NEURAL GRAPH EVOLUTION: AUTOMATIC ROBOT DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Despite the recent success in robotic locomotion control, the design of robot relies heavily on human engineering. Automatic robot design has been a long studied subject but with limited success due to the large combinatorial search space and the difficulty in evaluating the found solution. In this paper, We propose Neural Graph Evolution (NGE) to address these two challenges. We formulate automatic robot design as a graph search problem and perform evolution search in graph space. NGE uses graph neural networks to parameterize the control policies that allows skill transfer from previously evaluated control policy to a new robot design. The policy sharing and transfer greatly reducing the cost of re-training new candidates. Similar to ES, NGE performs selection on current candidates and evolves new ones iteratively. In addition, NGE applies Graph Mutation by incorporating model uncertainty, which reduces the search space by balancing exploration and exploitation. We show that NGE significantly outperforms both random graph search (RGS) and ES by an order of magnitude. As shown in experiments, NGE is the first algorithm that can automatically discover complex robotic graph structures, such as a fish with two symmetrical flat side-fins and a tail, or a cheetah with athletic front and back legs. Instead of using thousands of cores for weeks, NGE efficiently solves searching problem within a day on a single 64 CPU-core Amazon EC2 machine.

1 INTRODUCTION

The goal of robot design is to search for robot body structures and their means of locomotion to best achieve a given objective. Robot design often relies on careful human-engineering and expert knowledge. Automatic robot design has been a long-studied subject but with limited success. In (Sims, 1994), the authors evolved creatures with 3D-blocks. Recently, soft robots are studied in (Joachimczak et al., 2014), which are evolved by adding small cells connected to the old ones, and also in (Cheney et al., 2014), where 3D voxels are treated as the minimum element of the robot. Most evolutionary robots (Duff et al., 2001; Neri, 2010) require heavy engineering of the initial structures, evolving rules and careful human-guidance. Further, none of the algorithms are able to evolve kinematically reasonable structures. There are two major challenges in automatic robot design: 1) the search space of all possible design is a large combinatorial search problem, and 2) the evaluation of each design requires learning a separate optimal controller that is often expensive to obtain. Due to the combinatorial nature of the problem, evolutionary strategy (ES) has been the de facto algorithm of automatic robot design in the pioneering works (Steels, 1993; Mitchell & Forrest, 1994; Langton, 1997).

The problem of large combinatorial search space and the difficulty in evaluation are also present in automatic neural network design or neural architecture search. There have been several approaches to tackle these problems. Bayesian optimization approaches (Snoek et al., 2012) primarily focus on fine-tuning the number of hidden units and layers from a predefined set. Reinforcement learning (Zoph & Le, 2016) and genetic algorithms (Liu et al., 2017) are studied to evolve recurrent neural networks (RNNs) and convolutional neural networks (CNNs) from scratch to maximize the validation accuracy. These approaches are computationally expensive because a large number of candidate networks have to be trained from scratch. (Pham et al., 2018) and (Stanley & Miikkulainen, 2002) propose weight sharing among all possible candidates in the search space to effectively amortize the inner loop training time and thus speed up the architecture search.

Inspired by the recent progress on the neural architecture search, we propose an efficient search method, "Neural Graph Evolution" (NGE), for automatic robot design in simulation that co-evolves the robot design and control policy simultaneously. Unlike the recent deep reinforcement learning works (Mnih et al., 2013; Bansal et al., 2017; Heess et al., 2017) on learning locomotion control policy of specific robots carefully designed by human experts, we aim to adapt the robot design along with policy learning to maximize the agent's performance. NGE treats robot designs as a graph search problem. It uses graph as the main backbone of rich design representation and graph neural networks (GNN) as the controller. Similar to ES, NGE iteratively evolves new graphs and removes graphs based on the performance guided by the learnt GNN controller.

The specific contributions of this paper are as follows:

- We formulate the automatic robot design as a graph search problem.
- We utilize graph neural networks (GNNs) to share the weights between the controllers, which greatly reduces the computation time evaluate each new robot design.
- We propose a population-based search algorithm over the robot design space that iterates over generations of candidates.
- To balance exploration and exploitation during the search, we developed a mutation scheme that incorporates model uncertainty.
- We show NGE automatically discovers interesting robot designs in MuJoCo (Todorov et al., 2012b) that are comparable to the ones designed by human experts from stretch. While, random graph search and naive evolutionary strategy fails to discover meaningful results on these tasks.

2 Related Work

2.1 Reinforcement Learning

In reinforcement learning (RL), the problem is usually formulated as a Markov Decision Process (MDP). The infinite-horizon discounted MDP consists of a tuple of $(\mathcal{S}, \mathcal{A}, \gamma, P, R)$, respectively the state space, action space, discount factor, transition function, and reward function. The objective of the agent is to maximize the total expected reward $J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]$, where the state transition follows the distribution $P(s_{t+1}|s_t, a_t)$. Here, s_t and a_t denotes the state and action at time step t. In this paper, to evaluate each robot structure, we use PPO to train RL agents (Schulman et al., 2017; Heess et al., 2017). PPO uses a neural network parameterized as $\pi_\theta(a_t|s_t)$ to represent the policy, and adds a penalty for the KL-divergence between the new and old policy to prevent over-optimistic updates. PPO optimize the following surrogate objective function instead:

$$J_{\text{PPO}}(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} A^t(s_t, a_t) r^t(s_t, a_t)\right] - \beta\, \text{KL}\left[\pi_\theta(: |s_t) | \pi_{\theta_{old}}(: |s_t)\right]. \tag{1}$$

We denote the estimate of the expected total reward given the current state-action pair, the value and the advantage functions, as $Q^t(s_t, a_t)$, $V(s_t)$ and $A^t(s_t, a_t)$ respectively. PPO solves the problem by iteratively generating samples and optimizing J_{PPO}.

2.2 Graph Neural Network

Graph Neural Networks (GNNs) are very effective when the data is in in the form of graph (Bruna et al., 2014; Defferrard et al., 2016; Li et al., 2015; Kipf & Welling, 2017; Duvenaud et al., 2015; Henaff et al., 2015). Recently, the use of GNNs in locomotion control has greatly increased the transferability of controllers (Wang et al., 2018). A GNN operates on a graph whose nodes and edges are denoted respectively as $u \in V$ and $e \in E$. We consider the following GNN, where at timestep t each node in GNN receives an input feature and is supposed to produce an output at a node level.

Input Model: The input feature for node u is denoted as x_u^t. x_u^t is a vector of size d, where d is the size of features. In most cases, x_u^t is produced by the output of an embedding function used to encode information about u into d-dimensional space.

