NERVENET: LEARNING STRUCTURED POLICY WITH GRAPH NEURAL NETWORKS

by

Tingwu Wang

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

NerveNet: Learning Structured Policy with Graph Neural Networks

Tingwu Wang
Master of Science
Graduate Department of Computer Science
University of Toronto
2017

We address the problem of learning structured policies for continuous control. In traditional reinforcement learning, policies of agents are learned by multi-layer perceptrons (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 more transferable and generalized than policies learned by other models and are able to transfer even in a zero-shot setting.

Contents

List of Figures

List of Tables

Chapter 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 [42], and Go [53, 55]. Significant advances have also been made in robotics using the latest RL techniques,e.g. [33, 23].

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 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, [48]), 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, [7] 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 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.

The main contribution of this paper is the following: We explore the problem of learning transferable and generalized features by incorporating structure prior using graph neural network. NerveNet permits powerful transfer learning from one structure to the others, which has been beyond the ability of previous models. NerveNet is also more robust and has better potential of multi-task learning.

Chapter 2

Preliminary

The two major elements of this paper are reinforcement learning and graph neural network. In this chapter, we introduce the basic knowledge on reinforcement learning, while the detail of graph neural network is introduced in the next chapter.

2.1 Reinforcement Learning for Continuous Control

To control via reinforcement learning, an agent needs to interact with the environment. Starting from the current state, the agent chooses an action based on the observation. The environment receives the action and a transition to the next state is made. This process is repeated until the episode ends.

2.1.1 Markov Decision Process in Continuous Control

To start with, it is worth pointing out that the focus of this paper is finite-horizon Markov decision process (MDP), which means that the number of timesteps in one episode is limited and bounded, and the agent observes the whole state space. In another word, what happens before (history of observations) is not needed to for optimal control. We define the MDP to be a tuple of $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ denotes the state space of our problem, $\mathcal{A}$ denotes the action space of the agents. $\mathcal{P}$ denotes the transition probability distribution, which could be written as $p(S', S, a)$. Distribution $p(S', S, a) \to \mathbb{R}$ is not known by the agent in advance. $\mathcal{R}$ similarly describes the reward generated by the environment $r(S) \to \mathbb{R}$.
More specifically, we use the notation of γ to describe the discount factor and $s^{t=0}$ as the initial state, which follows the distribution of starting states $p_0(s^{t=0})$.
The expected total reward is the objective function for the agent to optimize, which is

$$\eta(\pi) = \mathbb{E}_{s^{t=0}} \left[\sum_{t=0}^{T=\infty} \gamma^t r(s_t) \right]. \tag{2.1}$$

In the continuous control problem, policy based reinforcement learning is usually used. In value-based methods [42, 53, 55], by estimating the value function of given state and actions available, the agent could choose the best action. While in continuous control problems, since the actions space $\mathcal{S}$ is continuous, there is infinite number of action choices. Therefore, more often, policy based methods are applied. And in our case, the Gaussian

policy is used. If we assume the action space is I dimentional, then

$$\pi(\mathbf{a}^t|s^t) = \prod_{i=0}^{I} \pi_i(a_i^t|s^t) = \prod_{i=0}^{I} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{(a_i^t-\mu_i)^2/(2\sigma_i^2)}, \tag{2.2}$$

where

$$\mathbf{a}^t = \begin{pmatrix} a_1^t \\ a_2^t \\ a_3^t \\ ... \\ a_{I-1}^t \\ a_I^t \end{pmatrix}, \ \boldsymbol{\mu}^t = \begin{pmatrix} \mu_1^t \\ \mu_2^t \\ \mu_3^t \\ ... \\ \mu_{I-1}^t \\ \mu_I^t \end{pmatrix}, \ \boldsymbol{\sigma}^t = \begin{pmatrix} \sigma_1^t \\ \sigma_2^t \\ \sigma_3^t \\ ... \\ \sigma_{I-1}^t \\ \sigma_I^t \end{pmatrix}, \tag{2.3}$$

and $\boldsymbol{\mu}^t$ is the mean of the action vector, while $\boldsymbol{\sigma}^t$ is the variance of the action vector. In [49, 51], the $\boldsymbol{\mu}^t$ is the output of a multi-layer perceptron (MLP)

$$\boldsymbol{\mu}^t = \mathcal{MLP}(s^t). \tag{2.4}$$

It is also possible to write the variance of the action vector distribution as the output of a either shared or separate MLP, but, in practice, simply using a trainable variable is more common.
Relatedly, traditional model based methods have been widely explored [62, 32, 17, 31]. Among them, one of the most successful algorithms is the iterative linear-quadratic regulator controller [63, 35, 5]. But these methods usually suffer from the computation cost, during both training and controlling, or the ability of generalization, which limits their performance in high-dimentional control problems. In this paper, all the algorithms are contrained in model free domain.

2.1.2 Objective Function

First, we introduce the definition of the action-state value function $Q_\pi(s^t, a^t)$ for continuous control

$$Q_\pi\left(s^t, a^t\right) = \mathbb{E}_{s_t,\pi}\left[\sum_{\tau=0}^{T=\infty} \gamma^\tau r(s_\tau)\right]. \tag{2.5}$$

And for the definition of advantage function, we introduce as

$$A_\pi\left(s^t, a^t\right) = Q_\pi\left(s^t, a^t\right) - V_\pi\left(s^t\right) = Q_\pi\left(s^t, a^t\right) - \mathbb{E}_{s_t,\pi}\left[\sum_{\tau=0}^{T=\infty} \gamma^\tau r(s_\tau)\right]. \tag{2.6}$$

Equation 2.5 and 2.6 define two very fundamental functions for continuous control. Their definitions are very similar to the action-state value function and advantage function used in discrete control problems. Following [49], we can rewrite the equation 2.1 in the following form:

$$\begin{aligned} \eta(\pi) &= \eta(\pi_{old}) + \mathbb{E}_{s^{t=0},\pi}\left[\sum_{t=0}^{T=\infty} \gamma^t A_{\pi_{old}}(s^t, a^t)\right] \\ &= \eta(\pi_{old}) + \sum_{t=0}^{T=\infty} \sum_s P(s_t = s|\pi) \sum_a \pi(a_t = a|s)\gamma^t A_{\pi_{old}}(s, a) \\ &= \eta(\pi_{old}) + \sum_s \rho_\pi(s) \sum_a \pi(a_t = a|s) A_{\pi_{old}}(s, a), \end{aligned} \tag{2.7}$$

where $\rho_\pi(s)$ is the unnormalized discounted visitation frequencies. During implementation, this discounted visitation frequencies term $\sum_s \rho_\pi(s)$ could be replaced by sampling during training. We prove 2.7 by following the proof of Theorem 4.1 in [27], which is also applied in [49].

$$\begin{aligned}
\mathbb{E}_{\tau|\pi}\left[\sum_{t=0}^{T=\infty}\gamma^t A_{\pi_{old}}(s^t,a^t)\right] &= \mathbb{E}_{\tau|\pi}\left[\sum_{t=0}^{T=\infty}\gamma^t\left(r(s^t)+\gamma V_{\pi_{old}}(s^{t+1})-V_{\pi_{old}}(s^t)\right)\right] \\
&= \mathbb{E}_{\tau|\pi}\left[\sum_{t=0}^{T=\infty}\gamma^t\left(r(s^t)+\gamma V_{\pi_{old}}(s^{t+1})-V_{\pi_{old}}(s^t)\right)\right] \\
&= \mathbb{E}_{\tau|\pi}\left[\sum_{t=0}^{T=\infty}\gamma^t r(s^t)\right]-\mathbb{E}_{s^{t=0}\sim p_0(s^{t=0})}V_{\pi_{old}}(s^{t=0}) \\
&= -\eta(\pi_{old})+\eta(\pi)
\end{aligned} \tag{2.8}$$

Equation 2.8 implies that, to optimize the objective function 2.1, we could equivalently optimize the following objective function iteratively:

$$L_{\pi_{old}}(\pi) = \mathbb{E}_{\tau|\pi}\left[\sum_{t=0}^{T=\infty}\gamma^t A_{\pi_{old}}\left(s^t,a^t\right)\right] \tag{2.9}$$

Note that this objective function is not directly usable, since the trajectory τ is heavily dependent on the policy to be optimized. Therefore, the dependency between the trajectory generated and the policy we are optimizing must be broken. In the next section, we show that this could be done by introducing the surrogate loss.

2.1.3 Surrogate Loss and Trust Region Optimization

To break the dependency, in [49], the authors introduce the first order approximation Γ of the objective function,

$$\Gamma_{\pi_{old}}(\pi) = \eta(\pi_{old}) + \sum_s \rho_{\pi_{old}}(s)\sum_a \pi(a_t=a|s)A_{\pi_{old}}(s,a). \tag{2.10}$$

This is a first-order approximation at the old policy π_{old} if we assume that the policy is parameterized by θ, since $\Gamma_{\pi_{old}}(\pi_{old}) = \eta(\pi_{old})$ and $\nabla\Gamma_{\pi_{old}}(\pi_{old})|_{\theta=\theta_{old}} = \nabla\eta(\pi_{old})|_{\theta=\theta_{old}}$. We also note that from that we could derive the low bound of the original objective function 2.1, which is defined as

$$\eta(\pi) \geq \Gamma_{\pi_{old}}(\pi) - CD_{\mathrm{KL}}^{\max}(\pi_{old},\pi), \tag{2.11}$$

where $C = \frac{4\epsilon\gamma}{1-\gamma}^2$ and $D_{\mathrm{KL}}^{\max}$ is the maximum of the KL-divergence $D_{\mathrm{KL}}^{\max} = \max_s D_{\mathrm{KL}}(\pi_{old}(\cdot|s)||\pi(\cdot|s))$. This is the trust region, i.e., if the step-size of optimizing is small enough, we should be able to increase the performance of the agent. In practice, we switch the $D_{\mathrm{KL}}\max$ with the average KL-divergence $\bar{D}_{\mathrm{KL}}$. The surrogate loss $\mathcal{M}$ is defined as

$$\mathcal{M}_{\pi_{old}}(\pi) = \Gamma_{\pi_{old}}(\pi) - \alpha\bar{D}_{\mathrm{KL}}(\pi_{old},\pi), \tag{2.12}$$

We can prove that by iteratively optimizing the $\mathcal{M}$, we are optimizing the original objective function $\eta(\pi)$. α is a heuristic chosen coefficient, which is chosen to avoid the calculation of the original variable C. For details of 2.11, 2.12, we refer to [49].

Optimization of 2.12 is straight-forward, as we can take the derivative of this function and apply standard gradient descent. More specifically, if we force the KL-divergence term to be a hard constraint, then the problem becomes a constrained optimization. In [49], second-order optimization is needed for this constrained optimization, and

conjugate gradient method with line search in the parameter space is applied. This is time consuming when the parameter's size is big. A large-scale method is to apply a hinge loss on KL-divergence, which is

$$\mathcal{M}'_{\pi_{old}}(\pi) = \mathcal{M}_{\pi_{old}}(\pi) - \beta \left| \text{KL}_{\text{target}} - \bar{D}_{\text{KL}}(\pi, \pi_{old}) \right|^2 . \tag{2.13}$$

$\text{KL}_{\text{target}}$ is usually heuristically chosen as 0.01. In this way, we circumvent second-order optimization by choosing a suitable scaler hyper-parameter β. Another way to circumvent second-order constraint is to clip the importance sampling ratio that is multiplied by the advantage function, as is used in PPO [51]. PPO clips the probability ratio and adds KL-divergence penalty term to the loss. The likelihood ratio is defined as $r^t(\theta; \theta_{old}) = \pi_\theta(a^\tau|s^\tau)/\pi_{\theta_{old}}(a^\tau|s^\tau)$. This is also discouraging the policy from over-confident updates.

Relation to Policy Gradient Methods In continuous control, policy gradient methods refer to the stochastic policy gradient method [57] and deterministic policy gradient [54] method. Both methods suffer from the variance of estimator. Respectively, the stochastic policy gradient and deterministic policy gradient are formulated as follow:

$$\nabla_\theta \mathcal{J}(\pi) = \mathbb{E}_{s^t=0\sim p_0(s^{t=0}), a\sim\pi} \left[\nabla_\theta \log \pi(a^t|s^t) Q_\pi(s^t, a^t) \right], \tag{2.14}$$

and

$$\nabla_\theta \mathcal{J}(\pi) = \mathbb{E}_{s^t=0\sim p_0(s^{t=0})} \left[\nabla_\theta \mu_\theta(s^t) \nabla_{\mu_\theta(s^t)} Q(s^t, \mu_\theta(s^t)) \right]. \tag{2.15}$$

Both 2.14 and 2.15 have reduced the computation of objective function' gradient into the calculation of expectation.

