.1: Comparison of substrate/network evolution approaches along dimensions relevant to GEENNS requirements. "Spatial" indicates whether the encoding preserves geometric regularities. "Per-node" indicates whether the method supports evolving per-node activation or aggregation functions at practical cost (a future direction enabled by EMR-HyperNEAT; see §8.4).

Algorithm	**Encoding**	**What It Scales**	**Spatial**	**Per-Node**	**GEENNS Suitability**
NEAT	Direct	Topology (∼1K nodes)	No	No	Foundation only; no substrates
HyperNEAT	Indirect (CPPN)	Substrate size	Yes	No	Spatial grids yes; fixed resolution
ES-HyperNEAT	Indirect (CPPN)	Adaptive density	Yes	No	Right approach; quadtree bottleneck
DeepNEAT	Direct (layers)	Network depth	No	No	Wrong abstraction for grids
CoDeepNEAT	Direct (blueprint + module)	Modular networks	No	No	Closest concept; wrong encoding
EMR-HyperNEAT	**Indirect (CPPN)**	**Tensor-vectorizable substrate**	**Yes**	**Yes**	**Built for GEENNS**

Co-evolution as future work. CoDeepNEAT's blueprint/module co-evolution raises a natural question for GEENNS's future: should mappers and grids co-evolve rather than following the current freeze-then-route pipeline? Co-evolving external parameters alongside NEAT populations would face a population alignment problem: NEAT's selection mechanism reorders population indices between generations, breaking the mapping between genomes and their co-evolved parameters without any explicit error signal. GEENNS's freeze-then-route approach sidesteps this entirely by decoupling the two evolutionary processes. A hybrid approach (co-evolving mappers with grids under constrained plasticity) is worth exploring in future work, but will require solving the alignment problem more reliably than current methods allow. The approach would likely differ from CoDeepNEAT's continuous co-evolution; a staged pipeline (evolve grids → partially freeze → co-refine mappers and grid plasticity) may be more tractable.

B.2 Why Evolution Determines Architecture

Conventional architectures (CNNs, Transformers) are hand-designed for specific hardware families and problem domains. GEENNS's substrate must be jointly optimized for hardware constraints (memory bandwidth, parallelism granularity, communication topology) *and* the target task distribution (required computational primitives, routing patterns). The joint design space (column count, layer count, node density, connectivity patterns at three spatial scales, and per-node activation and aggregation functions explored in companion work, not a thesis contribution) is combinatorially large and hardware-dependent. Hand-crafting a substrate for every hardware–task combination is intractable (Figure B.1).

Hornik et al. [41] showed that a feedforward network with a single hidden layer of sufficient width and a squashing activation function can approximate any continuous function on compact subsets of $\mathbb{R}^n$ (see also Cybenko [22] for the sigmoidal case). The theorem guarantees *existence* of a solution but not *discovery*: in fixed architectures, gradient descent adjusts weights but cannot alter topology. NEAT and its successors evolve topology alongside weights, searching the space of network structures. GEENNS extends this to hierarchical structures: evolution searches the space of columnar architectures for a substrate suited to both the target task distribution and the available hardware.

What the result does not guarantee matters more here: that evolution finds such a network, that the required network size is tractable, or that evolution outperforms gradient-based neural architecture search. Whether evolutionary search navigates this space for compositional substrates is an empirical question addressed throughout this thesis.

B.3 Architectural Detail: From Network to Node

This section decomposes the GEENNS architecture top-down, from the full multi-grid network to the per-node computation that unfolds at each propagation step. Figure B.2 locates the mappers relative to the shared columnar network; Figure B.3 zooms into a single column to expose the four CPPN connectivity levels; Figure B.4 traces a concrete inject–propagate–extract cycle; and Figure B.5 resolves the finest level of structural detail. The bold-headed paragraphs below walk that decomposition in the same order.

Full architecture and mapper placement. The outer frame of GEENNS is a columnar network flanked by task-specific mappers. Input mappers decompose a task signal into injection patterns for each grid; output mappers compose the grids' responses back into a task result. The columnar network itself is shared: all tasks route through the same underlying substrate, and only the mapper populations differ per task.

Intra-grid structure and CPPN hierarchy. Each grid in the architecture is itself a nested structure. Columns contain minicolumns, and each minicolumn is a stack of DES-HyperNEAT substrate layers. Four CPPNs govern connectivity at progressively finer scales, from the outermost between-column wiring down to the per-layer substrate. Figure B.3 makes the hierarchy explicit by expanding a single column.

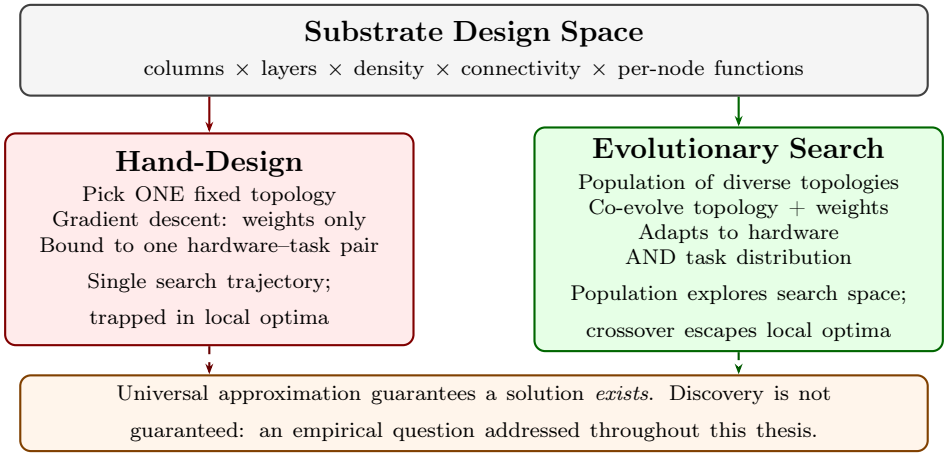


Figure B.1: Hand-designed versus evolutionary architecture search. The substrate design space spans column count, layer count, node density, connectivity patterns at three spatial scales, and per-node functions. Hand-design selects a single fixed topology; evolutionary search co-evolves topology and weights simultaneously. Universal approximation guarantees a solution exists but not that any search process finds it.

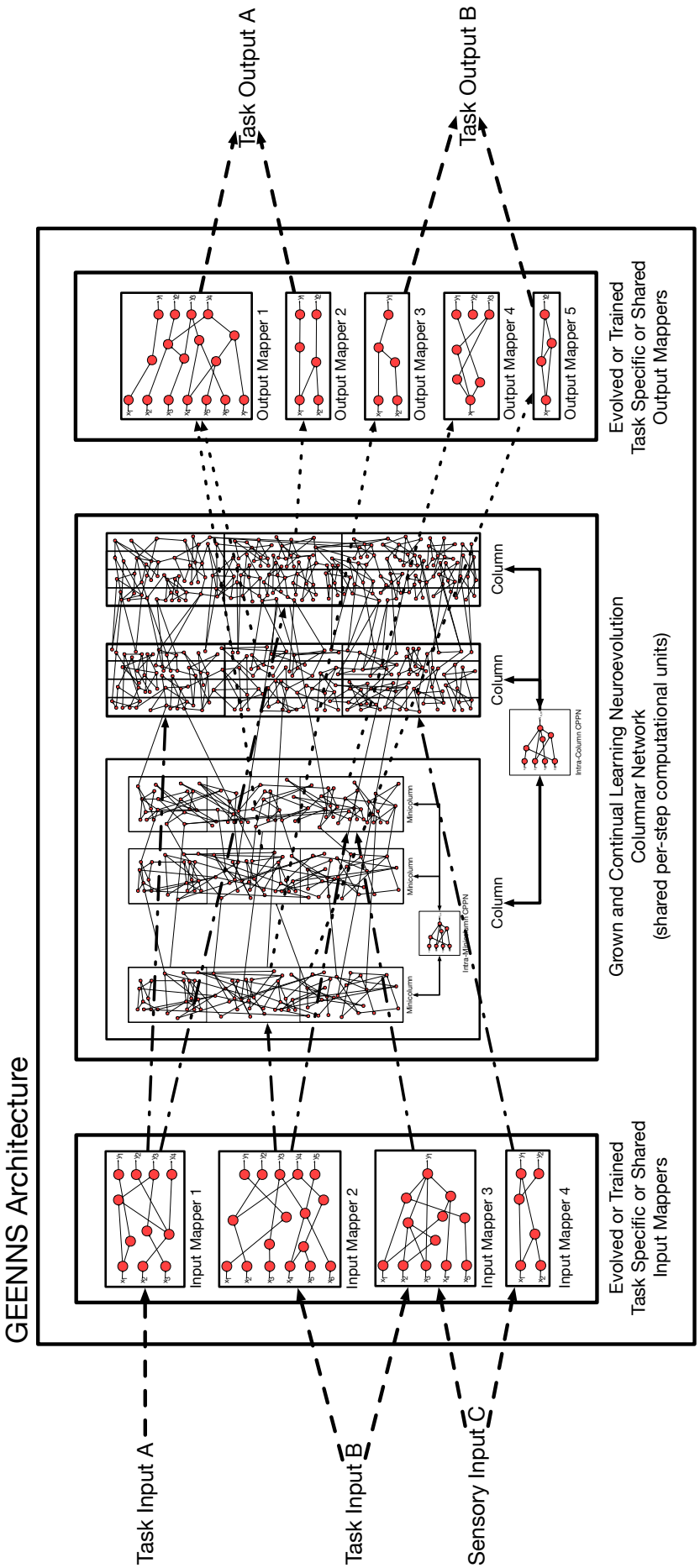


Figure B.2: Full GEENNS architecture. Input mappers (left) receive task and sensory signals and route them into the columnar network (center), which consists of three columns each containing multiple minicolumns with evolved internal connectivity. An Intra-Minicolumn CPPN governs within-column wiring; an Intra-Column CPPN governs between-column wiring. Output mappers (right) compose column responses into task-specific results. The columnar network is shared across all tasks; only mapper configurations change per task.

Figure B.3: GEENNS intra-column architecture. Three columns (outer rectangles) each contain multiple minicolumns, linked by an Intra-Column CPPN. The first column is expanded to show its internal structure: each minicolumn contains multiple layers, where each layer is a DES-HyperNEAT substrate with its own CPPN. Four levels of CPPNs govern connectivity at different spatial scales; the Minicolumn CPPN drawn alongside them is the canonical-microcircuit template shared across a column's minicolumns, not a fifth connectivity level.

Grid computation in action. The structural picture becomes concrete when we trace a single forward pass. Consider a small grid of 3 columns × 2 layers × 4 nodes (24 nodes in total). Injection happens at $t = 0$: the input mapper writes a signal vector into the grid's designated input nodes (typically the top-layer nodes of selected columns). At each propagation step $t \to t{+}1$, every node aggregates its weighted inputs through its evolved aggregation function (one of `sum`, `min`, `max`, `mean`), adds its bias, and applies its evolved activation function (drawn from `sin`, `tanh`, `sigmoid`, and related families). When neuromodulation is active, the aggregated input is scaled by the gain factor $\alpha_i(\mathbf{n}_\tau) = \mathbf{r}_i \cdot \mathbf{n}_\tau$ before the bias is added and the activation function applied: the dot product of node v_i's receptor density $\mathbf{r}_i$ with the task's neurotransmitter vector $\mathbf{n}_\tau$. After T_{prop} propagation steps (typically 2–5 for Boolean tasks), the output mapper reads designated output nodes and produces the task result. Figure B.4 shows the complete cycle.

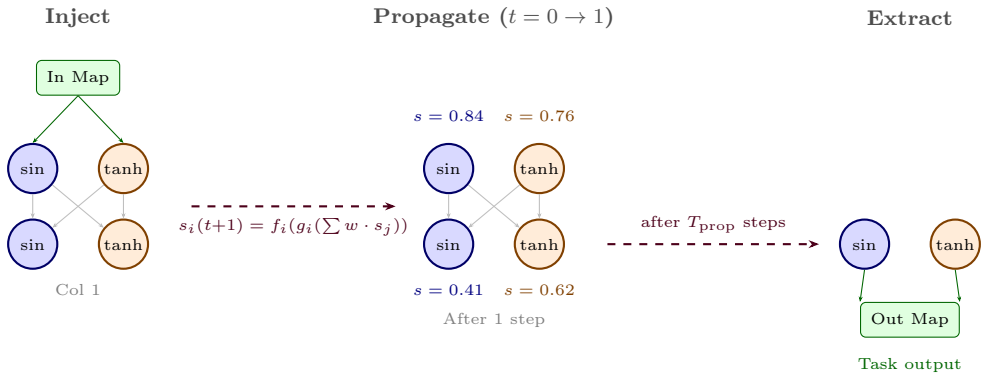


Figure B.4: Grid computation walkthrough. Left: the input mapper injects signals into designated input nodes (top layer). Center: after one propagation step, each node aggregates its weighted inputs and applies its per-node activation function (sin in blue, tanh in orange), producing updated state values. Right: after T_{prop} propagation steps, the output mapper reads designated output nodes. Under neuromodulation, each node's aggregated input is scaled by a task-specific gain factor before its activation function is applied. Per-node function selection is explored further in companion work beyond this thesis.

