NERVENET: LEARNING STRUCTURED POLICY WITH GRAPH NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

We address the problem of learning structured policies for continuous control. In traditional reinforcement learning, policies of agents are learned by MLPs which take the concatenation of all observations from the environment as input for predicting actions. In this work, we propose NerveNet to explicitly model the structure of an agent, which naturally takes the form of a graph. Specifically, serving as the agent's policy network, NerveNet first propagates information over the structure of the agent and then predict actions for different parts of the agent. In the experiments, we first show that our NerveNet is comparable to state-of-the-art methods on standard MuJoCo environments. We further propose our customized reinforcement learning environments for benchmarking two types of structure transfer learning tasks, i.e., *size and disability transfer*. We demonstrate that policies learned by NerveNet are significantly better than policies learned by other models and are able to transfer even in a zero-shot setting.

1 INTRODUCTION

Deep reinforcement learning (RL) has received increasing attention over the past few years, with the recent success of applications such as playing Atari Games, Mnih et al. (2015), and Go, (Silver et al., 2016; 2017). Significant advances have also been made in robotics using the latest RL techniques,e.g., Levine et al. (2016); Gu et al. (2017).

Many RL problems feature agents with multiple dependent controllers. For example, humanoid robots consist of multiple physically linked joints. Action to be taken by each joint or the body should thus not only depend on its own observations but also on actions of other joints.

Previous approaches in RL typically use multi-layer perceptrons (MLP) to learn the agent's policy. In particular, MLP takes the concatenation of observations from the environment as input, which may be measurements like positions, velocities of body and joints in the current time instance. The MLP policy then predicts actions to be taken by every joint and body. Thus the job of the MLP policy is to discover the latent relationships between observations. This typically leads to longer training times, requiring more exposure of the agent to the environment. In our work, we aim to exploit the body structure of an agent, and physical dependencies that naturally exist in such agents.

We rely on the fact that bodies of most robots and animals have a discrete graph structure. Nodes of the graph may represent the joints, and edges represent the (physical) dependencies between them. In particular, we define the agent's policy using a Graph Neural Network, Scarselli et al. (2009)), which is a neural network that operates over graph structures. We refer to our model as NerveNet due to the resemblance of the neural nervous system to a graph. NerveNet propagates information between different parts of the body based on the underlying graph structure before outputting the action for each part. By doing so, NerveNet can leverage the structure information encoded by the agents's body which is advantageous in learning the correct inductive bias, thus be less prone to overfitting. Moreover, NerveNet is naturally suitable for structure transferring tasks as most of the model weights are shared across nodes and edges respectively.

We first evaluate our NerveNet on standard RL benchmarks such as the OpenAI Gym, Brockman et al. (2016) which stem from MuJoCo. We show that our model achieves comparable results to state-of-the-art MLP based methods. To verify our claim regarding the structure transfer, we introduce our customized RL environments which are based on the ones of Gym. Two types of structure

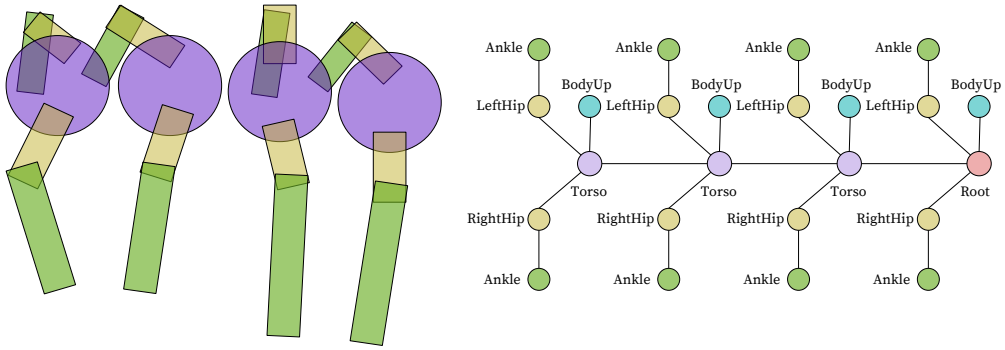


Figure 1: Visualization of the graph structure of the `CentipedeEight` agent in our environment. This agent is later used for testing the ability of transfer learning of our model. Since in this agent, each `body` node is paired with at least one `joint` node, we omit the `body` nodes and fill up the position with the corresponding `joint` nodes. By omitting the `body` nodes, a more compact graph is constructed, the details of which are illustrated in the experimental section.

transfer tasks are designed, *size transfer* and *disability transfer*. In particular, *size transfer* focuses on the scenario in which policies are learned for small-sized agents (simpler body structure) and applied directly to large-sized agents which are composed by some repetitive components shared with the small-sized agent. Secondly, *disability transfer* investigates scenarios in which policies are learned for one agent and applied to the same agent with some components disabled. Our experiments demonstrate that for structure transfer tasks our NerveNet is significantly better than all other competitors, and can even achieve zero-shot learning for some agents.

2 NERVENET

In this section, we first introduce the notation and basic settings of RL problems. We then explain how to construct the graph for our agents, followed by the description of the NerveNet. Finally, we describe the learning algorithm for our model.

We formulate the locomotion control problems as an infinite-horizon discounted Markov decision process (MDP). To fully describe the MDP for continuous control problems which include locomotion control, we define the state space or observation space as $\mathcal{S}$ and action space $\mathcal{A}$. To interact with the environments, the agent generates its stochastic policy $\pi_\theta(a^\tau|s^\tau)$ based on the current state $s^\tau \in \mathcal{S}$, where $a^\tau \in \mathcal{A}$ is the action and θ are the parameters of the policy function. The environment on the other hand, produces a reward $r(s^\tau, a^\tau)$ for the agent, and the agent's objective is to find a policy that maximizes the expected reward.

2.1 GRAPH CONSTRUCTION

In real life, skeletons of most robots and animals have discrete graph structures, and are most often trees. Simulators such as the MuJoCo engine Todorov et al. (2012), organize the agents using an XML-based kinematic tree. In our experiments, we will use the tree graphs as per MuJoCo's engine. Note that our model can be directly applied to arbitrary graphs. In particular, we assume two types of nodes in our tree: `body` and `joint`. The `body` node is an abstract node with physical structures inside, which is used to construct the kinematic tree via nesting. The `joint` node represents the degrees of freedom of motion between the two `body` nodes. Take a simple humanoid as an example; the `body` nodes `Thigh` and `Shin` are connected via the `Knee`, where `Knee` is a hinge `joint`. We further add a `root` node which observes additional information about the agent. For example, in the `Reacher` agent in MuJoCo, the `root` node will have access to the target position of the agent. We build edges follow the tree graph. Fig. 1 illustrates the graph structure of an example agent, `CentipedeEight`. We depict the sketch of the agent and its corresponding graph in the left and right parts of the figure respectively. Note that edges correspond to physical connections

of `joint` nodes. Different elements of the agent are parsed into nodes with different colors. The `BodyUp` node is not shown in the sketch, but it controls the rotation of up/down direction of the agent. Further details are provided in the experimental section.

