From Static Architectures to Evolutionary Neural Systems: A Bio-Inspired Approach to Artificial General Intelligence

Abstract

Modern deep learning has achieved remarkable success through scaling static architectures like Transformers and Mixture of Experts models. However, these approaches face fundamental limitations in generalization, compositional reasoning, and lifelong learning. We explore a bio-inspired paradigm where neural architectures are encoded as evolvable genotypes that undergo developmental processes to produce phenotypic networks. By combining evolutionary search over architectural space with intra-lifetime learning through gradient descent and local plasticity rules, such systems could potentially address core challenges including catastrophic forgetting, sample efficiency, and abstract reasoning. This document synthesizes concepts from neuroevolution, developmental encoding, and meta-learning to present a comprehensive framework for evolutionary neural architectures, while acknowledging the substantial implementation challenges and the existing body of related research in this domain.

---

1. The Limitations of Current Paradigms

1.1 The Transformer: Triumphs and Constraints

The Transformer architecture revolutionized deep learning by introducing self-attention mechanisms that can model long-range dependencies in sequential data. Models like GPT, BERT, and their successors have demonstrated impressive capabilities in language understanding, generation, and even reasoning tasks. However, several fundamental limitations have become increasingly apparent as we push these models toward more general intelligence.

The first major constraint is computational complexity. Self-attention operates with quadratic complexity relative to sequence length, making it prohibitively expensive to process truly long contexts. While various approximations exist—sparse attention, linear attention variants, and sliding window mechanisms—these typically involve trade-offs that reduce the model's ability to capture certain types of dependencies.

More fundamentally, Transformers exhibit poor compositional generalization. They excel at interpolating within the distribution of their training data but struggle when asked to systematically combine learned concepts in novel ways. This manifests clearly in abstract reasoning benchmarks like the Abstraction and Reasoning Corpus, where models must identify underlying rules from just a few examples and apply them to new situations. Transformers tend to rely on surface-level pattern matching rather than extracting the deep structural

regularities that humans use for such tasks.

The data efficiency problem is equally striking. While a human child can learn a new concept from a handful of examples, Transformers require exposure to massive datasets containing millions or billions of tokens. This suggests that their learning mechanism, while powerful for statistical pattern recognition, lacks the inductive biases and structured learning strategies that enable efficient human cognition.

1.2 Catastrophic Forgetting: The Stability-Plasticity Dilemma

Perhaps the most severe limitation of current architectures is catastrophic forgetting. When a pre-trained neural network is fine-tuned on new tasks, the weight updates required to learn the new information typically overwrite the representations that encoded previous knowledge. The network essentially "forgets" what it learned before, making true lifelong learning impossible.

This problem exists because standard neural networks have no mechanism to protect previously learned information while remaining plastic enough to acquire new knowledge. Various approaches have been proposed—elastic weight consolidation, progressive neural networks, and memory replay systems—but these are patches on a fundamentally static architecture rather than solutions to the underlying problem. The network's structure remains fixed, so all new information must be shoehorned into the same parameter space that already encodes existing knowledge.

1.3 Mixture of Experts: Scaling Without Solving

Mixture of Experts models represent an important architectural innovation. By maintaining multiple specialized sub-networks (experts) and using a gating mechanism to route inputs to the most relevant ones, MoE architectures can scale to trillions of parameters while keeping computational costs manageable. Each forward pass only activates a small subset of the total parameters, providing efficiency gains.

However, MoE fundamentally remains a static architecture. The number of experts, their internal structure, and the routing mechanism are all fixed at design time by human engineers. When the model encounters a truly novel domain—say, a language model trained on text and code suddenly being asked to process protein sequences or musical scores—it cannot spontaneously grow a new specialized expert. It must attempt to map the new domain onto its existing, pre-defined set of experts, which may be fundamentally ill-suited for the task.

The routing mechanisms in MoE also present challenges. Load balancing—ensuring that all experts are utilized efficiently during training—is difficult to achieve. Some experts may become dominant while others remain underused, wasting capacity. The gating network itself adds inference latency and intro-

duces another potential failure mode. Most critically, the specialization that emerges is entirely dependent on the training data distribution and cannot adapt to genuinely out-of-distribution scenarios.

---

2. The Bio-Inspired Alternative: Evolutionary Neural Architectures

2.1 Core Insight: Structure as a Learned, Evolvable Property

The fundamental limitation of current approaches is that architecture is treated as a fixed design choice. Researchers carefully craft model structures, tune hyperparameters, and engineer components, but once training begins, the core organizational structure is frozen. We optimize parameters within a fixed topology.

Biology suggests a radically different approach. Biological neural systems are not designed by an engineer. They grow from a genetic blueprint through a developmental process, guided by both inherited information and environmental signals. Moreover, the genetic blueprint itself evolves over generations through mutation and selection, allowing the space of possible architectures to be searched through evolutionary processes.

This suggests a paradigm where we encode neural architectures as compact genetic programs—genotypes—that undergo a developmental process to produce fully realized networks—phenotypes. These genotypes can then evolve through population-based search, with selection pressure applied based on performance across diverse tasks. Within each organism's lifetime, the phenotypic network learns through both gradient descent and local plasticity rules.

2.2 The Genotype-Phenotype Distinction

In this framework, the genotype is not the neural network itself but rather a high-level specification of how to build one. It encodes information about module types, connectivity patterns, plasticity rules, and developmental programs. Importantly, the genotype is much more compact than the phenotype it produces, creating a powerful compression that enables efficient search through architectural space.

The developmental process that maps genotype to phenotype is itself a computational procedure. It might be a simple interpretation of the genetic code, or it could be a learned neural program that reads the genotype and constructs the phenotype. This developmental mapping can produce networks far more complex than the genotype itself, just as the human genome (approximately 3 billion base pairs encoding roughly 20,000 genes) specifies the development of a brain with 86 billion neurons and 100 trillion synapses.

---

3. System Architecture and Components

3.1 Genotype Encoding Schema

The genotype needs to compactly encode several types of information. First, it must specify the modular components that will make up the network—different types of processing units such as convolutional layers, attention mechanisms, recurrent modules, and memory systems. Second, it must encode connectivity patterns that determine how these modules wire together. Third, it should specify plasticity rules that govern how connections change during the organism's lifetime. Finally, it may encode meta-parameters that control the developmental process itself.

One approach is to represent the genotype as a graph structure where nodes represent modules and edges represent connections. Each node contains parameters specifying the module type and its configuration. Each edge contains information about connection patterns, initialization schemes, and whether the connection is gated or sparse. Additional structures encode the plasticity rules and neuromodulatory signals.

```
class ModuleGene:
    """Encodes a single processing module in the architecture."""
    def __init__(self, module_id, module_type, params):
        self.id = module_id
        self.type = module_type  # e.g., "transformer_block", "conv_patch", "rnn_memory"
        self.params = params  # module-specific configuration

class ConnectionGene:
    """Encodes a connection between two modules."""
    def __init__(self, source_id, target_id, properties):
        self.source = source_id
        self.target = target_id
        self.sparse = properties.get('sparse', False)
        self.sparsity_level = properties.get('sparsity_level', 0.1)
        self.gated = properties.get('gated', False)
        self.recurrent = properties.get('recurrent', False)

class PlasticityRule:
    """Encodes local learning rules for synaptic plasticity."""
    def __init__(self, target_module, rule_type, params):
        self.target = target_module
        self.rule_type = rule_type  # e.g., "hebbian", "oja", "bcm"
        self.learning_rate = params.get('lr', 0.01)
        self.decay_rate = params.get('decay', 0.001)

class Genotype:
    """Complete genetic encoding of a neural architecture."""
```

```
    def __init__(self):
        self.modules = []  # List of ModuleGene objects
        self.connections = []  # List of ConnectionGene objects
        self.plasticity_rules = []  # List of PlasticityRule objects
        self.neuromod_params = {}  # Neuromodulatory signal parameters
        self.dev_program_id = None  # Reference to developmental program

    def add_module(self, module_gene):
        """Add a new module to the genetic specification."""
        self.modules.append(module_gene)

    def add_connection(self, conn_gene):
        """Add a connection specification between modules."""
        self.connections.append(conn_gene)

    def mutate(self, mutation_rate=0.1):
        """Apply random mutations to the genotype."""
        # This would contain logic for various mutation operators
        # - Add/remove modules
        # - Modify module parameters
        # - Add/remove connections
        # - Alter plasticity rules
        pass
```