Minicolumn-level detail. At the finest granularity, a minicolumn resolves into a vertical stack of DES-HyperNEAT substrate layers. Each layer carries its own CPPN for within-layer connectivity; Intra-Layer CPPNs link adjacent layers; and a Minicolumn CPPN defines the structural template shared across all minicolumns of a column. This is where the canonical microcircuit template Γ_{micro} takes physical form in the grid of Definition B.1.

B.4 Multi-Level CPPN Connectivity

GEENNS requires connectivity at multiple spatial scales, each generated by a dedicated set of CPPNs. Table B.2 specifies the complete CPPN hierarchy.

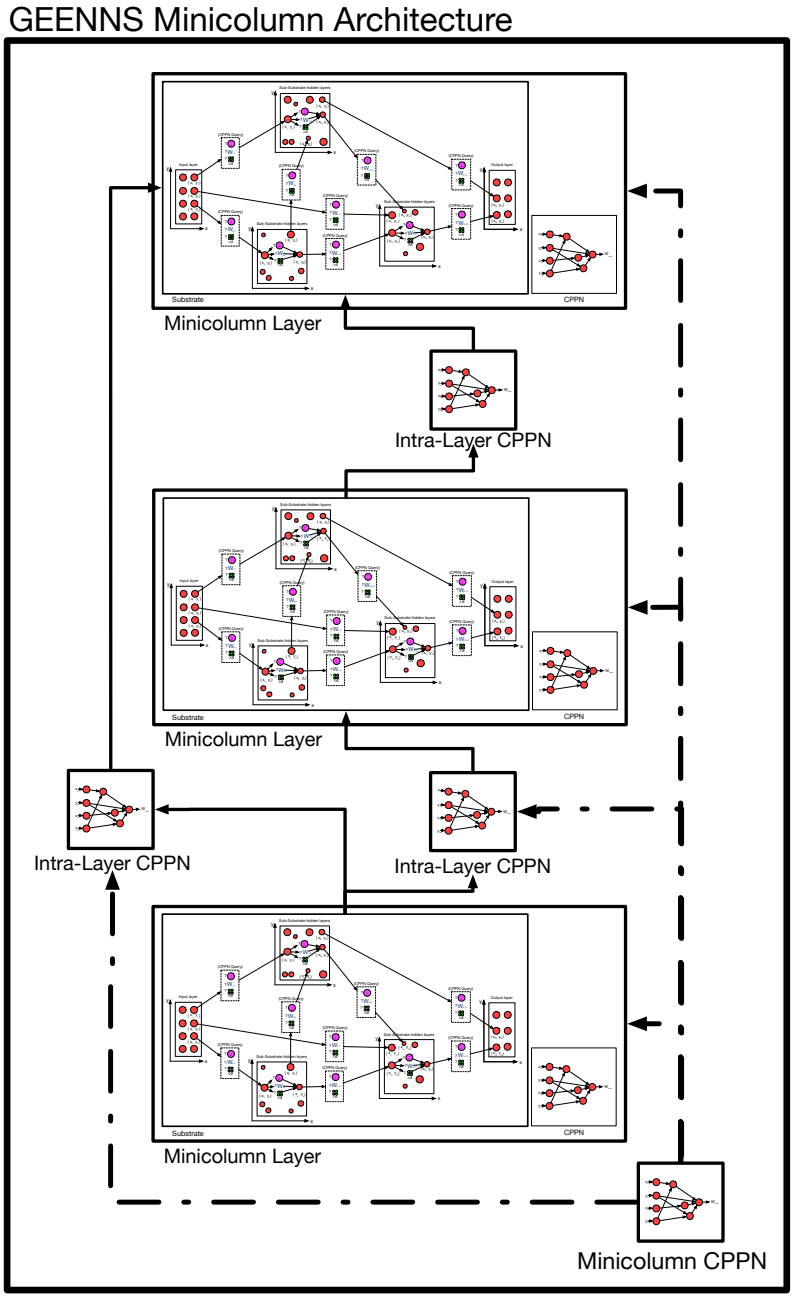


Figure B.5: GEENNS minicolumn architecture. Three minicolumn layers are stacked vertically, each containing a full DES-HyperNEAT substrate (with its own CPPN for intra-layer connectivity). Intra-Layer CPPNs generate the connectivity patterns linking adjacent layers. A Minicolumn CPPN governs the overall structural template shared across all minicolumns within a column.

Table B.2: CPPN hierarchy for GEENNS connectivity. "Shared across" indicates which structural elements use the same CPPN; "Scope" indicates what the CPPN connects. For k grids and T tasks, the total CPPN count is $4 + k(k-1)/2 + 2T$.

CPPN type	**Shared across**	**Scope**	**Input**	**Output**
Per-layer	All layers of same type	Within DES-HyperNEAT layer	4D coords	weight, prob
Intra-layer	All minicolumns	Between layers in minicolumn	4D coords	weight, prob
Intra-minicolumn	All columns	Between minicolumns	4D coords	weight, prob
Intra-column	All grids	Between columns	4D coords	weight, prob
Inter-grid	Per grid pair	Between grids ($k(k-1)/2$)	4D coords	weight, prob
Input mapper	Per task	Task input $\rightarrow$ grid inputs	varies	varies
Output mapper	Per task	Grid outputs $\rightarrow$ task output	varies	varies

The multi-level CPPN architecture is the primary source of the computational bottleneck identified in Section 7.5. With 4 base CPPNs for intra-grid connectivity at different scales, $k(k-1)/2$ inter-grid CPPNs, and 2 CPPNs per task (input and output mappers), a system with k grids and T tasks requires $4 + k(k-1)/2 + 2T$ CPPNs, each requiring its own hyperparameter optimization. For a small 3-grid, 3-task configuration: $4+3+6 = 13$ CPPNs. For a full 10-grid, 20-task system: $4 + 45 + 40 = 89$ CPPNs. Part II of this thesis addresses the per-CPPN cost; Part III removes the computational bottleneck and introduces the substrate architecture that makes function-level expressiveness feasible as future work.

Feasibility envelope. The CPPN count translates directly into compute. Table B.3 extrapolates the wall-clock cost of a (k=5, T=10) sweep under XOR-scale per-trial assumptions; Section 8.3.1 expands the assumption stack.

Table B.3: Speculative wall-clock envelope (illustrative only) for a GEENNS hyperparameter sweep at a modest (k=5, T=10) scale: 34 CPPNs (formula above) at 2,013 TPE trials each (the budget inherited from the MNIST optimization of Chapter 3) for ∼68,000 configurations. Baseline extrapolates XOR-scale ES-HyperNEAT per-trial timings; the pruner savings (41.6%) are measured on MNIST (Chapter 3) and assumed to transfer; the GPU row applies the XOR-at-depth-7 EMR speedup. EMR speedups vary with task, depth, and population: 5.5× on financial regression (depth 6, pop 100), 12–34× per-generation on XOR (depths 5–7, pop 1000; the 34× endpoint and the ∼33× cumulative rest on the depth-7 streaming estimate), and a cumulative ratio of ∼33× over a 30-generation XOR run at depth 7 (with an asymptotic limit near ∼34× as JIT overhead amortizes; Chapter 6). The projections are favorable but speculative: the arithmetic stacks three assumptions (XOR-scale per-trial cost, 41.6% pruner transfer to GEENNS-scale problems, and the depth-7 EMR speedup applying uniformly across 34 CPPNs of varying purpose); each is defensible individually, but their composition is not validated end-to-end. Harder tasks increase per-trial cost super-linearly, so these figures are lower bounds on wall-clock, not predictions. §8.3.1 expands the caveats.

Configuration	**Est. Wall-Clock**	**Feasible?**
Baseline: quadtree, CPU	∼250 CPU-days	No
+ pruner (41.6% savings, MNIST-trained)	∼146 CPU-days	Marginal
+ EMR-HyperNEAT on GPU	∼4.9 GPU-days	**Yes**

B.5 The Developmental Process

GEENNS adopts the Phylogenesis–Ontogenesis–Epigenesis (POE) framework from evolvable hardware [85], decomposing the system's adaptation into three timescales:

Phylogenesis (evolutionary timescale). The genome, comprising NEAT-encoded microcircuit templates and CPPN-encoded connectivity patterns, evolves across generations. This is the mechanism by which GEENNS discovers general-purpose computational substrates. Phylogenesis operates at the population level: selection, crossover, and mutation produce new genome variants; fitness evaluation determines which variants survive.

Ontogenesis (developmental timescale). A single genome unfolds into a functional substrate through a five-stage developmental process (Figure B.6):

1. **Node generation.** Create $W \times H \times D$ nodes organized into W columns, each with H layers of D nodes, using the canonical microcircuit template Γ_{micro}.
2. **Layer organization.** Assign layer-specific properties (type, density) based on Γ_{micro}. Normalize node coordinates to the $[-1, 1]$ range for CPPN queries.
3. **Connection formation.** Query each CPPN in the hierarchy (Table B.2) at all relevant node-pair coordinates. Retain connections where the existence output satisfies $|p_{ij}| > \theta_{\text{conn}}$. Assign per-node activation and aggregation functions from CPPN outputs.
4. **Refinement.** Prune connections with $|w_{ij}| < \theta_{\text{prune}}$. Apply variance-based filtering to the resulting connectivity. Strengthen surviving connections through normalization.
5. **Maturation.** Initialize the receptor density matrix $\mathbf{R}$ for neuromodulation. Set initial plasticity parameters (learning rates, homeostatic targets). The grid is now operational.

This function is deterministic: the same genome always produces the same grid.

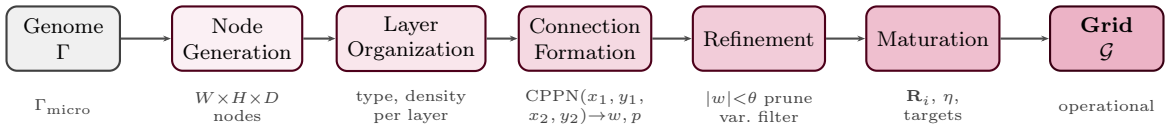


Figure B.6: Five-stage developmental pipeline. A genome Γ is transformed into a functional grid $\mathcal{G}$ through deterministic stages: node generation using the canonical microcircuit Γ_{micro}, layer organization assigning type and density, connection formation via CPPN queries over node-pair coordinates, refinement through pruning ($|w| < \theta$) and variance filtering, and maturation initializing the receptor density matrix $\mathbf{R}$ and plasticity parameters.

Epigenesis (operational timescale). During operation, plasticity mechanisms modify the substrate's behavior without altering its genome. GEENNS proposes four plasticity types, described in the following subsection.

B.6 Plasticity Mechanisms

GEENNS proposes four forms of plasticity operating during the epigenetic (operational) timescale. None has experimental support within this thesis; we record the status of each mechanism alongside its definition.

1. **Hebbian plasticity.** Correlation-based strengthening and weakening of connections [38] ("cells that fire together wire together"). Connection weight w_{ij} is updated according to $\Delta w_{ij} = \eta \cdot s_i \cdot s_j$, where η is the learning rate and s_i, s_j are the pre- and post-synaptic activities. *Status: theoretical. No implementation or experimental validation.*

2. **Homeostatic plasticity.** Regulatory mechanisms that maintain neural activity within functional ranges, preventing runaway excitation or silence. Each node adjusts its excitability to maintain a target firing rate. *Status: theoretical. No implementation.*

3. **Structural plasticity.** Long-term topology changes, including the formation of new connections and the elimination of unused ones. Over time, the substrate's connectivity adapts to the statistics of its inputs. *Status: theoretical. No implementation.*

4. **Neuromodulatory plasticity.** Global signals that modulate neuronal gain based on task identity. Each node's aggregated input is scaled by $\alpha_i(\mathbf{n}_\tau) = \mathbf{r}_i \cdot \mathbf{n}_\tau$ before its activation function is applied, where $\mathbf{r}_i$ is the evolved receptor density and $\mathbf{n}_\tau$ is the task's neurotransmitter vector. *Status: plausible future direction* (§8.4).

All four remain future directions. Whether Hebbian, homeostatic, or structural plasticity would provide additional value beyond neuromodulatory gating is an open question.

B.7 GEENNS among Continual Learning Approaches

Note: GEENNS has not been tested on continual learning tasks. This table positions the design intent, not validated capabilities.

Catastrophic forgetting (the tendency of neural networks to lose previously learned information when trained on new tasks) has been recognized as a fundamental challenge since the early connectionist literature [33, 65]. To understand what GEENNS contributes to this problem, it helps to position it among existing approaches to continual and multi-task learning. Table B.4 compares GEENNS against established methods along four dimensions, and Figure B.7 places the same methods in the isolation–flexibility plane.

Table B.4: GEENNS compared with established continual learning and modular approaches. "Encoding" distinguishes direct (one gene per weight) from indirect (CPPN-generated). "Forgetting prevention" describes the primary mechanism for retaining prior task performance.