2.2 NERVENET AS POLICY NETWORK

We now turn to NerveNet which builds on top of GNNs and servers as a policy network. Before delving into details, we first introduce our notation. We then specify the input model which helps to initialize the hidden state of each node. We further introduce the propagation model that updates these hidden states. At last, we describe the output model.

We denote the graph structure of the agent as $G = (V, E)$ where V and E are the sets of nodes and edges, respectively. We focus on the directed graphs as the undirected case can be easily addressed by splitting one undirected edge into two directed edges. We denote the out-going neighborhood of node u as $\mathcal{N}_{out}(u)$ which contains all endpoints v with (u, v) being an edge in the graph. Similarly, we denote the in-coming neighborhood of node u as $\mathcal{N}_{in}(u)$. Every node u has an associated node type $p_u \in \{1, 2, \ldots, P\}$, which in our case corresponds to `body`, `joint` and `root`. We also associate an edge type $c_{(u,v)} \in \{1, 2, \ldots, C\}$ with each edge (u, v). Node type can help in capturing different importances across nodes. Edge type can be used to describe different relationships between nodes, and thus propagate information between them differently. One can also add more than one edge type to the same edge which results in a multi-graph. We stick to simple graphs for simplicity. One interesting fact is that we have two notions of "time" in our model. One is the time step in the environment which is the typical time coordinate for RL problems. The other corresponds to the internal propagation step of NerveNet. These two coordinates work as follows. At each time step of the environment, NerveNet receives observation from the environment and performs a few internal propagation steps in order to decide on the action to be taken by each node. To avoid confusion, throughout this paper, we use τ to describe the time step in the environment and t for the propagation step.

2.2.1 INPUT MODEL

For each time step τ in the environment, the agent receives an observation $s^\tau \in \mathcal{S}$. The observation vector s^τ is the concatenation of observations of each node. We denote the elements of observation vector s^τ corresponding to node u with x_u. From now on, we drop the time step in the environment to derive the model for simplicity. The observation vector goes through an input network to obtain a fixed-size state vector as follows:

$$h_u^0 = F_{\text{in}}(x_u). \tag{1}$$

¡¡¡¡¡¡¡ HEAD where the subscript and superscript denote the node index and propagation step, respectively. Here, F_{in} may be a FC network and h_u^0 the state vector of node u at propagation step 0. ======= where the subscript and superscript denote the node index and propagation step respectively. Here, F_{in} may be a MLP and h_u^0 is the state vector of node u at propagation step 0. ¿¿¿¿¿¿¿ 329f560f8d292f32ec9604e1640be398e1bfc60d Note that we may need to pad zeros to the observation vectors if different nodes have observations of different sizes.

2.2.2 PROPAGATION MODEL

We now describe the propagation model of our NerveNet which mimics a synchronous message passing system studied in distributed computing Attiya & Welch (2004). We will show how the state vector of each node is updated from one propagation step to the next. This update process is recurrently applied during the whole propagation. We leave the details to the appendix.

Message Computation In particular, at propagation step t, for every node u, we have access to a state vector h_u^t. For every edge $(u, v) \in \mathcal{N}_{out}(u)$, node u computes a message vector as below,

$$m_{(u,v)}^t = M_{c_{(u,v)}}(h_u^t), \tag{2}$$

where $M_{c_{(u,v)}}$ is the message function which may be an identity mapping or a MLP. Note that the subscript $c_{(u,v)}$ indicates that edges of the same edge type share the same instance of the message function. For example, the second `torso` in Fig. 1 sends a message to the first and third `torso`, as well as the `LeftHip`, `RightHip` and `BodyUp`.

Message Aggregation Once every node finishes computing messages, we aggregate messages sent from all in-coming neighbors of each node. Specifically, for every node u, we perform the following aggregation:

$$\bar{m}_u^t = A(\{h_v^t | v \in \mathcal{N}_{in}(u)\}), \tag{3}$$

where A is the aggregation function which may be a summation, average or max-pooling function. Here, $\bar{m}_u^t$ is the aggregated message vector which contains the information sent from the node's neighborhood.

States Update We now update every node's state vector based on both the aggregated message and its current state vector. In particular, for every node u, we perform the following update:

$$h_u^{t+1} = U_{p_u}(h_u^t), \tag{4}$$

where U is the update function which may be a gated recurrent unit (GRU), a long short term memory (LSTM) unit or a MLP. From the subscript p_u of U, we can see that nodes of the same node type share the same instance of the update function. The above propagation model is then recurrently applied for a fixed number of time steps T to get the final state vectors of all nodes, i.e., $\{h_u^T | u \in V\}$.

2.2.3 Output Model

In RL, agents typically use MLP policy, where the network outputs the mean of the Gaussian distribution for the actions, while the standard deviation is a trainable vector Schulman et al. (2017). In our output model, we also treat standard deviation in the same way.

However, instead of predicting the action distribution of all nodes by single network all at once, we output for individual node. We denote the set of nodes which are assigned controllers for the actuators as $\mathcal{O}$. For each of such nodes, a MLP takes its final state vectors $h_{u \in \mathcal{O}}^T$ as input and produces the mean of the action of the Gaussian policy for the corresponding actuator. For each output node $u \in \mathcal{O}$, we define its output type as q_u. Nodes with the same node type should share the instance of MLP O_{q_u}. For example, in Fig. 1, two `LeftHip` should have a shared controller. Therefore, we have the following output model:

$$\mu_{u \in \mathcal{O}} = O_{q_u}(h_u^T), \tag{5}$$

where $\mu_{u \in \mathcal{O}}$ is the mean value for action applied on each actuator. In practice, we found that we can force controllers of different output type to share one unified controller which does not hurt the performance By integrating the produced Gaussian policy for each action, the probability density of stochastic policy is calculated as

$$\pi_\theta(a^\tau | s^\tau) = \prod_{u \in \mathcal{O}} \pi_{\theta,u}(a_u^\tau | s^\tau) = \prod_{u \in \mathcal{O}} \frac{1}{\sqrt{2\pi\sigma_u^2}} e^{(a_u^\tau - \mu_u)^2/(2\sigma_u^2)}, \tag{6}$$

where $a^\tau \in \mathcal{A}$ is the output action, and σ_u is the variable standard deviation for each action. Here, θ represents the parameters of the policy function.