Similar to the trust region methods introduced earlier in this chapter, policy gradient based methods also rely on sampling trajectory to approximate the expectation in the objective function. Therefore, the estimator of policy gradient suffers from having big variance. And the performance of the estimator is decreasing with the increase of action space size and state space size, thus requiring more samples.

Deterministic policy gradient estimator usually has better performance than stochastic policy gradient estimator, since the deterministic policy gradient estimator does not integrate over action space.

Note that for trust region methods, the $Q_\pi(s^t, a^t)$ is defined in 2.5, which is $\mathbb{E}_{s_t,\pi}\left[\sum_{\tau=0}^{T=\infty} \gamma^\tau r(s_\tau)\right]$. In policy gradient methods, this action-state function could be added or subtracted by any arbitary baseline functions that are not dependent on action a^t, and in practice the baseline function is usually chosen to be the estimation of value function $V_\pi(s^t)$. In trust region methods, $Q_\pi(s^t, a^t)$ is usually chosen to be the empirical return or the generalized advantage estimation return, which we refer the readers to [50] for more specific details.

In [14], the state-of-the-art continuous control algorithms are evaluated. The deep-learning based variant of deterministic policy gradient, which is [37], has comparable performance with trust region based methods in many benchmark tasks. In this paper, our algorithms are based on [51], but it is worth pointing out that our methods could fit in any optimization algorithms in continuous control.

Chapter 3

Related Work

3.1 Related Work in Reinforcement Learning

Reinforcement learning (RL) has recently achieved huge success in a variety of applications. Boosted by the progress of deep neural networks, [29], agents are able to solve problems from Atari Games and beat the best human players in the game of Go [42, 53, 55]. For continuous control problems, based on simulation engines like MuJoCo [62], numerous algorithms have been proposed to optimize agent's expected reward [51, 49, 26, 40].

Recently, more and more research experiments have been done for transfer learning tasks [60] in RL which mainly focus on transferring the policy learned from one environment to another. In [46, 47], 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. In model based method, tradtional control has been well studied on generalization [11]. [24] try to increase the transferability via learning invariant visual features. Efforts have also been made from the meta-learning perspective [15, 19, 18]. In [66], the authors propose a method of transfer learning by using imitation learning.

As for exploiting the structure in RL, most hierarchical reinforcement learning algorithms [30, 64] focus on modeling intrinsic motivation. In [25], the authors exploit the structure of the action space. Graph has been used in RL problems. In [41, 38, 52, 39], 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.

Multi-task learning has been receiving a lot of attention [65]. In [61], the authors use a distilled policy that captures common behaviour across tasks. [65, 43, 2] use a hierarchical approach, where multiple sub-policies are learnt. While at the same time, in [67, 44], shared visual features are used to solve Multi-task learning. In [1], a Multi-task policy gradient is proposed, while in [9], multi-task extensions of the fitted Q-iteration algorithm are proposed. Successor features in [4] are used to boost the performance of multiple tasks. In general, currently, most approaches try to improve the optimization algorithms, or use a simple hierarchical structure with sub-policies.

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

3.2 Related Work in 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 [8, 13, 28], CNNs are employed in the spectral domain relying on the graph Laplacian matrix. While [21, 16] used hash functions so that CNN can be applied to graphs. Another popular direction is based on recurrent neural networks (RNNs) [21, 22, 48, 56, 36, 59].

Among RNN based methods, some of them are only applicable to special structured graph, e.g., sequences or trees, [56, 59]. One class of models which are applicable to general graphs is called, graph neural networks (GNNs), [48]. 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, [3]. 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 [36], 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, [10]. It has been shown that GNNs, [36, 34, 45, 20] 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.

Chapter 4

Structured Model Using Graph Neural Network

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.

4.1 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 [62], 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. 4.1 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.

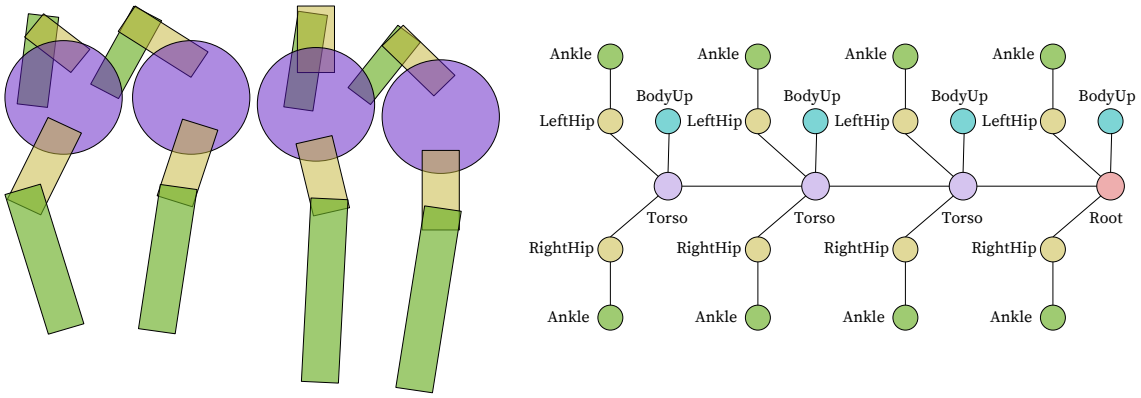


Figure 4.1: Visualization of the graph structure of the `CentipedeEight` agent in our environment. This agent is later used for testing the ability of transfer learning of our model. Since in this agent, each `body` node is paired with at least one `joint` node, we omit the `body` nodes and fill up the 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.

4.2 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.

4.2.1 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

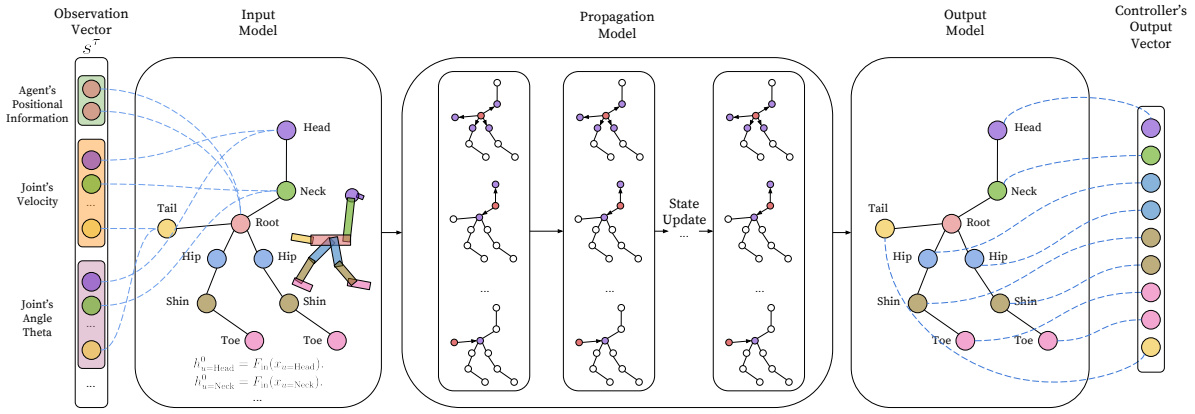


Figure 4.2: In this figure, we use `Walker-Ostrich` as an example of NerveNet. In the input model, for each node, NerveNet fetches the corresponding elements from the observation vector. NerveNet then computes the messages between neighbors in the graph, and update the hidden state of each node. This process is repeated for certain number of propagation. In the output model, the policy is produced by collecting the output from each controller.

observation vector goes through an input network to obtain a fixed-size state vector as follows:

$$h_u^0 = F_{\text{in}}(x_u). \tag{4.1}$$

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. Note that we may need to pad zeros to the observation vectors if different nodes have observations of different sizes.

4.2.2 Propagation Model

We now describe the propagation model of our NerveNet which mimics a synchronous message passing system studied in distributed computing [3]. 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,

$$m_{(u,v)}^t = M_{c_{(u,v)}}(h_u^t), \tag{4.2}$$

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. 4.1 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:

$$\bar{m}_u^t = A(\{h_v^t | v \in \mathcal{N}_{in}(u)\}), \tag{4.3}$$

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:

$$h_u^{t+1} = U_{p_u}(h_u^t, \bar{m}_u^t), \tag{4.4}$$

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., $\{h_u^T | u \in V\}$.

4.2.3 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 [51]. 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. Different sharing schemes are available for the instance of MLP O_{q_u}, for example, we can force the nodes with similar physical structure to share the instance of MLP. For example, in Fig. 4.1, two `LeftHip` should have a shared controller. Therefore, we have the following output model:

$$\mu_{u\in\mathcal{O}} = O_{q_u}(h_u^T), \tag{4.5}$$

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

$$\pi_\theta(a^\tau | s^\tau) = \prod_{u\in\mathcal{O}} \pi_{\theta,u}(a_u^\tau | s^\tau) = \prod_{u\in\mathcal{O}} \frac{1}{\sqrt{2\pi\sigma_u^2}} e^{(a_u^\tau - \mu_u)^2/(2\sigma_u^2)}, \tag{4.6}$$

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.

4.3 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

$$J(\theta) = \mathbb{E}_\pi \left[\sum_{\tau=0}^{\infty} \gamma^\tau r(s^\tau, a^\tau) \right]. \tag{4.7}$$

To optimize the expected reward, we apply the proximal policy optimization (PPO) by [51] 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. (4.7), KL-penalty and value function loss which is defined as:

$$\begin{aligned}
\tilde{J}(\theta) =& J(\theta) - \beta L_{KL}(\theta) - \alpha L_V(\theta) \\
=& \mathbb{E}_{\pi_\theta} \left[\sum_{\tau=0}^{\infty} \min \left(\hat{A}^\tau r^\tau(\theta), \hat{A}^\tau \text{ clip} \left(r^\tau(\theta), 1-\epsilon, 1+\epsilon \right) \right) \right] \\
& - \beta \mathbb{E}_{\pi_\theta} \left[\sum_{\tau=0}^{\infty} \text{KL} \left[\pi_\theta(:|s^\tau) | \pi_{\theta_{old}}(:|s^\tau) \right] \right] - \alpha \mathbb{E}_{\pi_\theta} \left[\sum_{\tau=0}^{\infty} \left(V_\theta(s^\tau) - V(s^\tau)^{\text{target}} \right)^2 \right],
\end{aligned} \tag{4.8}$$

where the $\hat{A}_t$ is the generalized advantage estimation (GAE) calculated using algorithm from [50], 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. (4.8), $V(s_t)^{\text{target}}$ 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 [58] 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.

Chapter 5

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, as well as the multi-task learning ability and robustness.

5.1 Comparison on Standard Benchmarks of MuJoCo

Experimental Setting We compare NerveNet with the standard MLP models utilized by [51] 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 [7] which is based on MuJoCo, [62]. 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 A.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 5.1. From the figures, we can see that MLP with the same setup as in [51] 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 [51], we obtained better performances than the ones reported in original paper.

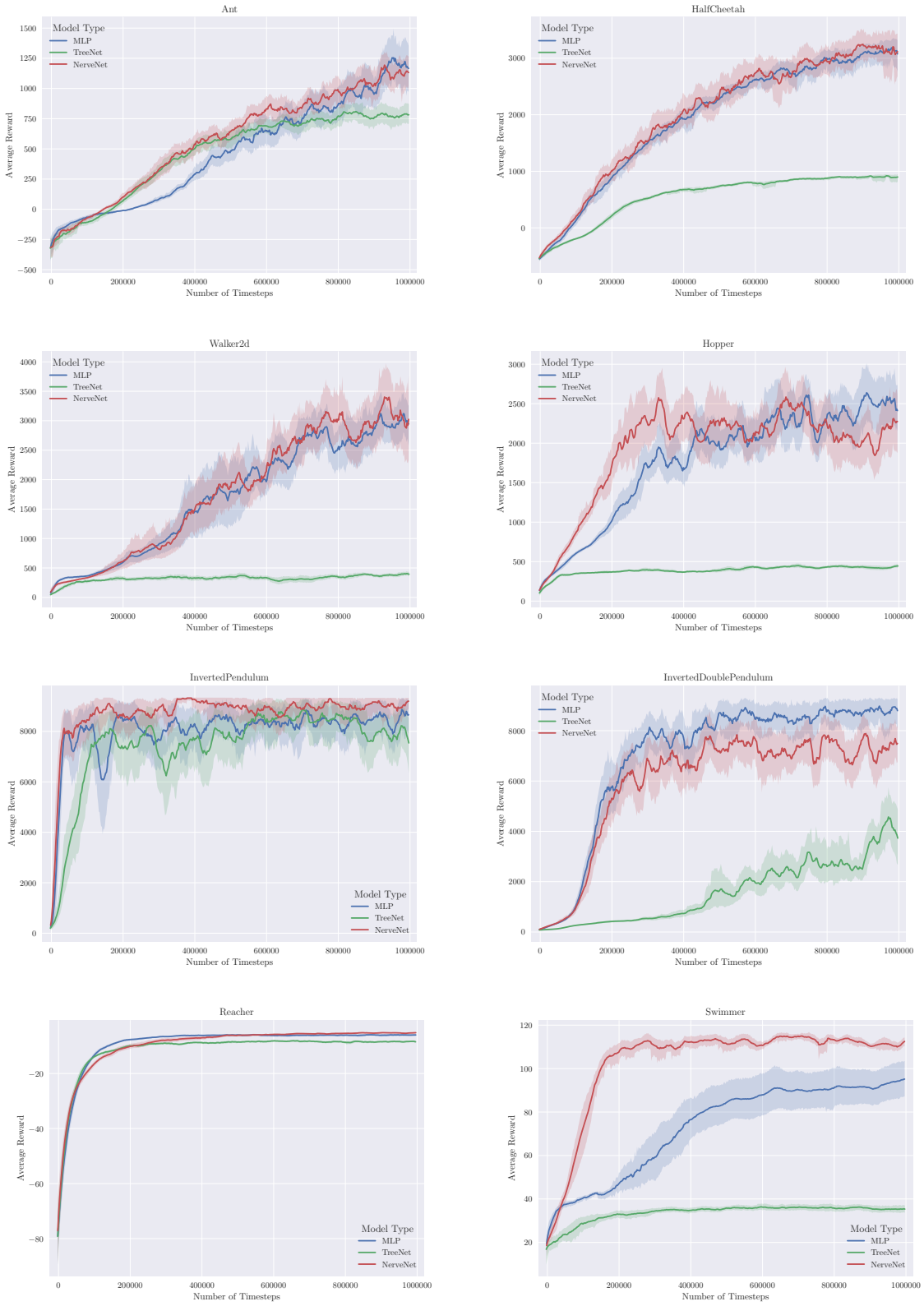


Figure 5.1: Results of MLP, TreeNet and NerveNet on 8 continuous control benchmarks from the Gym.