Method	**Topology**	**Encoding**	**Multi-task**	**Forgetting prevention**
EWC [51]	Fixed	Direct	Sequential	Weight regularization
PackNet [62]	Fixed	Direct	Sequential	Weight freezing by mask
PNN [80]	Growing	Direct	Sequential	Lateral connections
MoE [43]	Fixed gated	Direct	Simultaneous	Gated routing
Modular NE	Evolved modules	Direct	Separate	Separate populations
GEENNS	**Evolved adaptive**	**Indirect**	**Compositional**	**Frozen grids + evolved mappers**

Three properties position GEENNS within the table:

1. **Indirect encoding with evolved adaptive topology.** GEENNS substrates are generated by CPPNs through EMR-HyperNEAT, enabling compact genomes to produce large-scale structured networks with the potential for per-node function diversity. No other method in the table uses indirect encoding.

2. **Compositional multi-task operation.** Rather than learning tasks sequentially (EWC, PackNet, PNN) or training all tasks simultaneously (MoE), GEENNS trains specialist grids independently and then composes them through evolved mappers. Tasks are decomposed and delegated, not sequentially accumulated.

3. **Forgetting prevention through architectural separation.** Grids are frozen after specialist training. New tasks cannot overwrite grid weights because mappers, not grids, are evolved for each new task. This is structurally different from regularization (EWC), masking (PackNet), or growing (PNN).

The key caveat: this architectural separation has been validated up to 5-grid Boolean scale (with 6-grid recoverable via gated routing). Whether it scales to dozens of grids and non-trivial tasks, and whether the mapper complexity remains tractable, is unknown. The comparison table positions GEENNS among these methods; it does not claim superiority.

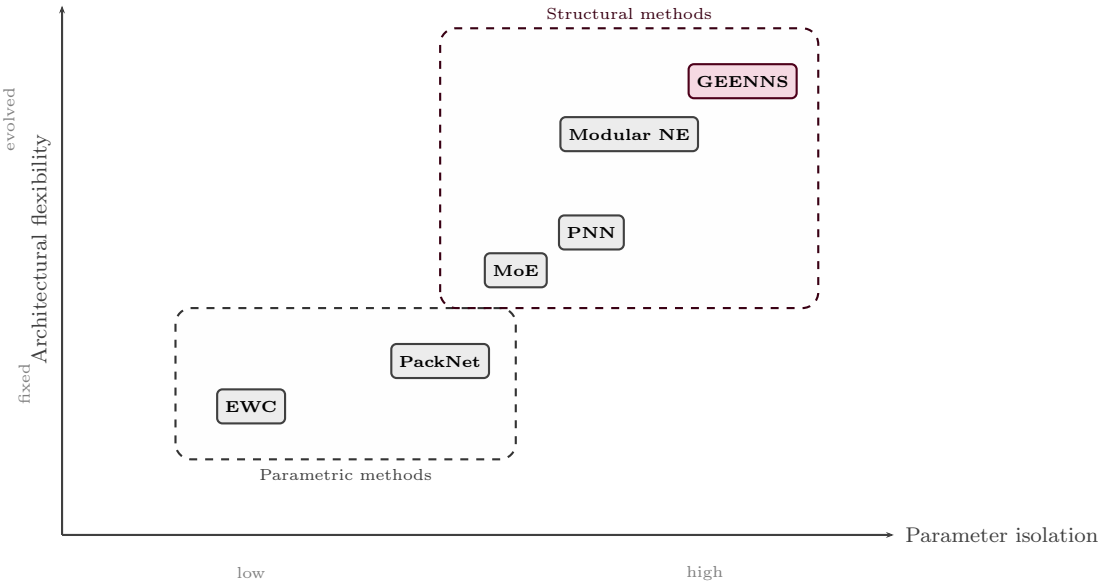


Figure B.7: GEENNS positioned among continual learning methods. The x-axis measures degree of parameter isolation (from shared weights to fully separate components); the y-axis measures architectural flexibility (from fixed topology to fully evolved). GEENNS occupies the upper-right: evolved adaptive topology with full parameter isolation through frozen specialist grids.

B.8 Formal Mathematical Specification

We present the mathematical formalization of GEENNS's core components. The formalism below captures the current design intent. None of the definitions, dynamics, or fitness

functions below have been realized in code beyond the simplified prototype of Section 7.6. This section provides the specification at a level sufficient for implementation and for connecting to the experimental contributions of Chapters 3–6.

B.8.1 Grid Formalism

Definition B.1 (Grid)**.** *A GEENNS grid is a tuple*

$$\mathcal{G} = (V, E, f, g, \mathbf{R})$$

where:

- $V = \{v_1, \ldots, v_n\}$ *is a set of nodes, each with spatial coordinates* $(c_i, l_i, n_i) \in \mathbb{N}^3$ *where* c_i *is the column index,* l_i *is the layer index, and* n_i *is the within-layer node index.*
- $E \subseteq V \times V \times \mathbb{R}$ *is a set of weighted directed connections, where* $(v_i, v_j, w_{ij}) \in E$ *indicates a connection from* v_i *to* v_j *with weight* $w_{ij} \in \mathbb{R}$*.*
- $f : V \to \mathcal{F}$ *assigns each node an activation function from the palette* $\mathcal{F} = \{f_1, \ldots, f_p\}$ *(e.g.,* $\{\texttt{tanh}, \texttt{sin}, \texttt{sigmoid}, \texttt{relu}, \ldots\}$*).*
- $g : V \to \mathcal{A}$ *assigns each node an aggregation function from the set* $\mathcal{A} = \{g_1, \ldots, g_q\}$ *(e.g.,* $\{\texttt{sum}, \texttt{min}, \texttt{max}, \texttt{mean}\}$*).*
- $\mathbf{R} \in \mathbb{R}^{|V| \times 4}$ *is the receptor density matrix, where row* $\mathbf{r}_i = [r_i^{DA}, r_i^{5HT}, r_i^{NE}, r_i^{ACh}]$ *specifies the sensitivity of node* v_i *to each neurotransmitter.*

Each node v_i additionally carries a bias $b_i \in \mathbb{R}$, and two distinguished subsets $V^{\text{input}}, V^{\text{output}} \subseteq V$ designate the nodes at which mappers inject and read signals (written $V_i^{\text{input}}, V_i^{\text{output}}$ when grid $\mathcal{G}_i$ must be named). The spatial coordinates enforce the columnar organization: nodes with the same column index c_i belong to the same column, and within a column, nodes are organized into layers by l_i. The canonical microcircuit template is captured by the constraint that the intra-columnar connectivity pattern $\{(v_i, v_j, w_{ij}) \in E \mid c_i = c_j\}$ is identical (up to weight values) across all columns.

B.8.2 State Dynamics

Given a grid $\mathcal{G}$, the state vector $\mathbf{s}(t) = [s_1(t), \ldots, s_{|V|}(t)]^T \in \mathbb{R}^{|V|}$ evolves under discrete-time dynamics. For a node v_i with activation function $f(v_i)$, aggregation function $g(v_i)$, and incoming connections from nodes $\{v_j : (v_j, v_i, w_{ji}) \in E\}$:

$$s_i(t+1) = f(v_i)(g(v_i)(\{w_{ji} \cdot s_j(t) : (v_j, v_i, w_{ji}) \in E\}) + b_i) \tag{B.1}$$

where b_i is the bias of node v_i. Under a neuromodulatory extension with task-specific neurotransmitter (NT) vector $\mathbf{n}_\tau \in \mathbb{R}^4$, the update becomes:

$$s_i(t+1) = f(v_i)(\alpha_i(\mathbf{n}_\tau) \cdot g(v_i)(\{w_{ji} \cdot s_j(t)\}) + b_i) \tag{B.2}$$

where the gain modulation factor $\alpha_i(\mathbf{n}_\tau) = \mathbf{r}_i \cdot \mathbf{n}_\tau$ is the dot product of node v_i's receptor density with the task's NT vector.

B.8.3 Mapper Formalism

Definition B.2 (Mapper)**.** *A mapper for task τ over grids $\mathcal{G}_1, \ldots, \mathcal{G}_k$ is a tuple*

$$\mathcal{M}_\tau = (M_{in}, M_{out}, M_{inter})$$

where:

- $M_{in} : \mathbb{R}^{d_{in}} \to \prod_{i=1}^{k} \mathbb{R}^{|V_i^{input}|}$ *is the input mapper that decomposes the task input $\mathbf{x} \in \mathbb{R}^{d_{in}}$ into injection signals for each grid's input nodes.*
- $M_{out} : \prod_{i=1}^{k} \mathbb{R}^{|V_i^{output}|} \to \mathbb{R}^{d_{out}}$ *is the output mapper that composes the grids' output signals into the task output $\mathbf{y} \in \mathbb{R}^{d_{out}}$.*
- $M_{inter} : \prod_{i=1}^{k} \mathbb{R}^{|V_i|} \to \prod_{i=1}^{k} \mathbb{R}^{|V_i|}$ *is the inter-grid mapper that routes signals between grids during processing.*

Each component of $\mathcal{M}_\tau$ is implemented as a NEAT-evolved network, allowing the mapper's topology and weights to evolve against frozen grid architectures. The compositional forward pass for a task with input $\mathbf{x}$ is:

$$\mathbf{x}_1, \ldots, \mathbf{x}_k = M_{\text{in}}(\mathbf{x}) \qquad \text{(decompose)} \tag{B.3}$$

$$\mathbf{s}_i^{(0)} = \text{inject}(\mathbf{x}_i, \mathcal{G}_i) \qquad \text{(inject)} \tag{B.4}$$

$$\mathbf{s}_i^{(t+1)} = F_{\mathcal{G}_i}(\mathbf{s}_i^{(t)}, M_{\text{inter}}(\mathbf{s}_1^{(t)}, \ldots, \mathbf{s}_k^{(t)})) \qquad \text{(propagate)} \tag{B.5}$$

$$\mathbf{y} = M_{\text{out}}(\text{extract}(\mathbf{s}_1^{(T_{\text{prop}})}), \ldots, \text{extract}(\mathbf{s}_k^{(T_{\text{prop}})})) \qquad \text{(compose)} \tag{B.6}$$

where $F_{\mathcal{G}_i}$ is the state update function of grid $\mathcal{G}_i$ (Equation B.1 or B.2), T_{prop} is the number of propagation steps, and inject/extract select the appropriate input/output nodes.

B.8.4 CPPN Connectivity Generation

Connection existence and weights are generated by CPPNs queried at spatial coordinates. For a connection between source node v_i at normalized coordinates $(\hat{c}_i, \hat{l}_i)$ and target node v_j at $(\hat{c}_j, \hat{l}_j)$, where the within-layer node index n_i is folded into the normalized horizontal coordinate so that each node retains a distinct query position:

$$(w_{ij}, p_{ij}) = \mathcal{C}(\hat{c}_i, \hat{l}_i, \hat{c}_j, \hat{l}_j) \tag{B.7}$$

where $\mathcal{C} : \mathbb{R}^4 \to \mathbb{R}^2$ is the CPPN, w_{ij} is the connection weight, and p_{ij} is the connection existence output, thresholded by magnitude. A connection exists if $|p_{ij}| > \theta_{\text{conn}}$, where θ_{conn} is the connection existence threshold (Chapter 2).

For multi-output CPPNs that could also assign per-node functions:

$$(w_{ij}, p_{ij}, f_j, g_j) = \mathcal{C}_{\text{ext}}(\hat{c}_i, \hat{l}_i, \hat{c}_j, \hat{l}_j) \tag{B.8}$$

where $f_j \in \{1, \ldots, p\}$ is the activation function index and $g_j \in \{1, \ldots, q\}$ is the aggregation function index assigned to the target node.

B.8.5 Development Function

The genome-to-phenotype transformation $\mathcal{D}$ maps a genome Γ to a functional grid:

$$\mathcal{D} : \Gamma \to \mathcal{G} \tag{B.9}$$

where $\Gamma = (\Gamma_{\text{NEAT}}, \Gamma_{\text{CPPN}}, \Gamma_{\text{micro}})$ consists of the NEAT genome for the canonical micro-circuit (Γ_{micro}) and the CPPN genomes for connectivity patterns (Γ_{CPPN}), both encoded within the NEAT framework (Γ_{NEAT}). The development function proceeds through the five stages described in Section B.5 and illustrated in Figure B.6.

Determinism, stated informally in Section B.5, is here the requirement that $\mathcal{D}$ be a function: the same genome yields the same grid.

B.8.6 Multi-Task Fitness

For tasks $\{\tau_1, \ldots, \tau_T\}$ with mappers $\{\mathcal{M}_{\tau_1}, \ldots, \mathcal{M}_{\tau_T}\}$, the fitness of a genome Γ is:

$$\phi(\Gamma) = \min_{\tau \in \{\tau_1, \ldots, \tau_T\}} \phi_\tau(\mathcal{D}(\Gamma), \mathcal{M}_\tau) \tag{B.10}$$

where $\phi_\tau(\mathcal{G}, \mathcal{M}_\tau)$ is the task-specific fitness of grid $\mathcal{G}$ using mapper $\mathcal{M}_\tau$. The min aggregation ensures that the genome must perform well on *all* tasks to achieve high fitness, preventing specialization on a subset. Convergence is defined as $\phi(\Gamma) \geq \phi^* = 0.95$.