2.3 Learning Algorithm

To interact with the environments, the agent generates its stochastic policy $\pi_\theta(a^\tau | s^\tau)$ after several propagation steps. The environment on the other hand, produces a reward $r(s^\tau, a^\tau)$ for the agent, and transits to the next state with transition probability $P(s^{\tau+1} | s^\tau)$. The target of the agent is to maximize its cumulative return

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{\tau=0}^{\infty} \gamma^\tau r(s^\tau, a^\tau) \right]. \tag{7}$$

To optimize the expected reward, we apply the proximal policy optimization (PPO) by Schulman et al. (2017) to our model. In PPO, the agents alternate between sampling trajectories with the latest policy and performing optimization on surrogate objective using the sampled trajectories. The algorithm tries to keep the KL-divergence of the new policy and the old policy within the trust

region. To achieve that, PPO clips the probability ratio and adds KL-divergence penalty term to the loss. The likelihood ratio is defined as $r^\tau(\theta;\theta_{old}) = \pi_\theta(a^\tau|s^\tau)/\pi_{\theta_{old}}(a^\tau|s^\tau)$. Following the notation and algorithm of PPO, our NerveNet tries to minimize the summation of the original loss in Eq. (7), KL-penalty and value function loss which is defined as:

$$\begin{aligned}\tilde{J}(\theta) =& J(\theta) - \beta L_{KL}(\theta) - \alpha L_V(\theta) \\ =& \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{\infty} \min\left(\hat{A}^\tau r^\tau(\theta), \hat{A}^\tau \text{ clip}\left(r^\tau(\theta), 1-\epsilon, 1+\epsilon\right)\right)\right] \\ & - \beta \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{\infty} \text{KL}\left[\pi_\theta(:|s^\tau)|\pi_{\theta_{old}}(:|s^\tau)\right]\right] - \alpha \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{\infty}\left(V_\theta(s^\tau) - V(s^\tau)^{\text{target}}\right)^2\right], \quad (8)\end{aligned}$$

where the $\hat{A}_t$ is the generalized advantage estimation (GAE) calculated using algorithm from Schulman et al. (2015b), and the ϵ is the clip value, which we choose to be 0.2. β is a dynamical coefficient adjusted to keep the KL-divergence constraints, and the α is used to balance the value loss. Note that in Eq. (8), $V(s_t)^{\text{target}}$ is the target state value in accordance with the GAE method. To optimize the $\tilde{J}(\theta)$, PPO make use of the policy gradient in Sutton et al. (2000) to do first-order gradient descent optimization.

Value Network To produce the state value $V_\theta(s^\tau)$ for given observation s^τ, we have several alternatives: (1) using one GNN as the policy network and using one MLP as the value network (NerveNet-MLP); (2) using one GNN as policy network and using another GNN as value network (NerveNet-2) (without sharing the parameters of the two GNNs); (3) using one GNN as both policy network and value network (NerveNet-1). The GNN for value network is very similar to the GNN for policy network. The output for value GNN is a scalar instead of a vector of mean action. We will compare these variants in the experimental section.

3 Related Work

Reinforcement Learning Reinforcement learning (RL) has recently achieved huge success in a variety of applications. Boosted by the progress of deep neural networks, Krizhevsky et al. (2012), agents are able to solve problems from Atari Games and beat the best human players in the game of Go (Mnih et al., 2015; Silver et al., 2016; 2017). For continuous control problems, based on simulation engines like MuJoCo Todorov et al. (2012), numerous algorithms have been proposed to optimize agent's expected reward (Schulman et al., 2017; 2015a; Heess et al., 2017).

Recently, more and more research experiments have been done for transfer learning tasks Taylor & Stone (2009) in RL which mainly focus on transferring the policy learned from one environment to another. In Rajeswaran et al. (2017), the authors show that agents in reinforcement learning are prone to over-fitting, and the learned policy generalizes poorly from one environment to the other. Gupta et al. (2017) try to increase the transferability via learning invariant visual features. Efforts have also been made from the meta-learning perspective (Duan et al., 2016; Finn et al., 2017b;a). In Wulfmeier et al. (2017), the authors propose a method of transfer learning by using imitation learning.

As for exploiting the structure in RL, most hierarchical reinforcement learning algorithms (Kulkarni et al., 2016; Vezhnevets et al., 2017) focus on modeling intrinsic motivation. In Hausknecht & Stone (2015), the authors exploit the structure of the action space. Graph has been used in RL problems. In Metzen (2013); Mabu et al. (2007); Shoeleh & Asadpour (2017); Mahadevan & Maggioni (2007), graph is used to learn the representation of environment. But these methods are limited to problems with simple dynamical models like 2d-navigation, and thus these problems are usually solved via model-based method. However, for complex multi-joint agents, learning the dynamical model and predicting the transition of states are time consuming and biased.

Currently, for problems of training a model-free multi-joint agents under complex physical environment, relatively little attention has been paid to modeling the physical structure of the agents.

Graph Neural Networks There have been many efforts to generalize neural networks to graph-structured data. One direction is based on convolutional neural networks (CNNs). In (Bruna et al.,

2014; Defferrard et al., 2016; Kipf & Welling, 2017), CNNs are employed in the spectral domain relying on the graph Laplacian matrix. While (Goller & Kuchler, 1996; Duvenaud et al., 2015) used hash functions so that CNN can be applied to graphs. Another popular direction is based on recurrent neural networks (RNNs) (Goller & Kuchler, 1996; Gori et al., 2005; Scarselli et al., 2009; Socher et al., 2011; Li et al., 2015; Tai et al., 2015).

Among RNN based methods, some of them are only applicable to special structured graph, e.g., sequences or trees, (Socher et al., 2011; Tai et al., 2015). One class of models which are applicable to general graphs is called, graph neural networks (GNNs), Scarselli et al. (2009). The inference procedure, a.k.a. forward pass, is a fixed-length propagation process which resembles synchronous message passing system in the theory of distributed computing, Attiya & Welch (2004). Nodes in the graph have state vectors which are recurrently updated based on their history and received messages. One of the representative work of GNNs, i.e., gated graph neural networks (GGNNs) by Li et al. (2015), uses gated recurrent unit to update the state vectors. Learning of such a model can be achieved by the back-propagation through time (BPTT) algorithm or recurrent back-propagation, Chauvin & Rumelhart (1995). It has been shown that GNNs, (Li et al., 2015; 2017; Qi et al., 2017) have a great capacity and achieve state-of-the-art performance in many applications which involve graph-structured data. In this paper, we model the structure of the reinforcement learning agents using GNN.

4 EXPERIMENTS

In this section, we first verify the effectiveness of NerveNet on standard MuJoCo environments in OpenAI Gym. We then investigate the transfer abilities of NerveNet and other competitors by customizing some of those environments.