Propagation Model: Within each timestep t, the GNN performs $\mathcal{T}$ internal propagations, so that each node has global (neighbourhood) information. In each propagation, every node communicates

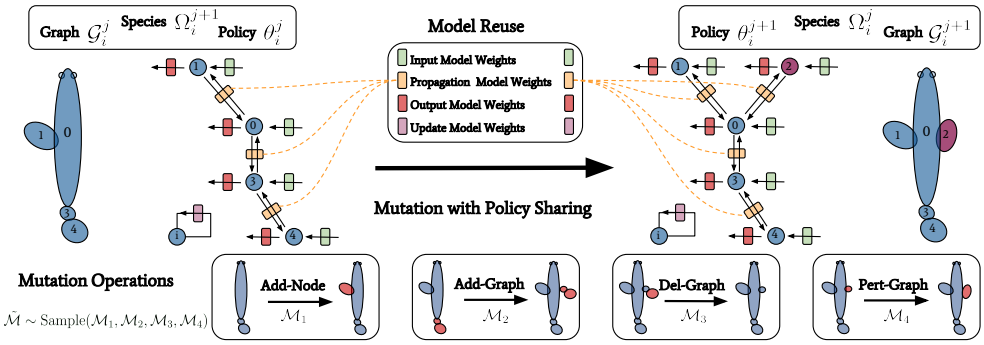


Figure 1: In NGE, several mutation operations are allowed. By using Policy Sharing, child species reuse weights from parents, even if the graphs are different. The same color indicates shared and reused weights. For better visualization, we only plot the sharing of propagation model (yellow curves).

with its neighbours, and updates its hidden state by absorbing the input feature and message. We denote the hidden state at the internal propagation step τ ($\tau \leq \mathcal{T}$) as $h_u^{t,\tau}$. Note that $h_u^{t,0}$ is usually initialized as $h_u^{t-1,\mathcal{T}}$, i.e., the final hidden state in the previous time step. $h^{0,0}$ is usually initialized to zeros. The message that u sends to its neighbors is computed as

$$m_u^{t,\tau} = M(h_u^{t,\tau-1}), \tag{2}$$

where M is the message function. To compute the updated $h_u^{t,\tau}$, we use the following equations:

$$r_u^{t,\tau} = R(\{m_v^{t,\tau} \,|\, \forall v \in \mathcal{N}_G(u)\}),\ h_u^{t,\tau} = U(h_u^{t,\tau-1}, (r_u^{t,\tau}; x_u^t)) \tag{3}$$

where R and U are the message aggregation function and the update function respectively, and $\mathcal{N}_G(u)$ denotes the neighbors of u.

Output Model: Output function F takes input the node's hidden states after the last internal propagation. The node-level output for node u is therefore defined as $u^t = F(h_u^{t,\mathcal{T}})$.

Functions M, R, U, F in GNNs can be trainable neural networks or linear functions. For details of GNN controllers, we refer readers to (Wang et al., 2018).

3 Neural Graph Evolution

In robotics design, every component, including the robot arms, finger and foot, can be regarded as a node. The connections between the components can be represented as edges. In locomotion control, the robotic simulators like MuJoCo (Todorov et al., 2012a) use an XML file to record the graph of the robot. As we can see, robot design is naturally represented by a graph. To better illustrate Neural Graph Evolution (NGE), we first introduce the following terminologies and summarize the algorithm.

Graph and Species. We use an undirected *graph* $\mathcal{G} = (V, E, A)$ to represent each robotic design. V and E are the collection of physical body nodes and edges in the graph respectively. The mapping $A : V \to \Lambda$ maps the node $u \in V$ to its structural attributes $A(u) \in \Lambda$, where Λ is the attributes space. For example, the fish in Figure 1 consists of a set of ellipsoid nodes, and vector $A(u)$ describes the configurations of each ellipsoid. The controller is a policy network parameterized by weights θ. The combination of graph and policy is defined as a *species*, denoted as $\Omega = (\mathcal{G}, \theta)$.

Generation and Policy Sharing. In the jth iteration, NGE evaluates a pool of species called a *generation*, denoted as $P^j = \{(\mathcal{G}_i^j, \theta_i^j), \forall i = 1, 2, ..., \mathcal{N}\}$, where $\mathcal{N}$ is the size of the generation. Different from ES, in NGE, the search space includes not only graph space, but also the weight space of the policy network. For better efficiency of NGE, we design a process called Policy Sharing (PS), where weights are reused from parent to child species.

Similar to ES, NGE performs optimization by iterating among *mutation*, *evaluation* and *selection*. The objective and performance metric of NGE are introduced in Section 3.1. In NGE, we randomly initialize the generation with $\mathcal{N}$ species. For each generation, NGE trains each species and evaluates their fitness separately, the policy of which is described in Section 3.2. During the selection, we

Algorithm 1 Neural Graph Evolution

Initialize generation $\mathcal{P}^0 \leftarrow \{(\theta_i^0, G_i^0)\}_{i=1}^{\mathcal{N}}$
while Evolving jth generation **do** ▷ Evolution outer loop
  for ith species $(\theta_i^j, \mathcal{G}_i^j) \in \mathcal{P}^j$ **do** ▷ Species fitness inner loop
    $\theta_i^{j+1} \leftarrow \text{Update}(\theta_i^j)$ ▷ Train policy network
    $\xi_i \leftarrow \xi(\theta_i^{j+1}, G_i^j)$ ▷ Evaluate fitness
  end for
  $\mathcal{P}^{j+1} \leftarrow \mathcal{P}^j - \{(\theta_k^j, \mathcal{G}_k^j) \in \mathcal{P}^j, \forall k \in \arg\min_{\mathcal{K}}(\{\xi_i\})\}$. ▷ Remove worst $\mathcal{K}$ species
  $\hat{\mathcal{P}} \leftarrow \{(\hat{\theta}_h, \hat{\mathcal{G}}_h = \mathcal{M}(\mathcal{G}_{h,p})), \text{ where } \mathcal{G}_{h,p} \sim \text{Uniform}(\mathcal{P}^{j+1})\}_{h=1}^{\mathcal{C}}$ ▷ Mutate from survivors
  $\mathcal{P}^{j+1} \leftarrow \mathcal{P}^{j+1} \cup \{(\hat{\theta}_k, \hat{\mathcal{G}}_k) \in \hat{\mathcal{P}}, \forall k \in \arg\max_{\mathcal{K}}(\{\xi_P(\hat{G}_h)\})\}$. ▷ Pruning
end while

eliminate $\mathcal{K}$ species with the worst AF. To mutate $\mathcal{K}$ new species from surviving species, we develop a novel mutation scheme called Graph Mutation (GM), described in Section 3.3, and efficiently inherit policies from the parent species by Policy Sharing, described in Section 3.4. Our method is outlined in Algorithm 1.