This fitness function would have a direct analog under neuromodulation, where a task-specific NT vector $\mathbf{n}_\tau$ would serve the role of the mapper $\mathcal{M}_\tau$: different NT vectors would route the same topology through different effective configurations, and fitness would be the minimum across all task-specific evaluations.

Appendix C

Supplementary Background Material

This appendix contains supplementary background material supporting Chapter 2 and the experimental chapters. The material provides additional detail on standard algorithms, benchmark tasks, and statistical methods referenced in the main text.

C.1 Parity Truth Tables and Scaling

The parity family recurs throughout the experimental chapters as the canonical non-monotonic Boolean task. Table C.1 gives the Parity-3 truth table in full, and Table C.2 summarizes how the input space and the all-ones output behave as N grows.

Table C.1: Truth table for Parity-3. The output is 1 iff an odd number of inputs are 1. The alternation pattern (every unit Hamming distance change flips the output) makes parity non-monotonic.

x_1	x_2	x_3	**Parity-3**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

C.2 GPU SIMT Architecture

NVIDIA GPUs execute programs using the Single Instruction, Multiple Threads (SIMT) architecture. Thousands of lightweight threads execute the same program on different data, organized into a hierarchy:

Warps. The fundamental unit of execution is the *warp*: a group of 32 threads (on NVIDIA hardware) that execute in lockstep. All 32 threads in a warp execute the same instruction at the same time. When a warp encounters a conditional branch (an `if`

Table C.2: Parity-N scaling summary. Input patterns grow exponentially with N. The all-ones output $f(\mathbf{1})$ alternates between 0 (even N) and 1 (odd N), illustrating the non-monotonic oscillation that makes parity an evolvability barrier for uniform monotonic activation functions (Chapter 7) (§2.1.9).

N	Input patterns (2^N)	$f(\mathbf{0})$	$f(\mathbf{1})$
2	4	0	0
3	8	0	1
4	16	0	0
5	32	0	1
6	64	0	0
7	128	0	1
8	256	0	0
9	512	0	1
10	1,024	0	0

statement), both branches must be executed if different threads take different paths: threads not on the active branch are masked (produce no output but still consume cycles). This phenomenon, called *warp divergence*, is the primary source of GPU inefficiency for irregular algorithms.

Thread blocks. Warps are grouped into thread blocks (typically 128–1024 threads). Threads within a block can communicate through fast shared memory (on-chip, ∼100 KB) and synchronize via barriers. Threads in different blocks cannot synchronize with each other and communicate only through slow global memory.

Grid. The full computation is organized as a grid of thread blocks. The GPU scheduler assigns thread blocks to streaming multiprocessors (SMs), with multiple blocks potentially executing on the same SM to hide memory latency.

C.3 GPU Memory Hierarchy

GPU memory is organized in a hierarchy with decreasing speed and increasing capacity:

1. **Registers**: Per-thread, fastest access (∼1 cycle), limited (∼256 per thread).
2. **Shared memory**: Per-block, fast (∼5 cycles), on-chip (∼100 KB per SM).
3. **L2 cache**: Shared across SMs, moderate speed (∼200 cycles).
4. **Global memory**: Device-wide, slow (∼400–800 cycles), large (8–80 GB).

Memory coalescing is the mechanism by which the GPU combines multiple memory accesses from threads in a warp into a single transaction. When consecutive threads access consecutive memory addresses, the accesses coalesce into one or a few transactions. When threads access scattered addresses, each access becomes a separate transaction, sharply reducing bandwidth. Irregular data structures (trees, linked lists, variable-length arrays) produce scattered access patterns and poor coalescing.

C.4 Hyperparameter Search Methods

C.4.1 Random Search

Random search samples hyperparameter configurations uniformly at random from the configuration space. Despite its simplicity, random search is competitive with or superior to grid search for high-dimensional hyperparameter spaces, as Bergstra and Bengio [3] demonstrated. The reason is that grid search wastes evaluations on parameters that have little effect on performance: if only 3 of 10 hyperparameters matter, a 10^{10}-point grid allocates 10 points per important dimension, whereas random search with the same budget distributes evaluations more effectively across the important dimensions.

Random search provides a natural baseline: any hyperparameter optimization method that cannot outperform random search is adding complexity without value. In the context of ES-HyperNEAT optimization, random search defines the lower bound of expected performance (Chapter 3).

C.4.2 Bayesian Optimization

Bayesian optimization [83, 86] models the relationship between hyperparameters and performance using a probabilistic surrogate model, then uses this model to decide which configurations to evaluate next. The framework has two components:

Surrogate model. A probabilistic model $p(y \mid \mathbf{x})$ that predicts the performance y of a hyperparameter configuration $\mathbf{x}$. Common surrogate models include Gaussian Processes [76] (which provide uncertainty estimates but scale poorly with the number of evaluations) and Tree-structured Parzen Estimators (TPE; see §2.2.2).

Acquisition function. A function that balances exploration (evaluating configurations where the surrogate is uncertain) with exploitation (evaluating configurations where the surrogate predicts good performance). The most common acquisition function is Expected Improvement (EI):

$$\mathrm{EI}(\mathbf{x}) = \mathbb{E}\left[\max(0, y^* - y(\mathbf{x}))\right] \tag{C.1}$$

where y^* is the best performance observed so far. EI is zero at configurations expected to be worse than the current best, and positive at configurations that might improve upon it, with the magnitude reflecting both the expected improvement and the uncertainty.

The optimization loop proceeds as follows: (1) fit the surrogate model to all evaluations so far, (2) select the configuration that maximizes the acquisition function, (3) evaluate that configuration, (4) update the surrogate model with the new result, and (5) repeat.

C.5 ES-HyperNEAT Hyperparameter Space

Chapter 3 optimizes a 16-parameter subset of ES-HyperNEAT's full configuration surface via TPE. Table C.3 enumerates that subset and the discrete search ranges used during the 2,013-trial campaign (band threshold is listed for completeness but was held at 0.3; the counts exclude it), grouped by which component the parameter governs: the NEAT evolutionary engine, the Compositional Pattern-Producing Network (CPPN), and the substrate discovery process.

Table C.3: ES-HyperNEAT hyperparameters varied during the 2,013-trial TPE search of Claret et al. [11]. The CPU-based library used in this work (neat-python 0.92) exposes 59 configurable keys spanning fitness criterion, genome encoding, compatibility coefficients, weight/bias/response properties, reproduction settings, species management, and stagnation detection; with the substrate-side parameters the surface reaches 67 keys, of which roughly 40 are genuinely tunable (the remainder are task-specific or library defaults). This table shows the 16-parameter subset optimized by TPE (nine NEAT-Python parameters and seven substrate-side parameters, of which five are Risi and Stanley 2012 ES-HyperNEAT additions plus `max_weight` and `activation`). The Cartesian product of discrete steps across these 16 yields more than 3×10^9 configurations; the band-threshold row is listed for completeness but was held at 0.3 and is excluded from the counts.

Level	Parameter	Search range	Controls
NEAT	Initial hidden nodes n_h	{0, 1, 2, 3, 4, 5}	Starting topology complexity
	Population size N	{10, 20, 40, 60, 80, 100}	Search capacity
	Add connection prob.	{0.3, 0.5, 0.7}	Structural growth
	Delete connection prob.	{0.3, 0.5, 0.7}	Structural simplification
	Add node prob.	{0.2, 0.4, 0.6, 0.8}	Topological growth
	Delete node prob.	{0.2, 0.4, 0.6, 0.8}	Topological simplification
	Initial connection scheme	5 schemes[a]	Starting connectivity pattern
	Connection fraction	{0.2, 0.4, 0.6, 0.8}	Density of initial connections
CPPN	Default activation f_{act}	{sigmoid, gauss, clamped, relu, tanh, softplus}	CPPN nonlinearity
Substrate	Initial depth d_0	{1, 2}	Starting grid resolution
	Maximum depth D	{3, 4, 5}	Max resolution (4^D positions)
	Division threshold θ_{div}	{0.3, 0.5, 0.7}	Quadtree subdivision (Phase 1)
	Variance threshold θ_{var}	{0.01, 0.02, 0.03, 0.04}	Pruning recursion depth (Phase 2)
	Band threshold θ_{band}	{0.3}	Connection expression (Phase 2)
	Maximum weight w_{max}	{3.0, 5.0, 7.0}	Connection strength cap
	Iteration level	{0, 1, 2, 3}	Hidden-node seeding iterations
	Substrate activation	{sigmoid, gauss, clamped, relu, tanh, softplus}	Substrate node nonlinearity

[a] full_nodirect, full_direct, unconnected, fs_neat_hidden, fs_neat_nohidden.

C.6 ES-HyperNEAT Division and Pruning: An Extended Walkthrough

This section expands the division-and-pruning walkthrough of §2.1.6 with the weight-landscape view in full and a worked example on a banded landscape.

The weight landscape's features are not designed, and the algorithm does not create them. A Gaussian component in the evolved CPPN produces a localized *band* whose position and width are set by the CPPN's evolved weights; a sine component produces periodic bands; compositions produce arbitrary structure (Figure 2.4 shows the one-dimensional profiles these components contribute). Mutating the CPPN moves and reshapes these features across generations, and each source neuron anchors its own field (Figure C.1 works one such landscape through both phases).

C.7 Speciation and CPPN Complexification Dynamics

This section presents supplementary population-dynamics evidence supporting the runtime-explosion analysis of Chapter 3 (§3.2.7). The 100-generation extended evaluation reveals two coupled trajectories, tracked in Figure C.2: NEAT's speciation behavior and the structural complexity of the best-evolved CPPN. The speciation dynamics reveal how NEAT's innovation-protection mechanism operates over extended evolution. Initial fragmentation (a median of one species splitting to two within the first few generations) occurs rapidly as topology mutations create incompatible genome lineages. Species count then continues to rise in widely spaced steps, the median never decreasing, and reaches its final median of 5 only in the closing third of the run.

The best individual's node count settles at its final value of 3 before the species count stops rising, and connection count fluctuates before settling at 6 in the final generations. While these absolute numbers appear modest, each added CPPN node compounds the quadtree subdivision cost: a more complex CPPN produces a substrate with more spatial variation, which triggers more subdivision. This steady complexification drives the runtime explosion analyzed in Chapter 3 (Figure 3.4): NEAT adds structural complexity that the substrate formation process must evaluate, but the resulting larger substrates produce no meaningful accuracy improvement beyond generation 20.

C.8 MoE Historical Development

Jacobs et al. [43] introduced the Mixture of Experts (MoE) architecture as a supervised learning method where multiple expert networks compete to handle different regions of the input space. The architecture consists of:

1. **Expert networks** $E_1, E_2, \ldots, E_K$: Each expert is a neural network that produces an output $\mathbf{y}_k = E_k(\mathbf{x})$ for input $\mathbf{x}$.

2. **Gating network** G: A network that produces a probability distribution over experts: $\mathbf{g} = G(\mathbf{x})$, where $g_k \geq 0$ and $\sum_k g_k = 1$.

3. **Mixture output**: $\mathbf{y} = \sum_{k=1}^{K} g_k \cdot E_k(\mathbf{x})$.