4.1 COMPARISON ON STANDARD BENCHMARKS OF MUJOCO

Experimental Setting We compare NerveNet with the standard MLP models utilized by Schulman et al. (2017) and another baseline which is constructed as follows. We first remove the physical graph structure and then introducing an additional super node which connects to all other nodes. This results in a singly rooted depth-1 tree. We refer to this baseline as TreeNet. The propagation model of TreeNet is similar to NerveNet whereas the policy is predicted by first aggregating the information from all children and then feeding the state vector of root to the output model. This simpler model can serve as a baseline to verify how important the graph structure is.

We run experiments on 8 simulated continuous control benchmarks from the Gym Brockman et al. (2016) which is based on MuJoCo, Todorov et al. (2012). In particular, we use `Reacher`, `InvertedPendulum`, `InvertedDoublePendulum`, `Swimmer`, and four walking or running tasks: `HalfCheetah`, `Hopper`, `Walker2d`, `Ant`. We set the maximum train step to be 1 million for all environments as it is enough to solve them. Note that for `InvertedPendulum`, different from the original one in Gym, we add the distance penalty of the cart and velocity penalty so that the reward is more consistent to the `InvertedDoublePendulum`. This change of design also makes the task more challenging.

Results We do grid search to find the best hyperparameters and leave the details in the appendix 6.3. As the randomness might have a big impact on the performance, for each environment, we run 3 experiments with different random seeds and plot the average curves and the standard deviations. We show the results in Figure 2. From the figures, we can see that MLP with the same setup as in Schulman et al. (2017) works the best in most of tasks.[1] NerveNet basically matches the performance of MLP in terms of sample efficiency as well as the performance after it converges. In most cases, the TreeNet is worse than NerveNet which highlights the importance of keeping the physical graph structure.

[1]By applying the adaptive learning rate schedule from Schulman et al. (2017), we obtained better performances than the ones reported in original paper.

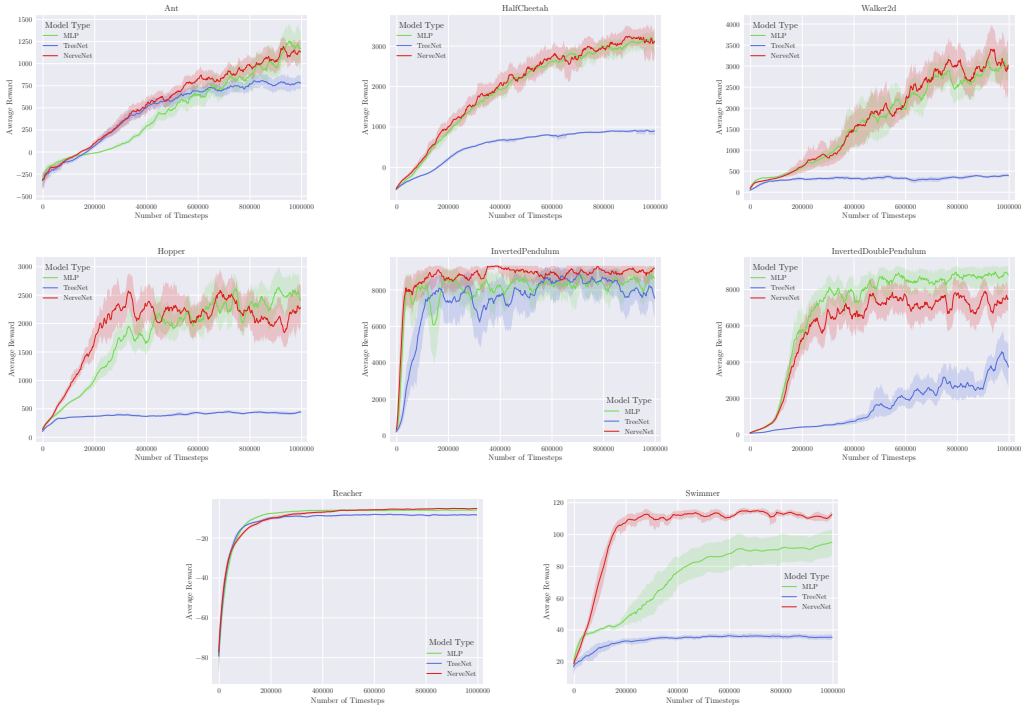


Figure 2: Results of MLP, TreeNet and NerveNet on 8 continuous control benchmarks from the Gym.

4.2 Structure Transfer Learning

We benchmark structure transfer learning by creating customized environments based on the existing ones from MuJoCo. We mainly investigate two types of structure transfer learning tasks. The first one is to train a model with an agent of small size (small graph) and apply the learned model to an agent with a larger size, i.e., *size transfer*. When increasing the size of agent, observation and action space also increase which makes learning more challenging. Another type of structure transfer learning is *disability transfer* where we first learn a model on the original agent and then apply it to the same agent with some components disabled. If one model over-fits the environment, disabling some components of the agent might bring catastrophic performance degradation. Note that for both transfer tasks, all factors of environments do not change except the structure of the agent.

Centipede We create the first environment where the agent is like a centipede. The goal of the agent is to run as fast as possible along the y-direction in the MuJoCo environment. The agent consists of repetitive torso bodies where each one has two legs attached. For two consecutive bodies, we add two actuators which control the rotation between them. Furthermore, each leg consists of a `thigh` and `shin`, which are controlled by two hinge actuators. By linking copies of torso bodies and corresponding legs, we create agents with different lengths. Specifically, the shortest Centipede is `CentipedeFour` and the longest one is `CentipedeFourty` due to the limit of supported resource of MuJoCo. For each time step, the total reward is the speed reward minus the energy cost and force feedback from the ground. Note that in practice, we found that training a `CentipedeFourteen` from scratch is already very difficult. For *size transfer* experiments, we create many instances which are listed in Table 2, like "Four2Six", "Six2Ten". For *disability transfer*, we create `CrippleCentipede` agents of which two back legs are disabled.

Snakes We also create a snake-like agent which is common in robotics, Crespi & Ijspeert (2008). We design the `Snake` environment based on the `Swimmer` model in Gym. The goal of the agent is to move as fast as possible. For details of the environment, please see the schematic figure 8.

Table 1: Performance of the Pre-trained models on `CentipedeSix` and `CentipedeSix`.

CentipedeFour	mean reward	reward std	max reward
NerveNet	2799.94	1247.21	3833.97
MLP	2398.52	1390.43	3936.3
TreeNet	1429.65	957.79	3021.76

CentipedeSix	mean reward	reward std	max reward
NerveNet	2507.12	738.46	2979.21
MLP	2793.05	1005.29	3529.58
TreeNet	2229.7	1109.44	3727.44

4.2.1 Experimental Settings

To fully investigate the performance of NerveNet, we build several baseline models for structure transfer learning which are explained below.