Table 5.1: Performance of the Pre-trained models on `CentipedeFour` and `CentipedeSix`.

C_Four	Reward Avg	Std	Max	C_Six	Reward Avg	Std	Max
NerveNet	2799.9	1247.2	3833.9	NerveNet	2507.1	738.4	2979.2
MLP	2398.5	1390.4	3936.3	MLP	2793.0	1005.2	3529.5
TreeNet	1429.6	957.7	3021.7	TreeNet	2229.7	1109.4	3727.4

5.2 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 `CentipedeEight` from scratch is already very difficult. For *size transfer* experiments, we create many instances which are listed in Figure 5.2, like "4to06", "6to10". For *disability transfer*, we create `CrippleCentipede` agents of which two back legs are disabled. In the Figure 5.2, `CrippleCentipede` is specified as "Cp".

Snakes We also create a snake-like agent which is common in robotics, [12]. 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 B.5.

5.2.1 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.

5.2.2 Results

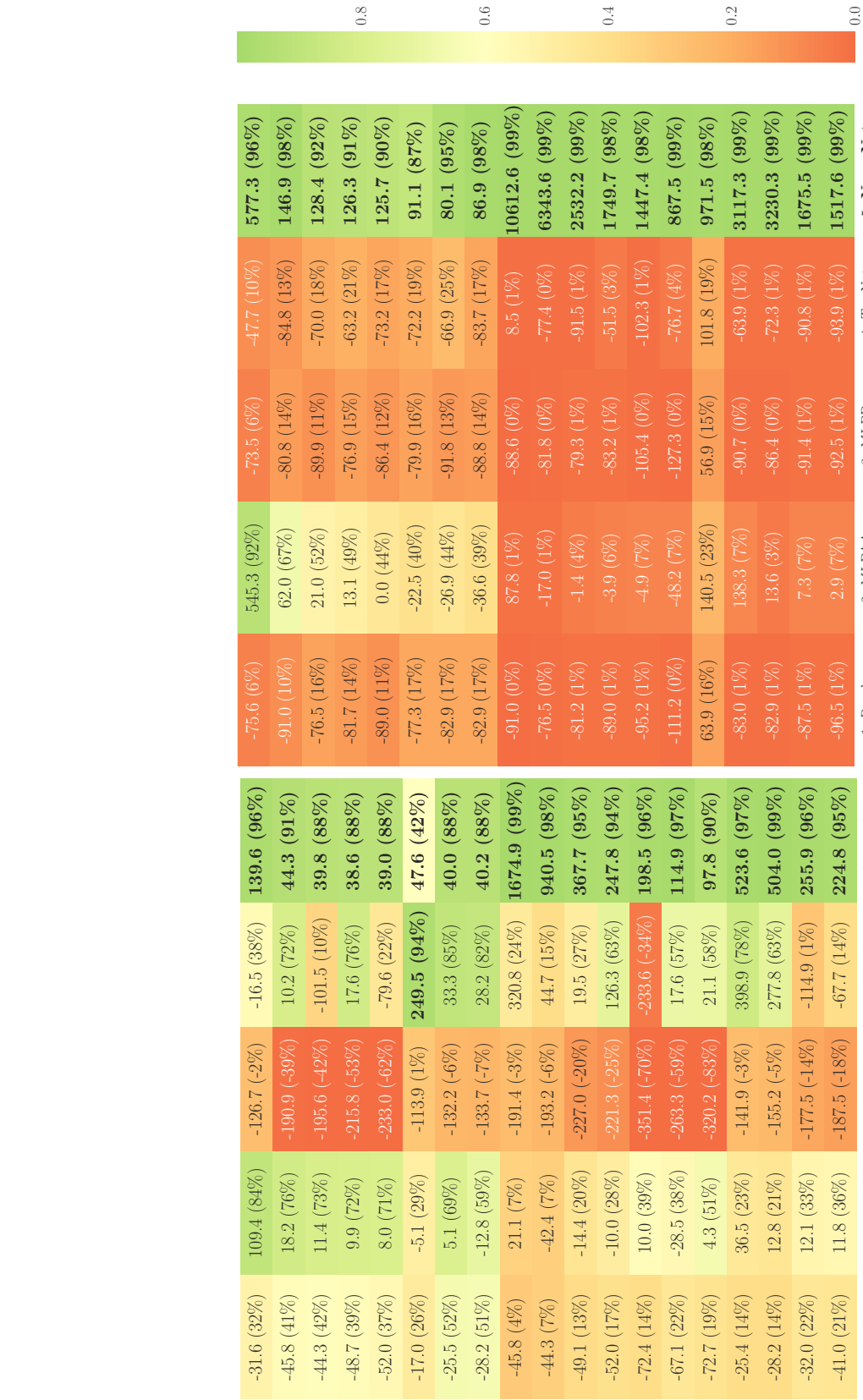
Centipedes For the Centipedes environment, we first run experiments of all models on `CentipedeSix` and `CentipedeFour` to get the pre-trained models for transfer learning. We train different models until these agents could run as equally well as possible, which is listed in Table 5.1. Note that, in practice, we train TreeNet on `CentipedeFour` for more than 8 million timesteps. However, due to the difficulty of optimizing TreeNet on `CentipedeFour`, the performance is still lower. But visually, the TreeNet agent is able to run in `CentipedeFour`.

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. To better visualize the results, we linearly normalize the performance to get a performance score, and color the results accordingly. The normalization scheme is recorded in appendix C.1.1. The performance score is less than 1, and is shown in the parentheses behind the original results. As we can see from Figure 5.2 (full chart in appendix C.1), NerveNet outperforms all competitors on all settings, except in the 4toCp06 scenario. Note that transferring from `CentipedeFour` is more difficult than from `CentipedeSix` 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 still and gaining alive bonus. We argue that the average running-length in each episode is also a very important metric.

By including the results of running-length, we notice that NerveNet is the only model able to walk in the zero-shot evaluations. As a matter of fact, the performance of NerveNet could be orders-of-magnitude better, and most of the time, agents from other methods cannot even move forward. We also notice that if transferred from `CentipedeSix`, NerveNet is able to provide walkable pretrained models on all new agents.

We fine-tune for both *size transfer* and *disability transfer* experiments and show the training curves in Fig. 5.3. 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



(a) Zero-shot average reward.

(b) Zero-shot average running-length.

Figure 5.2: Performance of zero-shot learning on centipedes. For each task, we run the policy for 100 episodes and record the average reward and average length the agent runs before falling down.

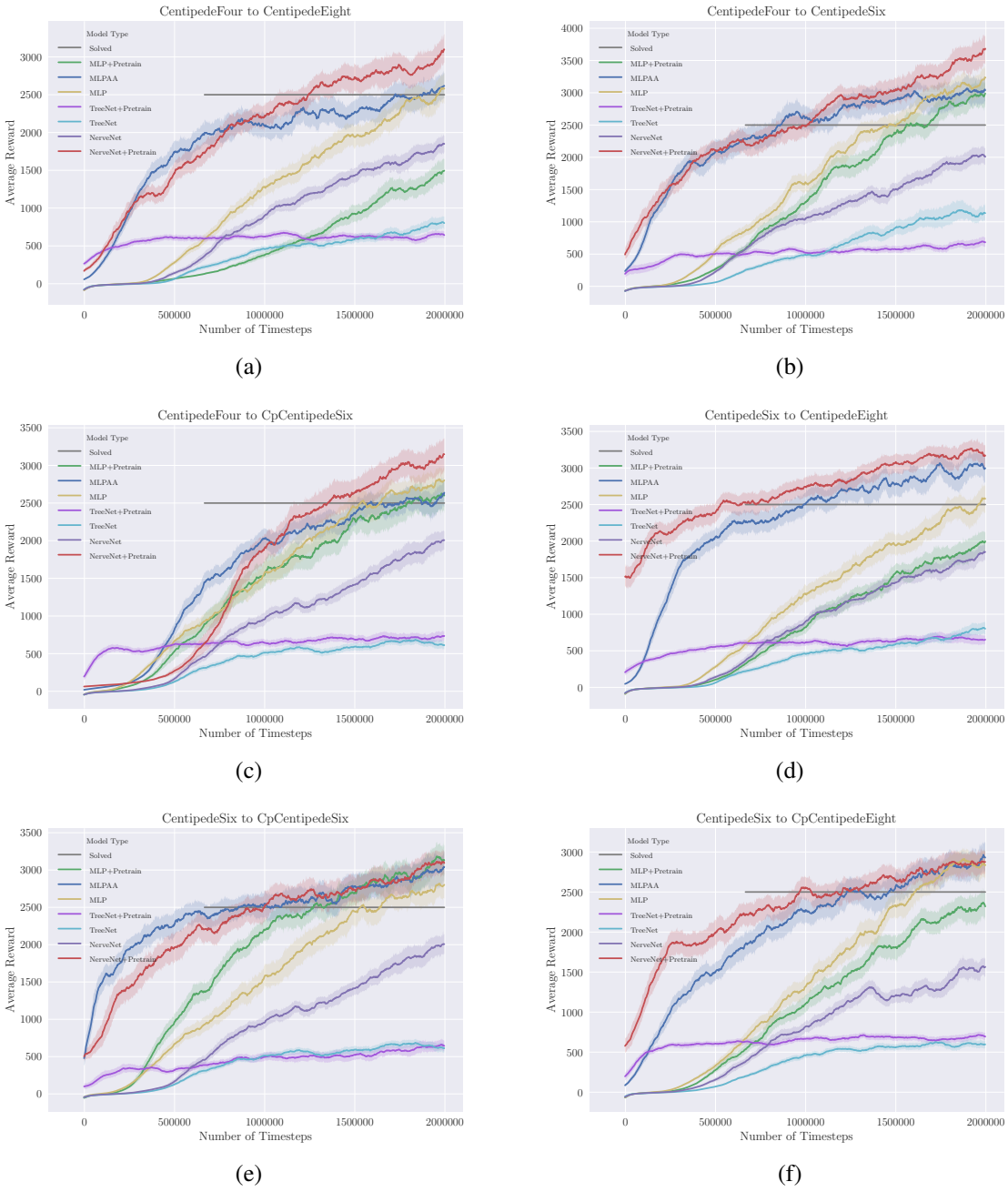


Figure 5.3: (a), (b), (d): Results of fine-tuning for *size transfer* experiments. (c), (e), (f) Results of fine-tuning for *disability transfer* experiments.

that this long time period is spent on learning to "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 [6]. 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 5.5.