3.2 Developmental Mapping: Growing the Phenotype

The developmental process takes the genotype and constructs an actual, executable neural network. This mapping can be implemented in several ways. The simplest approach is a deterministic interpretation where each gene directly specifies a component to instantiate. A more sophisticated approach uses a learned developmental network—itself a small neural model—that reads the genotype and generates the phenotype structure.

The developmental program must handle several tasks. It instantiates each module according to its gene specification, determining layer dimensions, activation functions, and initialization schemes. It creates the connectivity structure, forming synaptic weight matrices between modules according to the connection genes. For sparse connections, it determines which specific connections exist. For gated connections, it creates the gating mechanisms. It also sets up the plasticity machinery, associating each module or connection with its specified learning rules.

```
import torch
import torch.nn as nn

class PhenotypicModule(nn.Module):
    """Base class for instantiated neural modules in the phenotype."""
```

```
    def __init__(self, module_gene):
        super().__init__()
        self.gene = module_gene
        self.plasticity_active = False
        self.recent_activity = None

class TransformerBlockModule(PhenotypicModule):
    """A transformer block instantiated from a gene."""
    def __init__(self, module_gene):
        super().__init__(module_gene)
        params = module_gene.params
        self.attention = nn.MultiheadAttention(
            embed_dim=params['d_model'],
            num_heads=params['num_heads']
        )
        self.ffn = nn.Sequential(
            nn.Linear(params['d_model'], params['d_ff']),
            nn.ReLU(),
            nn.Linear(params['d_ff'], params['d_model'])
        )
        self.norm1 = nn.LayerNorm(params['d_model'])
        self.norm2 = nn.LayerNorm(params['d_model'])

    def forward(self, x):
        # Standard transformer block forward pass
        attn_out, _ = self.attention(x, x, x)
        x = self.norm1(x + attn_out)
        ffn_out = self.ffn(x)
        x = self.norm2(x + ffn_out)

        # Store activity for plasticity updates
        if self.plasticity_active:
            self.recent_activity = x.detach()
        return x

class Phenotype(nn.Module):
    """The fully realized neural organism grown from a genotype."""
    def __init__(self, genotype):
        super().__init__()
        self.genotype = genotype
        self.modules = nn.ModuleDict()
        self.connection_graph = {}
        self.plasticity_rules = {}

    def forward(self, x, task_context=None):
        """Execute the network by routing through modules."""
```

```
        # Initialize module activations
        activations = {mod.id: None for mod in self.genotype.modules}

        # Assume first module is input
        input_module_id = self.genotype.modules[0].id
        activations[input_module_id] = x

        # Topological sort to determine execution order
        execution_order = self._topological_sort()

        # Execute each module in order
        for module_id in execution_order:
            if module_id == input_module_id:
                continue

            # Gather inputs from connected modules
            inputs = []
            for source_id in self.connection_graph.get(module_id, []):
                if activations[source_id] is not None:
                    inputs.append(activations[source_id])

            if inputs:
                # Combine inputs (simple concatenation or learned aggregation)
                combined_input = torch.cat(inputs, dim=-1) if len(inputs) > 1 else inputs[0]
                # Execute the module
                activations[module_id] = self.modules[module_id](combined_input)

        # Return output from final module
        output_module_id = self.genotype.modules[-1].id
        return activations[output_module_id]

    def _topological_sort(self):
        """Determine execution order based on connectivity."""
        # Implementation of topological sort on the module graph
        # Returns list of module IDs in execution order
        pass

def develop_phenotype(genotype):
    """
    The developmental process that grows a phenotype from a genotype.
    This is where the genetic blueprint becomes a living neural organism.
    """
    phenotype = Phenotype(genotype)

    # Step 1: Instantiate all modules
    for module_gene in genotype.modules:
```

```
        if module_gene.type == "transformer_block":
            module = TransformerBlockModule(module_gene)
        elif module_gene.type == "conv_patch":
            # Create convolutional module
            module = ConvPatchModule(module_gene)
        elif module_gene.type == "rnn_memory":
            # Create recurrent memory module
            module = RNNMemoryModule(module_gene)
        else:
            raise ValueError(f"Unknown module type: {module_gene.type}")

        phenotype.modules[module_gene.id] = module

    # Step 2: Build connection graph
    for conn_gene in genotype.connections:
        if conn_gene.target not in phenotype.connection_graph:
            phenotype.connection_graph[conn_gene.target] = []
        phenotype.connection_graph[conn_gene.target].append(conn_gene.source)

    # Step 3: Set up plasticity rules
    for plasticity_rule in genotype.plasticity_rules:
        phenotype.plasticity_rules[plasticity_rule.target] = plasticity_rule

    return phenotype
```

3.3 Intra-Lifetime Learning: Plasticity and Neuromodulation

Once the phenotype is grown, it learns through two complementary mechanisms. Standard gradient descent optimization updates weights based on backpropagated error signals, just like conventional neural networks. This provides the strong learning signal needed for complex tasks.

Additionally, local plasticity rules enable fast adaptation without catastrophic forgetting. These rules operate at the level of individual synapses or modules, updating connections based on local activity patterns rather than global error signals. Hebbian learning, for instance, strengthens connections between neurons that fire together. These local rules can enable rapid learning of new associations without disrupting the global weight structure that encodes long-term knowledge.

Neuromodulatory signals control when and where plasticity occurs. These are scalar or vector signals that gate plasticity based on context—perhaps activating during exploration but not during exploitation, or enabling plasticity only in specific modules when learning a new task domain.

```
def apply_plasticity_updates(phenotype, learning_signals):
    """
```

```
    Apply local plasticity rules during or after a forward pass.
    This enables fast adaptation without catastrophic forgetting.
    """
    for module_id, plasticity_rule in phenotype.plasticity_rules.items():
        module = phenotype.modules[module_id]

        if not hasattr(module, 'recent_activity') or module.recent_activity is None:
            continue

        # Get neuromodulatory signal that gates plasticity
        neuromod_signal = learning_signals.get('novelty', 1.0)

        if plasticity_rule.rule_type == "hebbian":
            # Hebbian learning: strengthen connections between co-active units
            with torch.no_grad():
                for name, param in module.named_parameters():
                    if 'weight' in name and param.dim() >= 2:
                        # Simplified Hebbian rule: dW = lr * (pre * post^T)
                        # In practice, this would use stored pre/post activations
                        activity = module.recent_activity
                        if activity is not None:
                            # This is a simplified illustration
                            # Real implementation would properly compute pre/post products
                            hebbian_update = plasticity_rule.learning_rate * torch.randn_lik
                            param.add_(neuromod_signal * hebbian_update)

        elif plasticity_rule.rule_type == "oja":
            # Oja's rule: Hebbian learning with normalization
            with torch.no_grad():
                for name, param in module.named_parameters():
                    if 'weight' in name:
                        # dW = lr * (pre * post - post^2 * W)
                        # Prevents unlimited weight growth
                        pass

        elif plasticity_rule.rule_type == "bcm":
            # Bienenstock-Cooper-Munro rule with sliding threshold
            # Enables stable competition and selectivity
            pass

def meta_learning_step(phenotype, task_batch, optimizer):
    """
    Perform a standard gradient-based learning step.
    This is the 'slow' learning system that works alongside plasticity.
    """
    optimizer.zero_grad()
```

```
    total_loss = 0
    for task_data, task_labels in task_batch:
        # Enable plasticity during forward pass
        for module in phenotype.modules.values():
            module.plasticity_active = True

        predictions = phenotype(task_data)
        loss = nn.functional.cross_entropy(predictions, task_labels)
        total_loss += loss

        # Disable plasticity for backward pass
        for module in phenotype.modules.values():
            module.plasticity_active = False

    total_loss.backward()
    optimizer.step()

    return total_loss.item()
```