NerveNet For the NerveNet, since all the weights are exactly the same between the small-agent model and the large-agent model, we directly use the old weights trained on the small-agent model. When the large agent has repetitive structures, we further re-use the weights of the corresponding joints from the small-agent model.

MLP Pre-trained (MLPP) For the MLP based model, while transferring from one structure to another, the size of the input layer changes since the size of the observation changes. One straightforward idea is to reuse the weights from the first hidden layer to the output layer and randomly initialize the weights of the new input layer.

MLP Activation Assigning (MLPAA) Another way of making MLP transferable is assigning the weights of the small-agent model to the corresponding partial weights of the large-agent model and setting the remaining weights to be zero. Note that we do not add or remove any layers from the small-agent model to the large-agent except changing the size of the layers. By doing so, we can keep the output of the large-agent model to be same as the small-agent in the beginning, i.e., keeping the same initial policy.

TreeNet TreeNet is similar as the model described before. We apply the same way of assigning weights as MLPAA to TreeNet for the transfer learning task.

Random We also include the random policy which is uniformly sampled from the action space.

4.2.2 Results

Centipedes For the Centipedes environment, we first run experiments of all models on `CentipedeSix` and `CentipedeFour` to get the pre-trained models for transfer learning. We then examine the zero-shot performance where zero-shot means directly applying the model trained with one setting to the other without any fine-tuning. As we can see from Table 2, NerveNet outperforms all competitors on all settings, except in the Four2Ten scenario. Note that transferring from `CentipedeFour` is more difficult than from `CentipedesSix` since the situation where one torso connects to two neighboring torsos only happens beyond 4 bodies. TreeNet has a surprisingly good performance on tasks from `CentipedeFour`. However, by checking the videos, the learned agent is actually not able to "move" as good as other methods. The high reward is mainly due to the fact that TreeNet policy is better at standing and gaining alive bonus.

We fine-tune for both *size transfer* and *disability transfer* experiments and show the training curves in Fig. 3. From the figure, we can see that by using the pre-trained model, NerveNet significantly decreases the number of episodes required to reach the level of reward which is considered as solved. We found that for centipede, the bottleneck of learning for the agent is "how to stand". When training from scratch, it can be seen from the figure that almost 0.5 million time steps are spent on a very flat reward surface. By looking at the videos, we notice that this long time period is spent on learning to

Tasks	MLPAA	MLPP	Random	TreeNet	NerveNet
Four2Six	466.6/113.1	-23.7/-128.2	-5.7/-32	100.4/24.9	**587.5/133.5**
Four2Eight	107.1/18.5	-33.7/-188.2	-10.3/-45.6	84.7/10.2	**161.6/47.2**
Four2Ten	55.7/10.9	-26.2/-192.7	-5.5/-44.6	**518.5/453.9**	180.7/38.7
Four2Twelve	45.2/10.3	-24.8/-217.2	-9.9/-49.1	84.8/17	**261.4/38.9**
Six2Eight	92.9/21.1	-31/-188.4	-10.3/-45.8	394.9/321	**2586.6/1670.4**
Six2Ten	29.4/-9.9	-29.6/-196.9	-5.5/-44.4	257/44.6	**2154.9/952.1**
Six2Twelve	38.1/-18.4	-28.7/-220.9	-9.9/-49.1	157.1/20.4	**1421.3/367.6**
Six2Fourteen	33.3/-17.3	-27.6/-219.9	-10.5/-51.6	146.4/126.4	**1150.2/248.9**
Six2Thirty	29.6/-9.0	-4.6/-265.5	1.4/-69.2	96.7/20.1	**614.3/127.1**
Six2Forty	25.1/-19.2	-5.1/-275.5	-0.9/-74.5	374.4/51.6	**575.4/94.8**
Six2CpEight	169.1/36.2	-37.1/-145.6	0.1/-25.3	528.9/399	**2143.2/523.4**
Six2CpTen	55.8/12.8	-48.9/-155.7	-6.6/-28.4	389.7/277.8	**1904.6/504**
Six2CpTwelve	34.3/11.6	-34.7/-168.9	-6.2/-33.1	368.7/179.8	**1429.4/257.4**
Six2CpFourteen	48/11.9	-27.7/-187.5	-11.6/-41	214.2/41.4	**976.2/224.8**

Table 2: Performance of zero-shot learning on centipedes. For each task, we run the policy for 100 episodes and calculate the max and mean reward. Results are shown in the format of Max/Mean.

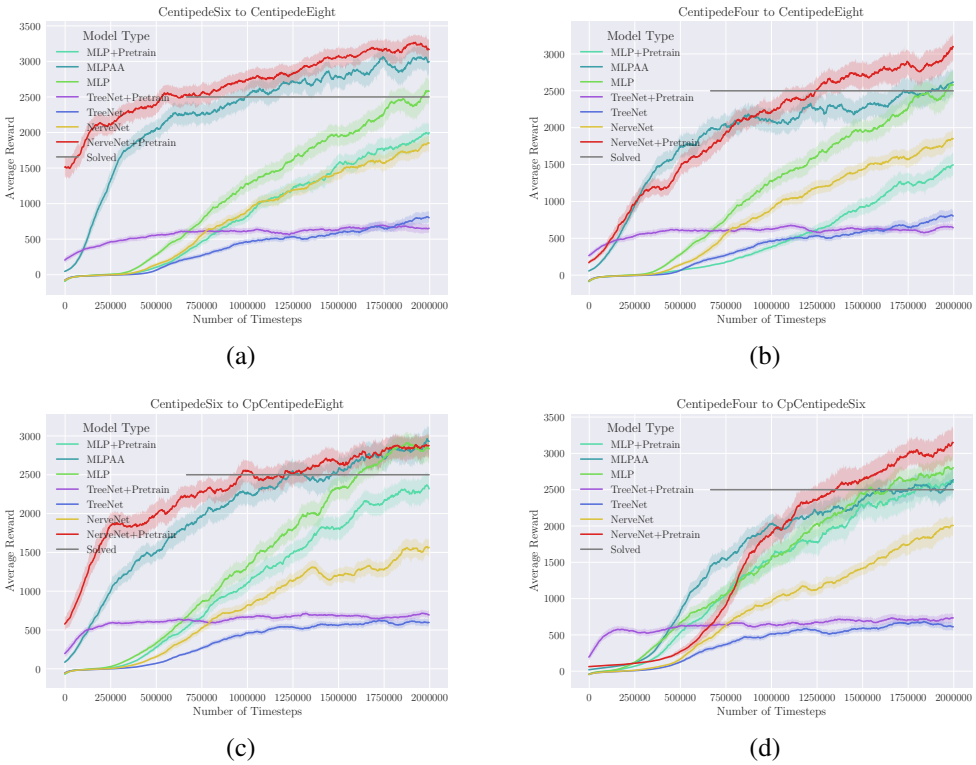


Figure 3: (a), (b): Results of fine-tuning for *size transfer* experiments. (c), (d) Results of fine-tuning for *disability transfer* experiments.