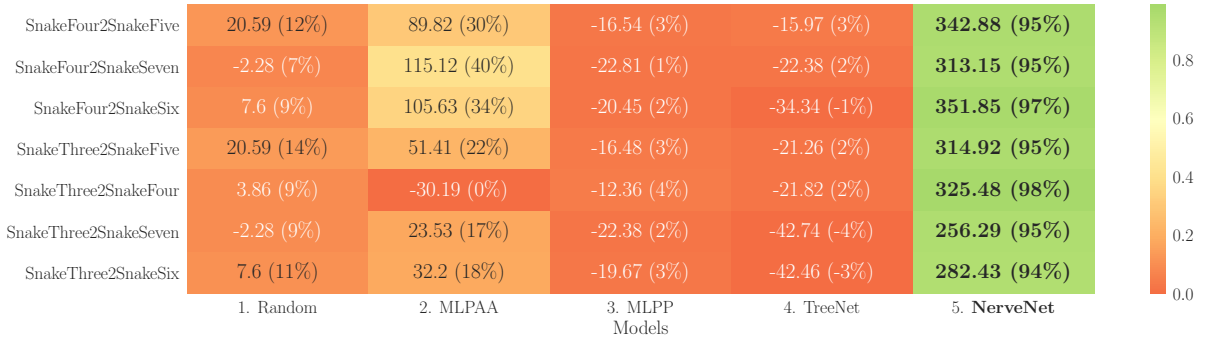


Figure 5.4: Results on zero-shot transfer learning on snake agents. Each tasks are simulated for 100 episodes.

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 Figure 5.4. 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.

We also summarize the training curves of fine-tuning in Fig. 5.5. 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.

5.3 Multi-task Learning

In this section, without devling into the specific optimization algorithms to be used in multi-task learning, we show that NerveNet has good potential of multi-task learning from the network structure's perspective. Before training the models to solve multiple tasks, it is important to point out that multi-task learning remains a very difficult, and more often the case, unsolved problem in RL. Most multi-task learning algorithms [61, 2, 43, 67] have not been applied to domain as difficult as locomotion under complex physical models, not to mention multi-task learning among different agents with different dynamics.

In this work, we constrain our problem domain, and design the `Walker` multi-task learning task-set, which contains five 2d-walkers. In this case, model's ability of multi-task learning, especially the ability to controll multiple agents, will be tested. The walkers are very different in terms of their dynamics, since they have very dinstinct

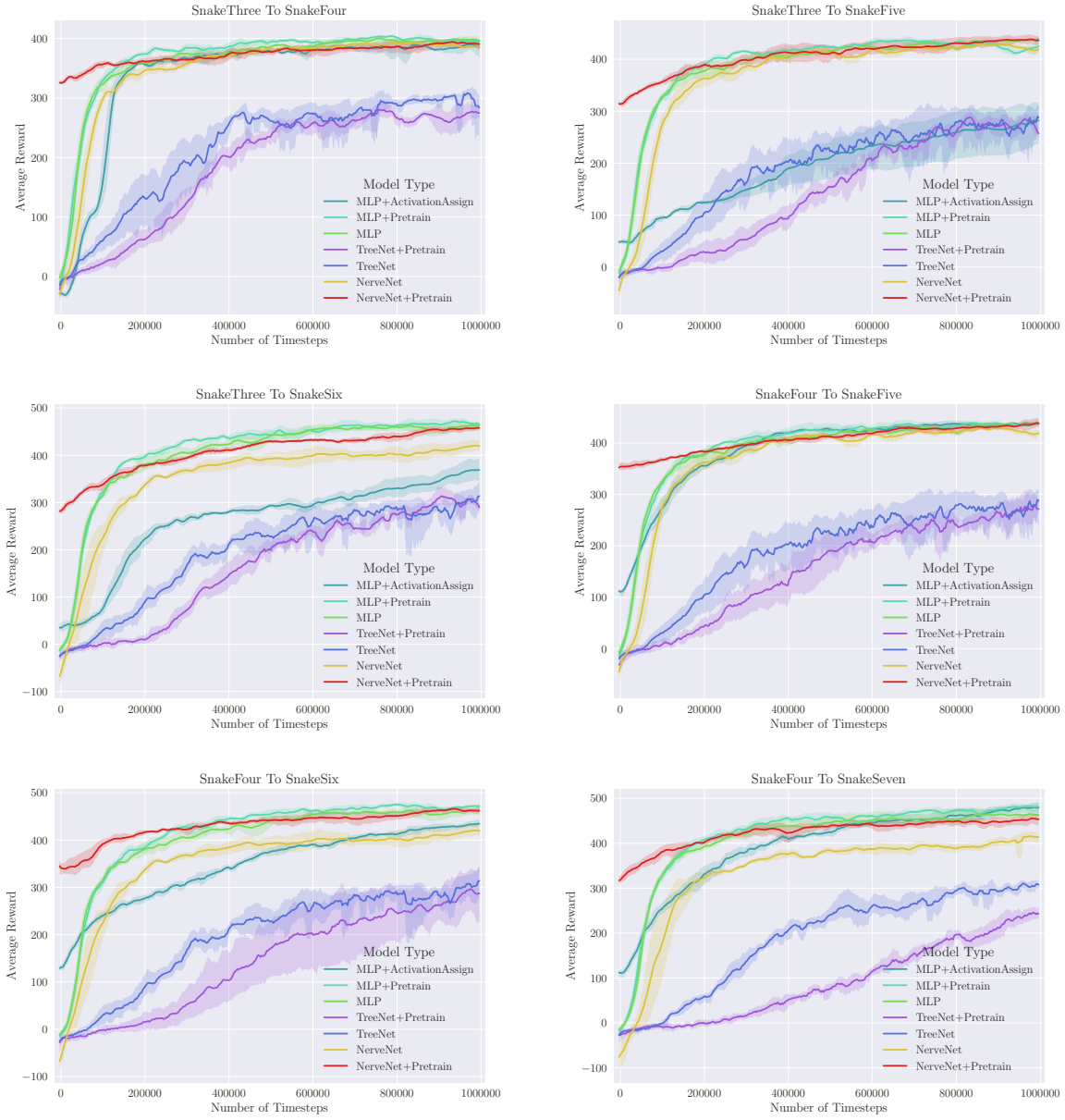


Figure 5.5: Results of finetuning on snake environments.

structures, different types and numbers of controllers. `Walker-HalfHumanoid` and `Walker-Hopper` are agents respectively variants of `Walker2d` and `Hopper` from the original MuJoCo Benchmarks. `Walker-Horse`, `Walker-Ostrich`, `Walker-Wolf` on the other hand, are agents mimicking the natural animals. Just like the real-animals, some of the agents have tails or necks to help the balance. The detailed schematic figures is in the appendix B.2.

5.3.1 Experimental Settings

To show the ability of multi-task learning of NerveNet, we design several baselines. Since we are more interested in the models' potential of multi-task learning from the network structure's perspective, we use a vanilla policy

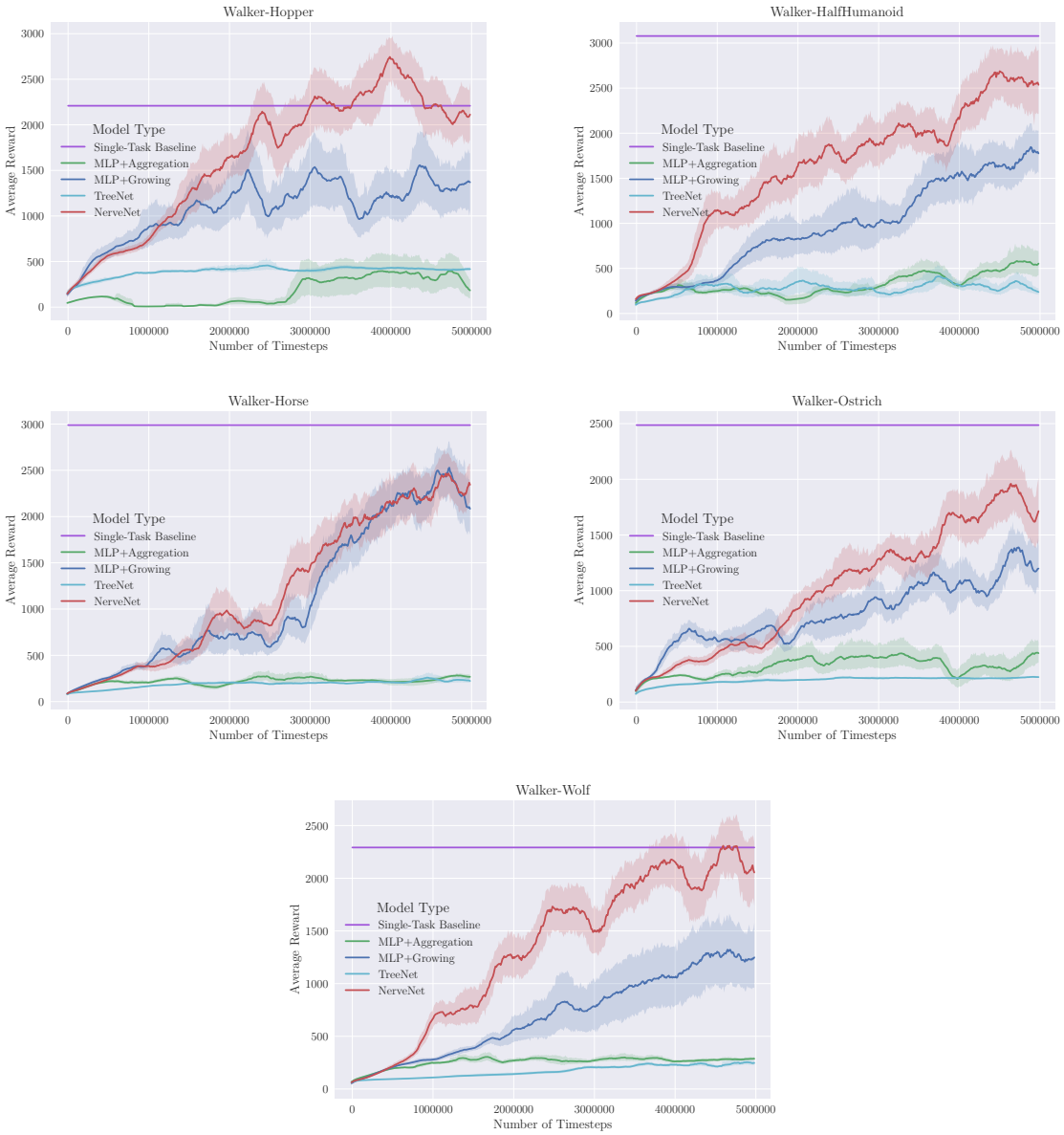


Figure 5.6: Results of Multi-task learning. We train the networks simultaneously on five different tasks from `Walker`.

optimization method for all models. More specifically, for each sub-task in the multi-task learning task-set, we generate equal number of timesteps for each policy update and calculate the gradients separately. We then aggregate the gradients and apply the mean value to update the network. To compensate the extra difficulty of training brought by more agents and tasks, we linearly increase the number of update epochs during each update in the training, as well as the total number of timesteps generated before the training is terminated. The hyper-parameter setting is summarized in appendix A.4.

NerveNet For NerveNet, the weights are naturally shared among different agents. More specifically, for different agents, the weight matrices for propagation and output are shared.

Model		HalfHumanoid	Hopper	Ostrich	Wolf	Horse	Average
MLP	Reward	1775.75	1369.59	1198.88	1249.23	2084.07	/
	Ratio	57.7%	62.0%	48.2%	54.5%	69.7%	58.6%
TreeNet	Reward	237.81	417.27	224.07	247.03	223.34	/
	Ratio	79.3%	98.0%	57.4%	141.2%	99.2%	94.8%
NerveNet	Reward	2536.52	2113.56	1714.63	2054.54	2343.62	/
	Ratio	96.3%	101.8%	98.8%	105.9%	106.4%	101.8%

Table 5.2: Results of Multi-task learning, with comparison of the single-task baselines. For three models, the first row is the mean reward of each model of the last 40 iterations, while the second row indicates the percentage of the performance of multi-task model compared with the single-task baseline respectively of each model.

MLP Sharing For the MLP method, we shared the weight matrices between hidden layers.

MLP Aggregation In the MLP Sharing method, the total size of the weight matrices grows with the number of tasks. Considering that the observation dimentions for different agents are different, to keep the size of matrices static, a matrix whose size is the hidden size by 1 is used. We multiple each element of the observation by this matrix and aggregate the resulting vectors from each observation.

TreeNet Similarly, TreeNet also has the benefits that its weights are naturally shared among different agents. But again, TreeNet has no knowledge of the agents' physical structure, and all the information of each node is aggregated into the root node.

Results We also include the baseline of training single-task MLP for each agent. We train the single-task MLP baselines for 1 million timesteps per agent. And in the figure, we align the results of single-task MLP baseline and the results of multi-task models by the number of episodes of one task.

As can be seen from the figure 5.6, NerveNet is able to have the best performance in all the sub-tasks. In `Walker-HalfHumanoid`, `Walker-Hopper`, `Walker-Ostrich`, `Walker-Wolf` NerveNet is able to outperform other agents by a large margin, And in `Walker-Horse`, the performance of NerveNet and MLP Sharing are almost similar. For MLP sharing method, the performance on other four agents are relatively limited, and in `Walker-Hopper`, the improvement of performance has been very limited from the middle of the experiments. MLP Aggregation method and TreeNet method are not able to solve the multi-task learning problem, with both of them stuck at a very low reward level. Under the vanilla optimization setting, we show that NerveNet has bigger potential than the baselines.