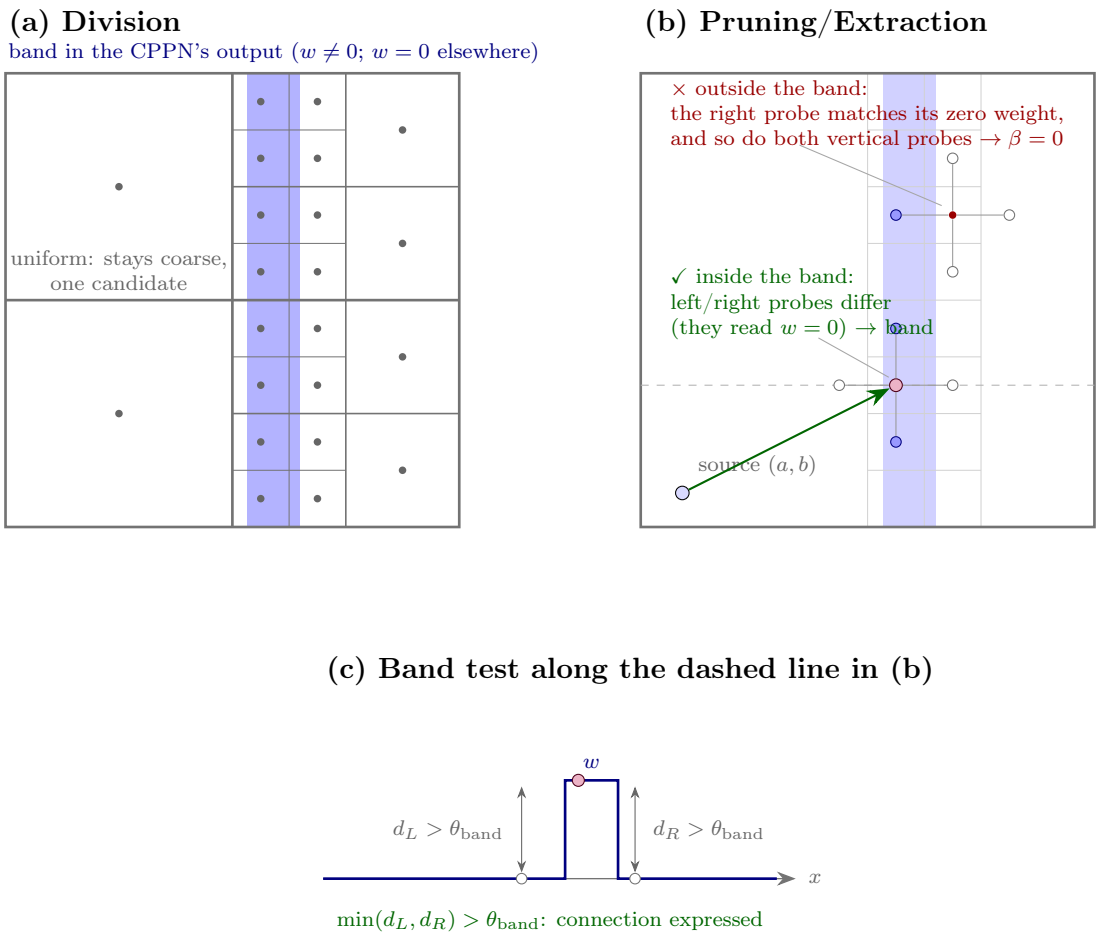


Figure C.1: How division and pruning/extraction cooperate, shown for one source neuron (a, b). The landscape is the CPPN's output $\mathcal{C}(a, b, \cdot, \cdot)$ over the plane; in this example the evolved CPPN produces a single thin vertical band (blue, $w \neq 0$) and zero elsewhere. The band's position and width are properties of the evolved CPPN (e.g., the offset and slope feeding a Gaussian component), not choices made by the algorithm, which only detects them. **(a)** Division recursively subdivides while the variance of the four child-center weights exceeds θ_{div}; line thickness encodes tree depth. The flat left half stops at the first level, while cells refine around the band; every cell center (dot) is a candidate carrying its queried weight. **(b)** Pruning descends only into subtrees whose weight variance exceeds θ_{var}, then applies the band test (Equation 2.6) at each stopping candidate, probing the CPPN one full cell side away on each side; probe fill shows the weight found there. The red candidate sits in the refined column but outside the band: its left probe lands inside the band, but the right probe matches its zero weight, so the horizontal pair contributes $\min(d_L, d_R) = 0$; the vertical probes match it as well, so $\beta = 0$ and no connection is expressed. The purple candidate sits inside the band: both horizontal probes fall outside it and differ, the test passes, and the candidate is extracted as a connection from (a, b) (green arrow) whose endpoint becomes a new hidden node. **(c)** The same test seen on the weight profile along the dashed line: d_L and d_R are the visible steps between the candidate's weight and its horizontal probes, both exceeding θ_{band}. A candidate whose weight differs from its surroundings on one side only fails: the min pairing keeps the smaller difference.

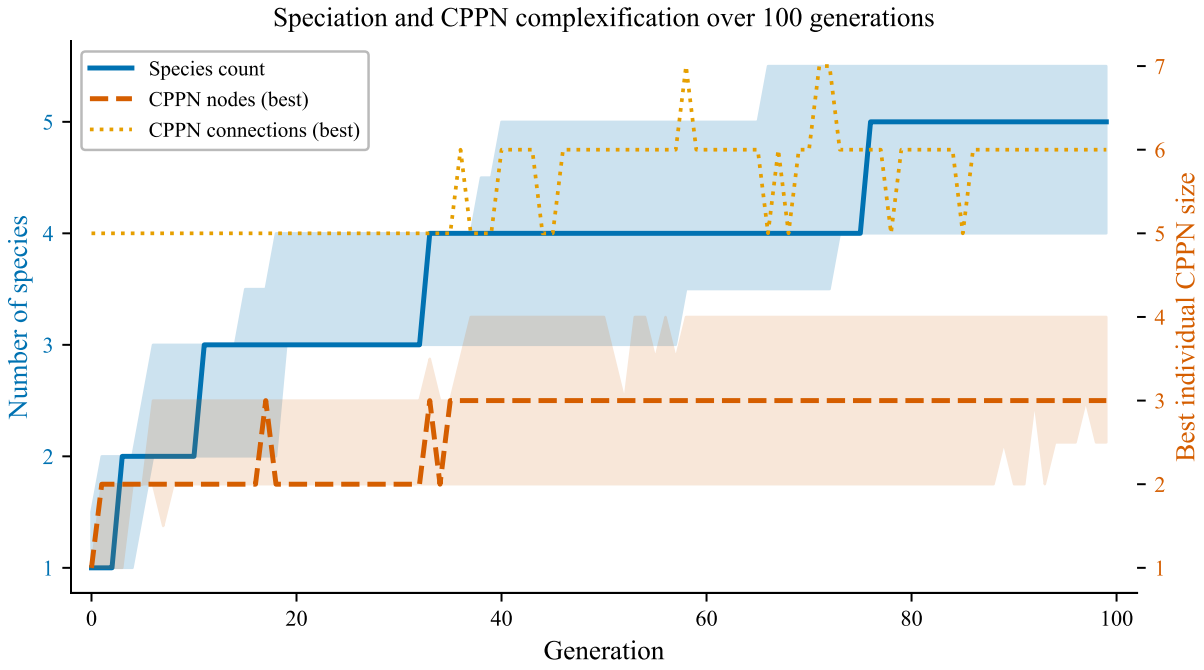


Figure C.2: Speciation and CPPN complexification over 100 generations ($N = 27$ runs). Left axis: number of species (blue, median ± IQR). Right axis: best individual's CPPN node count and connection count (red/orange). Species fragment rapidly in the earliest generations, then add further species in widely spaced steps; the median count never decreases. The best individual's CPPN node count settles at its final value while species are still being added.

The gating network learns to route each input to the most appropriate expert(s), while the experts specialize on the inputs they receive. Jordan and Jacobs [45] extended this to hierarchical MoE (HME), where experts can themselves be MoE systems, enabling multi-level decomposition of the input space.

Shazeer et al. [84] revitalized MoE at scale by introducing sparsely-gated MoE layers within deep networks. The key innovation is that only a small number of experts (typically 1–2) are activated for each input, via a top-k gating mechanism:

$$G(\mathbf{x}) = \text{softmax}(\text{top}_k(H(\mathbf{x}), k)) \tag{C.2}$$

where $H(\mathbf{x})$ is a linear transformation and top_k keeps only the k largest values, setting the rest to $-\infty$ before softmax. This sparse activation enables models with billions of parameters where each input activates only a small fraction. The result is conditional computation: the model's capacity scales with the number of experts, but computational cost scales only with the number activated per input.

Recent applications include the Switch Transformer [29], which routes each token to a single expert and scales to trillions of parameters, and GShard [59], which distributes MoE layers across multiple devices for efficient training.

C.9 Neuroevolution and the NAS Taxonomy

This section expands the definitional comparison of §2.6.

The relationship is definitional as much as technical, and it depends on which definition of NAS is in force. In the broad sense (automating the design of network architectures), NEAT has performed NAS since 2002: it searches topology, and by the literal reading of the term that is architecture search. The narrow modern usage implies a specific decomposition [27]: a search strategy proposes architectures from a predefined space, a separate training procedure fits each candidate's weights by gradient descent, and the returned artifact is an architecture that is then trained once more from scratch. NAS names the problem; a search strategy answers only where to look next. Evolutionary methods do appear inside this decomposition, as one search strategy among several, but there the weights are still learned by gradient descent. The neuroevolutionary lineage in this thesis breaks the decomposition itself: evolution carries the topology and the weights together, no gradient training occurs at any stage (Table 2.3, "Weight optim." row), and the search returns a functioning network rather than a blueprint awaiting training. NEAT is therefore NAS in the broad sense and not in the narrow one. ES-HyperNEAT complicates the classification once more: the object under search is not an architecture but a CPPN, an architecture *generator*, and each evaluation develops a substrate from it rather than selecting one from a space. The search operates one level above the modern NAS formulation (architecture search over architecture builders), which is precisely what makes its search space open where cell-based spaces are fixed.

C.10 Extended Benchmark Definitions

This section provides formal definitions of benchmark problems referenced throughout the thesis that extend beyond the core Boolean and Parity tasks defined in Chapter 2.

C.10.1 Visual Discrimination

The visual discrimination task, a size-discrimination variant in the family of HyperNEAT visual benchmarks [90], presents a 5×5 grid of binary pixels containing a variable number of active pixels, and the network must classify whether the object is small (fewer than 4 active pixels) or large (4 or more). The dataset holds 100 sampled patterns (50 per class) whose active pixels sit at random positions, so size must be judged independently of location. The task has 25 binary inputs (plus a bias) and 1 output; fitness is one minus the mean squared error against the binary size label over the pattern set. This task tests whether the evolved substrate integrates activation across the full visual field: estimating object size regardless of position requires connectivity that spans the grid. On this task, the cross-domain sweep reports 71.7% of runs solved, aggregated across the six connection-type presets (Section 6.5.10); within the per-preset breakdown at depth 5, feedforward alone reaches 90% and backward connections 100% (Section 6.5.11).

C.10.2 Retina Problem

The retina problem presents 8 binary inputs as a 4+4 split retina (left and right halves, plus a bias input) and asks which half carries more active pixels: the two outputs indicate left-has-more and right-has-more, with ties scored at 0.5 on both. All $2^8 = 256$ patterns form the dataset, and fitness is computed from per-output accuracy with a mean-squared-error component. The task tests geometric awareness: the substrate must separate the two halves spatially and compare their activation counts.

The pattern-recognition retina of Kashtan and Alon [49] — eight pixels whose halves are matched against fixed sets of valid patterns — is a related but distinct benchmark used to study modularity; on it, Risi and Stanley [78] report plain ES-HyperNEAT and fixed-substrate HyperNEAT-LEO each solving 30% of runs at a 2,000-generation budget, rising to 57% for ES-HyperNEAT-LEO with geometry seeding.

C.10.3 Orientation Selectivity

Orientation selectivity asks the network to reproduce a V1 simple cell's orientation tuning curve: given a 5×5 field of Gabor filter responses plus the stimulus angle encoded as $\sin\theta$ and $\cos\theta$ (27 inputs, plus a bias), the single continuous output must follow a Von Mises tuning curve centered on the preferred orientation. The task tests smooth periodic function approximation over spatial features, a natural fit for CPPNs with sinusoidal activations. Because the computation is spatial and feedforward, recurrent connections may introduce interference; we use this task in Chapter 6 to validate that connection type choice depends on task structure.

C.10.4 Two Spirals

The two spirals problem [55] requires classifying points belonging to two interleaved spirals in 2D space. The dataset consists of 194 training points distributed along two spirals with 3 full rotations, yielding a highly nonlinear, interleaved decision boundary. The task has 2 continuous inputs (x, y coordinates) and 1 binary output (spiral membership). Fitness is the fraction of correctly classified points.

Two Spirals is difficult: the decision boundary oscillates rapidly along the radial direction, with classes alternating at high frequency. Direct-encoding networks typically

require ∼20 hidden units; indirect encoding methods rarely solve the task at shallow substrate depths. In Chapter 6, we observe a fitness ceiling of ∼0.75 at depth 5 across all connection types, identifying this as a capacity-limited problem.

C.10.5 Concentric Circles

Concentric circles presents two nested circles in 2D and requires classifying whether a point lies in the inner or outer circle. With 2 continuous inputs (x, y) and 1 binary output, this tests the ability to learn a radial decision boundary. Fitness is the fraction of correctly classified points. Unlike Two Spirals, the decision boundary is smooth (a single circle), but it requires the substrate to implement a distance-from-center computation: a nonlinear function of both inputs. In Chapter 6, the uniform-function baseline plateaus at ∼0.78–0.82 at depth 5 (§6.5.11); under per-node activation evolution the problem solves in 100% of runs (Table 6.13).

C.10.6 Symmetry Detection

Symmetry detection presents a 7×7 binary pixel field and asks for its symmetry type (none, horizontal, vertical, or diagonal), with balanced samples per class over 49 inputs (plus a bias). The task exists in two forms. The discrete form outputs a one-hot classification over the four types; in Chapter 6 it remains unsolved at depth 5, plateauing at a fitness of exactly 0.8125. The continuous reformulation instead targets four symmetry-*strength* values: for each of the three mirrors, the fraction of compared cells that match, plus a fourth output encoding the absence of symmetry, with the class's solve threshold set at 0.92 so that both forms demand comparable improvement over their trivial baselines; the campaign reported in Table 6.13 applies its own 0.90 convergence threshold, under which this form solves in every cell.

The task tests *global structure recognition*: the correct output depends on comparing pixel positions across the full field, requiring long-range connections that span the grid. The discrete-versus-continuous contrast is itself informative for substrate methods: CPPN-generated networks produce smooth outputs naturally, and recasting a sharp classification as a continuous regression converts a task unsolved at depth 5 into one solved in every cell.