3.4 The Evolutionary Outer Loop

The most powerful aspect of this framework is the evolutionary population dynamics. We maintain not a single model but a diverse population of genotypes. Each generation, genotypes are evaluated on a battery of tasks, selected based on fitness, and recombined to produce offspring with mutations.

This evolutionary search explores the space of architectures in a way that is fundamentally different from typical neural architecture search. It's not just optimizing hyperparameters within a predefined search space. Mutations can add entirely new module types, create novel connection patterns, or invent new plasticity rules. Crossover can combine successful components from different genotypes, creating hybrid architectures that inherit strengths from multiple lineages.

The selection pressure is multi-objective. We care about task performance, but also about sample efficiency, computational cost, robustness to distribution shift, and architectural simplicity. Using Pareto-front selection, we can maintain a diverse population where different organisms represent different trade-offs in this multi-dimensional objective space.

```
import random
import copy

class EvolutionaryEngine:
    """Manages the population and evolutionary dynamics."""
    def __init__(self, population_size, mutation_rate, task_suite):
```

```
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.task_suite = task_suite
        self.population = []
        self.generation = 0
        self.fitness_history = []

    def initialize_population(self):
        """Create initial random population of genotypes."""
        for _ in range(self.population_size):
            genotype = self._create_random_genotype()
            self.population.append(genotype)

    def _create_random_genotype(self):
        """Generate a random but valid genotype."""
        genotype = Genotype()

        # Add random modules
        num_modules = random.randint(3, 8)
        module_types = ["transformer_block", "conv_patch", "rnn_memory"]

        for i in range(num_modules):
            module_type = random.choice(module_types)
            params = self._random_params_for_type(module_type)
            genotype.add_module(ModuleGene(f"M{i}", module_type, params))

        # Add random connections (ensure valid DAG)
        for i in range(1, len(genotype.modules)):
            # Connect each module to at least one previous module
            source_idx = random.randint(0, i-1)
            conn = ConnectionGene(
                genotype.modules[source_idx].id,
                genotype.modules[i].id,
                {'sparse': random.random() < 0.3}
            )
            genotype.add_connection(conn)

        return genotype

    def evaluate_population(self):
        """Evaluate all genotypes on the task suite."""
        fitness_scores = {}

        for genotype in self.population:
            try:
                # Develop phenotype from genotype
```

```
                phenotype = develop_phenotype(genotype)

                # Evaluate on multiple tasks
                scores = self._evaluate_phenotype(phenotype)

                # Compute multi-objective fitness
                fitness = {
                    'accuracy': scores['accuracy'],
                    'sample_efficiency': scores['sample_efficiency'],
                    'compute_cost': -scores['flops'],  # Negative because we want to minimi
                    'generalization': scores['ood_accuracy'] - scores['accuracy']
                }

                fitness_scores[id(genotype)] = fitness

            except Exception as e:
                # Invalid genotype produces invalid phenotype
                fitness_scores[id(genotype)] = {k: float('-inf') for k in ['accuracy', 'samp

        return fitness_scores

    def _evaluate_phenotype(self, phenotype):
        """Run phenotype through task battery."""
        scores = {
            'accuracy': 0.0,
            'sample_efficiency': 0.0,
            'ood_accuracy': 0.0,
            'flops': 0.0
        }

        # This would contain actual evaluation logic on tasks like ARC, HLE, etc.
        # For each task:
        #   - Train with limited samples (measure sample efficiency)
        #   - Test on in-distribution data (measure accuracy)
        #   - Test on out-of-distribution data (measure generalization)
        #   - Count FLOPs (measure compute cost)

        return scores

    def select_parents(self, fitness_scores):
        """Select parents using multi-objective Pareto selection."""
        # Compute Pareto front
        pareto_front = self._compute_pareto_front(fitness_scores)

        # Select from Pareto front with some randomness for diversity
        num_parents = self.population_size // 2
```

```
        parents = random.choices(pareto_front, k=num_parents)

        return parents

    def _compute_pareto_front(self, fitness_scores):
        """Find non-dominated individuals in multi-objective space."""
        pareto_front = []

        for genotype in self.population:
            genotype_fitness = fitness_scores[id(genotype)]
            is_dominated = False

            for other_genotype in self.population:
                if genotype is other_genotype:
                    continue
                other_fitness = fitness_scores[id(other_genotype)]

                # Check if other dominates genotype
                if self._dominates(other_fitness, genotype_fitness):
                    is_dominated = True
                    break

            if not is_dominated:
                pareto_front.append(genotype)

        return pareto_front if pareto_front else self.population[:10]

    def _dominates(self, fitness_a, fitness_b):
        """Check if fitness_a dominates fitness_b (better on all objectives)."""
        better_on_all = True
        strictly_better_on_one = False

        for objective in fitness_a.keys():
            if fitness_a[objective] < fitness_b[objective]:
                better_on_all = False
            if fitness_a[objective] > fitness_b[objective]:
                strictly_better_on_one = True

        return better_on_all and strictly_better_on_one

    def reproduce(self, parents):
        """Create next generation through crossover and mutation."""
        offspring = []

        while len(offspring) < self.population_size:
            # Select two parents
```

```
            parent1, parent2 = random.sample(parents, 2)

            # Crossover
            child = self._crossover(parent1, parent2)

            # Mutation
            if random.random() < self.mutation_rate:
                child = self._mutate(child)

            offspring.append(child)

        return offspring

    def _crossover(self, parent1, parent2):
        """Recombine genetic material from two parents."""
        child = Genotype()

        # Module-level crossover: take modules from both parents
        all_modules = parent1.modules + parent2.modules
        selected_modules = random.sample(all_modules,
                                         k=min(len(all_modules),
                                               random.randint(3, 8)))

        for module in selected_modules:
            child.add_module(copy.deepcopy(module))

        # Connection crossover: take compatible connections
        for conn in parent1.connections + parent2.connections:
            # Only add if both source and target exist in child
            if any(m.id == conn.source for m in child.modules) and \
               any(m.id == conn.target for m in child.modules):
                child.add_connection(copy.deepcopy(conn))

        return child

    def _mutate(self, genotype):
        """Apply random mutations to genotype."""
        mutation_type = random.choice([
            'add_module', 'remove_module', 'modify_module',
            'add_connection', 'remove_connection',
            'modify_plasticity'
        ])

        if mutation_type == 'add_module':
            new_module = ModuleGene(
                f"M{len(genotype.modules)}",
```

```
                random.choice(["transformer_block", "conv_patch", "rnn_memory"]),
                self._random_params_for_type("transformer_block")
            )
            genotype.add_module(new_module)

        elif mutation_type == 'remove_module' and len(genotype.modules) > 3:
            genotype.modules.pop(random.randint(1, len(genotype.modules)-2))

        elif mutation_type == 'modify_module' and genotype.modules:
            module = random.choice(genotype.modules)
            # Mutate module parameters
            for key in module.params:
                if isinstance(module.params[key], (int, float)):
                    module.params[key] *= random.uniform(0.8, 1.2)

        # Similar logic for connection and plasticity mutations

        return genotype

    def run_evolution(self, num_generations):
        """Execute the evolutionary process."""
        self.initialize_population()

        for gen in range(num_generations):
            self.generation = gen
            print(f"\n=== Generation {gen} ===")

            # Evaluate all organisms
            fitness_scores = self.evaluate_population()
            self.fitness_history.append(fitness_scores)

            # Report best fitness
            best_fitness = max(fitness_scores.values(),
                               key=lambda f: f['accuracy'])
            print(f"Best accuracy: {best_fitness['accuracy']:.3f}")

            # Selection
            parents = self.select_parents(fitness_scores)
            print(f"Selected {len(parents)} parents from Pareto front")

            # Reproduction
            self.population = self.reproduce(parents)

            # Optionally save best genotypes
            self._save_checkpoint(gen)
```

```
        return self.population

    def _random_params_for_type(self, module_type):
        """Generate random but reasonable parameters for a module type."""
        if module_type == "transformer_block":
            return {
                'd_model': random.choice([256, 512, 768]),
                'num_heads': random.choice([4, 8, 12]),
                'd_ff': random.choice([1024, 2048, 3072])
            }
        # Similar for other types
        return {}

    def _save_checkpoint(self, generation):
        """Save population state for reproducibility."""
        pass
```