"stand". Therefore, the MLPAA agents, which copy the learned policy, are able to stand and bypass this time-consuming process and reach to a good performance in the end.

Moreover, by examining the result videos, we noticed that the "walk-cycle" behavior is observed for NerveNet but is not common for others. Walk-cycle are adopted for many insects in the world Biewener (2003). For example, six-leg ants uses tripedal gait, where the legs are used in two separate triangles alternatively touching the ground. More details of walk-cycle will be in the following section 4.3.

Tasks	MLPAA	MLPP	Random	TreeNet	NerveNet
SnakeThree2SnakeFour	-30.19	-12.36	3.86	-21.82	**325.48**
SnakeThree2SnakeFive	51.41	-16.48	20.59	-21.26	**314.92**
SnakeThree2SnakeSix	32.2	-19.67	7.6	-42.46	**282.43**
SnakeThree2SnakeSeven	23.53	-22.38	-2.28	-42.74	**256.29**
SnakeFour2SnakeFive	89.82	-16.54	20.59	-15.97	**342.88**
SnakeFour2SnakeSix	105.63	-20.45	7.6	-34.34	**351.85**
SnakeFour2SnakeSeven	115.12	-22.81	-2.28	-22.38	**313.15**

Table 3: Results on zero-shot transfer learning on snake agents. Each tasks are simulated for 100 episodes.

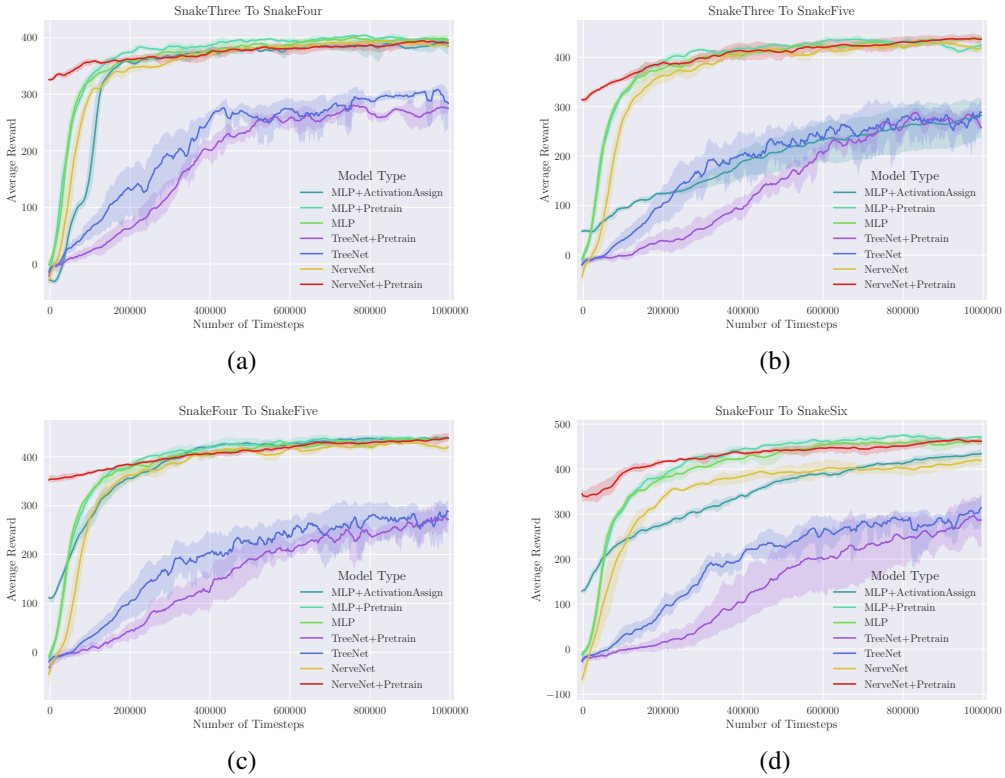


Figure 4: Results of finetuning on snake environments.

One possible reason is that the agent of MLP based method (MLPAA, MLPP) learns a policy that does not utilize all legs. From `CentipedeEight` and up, we do not observe any MLP agents to be able to coordinate all legs whereas almost all policies learned by NerveNet use all legs. Therefore, NerveNet is better at utilizing structure information and not over-fitting the environments.

Snakes The zero-shot performance for snakes is summarized in Table 3. As we can see, NerveNet has the best performance on all transfer learning tasks. In most cases, NerveNet has a starting reward value of more than 300, which is a pretty good policy since 350 is considered as solved for `snakeThree`. By looking at the videos, we found that agents of other competitors are not able to control the new actuators in the zero-shot setting. They either overfit to the original models, where the policy is completely useless in the new setting (e.g., the MLPAA is worse than random policy in `SnakeThree2SnakeFour`), or the new actuators are not able to coordinate with the old actuators trained before. While for NerveNet, the actuators are able to coordinate to its neighbors, regardless of whether they are new to the agents.

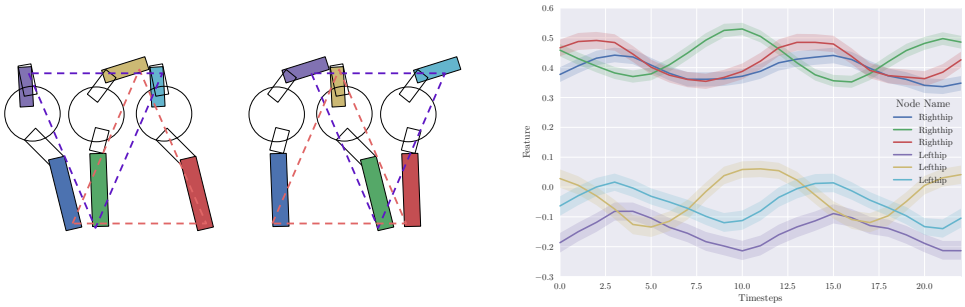


Figure 5: Diagram of the walk cycle. In the left figure, legs within the same triangle are used simultaneously. For each leg, we use the same color for their diagram on the left and their curves on the right.