C.10.7 Turing Patterns

Turing patterns are spatial patterns arising from reaction-diffusion dynamics [96]. In our benchmark formulation, the task takes 2 continuous inputs and produces 2 continuous outputs representing morphogen concentrations. Fitness measures how well the substrate's output matches a target reaction-diffusion pattern. This tests the substrate's capacity for generating structured spatial patterns: a generative task rather than a classification task. Picbreeder demonstrated that CPPNs can produce complex spatial patterns [82]. Chapter 6 finds Turing patterns capped near 0.89 at depth 5.

C.10.8 Tower of Hanoi

The Tower of Hanoi is a classic planning problem: move a stack of disks from one peg to another, one disk at a time, never placing a larger disk on a smaller one. We evaluate two formulations: *supervised* (given a state, predict the optimal move) and *goal-conditioned*

(given a state and goal, predict the next action). Input and output dimensions vary with the number of disks and pegs. Fitness is the fraction of correct move predictions across all reachable states.

Tower of Hanoi tests sequential reasoning and goal-conditioned behavior, capabilities beyond pattern matching. In Chapter 6, both formulations exhibit a fitness ceiling of $\sim$0.78 at depth 5. This ceiling suggests that the shallow substrate cannot represent the recursive structure of optimal Hanoi solutions.

C.10.9 Reinforcement Learning Control

Three classic control environments from the OpenAI Gym / Gymnasium suite [7] extend validation to sequential decision problems with continuous state spaces (evaluated in Section 6.5.10).

CartPole takes 4 continuous inputs (cart position, velocity, pole angle, angular velocity) and produces 1 binary output (left or right force). An episode succeeds if the pole remains upright for 200 steps. The smooth, low-dimensional state space makes CartPole the easiest control benchmark.

Acrobot takes 6 continuous inputs (joint angles and angular velocities of a two-link pendulum) and selects among 3 discrete torque options. The goal is to swing the pendulum tip above a height threshold. Acrobot requires coordinating two joints, testing the substrate's ability to handle coupled dynamics.

MountainCar takes 2 continuous inputs (position, velocity) and selects among 3 actions. The agent must build momentum to push a car up a hill, but the reward is sparse (given only upon reaching the goal), making exploration the primary challenge rather than representation.

C.10.10 CCM Financial Data

The CRSP/Compustat Merged (CCM) database [9] provides a real-world regression benchmark. We predict the earnings-price ratio from 5 accounting variables (book value, cash flow, dividends, earnings, and sales), using 94,464 firm-year observations from merged financial databases. With 5 continuous inputs and 1 continuous output, this is a nonlinear regression task. Fitness is measured as R^2 or MSE on a held-out test set.

CCM serves a different purpose from the Boolean/geometric benchmarks: it tests whether EMR's speedup transfers to *realistic data volumes* where each fitness evaluation requires processing tens of thousands of observations. In Chapter 6, we use CCM to validate that the eager reformulation scales to real-world problem sizes.

C.10.11 Earnings-Price Regression

The earnings-price benchmark extends CCM to higher input dimensionality, predicting the firm-level earnings-price ratio from 12 accounting variables including book value, cash flow, dividends, earnings, sales, and additional financial indicators. The same CRSP/Compustat data source is used, but the expanded feature set tests whether the substrate can handle moderate-dimensionality regression (Section 6.5.10).

C.10.12 California Housing

The California Housing dataset [71] contains 20,640 observations of 8 features (median income, house age, average rooms, average bedrooms, population, average occupancy, latitude, and longitude) predicting median house value. As a standard scikit-learn benchmark with well-characterized properties, it tests EMR on a medium-dimensionality regression task with known ground-truth performance levels from conventional methods. This benchmark is evaluated in Section 6.5.10.

C.11 Statistical Methodology

All experimental results in this thesis follow a standardized statistical reporting protocol.

C.11.1 Convergence Criterion Justification

For Boolean and Parity tasks evaluated at the default threshold, $\phi^* = 0.95$ corresponds to correctly classifying all input patterns, since the small number of patterns makes partial success unlikely: with 4 patterns, 3/4 correct gives $\phi = 0.75$ and 4/4 gives $\phi = 1.0$.

This default of $\phi^* = 0.95$ follows the convention established in the ES-HyperNEAT reference literature [78, 103]. Several experiments in this thesis adopt stricter thresholds where the experimental context warrants higher confidence. Chapter 5's main ES-HyperNEAT campaign uses $\phi^* = 0.98$ (§5.5.1, Figure 5.5) and its HSHG evolutionary confirmation uses $\phi^* = 0.975$ on Boolean tasks (§5.4.7); Chapter 6's EMR-HyperNEAT XOR validation uses $\phi^* = 0.99$ as the target fitness (§6.5). Chapter 7's compositional-reasoning prototype tightens to $\phi^* = 0.99$ at 5-grid and $\phi^* = 0.995$ at 6-grid, where class imbalance pushes the all-zeros predictor above 0.98 (§7.6.3). The stricter thresholds either force higher-confidence solutions or maintain a margin above class-imbalance floors; the relaxed $\phi^* = 0.95$ remains the default when neither concern applies. One campaign relaxes below the default: the cross-domain sweep of Table 6.13 applies 0.90 as its convergence criterion for the Retina and Symmetry preset sweeps, 0.85 for Parity-3/4, and 0.80 for Parity-5/6 (see that table's footnotes). These relaxed parity bars award partial credit: their solve rates are not all-patterns-correct rates and are not comparable across parity sizes.

We report the **convergence generation** (the first generation at which the convergence criterion is satisfied) as the primary measure of evolutionary efficiency. Lower convergence generation indicates faster evolutionary search.

Success rate is the fraction of runs (across random seeds) that converge within the maximum number of generations. For 30-seed experiments, we report success rate as a percentage with the count (e.g., "100% (30/30)" or "23.3% (7/30)").

C.11.2 Experimental Design

Unless otherwise stated, experiments use 30 independent random seeds (seeds 0–29). We chose 30 as a balance between statistical power and computational cost: 30 is the conventional threshold for applying the central limit theorem, and bootstrap confidence intervals are reliable at this sample size.

C.11.3 Confidence Intervals

We report 95% confidence intervals computed via the bias-corrected and accelerated (BCa) bootstrap method [25] with 10,000 resamples. The bootstrap is used because our primary metrics (convergence generation, success rate) are not normally distributed: convergence generation is right-skewed (many fast solutions, few slow ones), and success rate is a proportion.

C.11.4 Hypothesis Testing

For comparing two conditions, we use the Mann-Whitney U test [63], a non-parametric test that does not assume normality. We report the U statistic, p-value, and rank-biserial correlation r as the effect size:

$$r = 1 - \frac{2U}{n_1 n_2} \tag{C.3}$$

where n_1 and n_2 are the sample sizes of the two groups. The rank-biserial correlation ranges from -1 to 1, with $|r| > 0.5$ considered a large effect.

For comparing three or more conditions simultaneously, we use the Kruskal-Wallis H test as the non-parametric analog of one-way analysis of variance (ANOVA), followed by post hoc pairwise comparisons with appropriate correction.

C.11.5 Tests for Proportions

When comparing success rates (binary outcomes: converged or not), we use Fisher's exact test rather than the χ^2 approximation, because several experimental conditions yield cell counts below 5 where the χ^2 approximation is unreliable. Fisher's exact test computes the exact probability of observing the given contingency table under the null hypothesis of independence, making it appropriate for any sample size.

C.11.6 Why Non-Parametric Tests

Evolutionary fitness distributions violate the normality assumption required by parametric tests (Student's t, ANOVA). Convergence generation is right-skewed: most runs converge quickly but a few require many generations. Success rate is a proportion bounded by $[0, 1]$. Fitness values often cluster near boundaries (0.5 for chance, 1.0 for convergence). Non-parametric tests (Mann-Whitney U, Kruskal-Wallis, Fisher's exact) make no distributional assumptions and are therefore the default for solve rates and convergence generations; parametric tools appear where the metric supports them (Cohen's d for near-normal metrics and ANOVA with Tukey HSD on runtime, both documented later in this section).

C.11.7 Worked Example

To illustrate the reporting protocol with a hypothetical two-condition comparison at the scale of the Chapter 3 experiments: suppose an optimized configuration achieves 26.8% MNIST accuracy ($n = 30$ seeds) against a baseline's 22.3% ($n = 30$ seeds). A complete report then reads: Mann-Whitney $U = 287$, $p = 0.016$, rank-biserial $r = 0.36$ (medium

effect), Cohen's $d = 0.76$, and a 95% bootstrap CI of $[24.7, 28.9]$ for the optimized condition. The hypothetical demonstrates all required components (sample sizes, test statistic, p-value, effect size, confidence interval); the thesis's measured comparisons are reported in Chapter 3.

C.11.8 Effect Sizes

For normally distributed metrics, we report Cohen's d [21]:

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s_p} \tag{C.4}$$

where s_p is the pooled standard deviation. Conventional thresholds: $|d| = 0.2$ (small), $|d| = 0.5$ (medium), $|d| = 0.8$ (large).

C.11.9 Multiple Comparisons

When testing multiple hypotheses simultaneously, we apply Bonferroni correction [5] to control the family-wise error rate:

$$\alpha_{\text{corrected}} = \frac{\alpha}{m} \tag{C.5}$$

where m is the number of comparisons and $\alpha = 0.05$ is the nominal significance level. For large numbers of comparisons, we use the Holm-Bonferroni procedure [40], which provides greater power while maintaining family-wise error control.

C.11.10 ANOVA

For multi-factor experimental designs, we use ANOVA to decompose variance across factors. In the quadtree GPU analysis (Chapter 5), ANOVA quantifies how much of the runtime variance is attributable to the quadtree structure versus other factors, providing evidence for the incompatibility argument.

The F-statistic is reported as $F(\text{df}_1, \text{df}_2) = \text{value}$, where df_1 is the between-group degrees of freedom (number of groups $- 1$) and df_2 is the within-group degrees of freedom (total sample size $-$ number of groups). Higher F values indicate that between-group variation exceeds within-group variation. The associated p-value reports the probability of observing an F at least this extreme under the null hypothesis of no difference between groups. We complement F with the effect size η^2 (eta-squared, the proportion of total variance explained by the grouping factor); conventional thresholds are 0.01 (small), 0.06 (medium), and 0.14 (large) [21].

For example, $F(2, 3676) = 1090.7$ (Chapter 3) reports a one-way ANOVA with 3 groups ($\text{df}_1 = 3 - 1 = 2$) and 3,679 total samples ($\text{df}_2 = 3{,}679 - 3 = 3{,}676$).

C.11.11 Tukey HSD

When an ANOVA reveals a significant overall difference between groups, Tukey's Honestly Significant Difference (HSD) test identifies which specific pairs of groups differ. Tukey HSD computes all pairwise comparisons while controlling the family-wise error rate, so the reported p-values are already adjusted for the number of comparisons (no additional Bonferroni correction is needed). A Tukey HSD output of $p < 0.001$ for a specific pair

indicates that the difference between those two groups is statistically significant at the family-wise level.

C.11.12 Reporting Format

Results are reported in the following standard format:

> *Success rate*: X% (n/N seeds), *median* M generations, 95% CI [L, U], *effect size*: Cohen's d or rank-biserial r, p-value from Mann-Whitney U test.

For example: "100% convergence (30/30 seeds), median 14 generations, 95% CI [12.7, 19.2]." Median is preferred over mean for convergence generation due to the right-skewed distribution.

Differences between two percentage values use **percentage points (pp)** to disambiguate absolute change from relative change: an improvement from 23.9% to 29.0% accuracy is a 5.1 pp absolute gain, not a 5.1% relative gain.

C.12 JAX API Transformations

JAX provides four key composable function transformations:

- **jit** (Just-In-Time compilation): Compiles Python functions to optimized XLA (Accelerated Linear Algebra) kernels that run on CPU, GPU, or TPU. The compiled function must have fixed array shapes; dynamic shapes require recompilation.
- **vmap** (Vectorized Map): Transforms a function on a single example into one that runs on a batch. For example, `vmap(evaluate_cppn)` lifts a single-CPPN evaluator to the whole population.
- **pmap** (Parallel Map): Distributes computation across devices (GPUs/TPUs), with data partitioning and result aggregation.
- **grad** (Automatic Differentiation): Computes gradients of functions, enabling hybrid evolutionary-gradient approaches.

C.13 JAX-ESHN Implementation Algorithms

The two architecturally significant optimizations from Section 5.6.5 (O5 and O6 of the JAX-ESHN implementation) are presented below in pseudocode for completeness; the remaining optimizations (O7–O10) are described in prose only.