From the table 5.2, it could also be observed that the performance of MLP drops drastically (42% performance drop) when switching from single-task to multi-task learning, while for NerveNet, there is no obvious drop of performance.

Our intuition is that NerveNet is good at learning generalized features, and the learning of different agents could help the training of other agents, while for MLP methods, the performance decreases due the competition of different agents.

Model		Halfhumanoid	Hopper	Wolf	Ostrich	Horse	Average
Mass	MLP	33.28%	74.04%	94.68%	59.23%	40.61%	60.37%
	NerveNet	95.87%	93.24%	90.13%	80.2%	69.23%	85.73%
Strength	MLP	25.96%	21.77%	27.32%	30.08%	19.80%	24.99%
	NerveNet	31.11%	42.20%	42.84%	31.41%	36.54%	36.82%

Table 5.3: Results of robustness evaluations. Note that we show the average results for each type of parameters after perturbation. And the results are columned by the agent type. The ratio of the average performance of perturbed agents and the original performance is shown in the figure. Details are listed in C.2.

Figure 5.7: Results of table 5.3 visualized.

5.4 Robustness of Learnt Policies

In this section, we also report the robustness of our policy by perturbing the agent parameters. In reality, the parameters simulated might be different from the actual parameters of the agents. Therefore, it is important that the agent is robust to parameters perturbation. The model that has the better ability to learn generalized features are prone to be more robust.

We perturb the mass of the geometries (rigid bodies) in MuJoCo as well as the scale of the forces of the joints. We use the pretrained models with similar performance on the original task for both the MLP and NerveNet. The performance is tested in five agents from `Walker` task set. The average performance is recorded in 5.3, and the specific details are summarized in appendix C.2. The robustness of NerveNet' policy is likely due to the structure prior of the agent instilled in the network, which facilitates overfitting. The results are updated in the latest version of the paper.

5.5 Interpreting the Learned Representations

As we can see from the figure, NerveNet is significantly better than than MLP in terms of robustness. We argue that this might due to that MLP is more prone to over-fitting to specific agent. 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. 5.10, 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

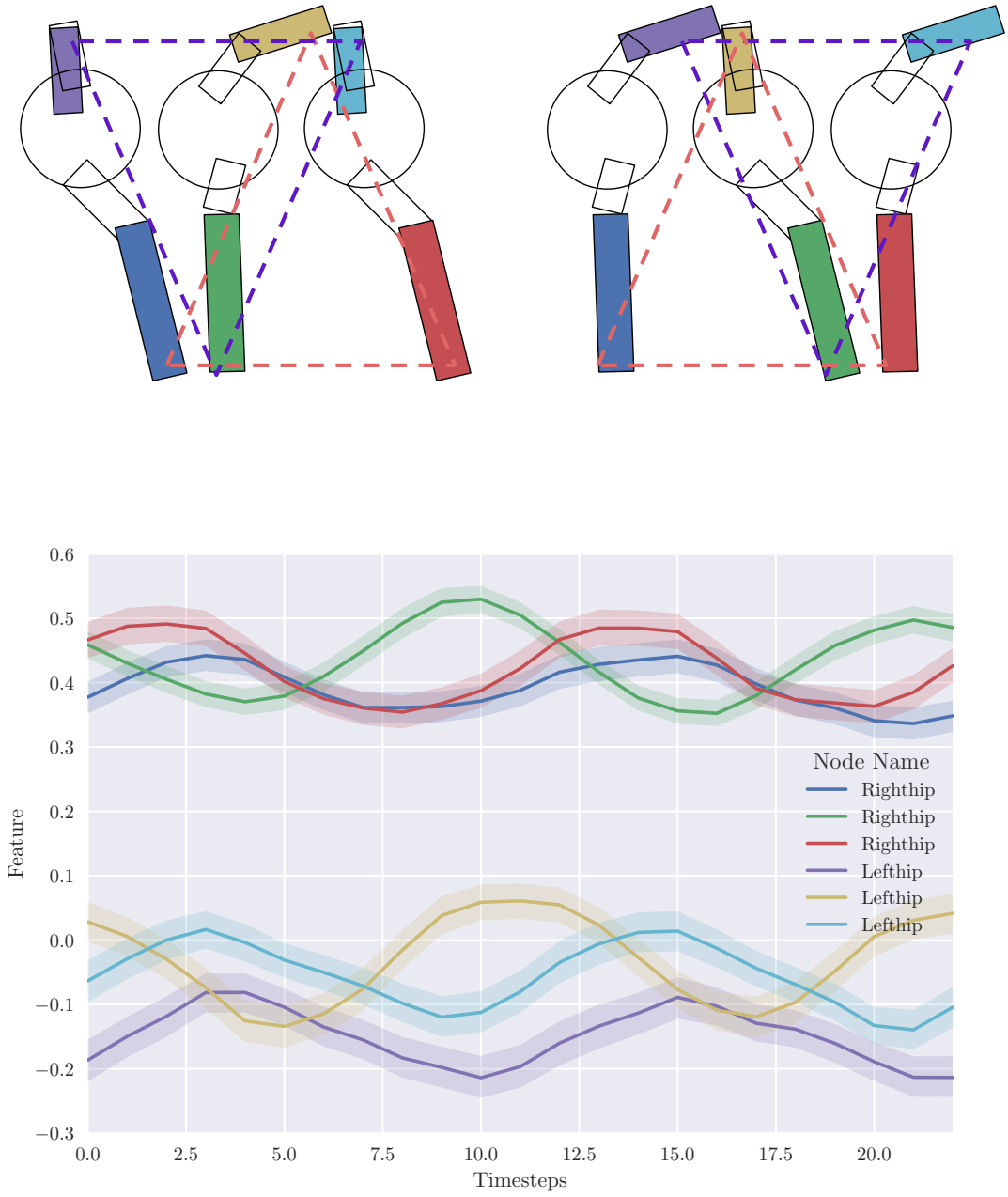
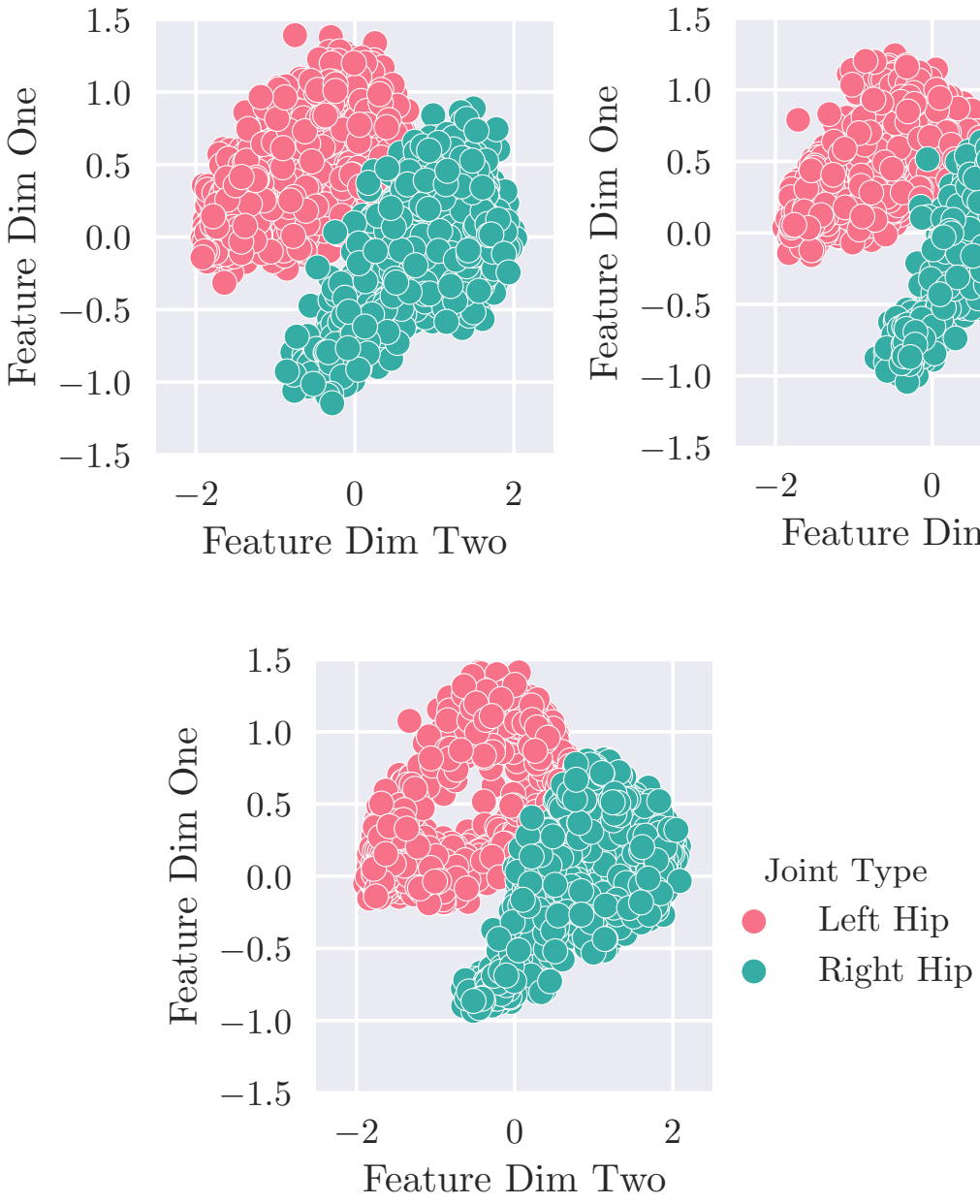


Figure 5.8: Diagram of the walk cycle. In the left figure, legs within the same triangle are used simultaneously. For each leg, we use the same color for their diagram on the left and their curves on the right.

transformed features of the 6 legs on Fig. 5.8. 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.



5.6 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. 5.11, 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 4.8. Based on the figure,

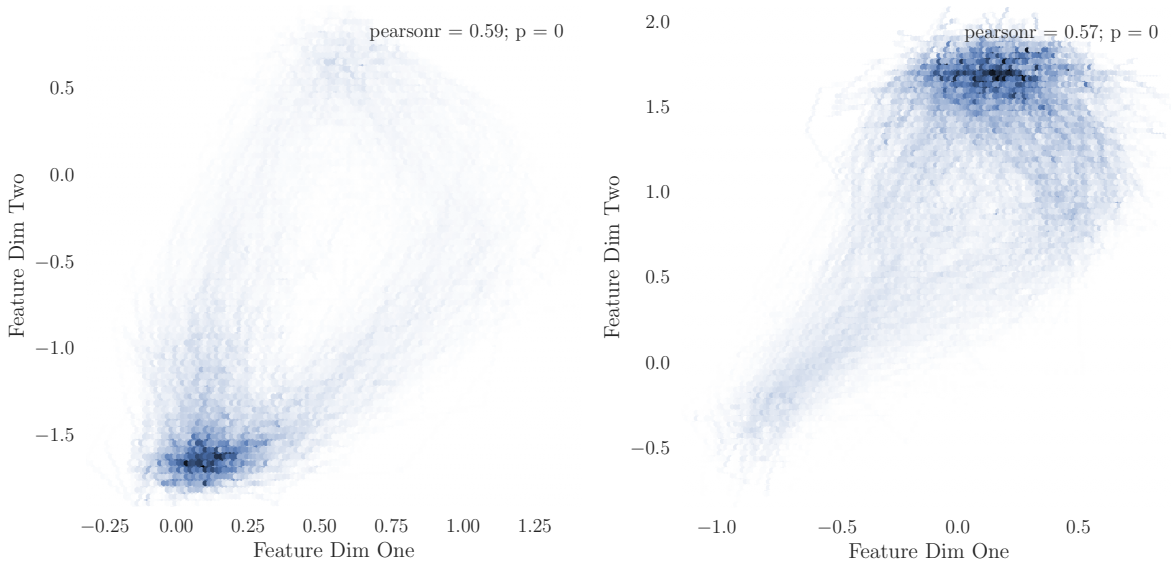


Figure 5.10: Results of visualization of feature distribution and trajectory density. As can be seen from the figure, NerveNet agent is able to learn shareable features for its legs, and certain walk-cycle is learnt during training.