We also summarize the training curves of fine-tuning in Fig. 4. We can observe that NerveNet has a very good initialization with the pre-trained model, and the performance increases with fine-tuning. When training from scratch, the NerveNet is less sample efficient compared to the MLP model which might be caused by the fact that optimizing our model is more challenging than MLP. Fine-tuning helps to improve the sample efficiency of our model by a large margin. At the same time, although the MLPAA has a very good initialization, its performance progresses slowly with the number of episodes increasing. In most experiments, the MLPAA and TreeNet could not even match the performance of its non-pretrained MLP baseline.

4.3 Interpreting the Learned Representations

In this section, we try to visualize and interpret the learned representations. We extract the final state vectors of nodes of NerveNet trained on `CentipedeEight`. We then apply 1-D and 2-D PCA on the node representations. In Fig. 6, we notice that each pair of legs is able to learn invariant representations, despite their different position in the agent. We further plot the trajectory density map in the feature map. By recording the period of the walk-cycle, we plot the transformed features of the 6 legs on Fig. 5. As we can see, there is a clear periodic behavior of our hidden representations learned by our model. Furthermore, the representations of adjacent left legs and the adjacent right legs demonstrate a phase shift, which further proves that our agents are able to learn the walk-cycle without any additional supervision.

4.4 Comparison of Model Variants

As mentioned in the model section, we have several variants of NerveNet, based on the type of network we use for the policy/value representation. We compare all variants. Again, we run experiments for each task three times. The details of hyper-parameters are left in the Appendix. For each environment, we train the network for one million time steps, with batch size 2050 for one update.

As we can see from Fig. 7, the NerveNet-MLP and NerveNet-2 variants perform better than NerveNet-1. One potential reason is that sharing the weights of the value and policy networks makes the trust-region based optimization methods, like PPO, more sensitive to the weight α of the value function in 8. Based on the figure, choosing α to be 1 is not giving good performance on the tasks we experimented on.

5 Conclusion

In this paper, we aimed to exploit the body structure of Reinforcement Learning agents in the form of graphs. We introduced a novel model called NerveNet which uses a Graph Neural Network to represent the agent's policy. At each time instance of the environment, NerveNet takes observations for each of the body joints, and propagates information between them using non-linear messages

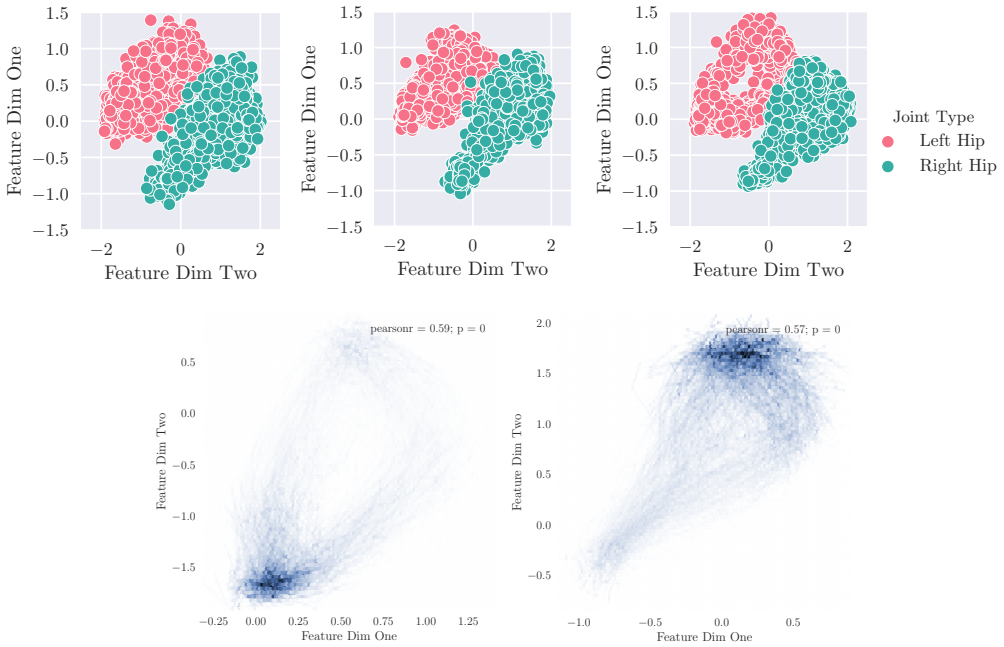


Figure 6: Results of visualization of feature distribution and trajectory density.

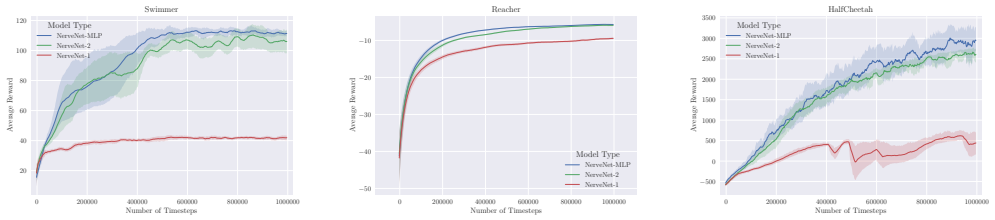


Figure 7: Results of several variants of NerveNet for the reinforcement learning agents.

computed with a neural network. Propagation is done through the edges which represent natural dependencies between joints, such as physical connectivity. We experimentally show that our NerveNet achieves comparable performance to state-of-the-art methods on standard MuJoCo environments. We further propose our customized reinforcement learning environments for benchmarking two types of structure transfer learning tasks, i.e., *size and disability transfer*. We demonstrate that policies learned by NerveNet are significantly better than policies learned by other models and are able to transfer even in a zero-shot setting.

REFERENCES

Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

Andrew A Biewener. *Animal locomotion*. Oxford University Press, 2003.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

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

Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

Alessandro Crespi and Auke Jan Ijspeert. Online optimization of swimming and crawling in an amphibious snake robot. *IEEE Transactions on Robotics*, 24(1):75–87, 2008.

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

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl 2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

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.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017a.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *arXiv preprint arXiv:1709.04905*, 2017b.

Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pp. 347–352. IEEE, 1996.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *IJCNN*, 2005.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *ICRA*, 2017.

Abhishek Gupta, Coline Devin, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *arXiv preprint arXiv:1703.02949*, 2017.

Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015.

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.

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

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pp. 3675–3683, 2016.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *J. Mach. Learn. Res.*, 17(1):1334–1373, January 2016. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2946645.2946684`.

Ruiyu Li, Makarand Tapaswi, Renjie Liao, Jiaya Jia, Raquel Urtasun, and Sanja Fidler. Situation recognition with graph neural networks. 2017.

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

Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A graph-based evolutionary algorithm: Genetic network programming (gnp) and its extension using reinforcement learning. *Evolutionary Computation*, 15(3):369–398, 2007.