---

4. Experimental Validation Strategy

4.1 Task Suite Design

To properly evaluate evolutionary neural architectures, we need a diverse battery of tasks that test different aspects of intelligence. The Abstraction and Reasoning Corpus provides abstract visual reasoning problems that require identifying underlying rules. The Humanity's Last Exam and similar benchmarks test knowledge, reasoning, and problem-solving across domains. We should also include algorithmic tasks, few-shot learning challenges, continual learning scenarios, and out-of-distribution robustness tests.

The key is that no single task should dominate the fitness function. We want to select for general problem-solving ability, not overfitting to a particular benchmark. Multi-objective optimization naturally handles this by maintaining diverse solutions that excel at different task combinations.

4.2 Baselines and Ablations

Rigorous evaluation requires proper baselines. We should compare against state-of-the-art Transformers, MoE models, and existing neural architecture search methods. Critically, we need ablation studies that isolate the contribution of each component. What happens with evolution but no plasticity? With plasticity but no evolution? With random architectures versus evolved ones? These ablations help us understand which mechanisms actually provide value.

4.3 Metrics Beyond Accuracy

Task accuracy is important but insufficient. We must measure sample efficiency—how many examples are needed to reach a given performance level. We should measure generalization gap—the difference between in-distribution and out-of-distribution performance. Computational cost matters, measured in FLOPs or wall-clock time. Architectural complexity can be quantified by counting modules and connections. Transfer learning ability can be assessed by training on one task set and evaluating on another.

---

5. Challenges and Open Questions

5.1 Computational Cost

The elephant in the room is computational expense. Evolutionary methods require evaluating many organisms per generation, and each organism requires training and evaluation on multiple tasks. This can quickly become prohibitive. However, several mitigation strategies exist. We can use surrogate models to predict fitness without full evaluation. We can employ early stopping, terminating unpromising organisms quickly. We can parallelize heavily, evaluating many organisms simultaneously. We can also start with toy tasks and scale up gradually, learning from small-scale experiments before committing to expensive large-scale runs.

5.2 Reward Hacking and Shortcuts

Evolutionary systems are notorious for finding unexpected shortcuts. When optimizing for a proxy metric, evolution may discover ways to maximize that metric without actually solving the underlying problem. This is the reward hacking problem, and it's particularly acute in open-ended search where the system has enormous creative freedom.

For instance, if we're not careful with how we measure sample efficiency, evolution might produce networks that memorize training data through simple lookup tables rather than learning generalizable representations. If we measure only final accuracy without considering the quality of intermediate representations, we might get brittle solutions that work on the test set through statistical flukes.

The solution requires carefully designed evaluation protocols. We need diverse, held-out test sets that the evolutionary process never sees directly. We should evaluate on genuinely novel task variations to prevent overfitting. We can include adversarial perturbations to test robustness. Most importantly, we need human evaluation and interpretability analysis to ensure that high-scoring solutions are actually solving problems in meaningful ways rather than exploiting evaluation artifacts.

5.3 The Credit Assignment Problem

When an evolved organism performs well, which parts of its genotype are responsible? This credit assignment problem becomes even more complex when we have both evolution (operating on genotypes across generations) and learning (operating on phenotypes within lifetimes). We need mechanisms to identify which genetic variations led to improved performance so that selection can amplify those variations.

Several approaches can help. We can track lineages to see which genetic features are preserved in successful organisms. We can use techniques from differentiable NAS to estimate gradients with respect to discrete architectural choices. We can employ information-theoretic measures to identify which genotypic features correlate with fitness improvements. None of these completely solves the problem, but they provide useful signals for guiding evolution.

5.4 Ensuring Reproducibility

Evolutionary experiments can be notoriously difficult to reproduce. Small changes in initialization, random seeds, or evaluation protocols can lead to dramatically different outcomes. This is especially problematic because evolution is inherently stochastic and may follow different trajectories on different runs.

Rigorous experimental practice is essential. We must log complete genotypes, not just fitness scores. We need to save checkpoints of entire populations at regular intervals. Random seeds must be carefully controlled and reported. The task suite and evaluation protocol must be precisely specified and version-controlled. Ideally, we should run multiple independent evolutionary runs and report statistics across runs rather than cherry-picking the best outcome.

---

6. Implementation Roadmap

6.1 Phase 1: Proof of Concept (Months 1-3)

The first phase should establish that the basic machinery works. Start with extremely simple tasks—algorithmic problems like copying sequences, sorting small lists, or simple pattern completion. Implement a minimal genotype encoding with just 2-3 module types. Create a simple developmental process that directly interprets genes into a PyTorch model. Implement basic mutation and crossover operators.

The goal here isn't to achieve state-of-the-art performance. It's to verify that genotypes can evolve, phenotypes can learn, and the system produces diverse solutions. We should see fitness improving over generations, even on toy problems. We should observe that evolved architectures differ from random ones in meaningful ways.

```python
# Phase 1: Minimal viable implementation
def proof_of_concept_experiment():
    """Run a minimal evolutionary experiment on toy tasks."""

    # Simple task: learn to copy sequences of varying length
    def copy_task_generator():
        length = random.randint(5, 20)
        sequence = torch.randint(0, 10, (length,))
        return sequence, sequence

    # Minimal task suite
    task_suite = {
        'copy': copy_task_generator,
        'reverse': lambda: reverse_task_generator(),
        'sort': lambda: sort_task_generator()
    }

    # Small-scale evolution
    engine = EvolutionaryEngine(
        population_size=20,
        mutation_rate=0.3,
        task_suite=task_suite
    )

    # Run for just 10 generations to verify mechanism
    final_population = engine.run_evolution(num_generations=10)

    # Analyze results
    print("Proof of concept complete!")
    print(f"Final population diversity: {measure_diversity(final_population)}")
    print(f"Best fitness achieved: {max(engine.fitness_history[-1].values())}")

    return final_population

def measure_diversity(population):
    """Measure genotypic diversity in population."""
    # Compare genotypes to see how different they are
    diversity_score = 0.0
    for i, g1 in enumerate(population):
        for g2 in population[i+1:]:
            diversity_score += genotype_distance(g1, g2)
    return diversity_score / (len(population) * (len(population) - 1) / 2)

def genotype_distance(g1, g2):
    """Measure distance between two genotypes."""
    # Count differences in modules, connections, etc.
```

```
    module_diff = abs(len(g1.modules) - len(g2.modules))
    conn_diff = abs(len(g1.connections) - len(g2.connections))
    return module_diff + conn_diff
```

6.2 Phase 2: Plasticity and Meta-Learning (Months 4-6)

Once basic evolution works, add intra-lifetime learning mechanisms. Implement local plasticity rules alongside gradient descent. Create neuromodulatory signals that gate plasticity. Design tasks that specifically benefit from fast adaptation—few-shot learning problems where the phenotype must quickly adapt to new examples within a single episode.