Algorithm 19 O5: Hierarchical Multi-Resolution Grid

Require: CPPN θ, Anchor a, Direction d, Max depth D, Coordinate bounds $[-1, 1]^2$

{Pre-compute all grid positions (done once at initialization)}
for $k = 0$ **to** $D - 1$ **do**
  $n_k \leftarrow 2^{k+1}$ {Grid dimension at level k}
  positions$[k] \leftarrow \text{MeshGrid}(n_k \times n_k)$
  parent_idx$[k] \leftarrow \lfloor \text{positions}[k]/2 \rfloor$ {Static parent mapping}
end for
{Evaluation: batch query all positions at each level}
for $k = 0$ **to** $D - 1$ **do**
  $\mathbf{w}_k \leftarrow \text{vmap}(\text{QueryCPPN})(\theta, a, \text{positions}[k], d)$
  $\sigma_k^2 \leftarrow \text{GroupVariance}(\mathbf{w}_k, \text{parent_idx}[k])$
  $\text{mask}_k \leftarrow \sigma_k^2 > \theta_{\text{div}}$ {Which cells to refine}
end for
return positions, masks

Algorithm 20 O6: GPU-Resident Multi-Generation Loop

Require: Initial state S_0, Fitness threshold ϕ, Max generations G

{Define pure JAX generation step (no Python control flow)}
function Step(S):
  pop $\leftarrow$ Ask(S)
  substrates $\leftarrow$ BuildSubstrates(pop)
  fitness $\leftarrow$ Evaluate(substrates)
  return Tell(S, fitness)
{Run entirely on GPU via `jax.lax.while_loop`}
$S \leftarrow S_0$
while $\max(\text{fitness}(S)) < \phi$ and $\text{gen}(S) < G$ **do**
  $S \leftarrow \text{Step}(S)$
end while
return S

Appendix D

Supplementary Experimental Material

This appendix collects experiment-level detail supporting conclusions stated compactly in Chapters 4, 6, and 7: mechanism walkthroughs, protocol detail, and the full transfer census. Sections D.1 and D.2, and each subsection of Section D.3, are cross-referenced from the body passages they expand; the headline numbers remain in the chapters, and nothing here alters a conclusion. Sections follow thesis order.

D.1 Expert Density versus Exploitation (Chapter 4)

This section expands the receptive-field density finding of §4.6.4, where density counts a pixel as active on a single non-zero connection regardless of whether that connection carries discriminative signal. The two quantities separate cleanly in the data. $N_e = 3$ experts frequently wire over 96% of their segments, yet the accuracy gains do not arrive until $N_e = 5$ (§4.5.2), because connections beyond the ∼50–60-dimension exploitation ceiling contribute wiring rather than discriminative signal. This is also why the prior campaign's champion baseline stopped at ∼48 pixels: with the informative central cluster already wired, additional pixels offer no fitness gradient, and the search settles at a satisficing point rather than a capacity wall.

D.2 Cross-Framework Positioning (Chapter 6)

This section records the measurement protocol behind Table 6.7, the CPU pass (the GPU pass of Table 6.8 reuses the same harness, seeds, and budgets on the CUDA server): batch structure, load sensitivity, and the one substrate parameter left unequalized. EMR's h→h cells use the default execution mode; a 3-seed, 10-generation probe outside the 30-generation protocol put the memory-saving depth-offload variant at roughly 2.1–4.4× the default mode's per-generation cost at depths 3–5. On the GPU pass, the depth-5 h→h median comes from the 23 of 30 seeds that survive mid-run out-of-memory failures; the survivors are the cheaper seeds (a +11.6% cost gap at depth 4), so the printed cell is a lower bound. The depth-7 ff cell is measured under forced depth-offload because the protocol configuration itself runs out of memory (a 1.40 GB allocation on a full device; the default mode's chunking does not engage). The results themselves are analyzed in Section 6.5.4.

The CPU-pass cells were collected in two measurement batches on the same machine, with runs interleaved within each batch and machine state recorded per run. Median

background load differed between batches (3.97 against 6.66 on 16 cores), so load sensitivity was measured rather than assumed: across the fully seeded cells above the sub-100-millisecond regime, the low-load and high-load halves agree to within 3.5% for every arm except EMR h→h (within 2.5% for the substrate-building arms), while the small-n ES-HyperNEAT depth-4 cell shows a 6.7% gap. The three sub-100-millisecond direct-encoding cells jitter by up to 8% of their small absolute values, which is why the bound is taken above that regime. The EMR h→h cells disperse more across seeds (coefficient of variation 14–26% at depths 3–5, under 10% at depths 1–2) because hidden-to-hidden cache-refresh generations recur at seed-dependent frequency; their load halves can differ far more, and inflating EMR's depth-4 cell by the full 19.5% gap (the most conservative attribution of that spread to load) would move the comparison against ES-HyperNEAT from roughly 144× (Table 6.7, depth-4 cells) to roughly 120×; no conclusion drawn here would change. One substrate parameter is not equalized: each substrate-building arm uses its own default weight scale, 5.0 for ES-HyperNEAT [78] and 8.0 for EMR (the framework's default throughout Chapter 6), the same pairing as Table 6.6. Table 6.14 bounds the sensitivity of the h→h configuration to this pair at under two percentage points of solve rate and about 3% of connectivity; feedforward timing is a dense product over the full grid and does not depend on how many weights survive thresholding.

D.3 Prototype Mechanism Detail (Chapter 7)

This section documents the mechanisms behind three prototype experiments of §7.6: the gated-mapper architecture, the continuous-gate protocol and its operator analysis, and the warm-start transfer census.

D.3.1 Gated Selective Routing: Mechanism

This subsection sets out the gate mechanism summarized in §7.6.2. The mechanism is deliberately minimal (Figure D.1). The augmented mapper's input matrix carries one column per grid-output channel; because every frozen grid is applied to every input pair in the reuse setting, two grids and two pairs yield four such columns, two pairing each grid with its intended inputs and two crossing them. A gate is one evolved scalar per column: each mapper individual carries a gate vector alongside its genome, and before evaluation every grid-output column is multiplied by the logistic transform $\sigma(g_i) \in (0, 1)$ of its gate, while the raw-input columns pass through ungated. The sigmoid makes each gate a soft valve rather than a switch: a channel can be attenuated toward silence or left open, but never amplified or sign-inverted, so converged gate values read directly as routing decisions. Gates initialize near $\sigma(0) = 0.5$ (every channel half-open, no routing preassumed) and co-evolve with the genomes under shared selection: fitness is computed for each genome–gate pair on the full truth table, and the next generation's gate vectors are drawn from the top decile of the current one with Gaussian perturbation. Nothing in the fitness signal identifies which channels matter. For the post hoc analysis, and only there, we label each channel by its wiring: *correct* when a grid meets its intended pair, *distractor* when the pairing is crossed.

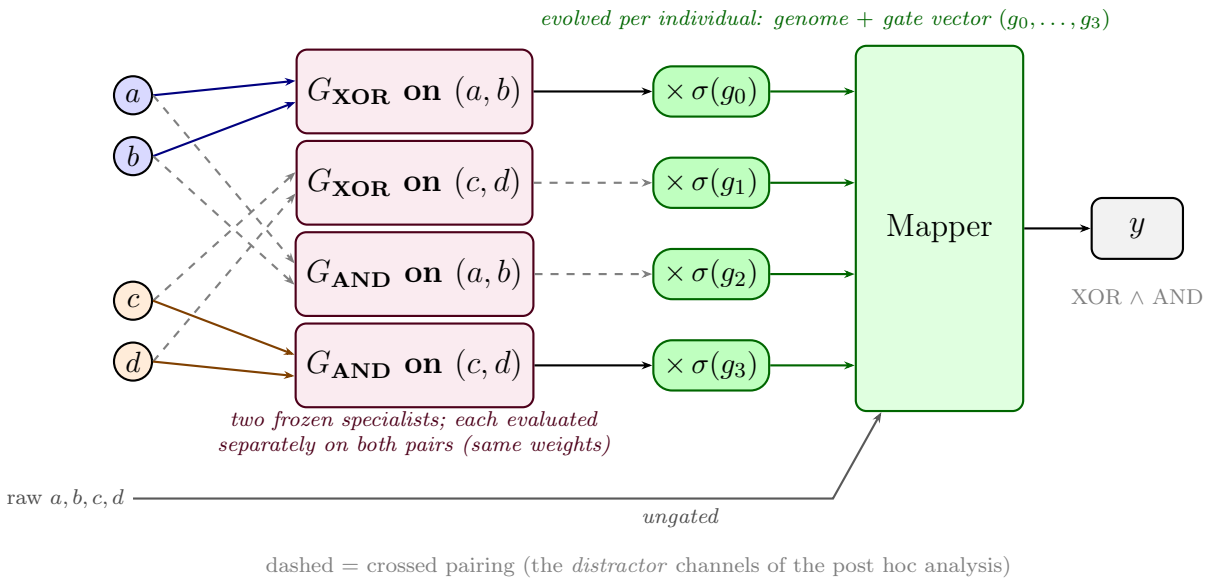


Figure D.1: The gated-mapper mechanism, shown for the 2-grid reuse task $\mathrm{XOR}(a, b) \wedge \mathrm{AND}(c, d)$. In the reuse setting each frozen specialist (purple) is evaluated separately on both input pairs through the same frozen weights, yielding four grid-output channels: two matched pairings (solid) and two crossed pairings (dashed), the *distractor* channels of the post hoc analysis. Each channel is multiplied by the sigmoid $\sigma(g_i) \in (0, 1)$ of one evolved gate before reaching the mapper; raw inputs bypass the gates. Each mapper individual carries its own gate vector alongside its genome, initialized near $\sigma(0) = 0.5$ and co-evolved under shared selection. Because σ is bounded, a gate can silence a channel but never amplify or invert it, so converged gate values read directly as routing decisions.

D.3.2 Continuous-Gate Protocol and Operator Analysis

This subsection details the regressor protocol of §7.6.5 and the analyses behind its operator conclusions.

Each specialist is evolved as a regressor of its own surface (Figure D.3; 64 uniform samples on $[0, 1]^2$, sin hidden activation, scored by R^2 with an $R^2 \geq 0.90$ acceptance gate) and frozen, mirroring the Boolean pipeline; seed 19 falls below the gate and is excluded, leaving 29 seeds in the composition analyses. The composition *operator* is a rule for generating the compound target, applied to the ideal gate surfaces before evolution begins; the mapper is never shown the rule and must approximate the resulting target from the frozen specialists' outputs alone (Figure D.2). Success is measured by $R^2 \geq 0.85$ rather than $1 - \mathrm{MSE}$: multiplicative targets concentrate near zero, where $1 - \mathrm{MSE}$ would reward predicting the mean much as the all-zeros predictor inflates Boolean scores, and R^2 scores that strategy at zero.

The mechanism behind successful composition is not literal multiplication. If the mapper simply multiplied the four frozen-grid predictions, compounding sigmoid compression across the chain would cap performance at the multiplicative ceiling $\prod R_i^2$ (≈ 0.764 at 4-gate); instead, every one of the 29 composition seeds at 3-gate and 4-gate exceeds its scale's ceiling, by mean gaps of $+0.029$ and $+0.059$ respectively. Replaying the converged mappers of three representative seeds per scale across a uniform grid of gate-output values lets us ask what function the mapper actually computes (the surrounding solve statistics use all 29). A degree-3 polynomial in the gate outputs accounts for ∼96.5% of its input–

output variance at both scales (mean $R^2 = 0.965 \pm 0.007$ at 3-gate, 0.965 ± 0.000 at 4-gate), whereas a least-squares rescaled product captures only 90% at 3-gate and 87% at 4-gate. Even at 4-gate, where the true target is itself a degree-4 polynomial, extending the fit to degree-4 gains only $+0.005$ R^2 over degree-3. The mapper has settled on a smooth low-order approximation that caps how much per-gate sigmoid compression can compound: degree-3 truncation holds regardless of gate count, while literal multiplication would lose accuracy with every added gate. This adaptation is specific to the compositional architecture: none of the monolithic conditions exhibit it.

Three additional candidates beyond multiplicative and additive-mean fail uniformly at continuous gate composition: $\max_i g_i$, $\min_i g_i$, and a soft-max-weighted sum with $\beta = 5$ each solve 0 of 29 seeds at both 3-gate and 4-gate (174 runs total, mean R^2 in the range 0.48–0.67). The operator landscape is narrower than its size suggests: among the five operators tested, only additive-mean reliably scales beyond 3-gate. The smooth sin-hidden mapper is plausibly mismatched to operators with kinks at switchover points, although optimization difficulty cannot be ruled out as a contributing factor.