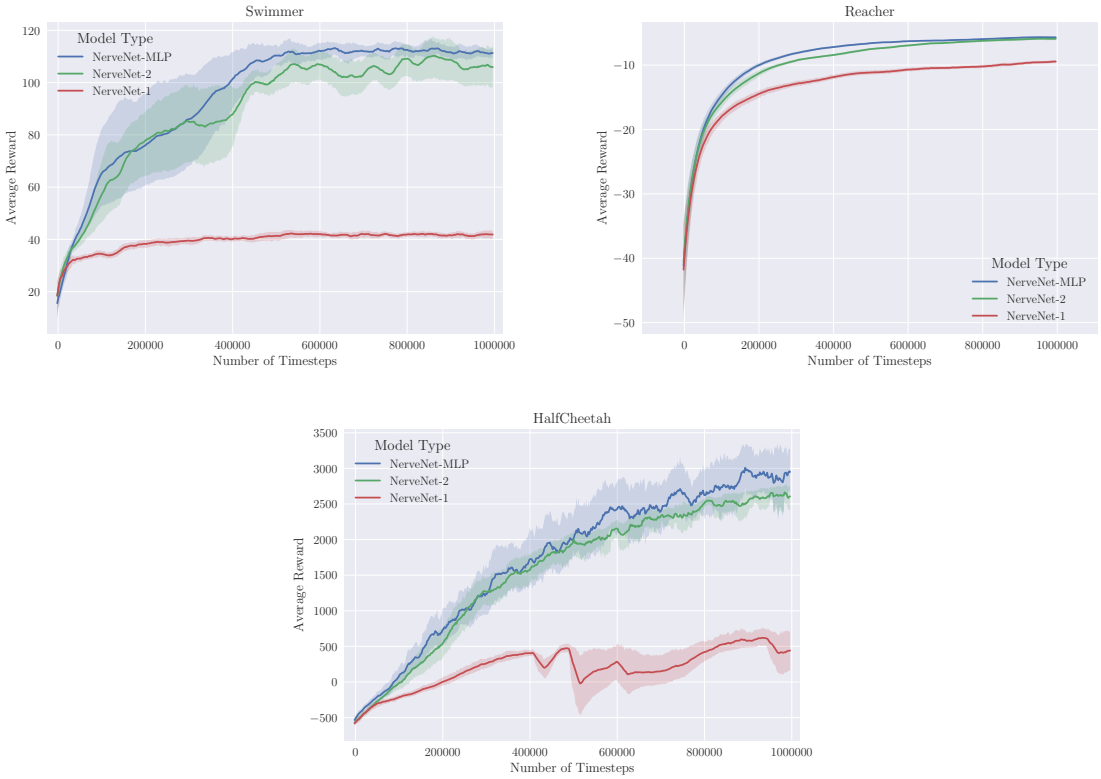


Figure 5.11: Results of several variants of NerveNet for the reinforcement learning agents.

choosing α to be 1 is not giving good performance on the tasks we experimented on.

Chapter 6

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 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.

Appendix A

Details of the Experiment Settings

A.1 Details of NerveNet Hyperparameters

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 table A.2, A.1.

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

Table A.1: Parameters used during training.

Parameters	Value Set	Parameters	Value Set
Value Discount Factor γ	0.99	GAE λ	0.95
PPO Clip Value	0.2	Starting Learning Rate	3e-4
Gradient Clip Value	5.0	Target KL	0.01

A.2 Graph of Agents

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 4.1, the `LeftHip` node will receive the angular velocity ϖ_j, the twist angle θ_j.

A.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 A.2, A.4, A.3

Table A.2: 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 A.3: 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

Table A.4: 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

A.4 Hyperparameters for Multi-task Learning

Table A.5: Hyperparameter settings for multi-task learning.

MLP, TreeNet and NerveNet	Value Tried
Network Shape	[64, 64]
Number of Iteration Per Update	10
Use KL Penalty	No
Learning Rate Scheduler	Adaptive
NerveNet	**Value Tried**
Number of Propogation Steps	4
NerveNet Variants	NerveNet-1
Size of Nodes' Hidden State	64
Merge `joint` and `body` Node	Yes
Output Network	Separate
Disable Edge Type	Yes
Add Skip-connection from / to `root`	No

Appendix B

Schematic Figures of Agents

B.1 Schematic Figures of the MuJoCo 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.

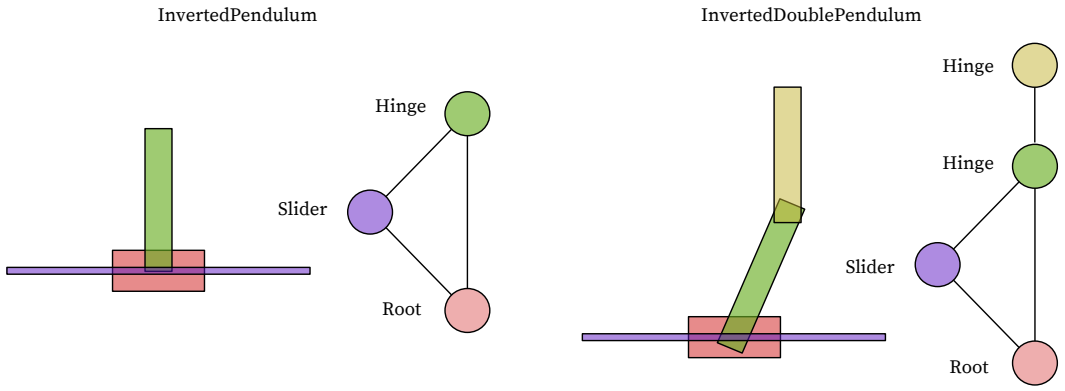


Figure B.1: Schematic diagrams and auto-parsed graph structures of the InvertedPendulum and InvertedDoublePendulum.

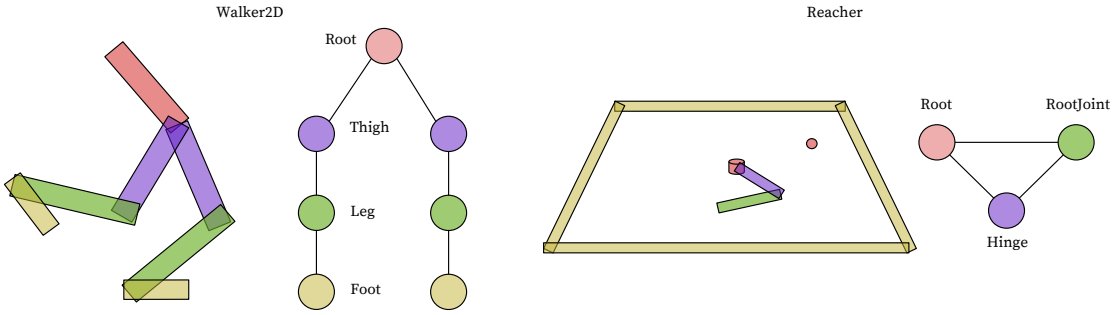


Figure B.2: Schematic diagrams and auto-parsed graph structures of the Walker2d and Reacher.

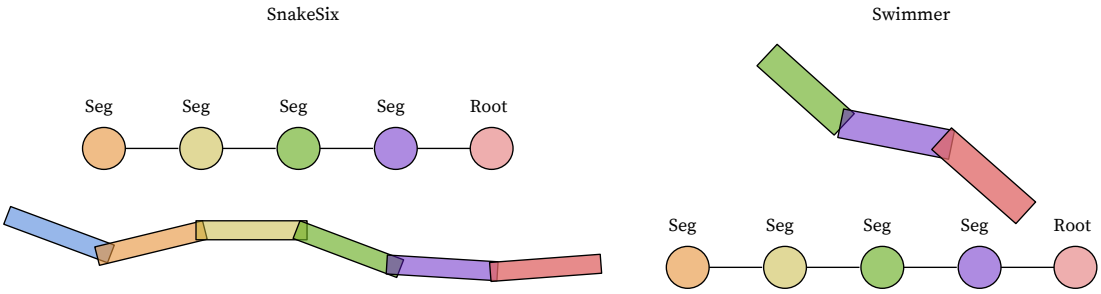


Figure B.3: Schematic diagrams and auto-parsed graph structures of the SnakeSix and Swimmer.

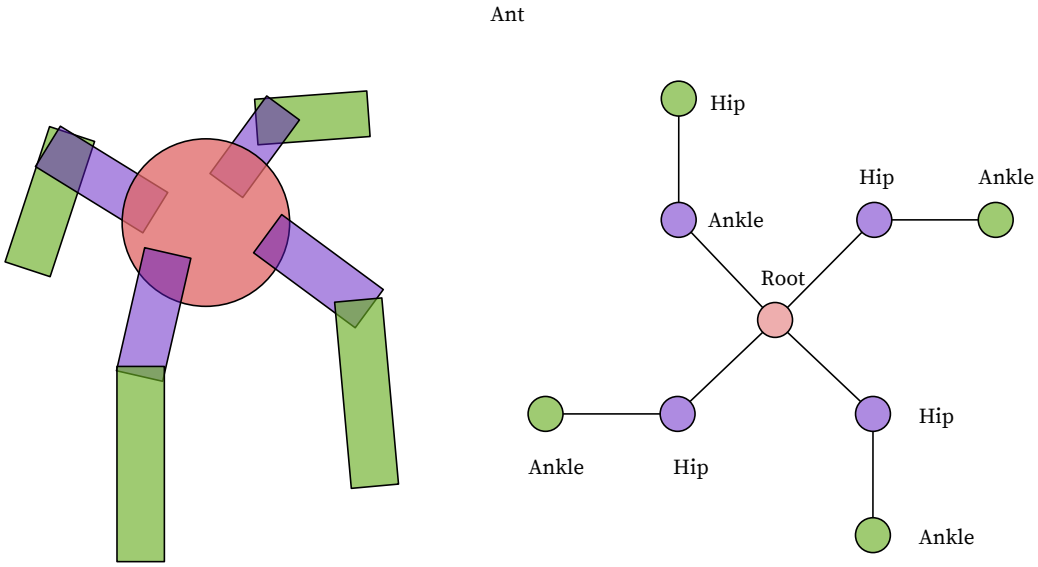


Figure B.4: Schematic diagrams and auto-parsed graph structures of the Ant.

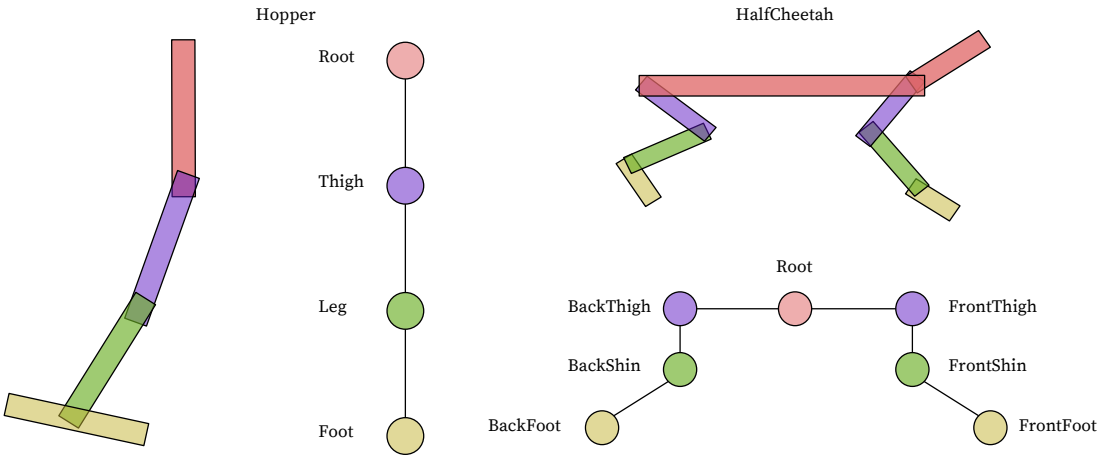


Figure B.5: Schematic diagrams and auto-parsed graph structures of the Hopper and HalfCheetah.

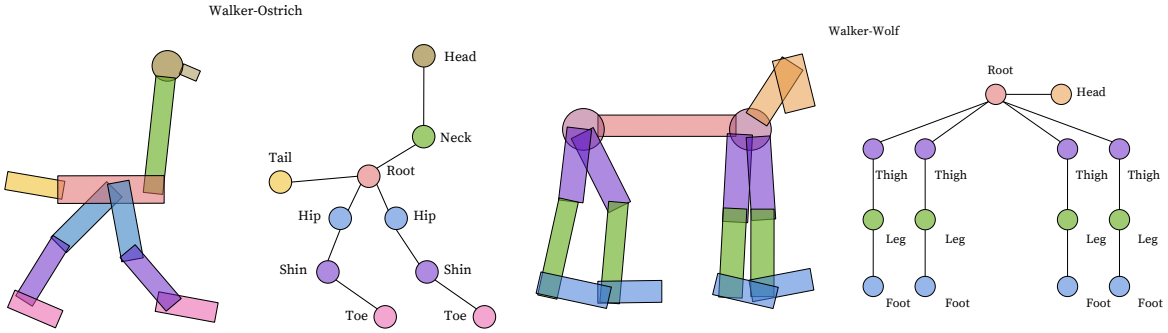


Figure B.6: Schematic diagrams and auto-parsed graph structures of Walker-Ostrich and Walker-Wolf.