This phase tests whether the combination of evolution and plasticity provides advantages over either alone. We should see organisms evolving plasticity rules that enable faster learning on novel tasks. We might observe specialization where different modules have different plasticity properties—some rapidly adaptive, others stable repositories of long-term knowledge.

```
# Phase 2: Adding plasticity mechanisms
class PlasticityExperiment:
    """Experiments testing evolution of learning rules."""

    def few_shot_learning_task(self, n_shot=5):
        """Generate a few-shot learning episode."""
        # Create a novel classification task
        n_classes = random.randint(3, 7)
        support_set = []
        query_set = []

        for class_id in range(n_classes):
            # Generate class-specific patterns
            class_pattern = self._generate_pattern()

            # Support examples (for learning)
            for _ in range(n_shot):
                example = class_pattern + torch.randn_like(class_pattern) * 0.1
                support_set.append((example, class_id))

            # Query examples (for testing)
            for _ in range(5):
                example = class_pattern + torch.randn_like(class_pattern) * 0.1
                query_set.append((example, class_id))

        return support_set, query_set

    def evaluate_with_plasticity(self, phenotype, num_episodes=100):
        """Evaluate how well plasticity enables fast learning."""
```

```
        accuracies = []

        for episode in range(num_episodes):
            support_set, query_set = self.few_shot_learning_task()

            # Enable plasticity for support set
            for module in phenotype.modules.values():
                module.plasticity_active = True

            # "Learn" from support set
            for example, label in support_set:
                prediction = phenotype(example.unsqueeze(0))
                # Plasticity updates happen automatically during forward pass

            # Disable plasticity and test on query set
            for module in phenotype.modules.values():
                module.plasticity_active = False

            correct = 0
            for example, label in query_set:
                prediction = phenotype(example.unsqueeze(0))
                if prediction.argmax() == label:
                    correct += 1

            accuracies.append(correct / len(query_set))

        return sum(accuracies) / len(accuracies)

    def _generate_pattern(self):
        """Generate a random pattern for a class."""
        return torch.randn(64)  # 64-dim pattern vector
```

6.3 Phase 3: Scaling to Real Benchmarks (Months 7-12)

With the core mechanisms validated, scale up to serious benchmarks. Implement efficient infrastructure for parallel evaluation. Create proper data loaders for ARC, design evaluation protocols for the Humanity's Last Exam, and include diverse reasoning tasks. Increase population sizes and run longer evolutionary timelines.

This is where we discover whether the approach actually delivers on its promise. Can evolved architectures match or exceed hand-designed models? Does plasticity enable better continual learning? Do we see novel architectural patterns emerging that humans wouldn't have designed? This phase requires significant computational resources but provides the empirical evidence needed to validate the approach.

```python
# Phase 3: Full-scale benchmark evaluation
class BenchmarkEvaluation:
    """Infrastructure for evaluating on real AGI benchmarks."""

    def __init__(self):
        self.arc_dataset = self.load_arc_dataset()
        self.hle_dataset = self.load_hle_dataset()
        self.other_tasks = self.load_additional_benchmarks()

    def load_arc_dataset(self):
        """Load Abstraction and Reasoning Corpus."""
        # Implementation would load actual ARC tasks
        # Each task has training examples and test examples
        return None  # Placeholder

    def evaluate_on_arc(self, phenotype, num_tasks=50):
        """
        Evaluate phenotype on ARC tasks.

        ARC requires:
        1. Processing visual grids (input/output examples)
        2. Identifying the underlying transformation rule
        3. Applying the rule to new test inputs
        """
        correct = 0
        total = 0

        for task in random.sample(self.arc_dataset, num_tasks):
            # Each task has training examples showing the rule
            training_examples = task['train']
            test_input = task['test']['input']
            test_output = task['test']['output']

            # Let the phenotype "learn" from training examples
            # using its plasticity mechanisms
            for train_input, train_output in training_examples:
                _ = phenotype(self._encode_grid(train_input))
                # Plasticity updates based on seeing input-output pair

            # Now test if it can apply the learned rule
            predicted_output = phenotype(self._encode_grid(test_input))
            predicted_grid = self._decode_grid(predicted_output)

            if self._grids_match(predicted_grid, test_output):
                correct += 1
            total += 1
```

```
        return correct / total if total > 0 else 0.0

    def evaluate_on_hle(self, phenotype):
        """Evaluate on Humanity's Last Exam questions."""
        # HLE covers diverse knowledge and reasoning
        correct = 0
        total = 0

        for question in self.hle_dataset:
            # Questions span math, science, logic, etc.
            answer = self._query_phenotype(phenotype, question)
            if self._check_answer(answer, question['correct_answer']):
                correct += 1
            total += 1

        return correct / total if total > 0 else 0.0

    def comprehensive_evaluation(self, phenotype):
        """Run complete benchmark suite."""
        results = {
            'arc_accuracy': self.evaluate_on_arc(phenotype),
            'hle_accuracy': self.evaluate_on_hle(phenotype),
            'sample_efficiency': self.measure_sample_efficiency(phenotype),
            'ood_robustness': self.measure_ood_performance(phenotype),
            'compute_cost': self.measure_compute_cost(phenotype)
        }

        # Aggregate into multi-objective fitness
        fitness = {
            'performance': (results['arc_accuracy'] + results['hle_accuracy']) / 2,
            'efficiency': results['sample_efficiency'],
            'robustness': results['ood_robustness'],
            'cost': -results['compute_cost']  # Negative because we minimize cost
        }

        return results, fitness

    def _encode_grid(self, grid):
        """Convert ARC grid to tensor representation."""
        # ARC grids are typically small (up to 30x30) with integer values
        return torch.tensor(grid, dtype=torch.float32)

    def _decode_grid(self, tensor):
        """Convert tensor back to ARC grid format."""
        return tensor.argmax(dim=-1).cpu().numpy()
```

```
    def _grids_match(self, predicted, target):
        """Check if predicted grid matches target."""
        return (predicted == target).all()
```

---

7. Connections to Existing Research

7.1 Neuroevolution and NEAT

The ideas presented here build on a rich history of neuroevolution research. NeuroEvolution of Augmenting Topologies (NEAT), introduced by Kenneth Stanley in 2002, demonstrated that evolving both network topology and weights could outperform fixed-architecture approaches on certain tasks. NEAT introduced innovations like speciation to protect innovation and historical markings to enable meaningful crossover.

Our approach extends these ideas by adding developmental processes that map compact genotypes to complex phenotypes, by incorporating modern deep learning modules rather than simple perceptrons, and by combining evolution with gradient-based learning and local plasticity. The multi-objective optimization and focus on general intelligence benchmarks also distinguishes this from classic neuroevolution work that typically targeted specific control or game-playing tasks.