3.1 Amortized Fitness and Objective Function

Fitness represents the performance of a given $\mathcal{G}$ using the optimal controller $\theta^*(\mathcal{G})$. However, it is impractical or impossible to obtain for the following reasons. First, each design is computationally expensive to evaluate. To evaluate one graph, the controller needs to be trained and tested. Model-free (MF) algorithms take more than one million in-game timesteps to train a simple 6-degree-of-freedom cheetah (Schulman et al., 2017), while model-based (MB) controllers, even given ground-truth dynamics, can be 7x slower than real time to test, while usually having lower performance than MF methods (Tassa et al., 2012; Nagabandi et al., 2017). Second, the search in robotic graph space easily or guaranteedly gets stuck in local-optima. In robotic design, local-optima are difficult to detect as it is hard to tell whether the controller has converged or has reached a temporary optimization plateau.

Therefore, to speed up and trade off between evaluating fitness and evolving new species, we introduce *amortized fitness* (AF), as defined in the following equation,

$$\xi(\mathcal{G}, \theta) = \mathbb{E}_{\pi_\theta, \mathcal{G}}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right]. \tag{4}$$

AF spreads the computation cost across generations. Within each iteration, the species update the policy θ for K times. Essentially, we use a surrogate objective function, as shown in Figure 1. With this surrogate objective function, NGE is able to co-adapt the controller θ and $\mathcal{G}$ simultaneously. Naive ES and RGS algorithms use the estimated controller $\tilde{\theta}(\mathcal{G})$ after a fixed number of updates, which is undertrained and biased, and therefore makes ES and RGS easy to overfit. AF scores are used to rank the performance of all the species during the selection.

Used By	Objective	Search Space
/	$\max_{\mathcal{G}} \xi(\mathcal{G}, \theta^*(\mathcal{G}))$	$\mathcal{G}$
ES, RGS	$\max_{\mathcal{G}} \xi(\mathcal{G}, \tilde{\theta}(\mathcal{G}))$	$\mathcal{G}$
NGE	$\max_{\mathcal{G},\theta} \xi(\mathcal{G}, \theta)$	$\mathcal{G}, \theta$

Table 1: The original objective functions and the objective function used by ES, RGS and NGE.

3.2 Policy Representation

Given a species with graph $\mathcal{G}$, we train the parameters θ of policy network $\pi_\theta(a^t|s^t)$ using reinforcement learning. Similar to (Wang et al., 2018), we use a GNN as the policy network of the controller. A graphical representation of our model is shown in Figure 1. We follow the notations in Section 2.2. For the *input model*, we parse the input state vector s^t obtained from the environment into a graph,

where each node $u \in V$ fetches the corresponding observation $o(u, t)$ from s^t, and extracts the feature $x_u^{O,t}$ with an embedding function Φ. We also encode the attribute information $A(u)$ into x_u^A with an embedding function denoted as ζ. The input feature x_u^t is thus calculated as:

$$\begin{aligned} x_u^{O,t} &= \Phi(o(u,t)),\ x_u^A = \zeta(A(u)), \\ x_u^t &= [x_u^{O,t}; x_u^A], \end{aligned} \tag{5}$$

where [.] denotes concatenation. We use θ_Φ, θ_ζ to denote the weights of embedding functions. The *propagation model* is the standard model we described in Section 2.2. We use summation as the aggregation function and a GRU (Chung et al., 2014) as the update function. For the *output model*, we define the collection of controlling nodes as $u \in \mathcal{F} \subseteq V$, and define Gaussian distributions on each node controller as follows:

$$\forall u \in \mathcal{F},\ \mu_u^t = F_\mu(h_u^{t,T}), \tag{6}$$

$$\sigma_u^t = F_\sigma(h_u^{t,T}), \tag{7}$$

where μ_u and σ_u are the mean and the standard deviation of the action distribution. The weights of output function are denoted as θ_F. By combining all the actions produced by each node controller, we have the policy distribution of the agent:

$$\pi(a^t|s^t) = \prod_{u \in \mathcal{F}} \pi_u(a_u^t|s^t) = \prod_{u \in \mathcal{F}} \frac{1}{\sqrt{2\pi(\sigma_u^t)^2}} \exp\left(\frac{(a_u^t - \mu_u^t)^2}{2(\sigma_u^t)^2}\right) \tag{8}$$

We optimize the $\pi(a^t|s^t)$ with PPO, the details of which are provided in Appendix A.

3.3 Graph Mutation

Between generations, the graphs of species evolve from parents to children. Without heavy engineering of the mutation primitives, in particular, we allow the following primitive operations to the parent's graph $\mathcal{G}$:

$\mathcal{M}_1$, **Add-Node**: In the $\mathcal{M}_1$ (Add-Node) operation, the growing of a new body part is done by sampling a node $v \in V$ from the parent, and append a new node u to it. We randomly initialize u's attributes from a uniform distribution in attribute space.

$\mathcal{M}_2$, **Add-Graph**: The $\mathcal{M}_2$ (Add-Graph) operation allows for faster evolution by reusing the subtrees in the graph with good functionality. We sample a sub-graph or leaf node $\mathcal{G}' = (V', E', A')$ from the current graph, and a placement node $u \in V(\mathcal{G})$ to which to append $\mathcal{G}'$. We randomly mirror the attributes of the root node in $\mathcal{G}'$ to incorporate a symmetry prior.

$\mathcal{M}_3$, **Del-Graph**: The process of removing body parts is defined as $\mathcal{M}_3$ (Del-Graph) operation. In this operation, a sub-graph $\mathcal{G}'$ from $\mathcal{G}$ is sampled and removed from $\mathcal{G}$.

$\mathcal{M}_4$, **Pert-Graph**: In the $\mathcal{M}_4$ (Pert-Graph) operation, we randomly sample a sub-graph $\mathcal{G}'$ and recursively perturb the parameter of each node $u \in V(\mathcal{G}')$ by adding Gaussian noise to $A(u)$.

We visualized a pair of example fish in Figure 1. The fish on the top-right is mutated from the fish on the top-left by applying $\mathcal{M}_1$. The new node (2) is colored magenta in the figure. To mutate each new candidate graph, we sample the operation $\mathcal{M}$ and apply $\mathcal{M}$ on $\mathcal{G}$ as

$$\mathcal{G}' = \mathcal{M}(\mathcal{G}),\ \text{where}\ \mathcal{M} \in \{\mathcal{M}_l,\ l = 1, 2, 3, 4\},\ \mathrm{P}(\mathcal{M} = \mathcal{M}_l) = p_m^l. \tag{9}$$

p_m^l is the probability of sampling each operation with $\sum_l p_m^l = 1$.

To facilitate evolution, we want to avoid wasting computation resources on species with low expected fitness, while encouraging NGE to test species with high uncertainty. We again employ a GNN to predict the fitness of the graph $\mathcal{G}$, denoted as $\xi_P(\mathcal{G})$. The weights of this GNN are denoted as $\theta_\mathcal{G}$. In particular, we predict the AF score with a similar propagation model as our policy network, but the observation feature is only x_u^A, i.e., the embedding of the attributes. The output model is a graph-level output (as opposed to node-level used in our policy), regressing to the score ξ. After each generation, we train the regression model using the L2 loss.