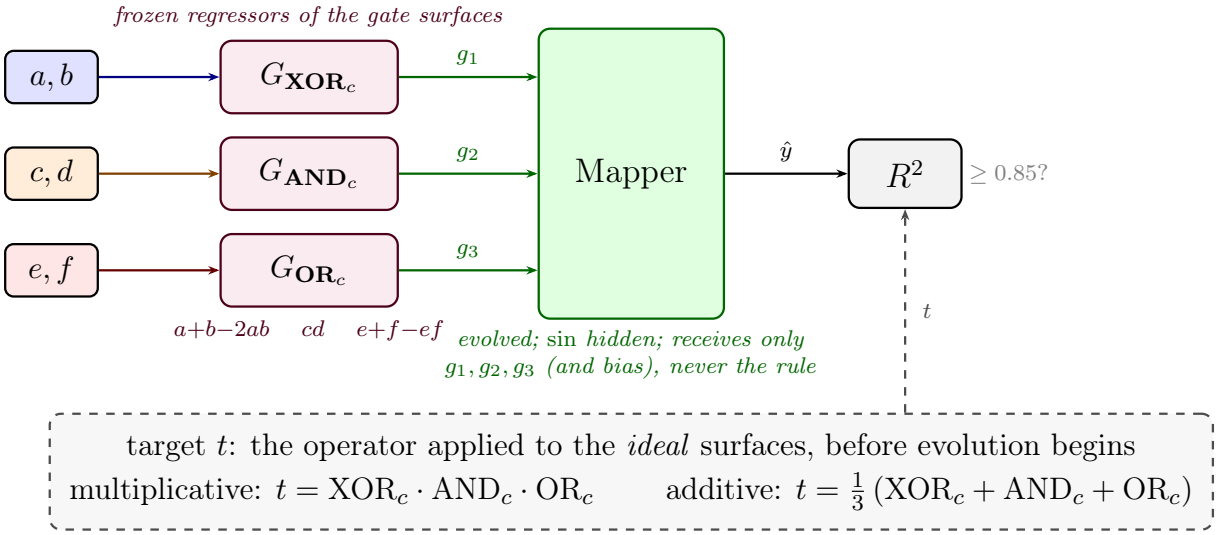


Figure D.2: Continuous gate composition for the 3-gate compound. Each frozen specialist (purple) is a regressor of one continuous gate surface (the multilinear extension of its Boolean truth table, exact at the corners of $[0, 1]^2$) and receives only its own input pair. The composition operator is applied to the *ideal* surfaces, offline, to produce the target column t; the evolved mapper (green) is never shown the rule and must approximate t from the specialists' outputs alone, scored by $R^2 \geq 0.85$. Between the multiplicative and additive regimes the architecture, frozen grids, and budgets are identical; only the target changes.

D.3.3 Warm-Start Transfer and the Latent-Solver Census

This subsection presents the warm-start mechanic and the full population census summarized in §7.6.7.

Warm-start transfer experiments reshape the negative answer of the compositional generalization test (§7.6.7) without overturning it. The mechanic is minimal (Figure D.4): evolve a population to convergence on a source compound, keep the entire final population, and continue evolving it on a related target it has never seen, against a cold-start control

that begins from fresh random initialization. In these experiments the channel assignment tracks the target (the input-pair permutation setting: the same specialists, re-wired to the target's gate arrangement), and under that alignment warm-started runs converge in a median of one generation at both scales. The 45–79× speedups at 3-grid and 223–351× at 5-grid are therefore not faster search but the cold-start medians themselves: the warm population already contains what the search is for, and the first evaluation pass finds it. Evaluating every individual of the source-evolved populations directly on the unseen targets makes the mechanism explicit (Figure D.5). Roughly 0.3% are latent solvers (individuals that, without any retraining, already satisfy the target objective), about one per population of 300, present in 28 of 29 seeds, and they are not hidden in the ranks: their median source-fitness rank is first, and in 56 of 58 seed–target views the source champion itself is among them (zero-shot fitness mean 0.960). The reason is visible in the figure's diagonal: for 93.5% of individuals, fitness is numerically identical on the source and the permuted target, because a mapper that conjoins the grid channels is indifferent to which gate each channel carries. Source selection does not conceal the transferable representation; it concentrates it at the top. Random-initialization populations scaled up to pop 3000 contain fewer than 0.01% such solvers, so the representation is a product of source evolution, not of sampling volume. Across the three per-seed solver sets (source task and both targets) the pairwise Jaccard overlap is 1.0 at the median, 0.88 on average: overwhelmingly, a single representation solves all three compositions. The implication is sharper than a population-versus-individual contrast: under input-pair permutation, transfer is carried by the converged champion itself, and what source evolution produces is a distribution whose top individuals are already general—concentrated at the top of the source ranking, and absent from random initialization at ten times the budget.

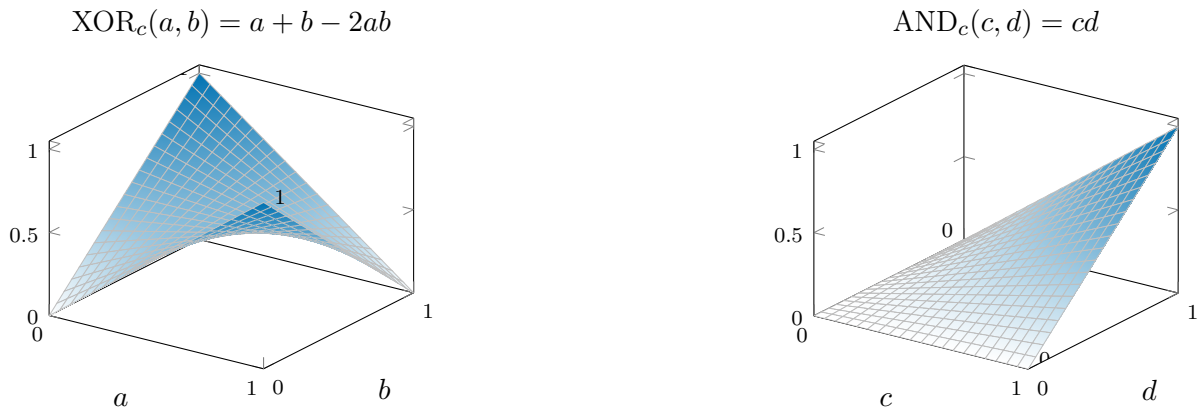


Figure D.3: What a continuous gate looks like. Each surface is the multilinear extension of a Boolean truth table: at the four binary corners (annotated) it reproduces the gate's exact Boolean values, and between them it interpolates smoothly. XOR_c is a saddle, rising toward the disagreeing corners $(0,1)$ and $(1,0)$ and passing through 0.5 at the center; AND_c is a monotone ramp toward $(1,1)$. Each specialist grid is evolved as a regressor of one such surface, then frozen; OR_c, NAND_c, NOR_c, and the 6–8-gate additions follow the same construction.

The speedups measured with the specialist set unchanged conceal one constraint. Substituting a single specialist at 5-grid Boolean scale (NOR→XNOR, the missing oscillatory primitive) reduces the warm-start advantage to a 2.06× ratio-of-medians (cold 828 gen, warm 402 gen) and lifts no seed beyond the cold-start solve rate (22/29 either way). Only

4/29 source-evolved populations contain a zero-shot solver of the substituted target at the strict $f \geq 0.99$ threshold, although all 29 clear the standard 0.95 threshold and mean zero-shot fitness remains high ($\bar{f} = 0.983$). Transfer at 5-grid Boolean is therefore conditional: it holds under input-pair permutation, where the source compositional structure is reused with relabeled coordinates, but it weakens under gate-set substitution, where the source-evolved distribution is partly tuned to the specific gate set. Under the additive operator the same mechanism transfers cleanly: at 5-gate continuous scale with $\text{NOR}_c \to \text{XNOR}_c$, 12 of 14 source-evolved populations contain a zero-shot solver of the unseen target (mean target $R^2 = 0.896$); at 6-gate ($\text{NOR}_c \to \text{IMPLY}_c$) the zero-shot rate drops to 59%; at 7-gate ($\text{NOR}_c \to \text{NIMPLY}_c$) it returns to 86%. The non-monotonic trajectory tracks the structural distance between the original and substituted gates rather than compound size, suggesting that what the source population encodes is partly compositional and partly distributional, a refinement of, not a contradiction to, the evolved-distribution transfer story.

One plausible reading of these results, which the prototype motivates but does not prove, is that the transfer regimes split along whether the channel assignment tracks the target. Where it does, extraction is already solved: for that family of recompositions, the champion is the general mapper. Where it does not, transfer mostly fails (no seed under the fixed-channel test, 4 of 29 populations under gate substitution), and the open problem is genuine: evolving mappers whose strategies survive changes in what the channels mean, not merely in which pair they are wired to. Whether the champion-carried transfer extends to larger grid counts, other task families, or other source tasks is untested; what the prototype establishes is that the permutation-invariant representation reliably emerges from source evolution, is recovered by a single generation of warm-start, and does not arise in random initialization at ten times the search budget.

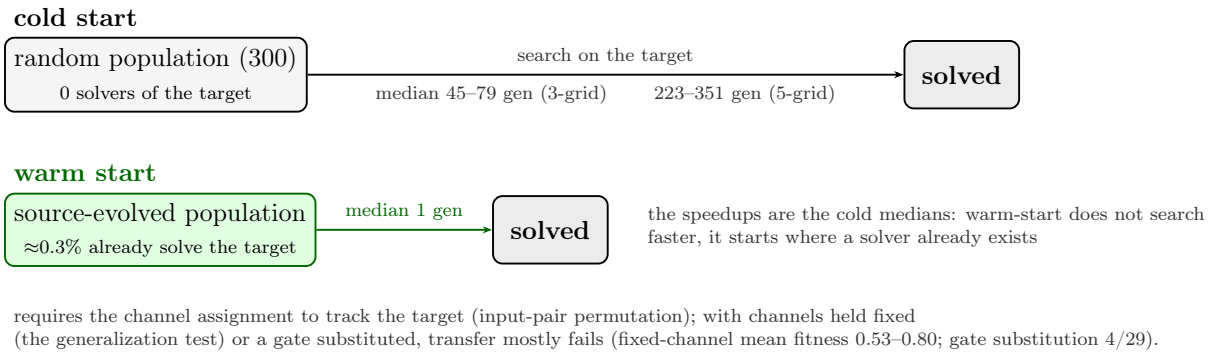


Figure D.4: The warm-start mechanic. Cold start evolves a fresh random population against the target; warm start keeps the entire population evolved on the source compound and continues on the target, with the channel assignment re-wired to the target's gate arrangement. Warm-started runs converge in a median of one generation at both scales, so the reported speedups are the cold-start medians themselves: the warm population already contains what the search is for (Figure D.5), and the first evaluation pass finds it. The mechanic is regime-bound: with the channel assignment held fixed no seed transfers, and under gate substitution only 4 of 29 populations retain one.

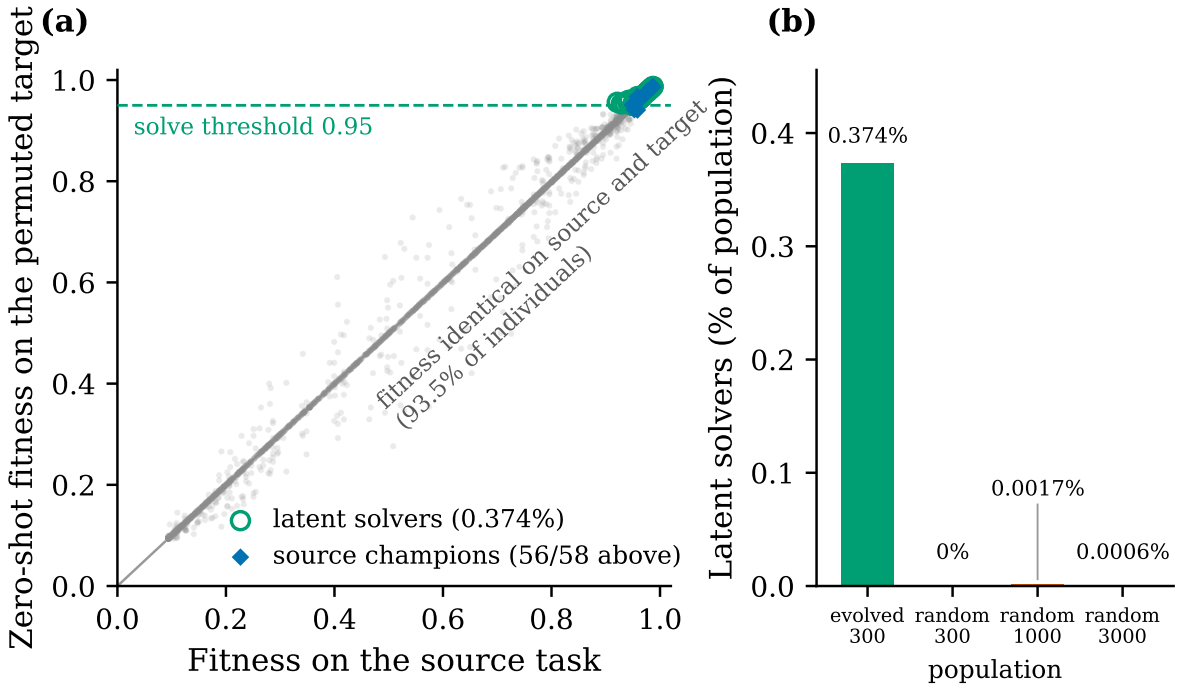


Figure D.5: Warm-start transfer at 3-grid scale, from the population census (29 seeds × 2 permuted targets; all 17,400 individual evaluations shown). (a) Zero-shot fitness on the unseen target against fitness on the source task. For 93.5% of individuals the two are numerically identical (the diagonal): a mapper that conjoins the grid channels is indifferent to which gate each channel carries, so relabeling the channels changes nothing. The latent solvers (0.374%, green) therefore sit at the top of the source ranking, and in 56 of 58 views the source champion (blue) is among them. (b) Latent-solver rate: source-evolved populations versus random initialization at pop 300, 1,000, and 3,000. Random populations contain essentially none, so the transferable representation is a product of source evolution, not of sampling volume. Both panels report the mean rate (0.374%); the median population contains exactly one latent solver in 300 (0.33%), which is the ≈0.3% quoted in the text.