7.2 Neural Architecture Search

Neural Architecture Search has become a major research area, with methods ranging from reinforcement learning-based search to gradient-based approaches like DARTS. However, most NAS methods search within constrained spaces defined by human designers—they might optimize layer counts or connection patterns but don't fundamentally change the architectural paradigm.

The evolutionary approach here is more open-ended. Mutations can introduce qualitatively new module types or connection patterns that weren't explicitly programmed into the search space. The developmental encoding also provides a richer representation than typical NAS encodings, potentially enabling discovery of more sophisticated hierarchical architectures.

7.3 Meta-Learning and Learning-to-Learn

Meta-learning research aims to create systems that learn how to learn—that improve their learning algorithm based on experience across multiple tasks. Model-Agnostic Meta-Learning (MAML) and similar approaches optimize for rapid adaptation to new tasks. Learned plasticity rules can be seen as a form of meta-learning where the evolutionary process discovers learning algorithms rather than just learned parameters.

The key distinction is that meta-learning typically occurs within a fixed architecture, while evolutionary neural systems can modify the architecture itself based on what learning strategies work across tasks. This provides a broader space for adaptation.

7.4 Continual Learning and Catastrophic Forgetting

The continual learning community has developed numerous techniques to combat catastrophic forgetting: elastic weight consolidation, progressive neural networks, memory replay, and dynamic architectures. The evolutionary approach offers a different angle: rather than protecting existing weights, grow new modules for new tasks while preserving the genetic encoding of old capabilities.

This is closer to how biological brains handle lifelong learning—through structural plasticity and modularity rather than just synaptic weight protection. The combination of evolution, development, and local plasticity rules may provide a more robust solution to the stability-plasticity dilemma.

---

8. Limitations and Realistic Expectations

8.1 This Is Not a Silver Bullet

It's crucial to maintain realistic expectations. Evolutionary neural architectures are not guaranteed to outperform well-engineered static models on all tasks. Evolution is a powerful search mechanism, but it requires enormous amounts of evaluation and doesn't guarantee finding global optima. The no-free-lunch theorem tells us that no single learning algorithm dominates across all possible problems.

What this approach offers is a different set of trade-offs. It may excel at tasks requiring compositional generalization, continual learning, and transfer across diverse domains—precisely the areas where current static architectures struggle most. But it will likely be computationally expensive and may underperform specialized architectures on narrow, well-defined tasks where human engineering insight is available.

8.2 Implementation Complexity

Building a working evolutionary neural system is significantly more complex than training a standard neural network. It requires infrastructure for population management, parallel evaluation, genetic operators, developmental processes, and multi-objective optimization. Debugging is challenging because failures can occur at multiple levels—genetic encoding, developmental mapping, phenotypic learning, or evolutionary dynamics.

This complexity means that early implementations will likely be buggy, inefficient, and frustrating. Extensive iteration and refinement will be necessary.

The path from concept to working system is long and requires substantial engineering effort alongside theoretical insight.

8.3 Computational Requirements

Even with efficient implementations, evolutionary methods are computationally demanding. Evaluating hundreds or thousands of candidate architectures, each requiring training on multiple tasks, quickly consumes enormous computational budgets. While there are strategies to mitigate this—surrogate models, early stopping, transfer learning—the fundamental requirement for broad exploration means that evolution won't be the most sample-efficient approach for any single task.

The computational cost may be justified if the resulting organisms generalize far better than static models or can continually learn across tasks. But researchers pursuing this direction need access to significant computational resources or must be very clever about reducing evaluation costs.

---

9. Conclusion and Future Directions

The limitations of current AI paradigms—catastrophic forgetting, poor compositional reasoning, data inefficiency—point toward the need for architectural innovation rather than just scaling. Bio-inspired evolutionary neural architectures offer a compelling direction by treating structure as a learned, evolvable property rather than a fixed design choice.

The framework presented here combines several powerful ideas: genetic encoding of architectures, developmental processes that grow complex networks from compact specifications, local plasticity rules that enable fast adaptation, and evolutionary search over populations of organisms. Together, these mechanisms could potentially address many current limitations.

However, this remains largely a research program rather than a proven solution. Significant theoretical and engineering work is needed to determine whether these ideas can deliver on their promise. Key questions remain open:

- Can evolutionary search discover genuinely novel architectural principles that humans wouldn't conceive?
- Does the combination of evolution and plasticity provide substantial advantages over either mechanism alone?
- Can evolved organisms achieve state-of-the-art performance on demanding benchmarks like ARC while also demonstrating better generalization and continual learning?
- What are the minimal computational requirements to make this approach practical?

Answering these questions requires careful empirical work, rigorous evaluation, and honest reporting of both successes and failures. The path forward involves starting small, validating core mechanisms, and scaling up incrementally while maintaining scientific rigor.

For researchers inspired by these ideas, the advice is to ground enthusiasm in practical experimentation. Build working prototypes, test specific hypotheses, compare against proper baselines, and be honest about limitations. The goal is not to claim revolutionary breakthroughs but to make steady, well-validated progress toward more general, adaptable AI systems.

The vision of neural networks that can grow, learn, and evolve their own structure is compelling. Whether it represents a viable path to artificial general intelligence remains to be discovered through rigorous research. But the journey of exploration itself will deepen our understanding of learning, adaptation, and intelligence—and that understanding has value regardless of where the path ultimately leads.

References and Further Reading

Neuroevolution: - Stanley, K.O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. Evolutionary Computation. - Stanley, K.O., et al. (2019). Designing Neural Networks through Neuroevolution. Nature Machine Intelligence.

Neural Architecture Search: - Zoph, B., & Le, Q.V. (2017). Neural Architecture Search with Reinforcement Learning. ICLR. - Liu, H., et al. (2019). DARTS: Differentiable Architecture Search. ICLR. - Real, E., et al. (2019). Regularized Evolution for Image Classifier Architecture Search. AAAI.

Meta-Learning: - Finn, C., et al. (2017). Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. ICML. - Hospedales, T., et al. (2021). Meta-Learning in Neural Networks: A Survey. IEEE PAMI.

Continual Learning: - Kirkpatrick, J., et al. (2017). Overcoming Catastrophic Forgetting in Neural Networks. PNAS. - Rusu, A., et al. (2016). Progressive Neural Networks. arXiv:1606.04671.

Developmental Encoding: - Stanley, K.O. (2007). Compositional Pattern Producing Networks. Genetic Programming and Evolvable Machines. - Clune, J., et al. (2011). On the Performance of Indirect Encoding Across the Continuum of Regularity. IEEE Transactions on Evolutionary Computation.

Plasticity and Biological Learning: - Miconi, T., et al. (2018). Differentiable Plasticity: Training Plastic Neural Networks with Backpropagation. ICML. - Najarro, E., & Risi, S. (2020). Meta-Learning Through Hebbian Plasticity in Random Networks. NeurIPS.

Quality Diversity and Open-Ended Evolution: - Mouret, J.B., & Clune, J. (2015). Illuminating the Space of Possible. Artificial Life. - Lehman, J., & Stanley, K.O. (2011). Abandoning Objectives: Evolution Through the Search for Novelty Alone. Evolutionary Computation.