However, pruning the species greedily may easily overfit the model to the existing species since there is no modeling of uncertainty. We thus propose Graph Mutation (GM) based on Thompson Sampling

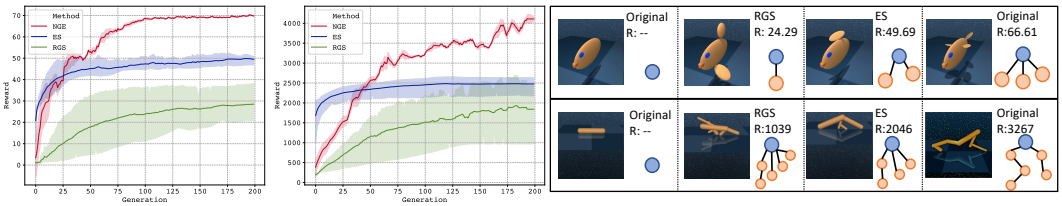


Figure 2: The performance of the graph search for RGS, ES and NGE. The figures on are the example creatures obtained from each of the method. The graph structure next to the figure are the corresponding graph structure. We included the original species for reference.

to balance between exploration and exploitation. We denote the dataset of past species and their AF score as $\mathcal{D}$. GM selects the best graph candidates by considering the posterior distribution of the surrogate $P\left(\theta_{\mathcal{G}} | \mathcal{D}\right)$:

$$\mathcal{G}^{*} = \arg\max_{\mathcal{G}} \mathbb{E}_{P(\theta_{\mathcal{G}}|\mathcal{D})}\left[\xi_{P}\left(\mathcal{G} | \theta_{\mathcal{G}}\right)\right]. \tag{10}$$

Instead of sampling the full model with $\widetilde{\theta_{\mathcal{G}}} \sim P\left(\theta_{\mathcal{G}} | \mathcal{D}\right)$, we follow Gal & Ghahramani (2016) and perform dropout during inference, which can be viewed as an approximate sampling from the model posterior. At the end of each generation, we randomly mutate $\mathcal{C} \geq \mathcal{N}$ new species from surviving species. We then sample a single dropout mask for the surrogate model and only keep $\mathcal{N}$ species with highest ξ_{P}. The details of GM are given in Appendix F.

3.4 Rapid Adapting using Policy Sharing

To leverage the transferability of GNNs across different graphs, we propose Policy Sharing (PS) to reuse old weights from parent species. The weights of a species are as follows:

$$\theta = \left(\theta_{\Phi}, \theta_{\zeta}, \theta_{M}, \theta_{U}, \theta_{F}\right), \tag{11}$$

where $\theta_{\Phi}, \theta_{\zeta}, \theta_{M}, \theta_{U}, \theta_{F}$ are the weights for the models we defined earlier in Section 3.2 and 2.2. Since our policy network is based on GNNs, as we can see from Figure 1, model weights of different graphs share the same cardinality (shape). A different graph will only alter the paths of message propagation. With PS, new species are provided with a strong weight initialization, and the evolution will less likely be dominated by species that are more ancient in the genealogy tree.

4 Experiments

In this section, we demonstrate the effectiveness of NGE on various evolution tasks. In particular, we evaluate graph search from scratch in Section 4.1, and fine-tuning from human-engineered species in Section 4.2. We also provide an ablation study on GM in Section 4.3, and ablation study on computational cost or generation size in Section 4.4. Experiments are simulated with MuJoCo. The code *will be released* upon acceptance. We do a grid search on the hyper-parameters as summarized in Appendix E, and show the averaged curve of each method. In the Random Graph Search (RGS) baseline, a large amount of graphs are generated randomly. RGS focuses on exploiting given structures, and relies no evolution to generate new graphs. In the Evolutionary Strategy (ES) baseline, no Graph Mutation is used during training. Old species are allowed to continue their training across different generations (otherwise the controller will be under-trained). For both RGS and ES algorithm, we use a multi-layer perceptron as the policy network. We also introduce the test environments here:

Fish Env: In the `fish` environment, graph consists of ellipsoids. The reward is the swimming-speed along the y-direction. We denote reference human-engineered graph (Tassa et al., 2018) as $\mathcal{G}_{F}$.

Walker Env: We also define a 2D environment `walker` constructed by cylinders, where the goal is to move along x-direction as fast as possible. We denote the reference human-engineered walker as $\mathcal{G}_{W}$ and cheetah as $\mathcal{G}_{C}$ (Tassa et al., 2018).

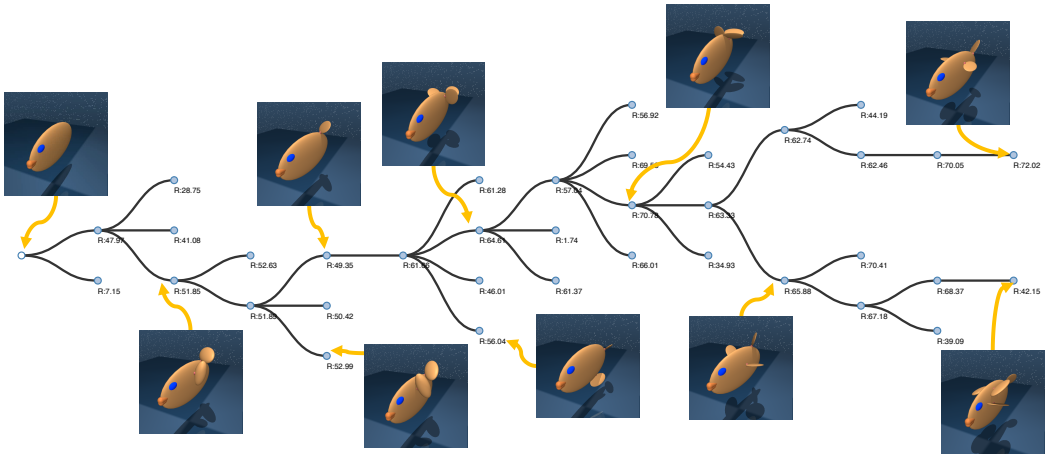


Figure 3: The genealogy tree generated using NGE for `fish`. The number next to the node is the reward (the averaged speed of the fish). For better visualization, we down-sample genealogy sub-chain of the winning species. NGE agents gradually grow symmetrical side-fins.