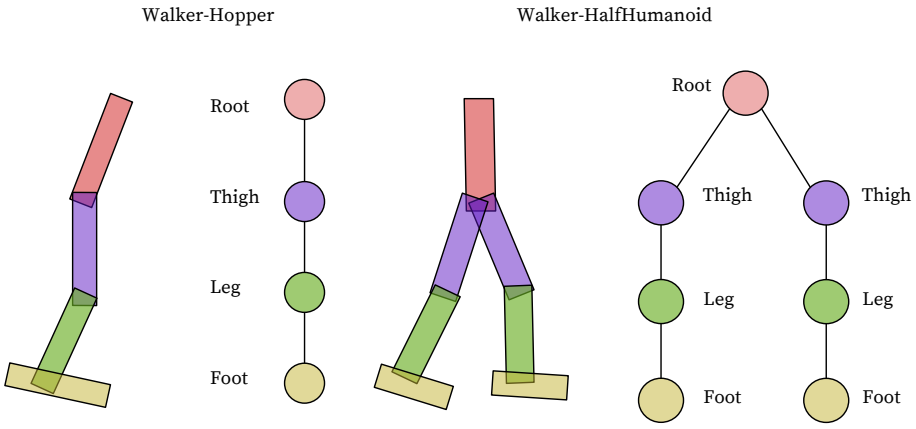


Figure B.7: Schematic diagrams and auto-parsed graph structures of Walker-Hopper and Walker-HalfHumanoid.

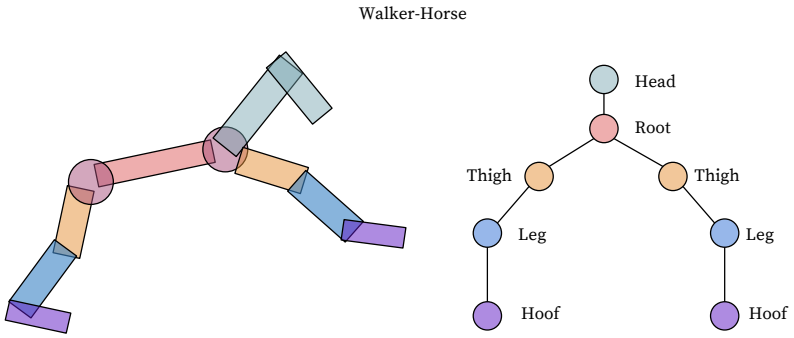


Figure B.8: Schematic diagrams and auto-parsed graph structures of Walker-Horse.

B.2 Schematic Figures of `Walkers` Agents

We design `Walker` task-set, which contains five 2d walkers in the MuJoCo engine.

Appendix C

Detailed Results

C.1 Details of Zero-shot Learning Results

C.1.1 Linear Normalization Zero-shot Results and colorization

As the scale of zero-shot results is very different, we normalize the results across different models for each transfer learning task. For each task, we record the worst value of results from different models and the pre-set worst value V_{min}. we set the normalization minimun value as this worst value. We calculate the normalization maximum value by $\lfloor \max(V)/\text{IntLen} \rfloor * \text{IntLen}$.

parameters	V_{min}	IntLen
value	-100, -100, -20	30, 30, 30

Table C.1: Parameters for linear normalization during zero-shot results. Results are shown in the order of Centipedes' reward, Centipedes' running-length, Snakes' reward

Table C.2: Results of the zero-shot learning of `Centipede`

	Reward Max					Reward Std					Reward Average				
Task	Random	MLPAA	MLPP	TreeNet	NerveNet	Random	MLPAA	MLPP	TreeNet	NerveNet	Random	MLPAA	MLPP	TreeNet	NerveNet
CentipedeFour2CentipedeSix	-5.7	396.4	-23.7	100.4	561.4	20.3	86.2	74	73.6	125.9	-31.6	109.4	-126.7	-16.5	139.6
CentipedeFour2CentipedeEight	-10.3	107.1	-33.7	84.7	161.6	30.7	19.5	146.3	17.8	35.2	-45.8	18.2	-190.9	10.2	44.3
CentipedeFour2CentipedeTen	-5.5	55.7	-47	510.1	136.7	30.3	13.3	132	228.5	28.7	-44.3	11.4	-195.6	-101.5	39.8
CentipedeFour2CentipedeTwelve	-9.9	45.2	-24.8	84.8	147.7	30.5	11.9	173.2	14.9	27.4	-48.7	9.9	-215.8	17.6	38.6
CentipedeFour2CentipedeFourteen	-10.5	46.9	-21.8	550	142.8	38.1	11.8	170.9	217.6	31.2	-52	8	-233	-79.6	39
CentipedeFour2CentipedeTwenty	-11.9	59.6	-31.8	420	176.8	52.2	13.4	299.8	84.7	37.2	-72.4	5.8	-353.4	158.4	39
CentipedeFour2CentipedeThirty	1.4	44.3	-5.8	169.4	137.9	64.8	10.8	262.2	29.2	35	-67.6	-1.1	-289.8	30	30.4
CentipedeFour2CentipedeForty	-0.9	22	-3.7	105.1	244.1	64.9	8.4	220.2	190.8	32.4	-71.5	-4.3	-272	-56.2	24.3
CentipedeSix2CentipedeEight	-10.3	92.9	-31	394.9	2586.6	30.7	16.1	127.4	75.2	884.4	-45.8	21.1	-191.4	320.8	1674.9
CentipedeSix2CentipedeTen	-5.5	29.4	-29.6	257	2154.9	30.3	115.7	157.2	76.4	695	-44.3	-42.4	-193.2	44.7	940.5
CentipedeSix2CentipedeTwelve	-9.9	29.6	-40.2	157.1	1421.3	30.5	64.6	177.8	30.9	351.6	-49.1	-14.4	-227	19.5	367.7
CentipedeSix2CentipedeFourteen	-10.5	33.3	-27.6	146.4	1150.2	38.1	108.5	141.3	9.9	249	-52	-10	-221.3	126.3	247.8
CentipedeSix2CentipedeTwenty	-11.9	39.6	-29.1	163.5	1452	52.2	10.1	379	258.9	197.6	-72.4	10	-351.4	-233.6	198.5
CentipedeSix2CentipedeThirty	1.4	29.3	-4.3	80.2	614.3	64.6	158.2	225.3	16.2	124.3	-67.1	-28.5	-263.3	17.6	114.9
CentipedeSix2CentipedeForty	-0.9	27	-5.1	87.3	575.4	65.1	7.9	288.6	18.6	114.2	-72.7	4.3	-320.2	21.1	97.8
CentipedeFour2CpCentipedeSix	7.9	44.2	-30.2	729.8	175.1	12	44.5	63.5	242.8	30.4	-17	-5.1	-113.9	249.5	47.6
CentipedeFour2CpCentipedeEight	0.1	23.3	-23.7	113.3	115	14	4.5	64.6	22.8	24.1	-25.5	5.1	-132.2	33.3	40
CentipedeFour2CpCentipedeTen	-6.6	27.8	-29.9	141.4	96.6	16	50.6	80.8	22.8	22.3	-28.2	-12.8	-133.7	28.2	40.2
CentipedeFour2CpCentipedeTwelve	-6.2	27.1	-44	665.6	84.6	14.9	5.7	87.9	106.8	18.7	-32.2	4.4	-174	578.8	34.8
CentipedeFour2CpCentipedeFourteen	-11.6	25.5	-24.8	383	101.9	20.5	5.7	99.8	82.9	21.8	-41	3.8	-194	157.9	33.9
CentipedeSix2CpCentipedeEight	0.1	169.1	-37.1	528.9	2143.2	13.9	31.8	76.2	152.8	494.6	-25.4	36.5	-141.9	398.9	523.6
CentipedeSix2CpCentipedeTen	-6.6	55.8	-29.9	389.7	1904.6	16	8.9	81.9	112.9	444.5	-28.2	12.8	-155.2	277.8	504
CentipedeSix2CpCentipedeTwelve	-6.2	49.2	-34.7	343.4	1429.4	15	8.6	88.3	177.8	234.7	-32	12.1	-177.5	-114.9	255.9
CentipedeSix2CpCentipedeFourteen	-11.6	48	-27.7	171.4	976.2	20.5	8.3	104.5	136.2	190.2	-41	11.8	-187.5	-67.7	224.8

	Distance Max					Distance Std					Distance Average				
Task	Random	MLPAA	MLPP	TreeNet	NerveNet	Random	MLPAA	MLPP	TreeNet	NerveNet	Random	MLPAA	MLPP	TreeNet	NerveNet
CentipedeFour2CentipedeSix	63.3	2082.6	40.5	61	2523.5	55.1	460.7	48.9	35.5	578.8	-75.6	545.3	-73.5	-47.7	577.3
CentipedeFour2CentipedeEight	70.5	528.1	77.8	-47.9	676.4	60.8	101	68.7	13.7	164.6	-91	62	-80.8	-84.8	146.9
CentipedeFour2CentipedeTen	29.5	283.2	104.5	79.9	580.2	51.8	69	56.3	54.5	136	-76.5	21	-89.9	-70	128.4
CentipedeFour2CentipedeTwelve	31.3	179.8	43.1	-34.3	674.6	45.8	58.1	47.9	12.4	133.7	-81.7	13.1	-76.9	-63.2	126.3
CentipedeFour2CentipedeFourteen	20.1	188.8	131.4	140.1	660.5	50.4	57.5	59.6	51.4	151.8	-89	0	-86.4	-73.2	125.7
CentipedeFour2CentipedeTwenty	59.7	191.3	53.4	-3.9	852.2	46.4	57	53.2	16.1	188.5	-95.2	-16.8	-98.4	-44.2	131.8
CentipedeFour2CentipedeThirty	-7.5	159.9	-28.5	-40.7	675.4	47.5	47.2	58.8	25.2	181.3	-111.9	-60.5	-122.5	-84.4	82
CentipedeFour2CentipedeForty	166.5	194.5	162.2	156.5	1332.4	61.9	37.2	59.3	42.7	160.9	64	114.9	60.3	90.8	247.5
CentipedeSix2CentipedeEight	70.5	461.9	109.7	73.8	16225.5	60.8	93.3	65.7	26.4	5606.5	-91	87.8	-88.6	8.5	10612.6
CentipedeSix2CentipedeTen	29.5	101.6	13.1	-36.5	14438.9	51.8	58.7	43.8	15.9	4669.9	-76.5	-17	-81.8	-77.4	6343.6
CentipedeSix2CentipedeTwelve	31.3	130	163.9	-53.4	10110.1	46	51.2	59.8	13.4	2409.6	-81.2	-1.4	-79.3	-91.5	2532.2
CentipedeSix2CentipedeFourteen	20.1	121.6	25.3	-5.2	8031.4	50.4	54.7	53.7	19.4	1748.2	-89	-3.9	-83.2	-51.5	1749.7
CentipedeSix2CentipedeTwenty	59.7	130.9	3.8	70.3	10422.7	46.4	46.1	58.3	51.3	1415.8	-95.2	-4.9	-105.4	-102.3	1447.4
CentipedeSix2CentipedeThirty	-7.5	57.9	-38.3	-41.1	4693.1	46.4	48.7	52.9	21.1	941.7	-111.2	-48.2	-127.3	-76.7	867.5
CentipedeSix2CentipedeForty	166.5	232	158.7	156.9	4769.4	61.7	33.7	64.3	27.3	884.5	63.9	140.5	56.9	101.8	971.5
CentipedeFour2CpCentipedeSix	61.9	122.4	60.6	36.8	675.4	50.8	35.1	57.4	42.3	126	-77.3	-22.5	-79.9	-72.2	91.1
CentipedeFour2CpCentipedeEight	27.7	49.1	35.7	-5.1	408.7	41	16.2	57.5	27.1	101.7	-82.9	-26.9	-91.8	-66.9	80.1
CentipedeFour2CpCentipedeTen	18.3	61.1	0	-53.9	363	40.5	31.7	35.5	12	95.4	-82.9	-36.6	-88.8	-83.7	86.9
CentipedeFour2CpCentipedeTwelve	1.9	15.3	57.8	10.8	299.6	39.2	21	49.9	51.2	76.2	-87.9	-30.4	-98.3	-62.2	72
CentipedeFour2CpCentipedeFourteen	-1.1	23.5	43.6	-43.8	349.2	42	19.8	48.3	16	93.7	-96.5	-33.5	-92.5	-76.3	74.8
CentipedeSix2CpCentipedeEight	27.7	788.5	66.7	-23.3	12628.5	41.1	157.2	58.6	18.1	2939.4	-83	138.3	-90.7	-63.9	3117.3
CentipedeSix2CpCentipedeTen	18.3	215	16.2	-28.4	12256.1	40.5	42.7	46.6	16.9	2844.5	-82.9	13.6	-86.4	-72.3	3230.3
CentipedeSix2CpCentipedeTwelve	1.9	209.6	97.2	52.7	9370.8	39.1	41.4	46.5	46.9	1538.9	-87.5	7.3	-91.4	-90.8	1675.5
CentipedeSix2CpCentipedeFourteen	-1.1	148.8	22.8	11.3	6456.2	42	38	46.5	33.6	1293.4	-96.5	2.9	-92.5	-93.9	1517.6