Appendix A: Complete Implementation Example

Below is a more complete, runnable implementation that integrates all the key components discussed. This code is intended as an illustrative template rather than production-ready software.

```
import torch
import torch.nn as nn
import torch.optim as optim
import random
import copy
from typing import List, Dict, Tuple, Any
from dataclasses import dataclass
import numpy as np

# ==============================================================================
# GENOTYPE COMPONENTS
# ==============================================================================

@dataclass
class ModuleGene:
    """Genetic specification for a neural module."""
    id: str
    type: str
    params: Dict[str, Any]

@dataclass
class ConnectionGene:
    """Genetic specification for connections between modules."""
    source: str
    target: str
    sparse: bool = False
    sparsity_level: float = 0.1
    gated: bool = False

@dataclass
class PlasticityRule:
    """Specification for local learning rules."""
    target_module: str
    rule_type: str
```

```
    learning_rate: float
    decay: float = 0.0

class Genotype:
    """Complete genetic encoding of neural architecture."""

    def __init__(self):
        self.modules: List[ModuleGene] = []
        self.connections: List[ConnectionGene] = []
        self.plasticity_rules: List[PlasticityRule] = []
        self.fitness_history: List[float] = []

    def add_module(self, module: ModuleGene):
        self.modules.append(module)

    def add_connection(self, conn: ConnectionGene):
        self.connections.append(conn)

    def add_plasticity_rule(self, rule: PlasticityRule):
        self.plasticity_rules.append(rule)

    def clone(self):
        """Create deep copy of genotype."""
        return copy.deepcopy(self)

    def mutate(self, mutation_rate: float = 0.1):
        """Apply mutations to this genotype."""
        mutations = []

        if random.random() < mutation_rate:
            mutation_type = random.choice([
                'add_module', 'remove_module', 'modify_params',
                'add_connection', 'modify_plasticity'
            ])
            mutations.append(mutation_type)

            if mutation_type == 'add_module' and len(self.modules) < 10:
                new_id = f"M{len(self.modules)}"
                module_type = random.choice(['linear', 'attention', 'rnn'])
                params = self._random_params(module_type)
                self.add_module(ModuleGene(new_id, module_type, params))

            elif mutation_type == 'remove_module' and len(self.modules) > 3:
                idx = random.randint(1, len(self.modules) - 2)
                removed = self.modules.pop(idx)
                # Remove connections involving this module
```

```
                    self.connections = [c for c in self.connections
                                        if c.source != removed.id and c.target != removed.id]

                elif mutation_type == 'modify_params' and self.modules:
                    module = random.choice(self.modules)
                    for key in module.params:
                        if isinstance(module.params[key], (int, float)):
                            module.params[key] *= random.uniform(0.8, 1.2)

                elif mutation_type == 'add_connection' and len(self.modules) > 1:
                    source = random.choice(self.modules[:-1])
                    target = random.choice(self.modules[1:])
                    if not any(c.source == source.id and c.target == target.id
                               for c in self.connections):
                        self.add_connection(ConnectionGene(source.id, target.id))

            return mutations

        def _random_params(self, module_type: str) -> Dict:
            if module_type == 'linear':
                return {'dim': random.choice([64, 128, 256, 512])}
            elif module_type == 'attention':
                return {
                    'dim': random.choice([128, 256, 512]),
                    'heads': random.choice([2, 4, 8])
                }
            elif module_type == 'rnn':
                return {
                    'hidden_dim': random.choice([64, 128, 256]),
                    'num_layers': random.choice([1, 2])
                }
            return {}

    # ============================================================================
    # PHENOTYPE MODULES
    # ============================================================================

    class LinearModule(nn.Module):
        """Simple linear transformation module."""
        def __init__(self, gene: ModuleGene, input_dim: int):
            super().__init__()
            self.gene = gene
            dim = gene.params.get('dim', 128)
            self.layer = nn.Linear(input_dim, dim)
            self.activation = nn.ReLU()
            self.recent_activity = None
```

```
        self.plasticity_active = False

    def forward(self, x):
        out = self.activation(self.layer(x))
        if self.plasticity_active:
            self.recent_activity = out.detach().clone()
        return out

class AttentionModule(nn.Module):
    """Self-attention module."""
    def __init__(self, gene: ModuleGene, input_dim: int):
        super().__init__()
        self.gene = gene
        dim = gene.params.get('dim', 128)
        heads = gene.params.get('heads', 4)
        self.attention = nn.MultiheadAttention(dim, heads, batch_first=True)
        self.norm = nn.LayerNorm(dim)
        self.projection = nn.Linear(input_dim, dim) if input_dim != dim else nn.Identity()
        self.recent_activity = None
        self.plasticity_active = False

    def forward(self, x):
        if x.dim() == 2:
            x = x.unsqueeze(1)
        x = self.projection(x)
        attn_out, _ = self.attention(x, x, x)
        out = self.norm(x + attn_out)
        if self.plasticity_active:
            self.recent_activity = out.detach().clone()
        return out.squeeze(1) if out.size(1) == 1 else out

class RNNModule(nn.Module):
    """Recurrent module for sequential processing."""
    def __init__(self, gene: ModuleGene, input_dim: int):
        super().__init__()
        self.gene = gene
        hidden_dim = gene.params.get('hidden_dim', 128)
        num_layers = gene.params.get('num_layers', 1)
        self.rnn = nn.GRU(input_dim, hidden_dim, num_layers, batch_first=True)
        self.recent_activity = None
        self.plasticity_active = False

    def forward(self, x):
        if x.dim() == 2:
            x = x.unsqueeze(1)
        out, _ = self.rnn(x)
```

```
        if self.plasticity_active:
            self.recent_activity = out.detach().clone()
        return out[:, -1, :] if out.size(1) > 1 else out.squeeze(1)

# ============================================================================
# PHENOTYPE: THE GROWN ORGANISM
# ============================================================================

class Phenotype(nn.Module):
    """The actual neural network grown from a genotype."""

    def __init__(self, genotype: Genotype, input_dim: int = 64, output_dim: int = 10):
        super().__init__()
        self.genotype = genotype
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.modules_dict = nn.ModuleDict()
        self.connection_graph = {}
        self._build_from_genotype()

    def _build_from_genotype(self):
        """Construct the neural network from genetic specification."""
        current_dim = self.input_dim

        for module_gene in self.genotype.modules:
            if module_gene.type == 'linear':
                module = LinearModule(module_gene, current_dim)
                current_dim = module_gene.params.get('dim', 128)
            elif module_gene.type == 'attention':
                module = AttentionModule(module_gene, current_dim)
                current_dim = module_gene.params.get('dim', 128)
            elif module_gene.type == 'rnn':
                module = RNNModule(module_gene, current_dim)
                current_dim = module_gene.params.get('hidden_dim', 128)
            else:
                raise ValueError(f"Unknown module type: {module_gene.type}")

            self.modules_dict[module_gene.id] = module

        # Build connection graph
        for conn in self.genotype.connections:
            if conn.target not in self.connection_graph:
                self.connection_graph[conn.target] = []
            self.connection_graph[conn.target].append(conn.source)

        # Add output layer
```

```
        self.output_layer = nn.Linear(current_dim, self.output_dim)

    def forward(self, x):
        """Execute forward pass through the evolved architecture."""
        activations = {}

        # Process modules in order
        for i, module_gene in enumerate(self.genotype.modules):
            if i == 0:
                # First module gets input directly
                activations[module_gene.id] = self.modules_dict[module_gene.id](x)
            else:
                # Gather inputs from connections
                inputs = []
                for source_id in self.connection_graph.get(module_gene.id, []):
                    if source_id in activations:
                        inputs.append(activations[source_id])

                if inputs:
                    # Combine inputs (simple concatenation or addition)
                    if len(inputs) == 1:
                        combined = inputs[0]
                    else:
                        # Resize to match dimensions if needed
                        min_dim = min(inp.size(-1) for inp in inputs)
                        resized = [inp[..., :min_dim] for inp in inputs]
                        combined = torch.stack(resized).mean(dim=0)

                    activations[module_gene.id] = self.modules_dict[module_gene.id](combined

        # Get final activation
        final_module_id = self.genotype.modules[-1].id
        final_activation = activations.get(final_module_id, x)

        return self.output_layer(final_activation)

    def apply_plasticity(self):
        """Apply local plasticity rules to update weights."""
        for rule in self.genotype.plasticity_rules:
            if rule.target_module in self.modules_dict:
                module = self.modules_dict[rule.target_module]
                if module.recent_activity is not None:
                    self._apply_hebbian_update(module, rule)

    def _apply_hebbian_update(self, module, rule):
        """Apply Hebbian learning rule."""
```

```
        with torch.no_grad():
            for name, param in module.named_parameters():
                if 'weight' in name and param.grad is not None:
                    # Simplified Hebbian: strengthen active connections
                    activity_scale = module.recent_activity.abs().mean()
                    hebbian_delta = rule.learning_rate * activity_scale * torch.randn_like(p
                    param.add_(hebbian_delta)

                    # Decay to prevent runaway growth
                    if rule.decay > 0:
                        param.mul_(1 - rule.decay)

# ============================================================================
# EVOLUTIONARY ENGINE
# ============================================================================

class EvolutionarySystem:
    """Manages population evolution and evaluation."""

    def __init__(self,
                 population_size: int = 50,
                 mutation_rate: float = 0.2,
                 input_dim: int = 64,
                 output_dim: int = 10):
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.population: List[Genotype] = []
        self.generation = 0
        self.best_fitness_history = []

    def initialize_population(self):
        """Create initial random population."""
        for _ in range(self.population_size):
            genotype = self._create_random_genotype()
            self.population.append(genotype)

    def _create_random_genotype(self) -> Genotype:
        """Generate a random valid genotype."""
        genotype = Genotype()

        num_modules = random.randint(2, 5)
        module_types = ['linear', 'attention', 'rnn']

        for i in range(num_modules):
```

```
            module_type = random.choice(module_types)
            module_id = f"M{i}"
            params = self._random_params(module_type)
            genotype.add_module(ModuleGene(module_id, module_type, params))

        # Create connections
        for i in range(1, len(genotype.modules)):
            source_idx = random.randint(0, i-1)
            genotype.add_connection(ConnectionGene(
                genotype.modules[source_idx].id,
                genotype.modules[i].id
            ))

        # Add some plasticity rules
        if len(genotype.modules) > 1 and random.random() < 0.5:
            target = random.choice(genotype.modules[1:])
            genotype.add_plasticity_rule(PlasticityRule(
                target.id,
                'hebbian',
                learning_rate=random.uniform(0.001, 0.01)
            ))

        return genotype

    def _random_params(self, module_type: str) -> Dict:
        if module_type == 'linear':
            return {'dim': random.choice([64, 128, 256])}
        elif module_type == 'attention':
            return {'dim': random.choice([128, 256]), 'heads': random.choice([2, 4])}
        elif module_type == 'rnn':
            return {'hidden_dim': random.choice([64, 128]), 'num_layers': 1}
        return {}

    def evaluate_organism(self, genotype: Genotype, num_epochs: int = 10) -> Dict[str, float
        """Evaluate a single organism on tasks."""
        try:
            phenotype = Phenotype(genotype, self.input_dim, self.output_dim)
            optimizer = optim.Adam(phenotype.parameters(), lr=0.001)

            # Simple classification task for demonstration
            train_losses = []
            for epoch in range(num_epochs):
                # Generate random data
                x = torch.randn(32, self.input_dim)
                y = torch.randint(0, self.output_dim, (32,))
```

```
                optimizer.zero_grad()
                output = phenotype(x)
                loss = nn.functional.cross_entropy(output, y)
                loss.backward()
                optimizer.step()

                phenotype.apply_plasticity()
                train_losses.append(loss.item())

            # Measure performance
            final_loss = np.mean(train_losses[-3:])
            learning_speed = train_losses[0] - train_losses[-1]  # How much it improved

            return {
                'accuracy': max(0, 1.0 - final_loss),  # Proxy for accuracy
                'learning_speed': max(0, learning_speed),
                'complexity': len(genotype.modules) + len(genotype.connections)
            }

        except Exception as e:
            return {'accuracy': 0.0, 'learning_speed': 0.0, 'complexity': 100}

    def evolve_generation(self):
        """Run one generation of evolution."""
        # Evaluate population
        fitness_scores = {}
        for genotype in self.population:
            scores = self.evaluate_organism(genotype)
            # Multi-objective fitness
            fitness = (scores['accuracy'] * 0.6 +
                       scores['learning_speed'] * 0.3 -
                       scores['complexity'] * 0.001)
            fitness_scores[id(genotype)] = fitness
            genotype.fitness_history.append(fitness)

        # Track best
        best_fitness = max(fitness_scores.values())
        self.best_fitness_history.append(best_fitness)

        # Select parents
        sorted_pop = sorted(self.population,
                            key=lambda g: fitness_scores[id(g)],
                            reverse=True)
        parents = sorted_pop[:self.population_size // 2]

        # Create offspring
```

```
        offspring = []
        while len(offspring) < self.population_size:
            parent1, parent2 = random.sample(parents, 2)
            child = self._crossover(parent1, parent2)
            child.mutate(self.mutation_rate)
            offspring.append(child)

        self.population = offspring
        self.generation += 1

        return best_fitness

    def _crossover(self, parent1: Genotype, parent2: Genotype) -> Genotype:
        """Create offspring from two parents."""
        child = Genotype()

        # Take modules from both parents
        all_modules = parent1.modules + parent2.modules
        selected = random.sample(all_modules,
                                 k=min(len(all_modules), random.randint(2, 6)))
        for module in selected:
            child.add_module(copy.deepcopy(module))

        # Take compatible connections
        for conn in parent1.connections + parent2.connections:
            if any(m.id == conn.source for m in child.modules) and \
               any(m.id == conn.target for m in child.modules):
                if not any(c.source == conn.source and c.target == conn.target
                           for c in child.connections):
                    child.add_connection(copy.deepcopy(conn))

        return child

    def run(self, num_generations: int):
        """Run the evolutionary process."""
        print(f"Starting evolution with population size {self.population_size}")
        self.initialize_population()

        for gen in range(num_generations):
            best_fitness = self.evolve_generation()
            print(f"Generation {self.generation}: Best fitness = {best_fitness:.4f}")

            if gen % 10 == 0:
                avg_modules = np.mean([len(g.modules) for g in self.population])
                print(f"  Average modules per organism: {avg_modules:.1f}")
```

```
        # Return best organism
        final_fitness = {id(g): g.fitness_history[-1] for g in self.population}
        best_genotype = max(self.population, key=lambda g: final_fitness[id(g)])
        return best_genotype

# ================================================================================
# EXAMPLE USAGE
# ================================================================================

if __name__ == "__main__":
    print("=" * 60)
    print("Evolutionary Neural Architecture System")
    print("=" * 60)

    # Create evolutionary system
    system = EvolutionarySystem(
        population_size=20,
        mutation_rate=0.2,
        input_dim=64,
        output_dim=10
    )

    # Run evolution
    best_organism = system.run(num_generations=30)

    print("\n" + "=" * 60)
    print("Evolution complete!")
    print(f"Best organism has {len(best_organism.modules)} modules")
    print(f"Final fitness: {best_organism.fitness_history[-1]:.4f}")
    print("=" * 60)
```

This implementation demonstrates the core concepts while remaining tractable for experimentation. It can be extended with more sophisticated evaluation protocols, additional module types, better developmental processes, and connection to real benchmarks like ARC.