4.1 Evolution Topology Search

In this experiment, the task is to evolve the graph and the controller from scratch. For both `fish` and `walker`, species are initialized as random $(\mathcal{G}, \theta)$. Computation cost is often a concern among evolutionary algorithms. In our comparison results, we allocate the same computation budget to all methods, which is approximately 12 hours on a EC2 `m4.16xlarge` cluster with 64 cores for one session. A grid search on hyper-parameters is performed (details in Appendix E). The averaged curves from different runs are shown in Figure 2. In both `fish` and `walker` environments, NGE is the best model. We find RGS is not able to efficiently search the space of $\mathcal{G}$ even after evaluating $12,800$ different graphs. The performance of ES grows faster for the earlier generations, but is worse than our method in the end. By looking at the generated species, ES overfits to local species that dominate the rest of generations.

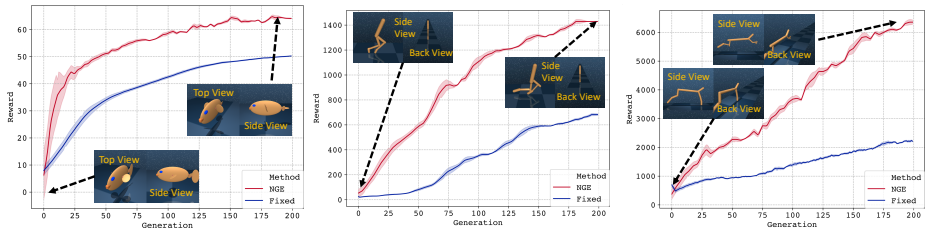
To better understand the evolution process, we visualize the genealogy tree of fish using our model in Figure 3. Our fish species gradually generates three fins with preferred $\{A(u)\}$, with two side-fins symmetrical about the fish torso, and one tail-fin lying on the middle line. We obtain similar results for `walker`, as shown in Appendix C.

4.2 Fine-tuning Species

Evolving every species from scratch is costly in practice. For many locomotion control tasks, we already have a decent human-engineered robot as a starting point. In this fine-tuning task, we study how our algorithm can improve upon the human-engineered design.We examine the scenario in which (V, E) of graph is fixed, and our model is tasked to fine-tune (evolve) only the node attributes $\{A(u)\}$. In the baseline models, the graph (V, E, A) is fixed, and only the controllers are trained. As we can see from Figure 4, when given the same wall-clock time, NGE can discover fine adjustment that significantly improves the initial human-engineered design. The cheetah gradually transforms the forefoot into a *claw*, the 3D-fish rotates the pose of the side-fins and tail, and 2D-walker evolves bigger feet.

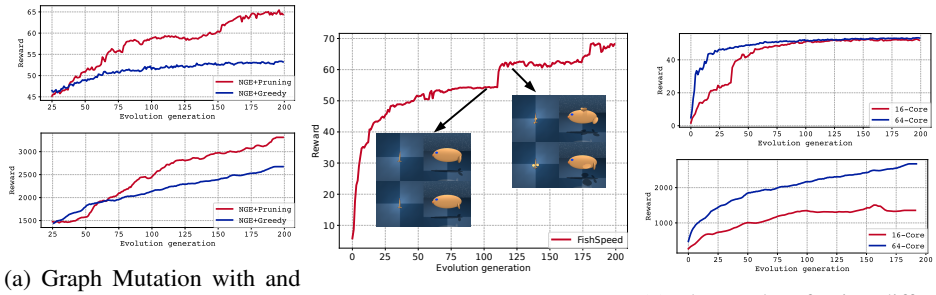
4.3 Greedy Search v.s. Exploration under Uncertainty

We also investigate the performance of NGE with and without Graph Mutation, whose hyper-parameters summarized in Appendix E. In Figure 5a, we applied GM to the evolution graph search task. The final performance of the GM outperforms the baseline on both `fish` and `walker` environments. The proposed GM is able to better explore the graph space.



(a) Fine-tuning `3D-fish`. (b) Fine-tuning `2D-walker`. (c) Fine-tuning `cheetah`.

Figure 4: Fine-tuning results on different creatures compared with baseline where structure is fixed. The figures included the species looking from 2 different angles.



(a) Graph Mutation with and without uncertainty in `fish` (upper) and `walker` environment (b) Rapid graph evolution in the `fish` environment (c) The results of using different computation resource.

Figure 5: Results of ablation study. Graph Mutation without uncertainty results and rapid evolution during experiments.

4.4 Computation Cost and Generation Size

We also investigate how the generation size $\mathcal{N}$ affect the final performance of NGE. We note that as we increase the generation size and computing resources, NGE achieves marginal improvement on the simple `Fish` task. A NGE session with 16-core `m5.4xlarge` ($0.768 per Hr) AWS machine could achieve almost the same performance with 64-core `m4.16xlarge` ($3.20 per Hr) in `Fish` environment in the same wall-clock time, However, we do notice there is a trade off between computational resources and performance on the more difficult task. In general, NGE is effective even when the computing resources are limited and it significantly outperforms RGS and ES by using only a small generation size of 16.

5 Discussion

In this paper, we introduced NGE, an efficient graph search algorithm for automatic robot design that co-evolves the robot design graph and its controllers. NGE greatly reduces evaluation cost by transferring the learned GNN-based control policy from previous generations, and better explores the search space by incorporating model uncertainties. Our experiments show that searching for robotic graph is challenging where both random graph search and evolutionary strategy fail to discover meaning robot designs. NGE outperforms the naive approaches in both final performance and computation time by a order of magnitude, and is the first algorithm that can discovers graphs similar to carefully hand-engineered design. We believe this work is an important step towards automated robot design, and is inspiring to other graph search problems.

References

Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *ICLR*, 2014.

Nick Cheney, Robert MacCurdy, Jeff Clune, and Hod Lipson. Unshackling evolution: Evolving soft robots with multiple materials and a powerful generative encoding. *ACM SIGEVOlution*, 7(1): 11–23, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

David Duff, Mark Yim, and Kimon Roufas. Evolution of polybot: A modular reconfigurable robot. In *Proc. of the Harmonic Drive Intl. Symposium, Nagano, Japan*, 2001.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pp. 1050–1059, 2016.

Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Michał Joachimczak, Reiji Suzuki, and Takaya Arita. Fine grained artificial development for body-controller coevolution of soft-bodied animats. *Artificial life*, 14:239–246, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR*, 2017.

Christopher G Langton. *Artificial life: An overview*. Mit Press, 1997.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

Melanie Mitchell and Stephanie Forrest. Genetic algorithms and artificial life. *Artificial life*, 1(3): 267–289, 1994.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *arXiv preprint arXiv:1708.02596*, 2017.

Ferrante Neri. Memetic compact differential evolution for cartesian robot control. *IEEE Computational Intelligence Magazine*, 5(2):54–65, 2010.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 15–22. ACM, 1994.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Luc Steels. The artificial life roots of artificial intelligence. *Artificial life*, 1(1_2):75–110, 1993.

Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 4906–4913. IEEE, 2012.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033. IEEE, 2012a.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pp. 5026–5033. IEEE, 2012b.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pp. 1058–1066, 2013.

Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *ICLR*, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

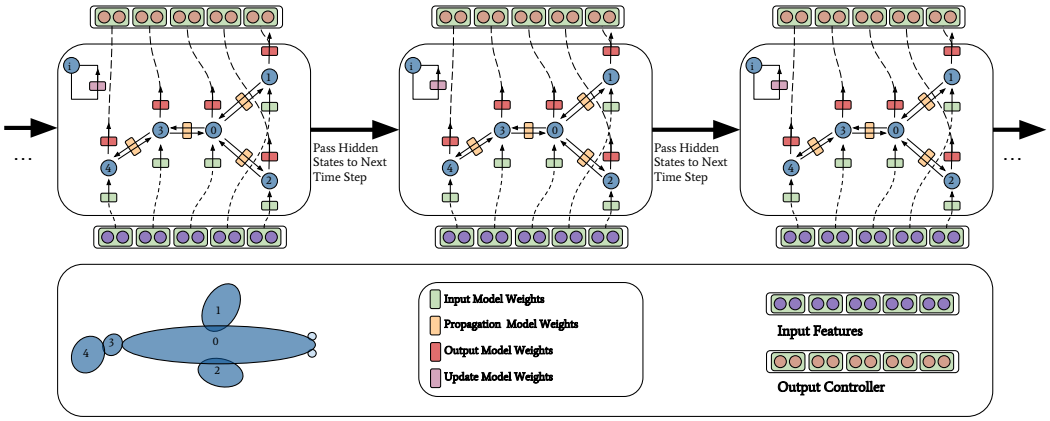


Figure 6: In this figure, we show the computation graph of the NerveNet++, at each timestep, every node in the graph updates it's hidden state by absorbing the messages as well as the input feature. The output function takes the hidden states as input and outputs the controller (or policy) of the agent.

A Details of NerveNet++

Similar to NerveNet, we parse the agent into a graph, where each node in the graph corresponds to the physical body part of the agents. For the example, the `fish` in figure 1 could be parsed into a graph of five nodes, namely the torso (0), left-fin (1), right-fin (2), and tail-fin bodies (3, 4). By replacing MLP with NerveNet, the learnt policy has much better performance on robustness and transfer learning. Since our GNN policy network is based on Wang et al. (2018), we name our model NerveNet++.

In the origin NerveNet, at every timestep, several propogation steps are performed so that every node could observe global information before producing the control signal. It is time and memory consuming, and the minimum number of propogation steps is constrained by the depth of the graph. Therefore in NerveNet++, we propose a propogation model with memory state, where node updates its hidden state by absorbing the input feature and message with time.

Since the episode of each game usually lasts for several hundred timesteps, it is computationally expensive and ineffective to build the full back-propagation graph. Inspired by Mnih et al. (2016), we employ the truncated graph back-propagation to optimize the policy. NerveNet++ is suitable for evolutionary algorithm, as it brings speed-up in wall-clock time, and decreases the amount of memory usage. The computational performance evaluation are provided in Appendix B. The parameters of the NerveNet++ model is trained by PPO algorithm Schulman et al. (2017); Heess et al. (2017),

B Optimization with Truncated Backpropagation

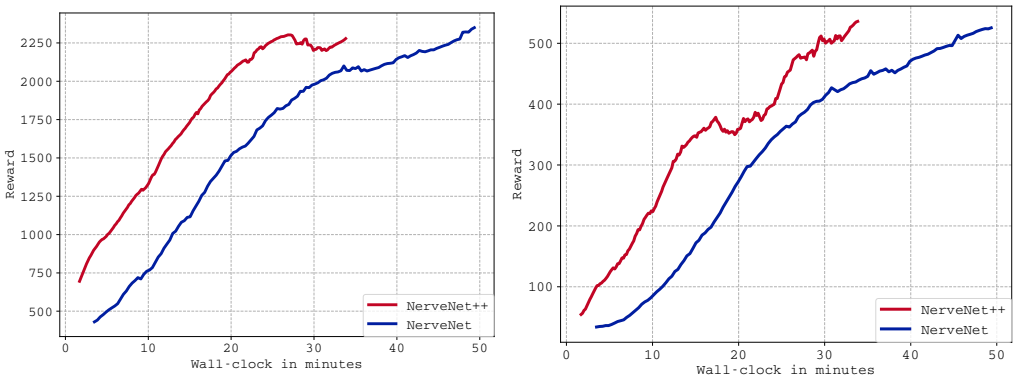
During training, the agent generates the rollout data by sampling from the distribution $a_t \sim \pi(a_t|s_t)$ and store the training data of $\mathcal{D} = \{a^t, s^t, \{h_u^{t,\tau=0}\}\}$. For the training of reinforcement learning agents with memory, the original training objective is

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s^t, a^t, \{h_u^{t,\tau=0}\}) \right], \tag{12}$$

where we denote the whole update model as H and

$$h_u^{t+1,\tau=0} = H(\{h_v^{t,\tau=0}\}, s^t, a^t). \tag{13}$$

The memory state $h_{u,t+1}$ is depend on previous actions, observations, and states. Therefore, the full back-propagation graph will be the same length as the episode length, which is very computationally intensive. The intuition from the authors in Mnih et al. (2016) is that, for the RL agents, the dependency of the agents on timesteps that are far-away from the current timestep is limited. And not



(a) Results on `Cheetah-V1` environment.

(b) Results on `Walker2d-V1` environment.

Figure 7: In these 2 figures, we show that to reach similar performance, NerveNet++ took shorter time comparing to original NerveNet.

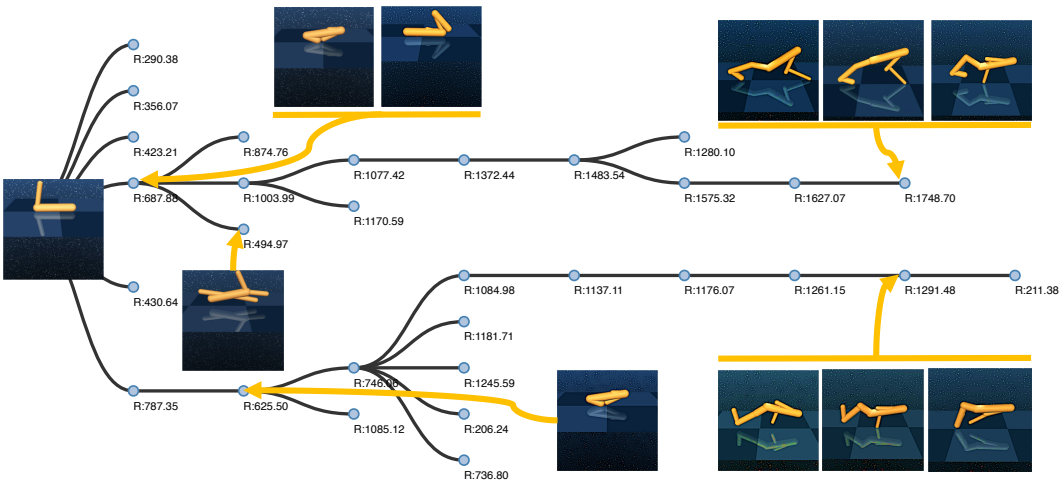


Figure 8: Our `walker` species gradually grows two foot-like structures from randomly initialized body graph.