Sridhar Mahadevan and Mauro Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(Oct):2169–2231, 2007.

Jan Hendrik Metzen. Learning graph-based representations for continuous reinforcement learning domains. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 81–96. Springer, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Xiaojuan Qi, Renjie Liao, Jiaya Jia, Sanja Fidler, and Raquel Urtasun. 3d graph neural networks for rgbd semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5199–5208, 2017.

Aravind Rajeswaran, Kendall Lowrey, Emanuel Todorov, and Sham Kakade. Towards generalization and simplicity in continuous control. *arXiv preprint arXiv:1703.02660*, 2017.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE TNN*, 2009.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1889–1897, 2015a.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015b.

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

Farzaneh Shoeleh and Masoud Asadpour. Graph based skill acquisition and transfer learning for continuous reinforcement learning domains. *Pattern Recognition Letters*, 87:104–116, 2017.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136, 2011.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *ACL*, 2015.

Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.

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, 2012.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

Markus Wulfmeier, Ingmar Posner, and Pieter Abbeel. Mutual alignment transfer learning. *arXiv preprint arXiv:1707.07907*, 2017.

6 APPENDIX

6.1 DETAILS OF NERVENET

We use MLP to compute the messages which uses $tanh$ nonlinearities as the activation function. We do a grid search on the size of the MLP to compute the messages, the details of which is listed in the below table4.

Throughout all of our experiments, we use average aggregation and GRU as update function.

6.2 GRAPH OF AGENT

In MuJoCo, we observe that most `body` nodes are paired with one and only one `joint` node. Thus, we simply merge the two paired nodes into one. We point out that this model is very compact, and is the standard graph we use in our experiments.

In the Gym environments, observation for the `joint` nodes normally includes the angular velocity, twist angle and optionally the torque for the hinge `joint`, and position information for the positional `joint`. For the `body` nodes, velocity, inertia, and force are common observations. For example in the centipede environment 1, the `LeftHip` node will receive the angular velocity ϖ_j, the twist angle θ_j.

6.3 HYPERPARAMETER SEARCH

For MLP, we run grid search with the hidden size from two layers to three layers, and with hidden size from 32 to 256. For NerveNet, to reduce the time spent on grid search, we constrain the propagation network and output network to be the same shape. Similarly, we run grid search with the network's hidden size, and at the same time, we run a grid search on the size of node's hidden states from 32 to 64. For the TreeNet, we run similar grid search on the node's hidden states and output network's shape.

For details of hyperparameter search, please see the attached table 4, 5, 6

Table 4: Hyperparameter grid search options for MLP.

MLP	Value Tried
Network Shape	[64, 64], [128,128], [256, 256], [64,64,64]
Number of Iteration Per Update	10, 20
Use KL Penalty	Yes, No
Learning Rate Scheduler	Linear Decay, Adaptive, Constant

Table 5: Hyperparameter grid search options for NerveNet.

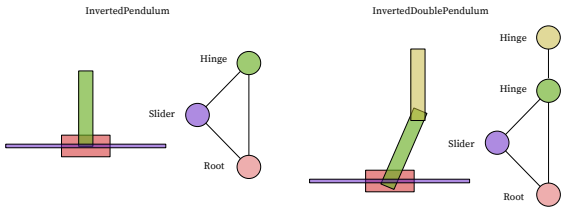
NerveNet	Value Tried
Network Shape	[64, 64], [128,128], [256, 256]
Number of Iteration Per Update	10, 20
Use KL Penalty	Yes, No
Learning Rate Scheduler	Linear Decay, Adaptive, Constant
Number of Propogation Steps	3, 4, 5, 6
NerveNet Variants	NerveNet-1, NerveNet-2, NerveNet-MLP
Size of Nodes' Hidden State	32, 64, 128
Merge `joint` and `body` Node	Yes, No
Output Network	Shared, Separate
Disable Edge Type	Yes, No
Add Skip-connection from / to `root`	Yes, No

Table 6: Hyperparameter grid search options for TreeNet.

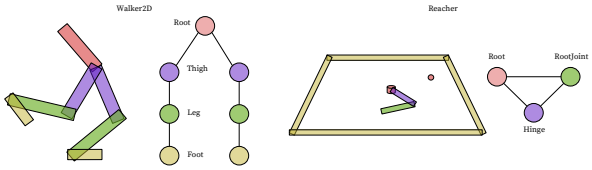
TreeNet	Value Tried
Network Shape	[64, 64], [128,128], [256, 256]
Number of Iteration Per Update	10, 20
Use KL Penalty	Yes, No
Learning Rate Scheduler	Linear Decay, Adaptive, Constant

6.4 Schematic Figures of the Parsed Agents

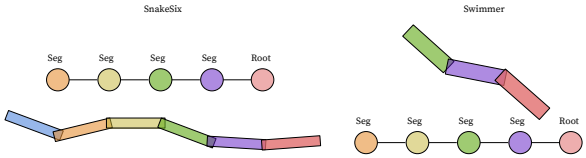
In this section, we also plot the schematic figures of the agents for readers' reference. Graph structures could be automatically parsed from the MuJoCo XML configuration files. Given any MuJoCo configuration files, or other configuration files organized in a kinematic tree, we could construct the graph for the agents.



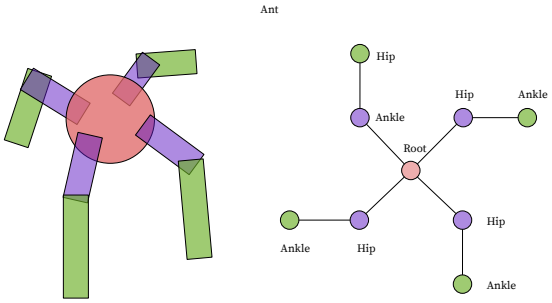
(a) Schematic diagram of the `InvertedPendulum` and `InvertedDoublePendulum`.



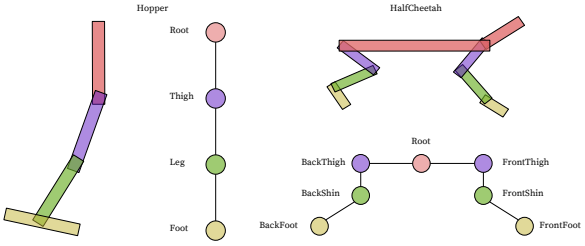
(b) Schematic diagram of the `Walker2d` and `Reacher`.



(c) Schematic diagram of the `SnakeSix` and `Swimmer`.



(d) Schematic diagram of the `Ant`.



(e) Schematic diagram of the `Hopper` and `HalfCheetah`.

Figure 8: Schematic diagram of the agents with their auto-parsed graph structures next to it.