much accuracy of the gradient estimator will be lost if we truncate the back-propagation graph. We define a back-propagation length Γ, and optimize the following objective function instead:

$$J_T(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \sum_{\kappa=0}^{\Gamma-1} \gamma^{t+\kappa} r(s_{t+\kappa}, a_{t+\kappa}, \{h_u^{t,\tau=0}\}) \right], \text{ where} \tag{14}$$

$$h_u^{t+\kappa,\tau=0} = \begin{cases} H(\{h_v^{t+\kappa-1,\tau=0}, \forall v\}, s_{t+\kappa-1}, a_{t+\kappa-1}) & \kappa \neq 0, \\ h_u^{t,\tau=0} \in \mathcal{D} & \kappa = 0, \end{cases} \tag{15}$$

Essentially this optimization means that we only back-propagate up to Γ timesteps, namely at the places where $\kappa = 0$, we treat the hidden state as input to the network and stop the gradient. To optimize the objective function, we follow same optimization procedure from Wang et al. (2018), which is a variant of PPO Schulman et al. (2017), where a surrogate loss $J_{\text{ppo}}(\theta)$ is optimized. We refer readers to these papers for algorithm details.

C Full NGE Results

Similar to the fish genealogy tree, in Fig. 8, the simple initial walking agent evolves into a cheetah-like structure, and is able to run with high speed. We also show the species generated by NGE, ES and RGS. Note that we do not cherry-pick the results in Figure 9.

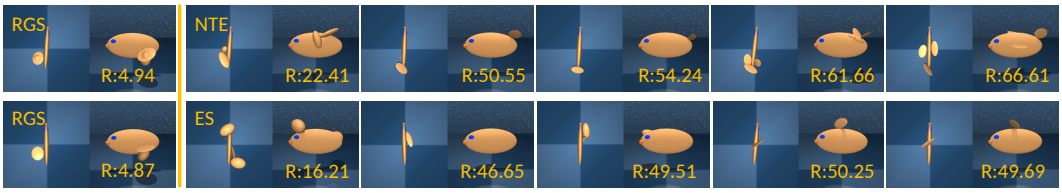


Figure 9: We present qualitative comparison between the 3 algorithms in the figure. Specifically, the aligned comparison between our method and naive baseline are the representative creatures at the same generation (using same computation resources). Our algorithm notably display stronger dominance in terms of its structure as well as reward.

D Resetting Controller for Fair Competition

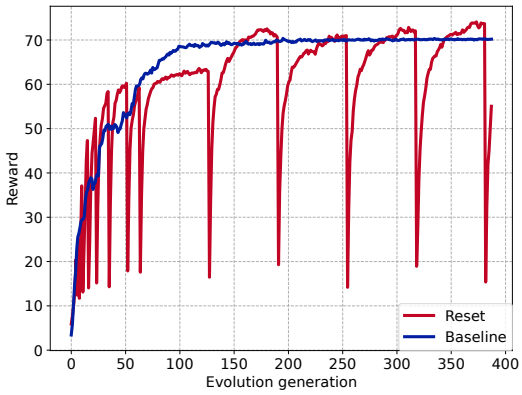


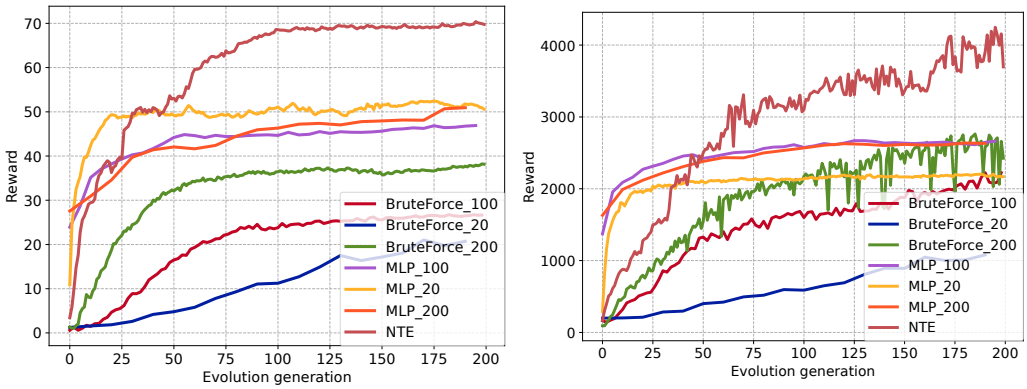
Figure 10: The results of resetting controller scheme and baselines.

Although amortized fitness is a better estimation of the ground-truth fitness, it is still biased. Species that appear earlier in the experiment will be trained for more updates if it survives. Indeed, intuitively, it is possible that in the real nature, species that appear earlier will dominate the generation by number, and new species are eliminated even if the new species has better fitness. Therefore, we design the experiment where we reset the weights for all species $\theta = (\theta_\Phi, \theta_\zeta, \theta_M, \theta_U, \theta_F)$ randomly. By doing this, we are forcing the species to compete fairly. From the figure 10, we notice that this method helps the exploration, which leads to higher reward in the end. But it usually takes longer time for the algorithm to converge. And therefore for the graph search task figure 2, we do not include the results with controller-resetting.

E Hyper-parameters Searched

All methods are given equal amount of computation budget. To be more specific, the number of total timestep generated by all species for all generation is the same for all methods. For example, if we use 10 training epochs in one generation, each of the epoch with 2000 sampled timesteps, then the computation budget could let NGE evolve for 200 generation, where each generation has a species size of 64. For NGE, RGS, ES models in Figure 11, we run a grid search on the hyper-parameters recorded in Table 2, and Table 3, and plot the curve with the best results respectively. Since the number of generation for RGS baseline could be regarded as 1, its curve is plotted with the number of updates normalized by computation resource as x-axis.

Here we show the detail figures of six baselines, which are namely RGS-20, RGS-100, RGS-200, and ES-20, ES-100, ES-200. The number attached to the baseline names is indicating the number of inner-loop policy training epochs. In the case of RGS-20, where more than 12800 different graphs are searched, the average reward is still very low. Increasing the number of inner-loop training of species to 100 and 200 does not help the final performance significantly.



(a) Detailed results in `Fish` environment. (b) Detailed results in `Walker` environment.

Figure 11: The results of the graph search

To test the performance with and without GM, we use 64-core clusters (generations of size 64). Different from graph search experiment, we do not want to run excessive amount experiment with a grid search. Therefore, the hyper-parameters are chosen to be the first value available in Table 2 and Table 3.

Items	Value Tried
Number of Iteration Per Update	10, 20, 100, 200
Number of Species per Generation	16, 32, 64, 100
Elimination Rate	0.15, 0.20, 0.3
Discrete Socket	Yes, True
Timesteps per Updates	2000, 4000, 6000
Target KL	0.01
Learning Rate Schedule	Adaptive
Number of Maximum Generation	400
Prob of `Add-Node`, `Add-Graph`	0.15
Prob of `Pert-Graph`	0.15
Prob of `Del-Graph`	0.15
Allow Mirrowing Attrs in `Add-Graph`	Yes, No
Allow Resetting Controller	Yes, No
Resetting Controller Freq	50, 100

Table 2: Hyperparameter grid search options.

Items	Value Tried
Allow `Graph-Add`	True, False
Graph Mutation	True, False
Pruning Temperature	0.01, 0.1, 1
Network Structure	NerveNet, NerveNet++
Number Candidates before Pruning	200, 400

Table 3: Hyperparameters grid search options for NGE.

F MODEL BASED SEARCH USING THOMPSON SAMPLING

Thompson Sampling is a simple heuristic search strategy that is typically applied to the multi-armed bandit problem. The main idea is to select an action proportional to the probability of the action

being optimal. When applied to the graph search problem, Thompson Sampling allows the search to balance the trade-off between exploration and exploitation by maximizing the expected fitness under the posterior distribution of the surrogate model.

Formally, Thompson Sampling selects the best graph candidates at each round according to the expected estimated fitness ξ_P using a surrogate model. The expectation is taken under the posterior distribution of the surrogate $P\left(\text{model}|\text{data}\right)$:

$$\mathcal{G}^* = \arg\max_{\mathcal{G}} \mathbb{E}_{P(\text{model}|\text{data})}\left[\xi_P\left(\mathcal{G}|\text{model}\right)\right]. \tag{16}$$

F.1 SURROGATE MODEL ON GRAPHS

Here we consider a graph neural network (GNN) surrogate model to predict the average fitness of a graph as a Gaussian distribution, namely $P\left(f(\mathcal{G})\right) \sim \mathcal{N}\left(\xi_P(\mathcal{G}), \sigma^2(\mathcal{G})\right)$. We use a simple architecture that predicts the mean of the Gaussian from the last hidden layer activations, $h_W(\mathcal{G}) \in \mathbb{R}^D$, of the GNN, where W are the weights in the GNN up to the last hidden layer.

Greedy search We denoted the size of dataset as N. The GNN weights are trained to predict the average fitness of the graph as a standard regression task:

$$\min_{W, W_{out}} \frac{\beta}{2} \sum_{n=1}^{N} \left(\xi(\mathcal{G}_n) - \xi_P(\mathcal{G}_n)\right)^2, \quad \text{where} \quad \xi_P(\mathcal{G}_n) = W_{out}^T h_W(\mathcal{G}_n) \tag{17}$$

Algorithm 2 Greedy Search

```
Initialize generation P^0
for j < maximum generations do
    Collect the (ξ_i^k, G_i^k) from previous k ≤ j generations          ▷ Update dataset
    Train W and W_out on {(ξ_i^k, G_i^k)}_{n=1}^N                       ▷ Train GM
    Propose C new graph {G_i}_{i=1}^C, C >> M.                          ▷ Propose new candidates
    Rank {ξ_P(G_i|W, W_out)}_{i=1}^C on the proposals and pick the top K ▷ Prune candidates
    Update generation P^j
    for m < N do                                                        ▷ Train and evaluate each species
        for k < maximum parameter updates do
            Train policy π_{G_m}
        end for
        Evaluate the fitness ξ(G_m, θ_m)
    end for
end for
```

Thompson Sampling In practice, Thompson Sampling is very similar to the previous greedy search algorithm. Instead of picking the top action according to the best model parameters, at each generation, it draws a sample of the model and takes a greedy action under the sampled model.

Approximating Thompson Sampling using Dropout Performing dropout during inference can be viewed as an approximately sampling from the model posterior. At each generation, we will sample a single dropout mask for the surrogate model and rank all the proposed graphs accordingly.

Algorithm 3 Thompson Sampling using Bayesian Neural Networks

Initialize generation $\mathcal{P}^0$
for $j <$ maximum generations **do**
  Collect the $(\xi_i^k, \mathcal{G}_i^k)$ from previous $k \leq j$ generations ▷ Update dataset
  Train W and W_{out} on $\{(\xi_i^k, \mathcal{G}_i^k)\}_{n=1}^N$ ▷ Train GM
  Propose $\mathcal{C}$ new graph $\{\mathcal{G}_i\}_{i=1}^{\mathcal{C}}$, $\mathcal{C} >> M$. ▷ Propose new candidates
  Sample a model from the posterior of the weights.
    e.g. $\widetilde{W}, \widetilde{W}_{out} \sim P\left(W, W_{out}|\mathcal{D}\right) \approx \mathcal{N}\left([W, W_{out}], [W, W_{out}]\right)$
    (similar to DropConnect Wan et al. (2013))
  Rank $\{\xi_P(\mathcal{G}_i|\widetilde{W}, \widetilde{W}_{out})\}_{i=1}^{\mathcal{C}}$ on the proposals and pick the top $\mathcal{K}$
  for $m < \mathcal{N}$ **do** ▷ Train and evaluate each species
    for $k <$ maximum parameter updates **do**
      Train policy $\pi_{\mathcal{G}_m}$
    end for
    Evaluate the fitness $\xi(\mathcal{G}_m, \theta_m)$
  end for
end for

Algorithm 4 Thompson Sampling with Dropout

Initialize generation $\mathcal{P}^0$
for $j <$ maximum generations **do**
  Collect the $(\xi_i^k, \mathcal{G}_i^k)$ from previous $k \leq j$ generations ▷ Update dataset
  Train W and W_{out} on $\{\mathcal{G}_n, \xi(\mathcal{G}_n)\}_{n=1}^N$ using dropout rate 0.5 on the inputs of the fc layers.
  Propose $\mathcal{C}$ new graph $\{\mathcal{G}_i\}_{i=1}^{\mathcal{C}}$, $\mathcal{C} >> M$. ▷ Propose new candidates
  Sample a dropout mask m_i for the hidden units
  Rank $\{\xi_P(\mathcal{G}_i|W, W_{out}, \mathrm{m}_i)\}_{i=1}^J$ on the proposals and pick the top $\mathcal{K}$
  for $m < \mathcal{N}$ **do** ▷ Train and evaluate each species
    for $k <$ maximum parameter updates **do**
      Train policy $\pi_{\mathcal{G}_m}$
    end for
    Evaluate the fitness $\xi(\mathcal{G}_m, \theta_m)$
  end for
end for