Table C.3: Results of the zero-shot learning of `Snake`

	Reward Min				
	Random	MLPAA	MLPP	TreeNet	NerveNet
SnakeFour-v1	-31.719	-43.328	-64.958	-58.811	308.994
SnakeFive-v1	-56.382	29.262	-55.562	-39.346	301.4
SnakeSix-v1	-44.888	-5.306	-42.744	-76.42	266.066
SnakeSeven-v1	-52.764	-25.511	-49.765	-59.52	231.557
SnakeFive-v1	-56.382	-30.466	-49.953	-47.339	329.456
SnakeSix-v1	-44.888	-35.655	-52.18	-90.837	333.237
SnakeSeven-v1	-52.764	89.358	-57.334	-52.015	253.828
	Reward Std				
	Random	MLPAA	MLPP	TreeNet	NerveNet
SnakeFour-v1	9.544	5.792	10.769	10.146	5.513
SnakeFive-v1	13.318	6.191	11.715	7.883	5.788
SnakeSix-v1	11.515	13.535	10.304	16.869	7.395
SnakeSeven-v1	11.21	12.131	10.981	6.953	7.179
SnakeFive-v1	13.318	52.71	11.544	8.519	5.85
SnakeSix-v1	11.515	64.549	10.528	23.109	6.604
SnakeSeven-v1	11.21	10.683	11.615	13.529	14.084
	Reward Max				
	Random	MLPAA	MLPP	TreeNet	NerveNet
SnakeFour-v1	3.863	-19.409	10.118	0.918	338.374
SnakeFive-v1	20.585	63.866	3.136	-2.329	326.974
SnakeSix-v1	7.6	58.779	3.51	-2.139	305.011
SnakeSeven-v1	-2.28	51.265	1.325	-28.087	275.536
SnakeFive-v1	20.585	139.617	5.305	1.487	356.366
SnakeSix-v1	7.6	156.771	2.316	-4.159	367.768
SnakeSeven-v1	-2.28	143.824	2.596	8.394	336.561
	Reward Average				
	Random	MLPAA	MLPP	TreeNet	NerveNet
SnakeFour-v1	-13.531	-30.185	-12.363	-21.824	325.476
SnakeFive-v1	-17.829	51.411	-16.481	-21.256	314.919
SnakeSix-v1	-19.89	32.2	-19.674	-42.461	282.426
SnakeSeven-v1	-22.099	23.533	-22.378	-42.742	256.293
SnakeFive-v1	-17.829	89.823	-16.54	-15.975	342.881
SnakeSix-v1	-19.89	105.632	-20.454	-34.343	351.85
SnakeSeven-v1	-22.099	115.123	-22.81	-22.383	313.149

C.2 Details of Robustness results

Model	Mass				
	Hopper	Halfhumanoid	Horse	Wolf	Ostrich
NerveNet	1981.59	2519.6	2304.25	1862.21	922.12
	2047.92	2392.05	1900.43	1955.16	932.48
	1968.87	2282.84	1703.47	1648.28	983.11
	2062.36	2161.61	1239.72	1576.21	910.29
	1852.76	1804.52	946.08	1354.51	791.84
MLP	2268.64	421.8	1692.08	1932.64	878.75
	1932.98	2115.75	1083.62	1754.94	945.94
	1637.47	278.92	690.22	1459.93	879.66
	1591.51	1224.98	538.14	1039.9	805.99
	1073.1	353.56	443.92	814.55	802.41
Model	Strength				
	Hopper	Halfhumanoid	Horse	Wolf	Ostrich
NerveNet	1961.62	1528.89	2316.82	1827.73	794.56
	1117.91	940.96	960.34	862.19	539.45
	492.33	594.53	488.51	274.77	371.19
	468.4	308.08	305.16	189.49	255.12
	446.14	249.6	201.42	133.84	197.84
MLP	800.15	1439.89	1236.88	1730.88	230.85
	431.97	1040.34	209.54	1108.32	483.36
	429.76	765.7	306.08	519.73	163.36
	426.27	117.05	228.26	88.28	232.8
	412.39	65.29	187.72	108.97	133.99

Table C.4: Detail results of Robustness testing.

Appendix D

Structured Weight Sharing in MLP

In the MLP-Bind method, we bind the weights of MLP. By doing this, the weights of the agent from the similar structures will be shared. For example, in the `Centipede` environment, the weights from observation to action of all the LeftHips are constrained to be same.

Note that MLP-Bind and TreeNet are equivalent for the Snake agents, since the snakes only have one type of joint. We ran MLP-Bind for the zero-shot (table D.1) and fine-tuning experiments (figure D.1) on centipedes.

We summarize the results here:

1. Zero-shot performances of MLP-Bind and MLPAA are very similar. Both models have limited performance in the zero-shot scenario.
2. For fine-tuning on ordinary centipedes from pretrained models, the performance is only slightly worse than when using MLP. In our experiment, in the two curves of transferring from CentipedeFour to CentipedeEight as well as CentipedeSix to CentipedeEight, MLP-Binds reward is from 100 to 1000 worse than MLPAA during fine-tuning.
3. For the Crippled agents, the MLP-Bind agent is very bad. This might be due to MLP-Bind not being able to efficiently exploit the information of crippled and well-functioning legs.

D.1 Zero-shot Results for `Centipede`

The results are recorded in table D.1.

D.2 Results of fine-tuning on `Centipede`

The results are recorded in figure D.1.

Table D.1: Results of the zero-shot learning of `Centipede` using MLP-Bind method.

Task	Distance Max	Distance Std	Distance Average	Reward Max	Reward Std	Reward Average
CentipedeFour2CentipedeSix	7.09	157.46	141.51	274.99	52.26	62.13
CentipedeFour2CentipedeEight	2.8	53.23	-2	144.44	21.83	24.62
CentipedeFour2CentipedeTen	1.32	38.12	-5.84	256.34	31.45	30.06
CentipedeFour2CentipedeTwelve	0.83	36.14	-26.92	160.02	30.8	28.79
CentipedeFour2CentipedeFourteen	1.28	39.76	-17.9	323.97	46.03	41.41
CentipedeFour2CentipedeTwenty	1.07	39.98	-34.45	527.04	76.31	44.94
CentipedeFour2CentipedeThirty	0.51	34.58	-56.76	565.61	82.94	36.53
CentipedeFour2CentipedeForty	2.1	38.05	127.39	484.13	57.71	27.76
CentipedeSix2CentipedeEight	53.73	1073.33	1158.73	1070.86	209.24	235.97
CentipedeSix2CentipedeTen	1.94	50.27	33.14	63.05	11.88	18.65
CentipedeSix2CentipedeTwelve	1.76	44.27	9.4	60.63	10.68	14.92
CentipedeSix2CentipedeFourteen	2	47.81	8.09	60.44	12.21	15.69
CentipedeSix2CentipedeTwenty	1.07	38.46	-10.27	49.71	11.42	14.25
CentipedeSix2CentipedeThirty	0.93	43.25	-37.93	55.97	13.68	10.74
CentipedeSix2CentipedeForty	2.28	37.6	140.06	62.63	10.55	6.63
CentipedeFour2CpCentipedeSix	2.03	35.71	-19.01	71.5	13.57	11.47
CentipedeFour2CpCentipedeEight	0.5	23.52	-25.71	85.42	10.71	7.34
CentipedeFour2CpCentipedeTen	0.28	18.64	-26.68	32.86	7.64	7.74
CentipedeFour2CpCentipedeTwelve	0.1	15.99	-29.97	50.4	7.98	6.35
CentipedeFour2CpCentipedeFourteen	0.23	18.74	-30.56	50.78	9.74	7.52
CentipedeSix2CpCentipedeEight	4.43	90.32	39.57	162.91	29.33	29.09
CentipedeSix2CpCentipedeTen	0.65	25.08	-23.72	41.9	7.38	8.82
CentipedeSix2CpCentipedeTwelve	0.6	26.51	-27.88	30.09	7.2	8.32
CentipedeSix2CpCentipedeFourteen	0.67	28.85	-20.84	43.44	8.66	11.02

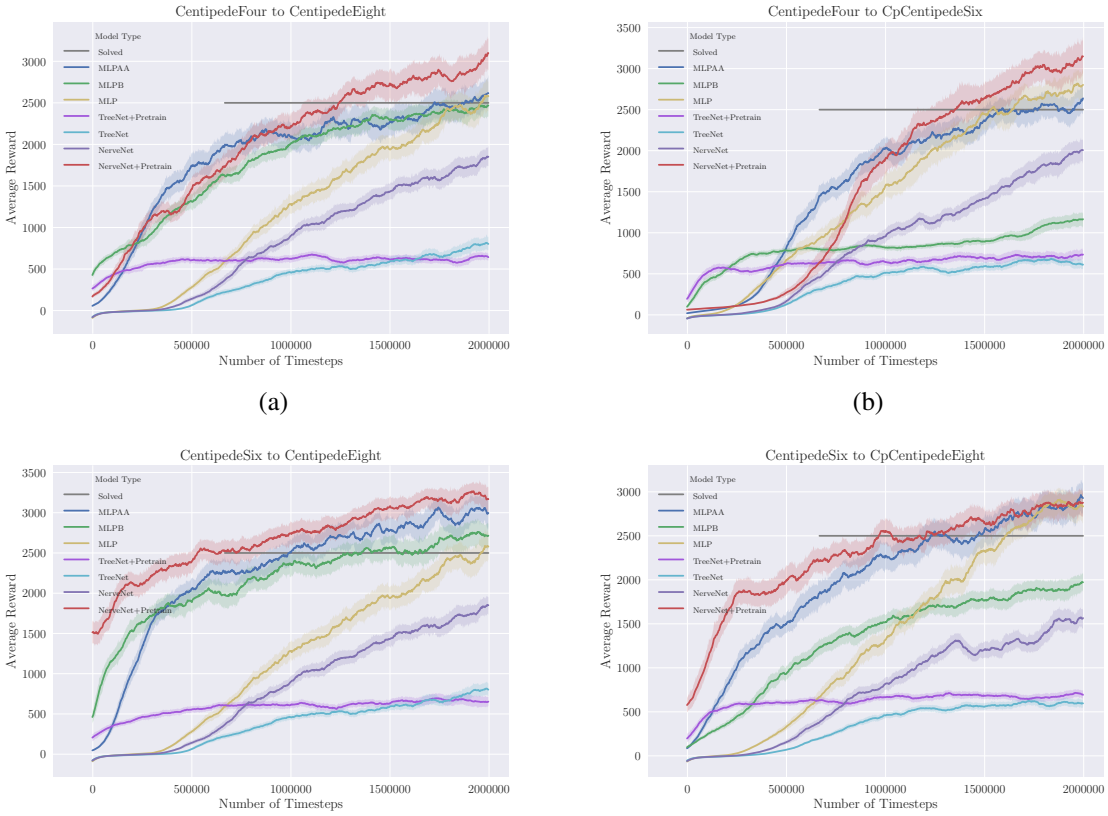


Figure D.1: (a), (c): Results of fine-tuning for *size transfer* experiments. (b), (d) Results of fine-tuning for *disability transfer* experiments.

Bibliography

[1] Haitham B Ammar, Eric Eaton, Paul Ruvolo, and Matthew Taylor. Online multi-task learning for policy gradient methods. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1206–1214, 2014.

[2] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. *arXiv preprint arXiv:1611.01796*, 2016.

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

[4] André Barreto, Will Dabney, Rémi Munos, Jonathan J Hunt, Tom Schaul, David Silver, and Hado P van Hasselt. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4056–4066, 2017.

[5] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002.

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

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

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

[9] Daniele Calandriello, Alessandro Lazaric, and Marcello Restelli. Sparse multi-task reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 819–827, 2014.

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

[11] Stelian Coros, Philippe Beaudoin, and Michiel Van de Panne. Generalized biped walking control. *ACM Transactions on Graphics (TOG)*, 29(4):130, 2010.

[12] 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.

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

[14] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[15] 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.

[16] 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.

[17] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 4397–4404. IEEE, 2015.

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

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

[20] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

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

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

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

[24] 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.

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

[26] 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.

[27] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002.

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

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

[30] 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*, pages 3675–3683, 2016.

[31] Vikash Kumar, Emanuel Todorov, and Sergey Levine. Optimal control with learned local models: Application to dexterous manipulation. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 378–383. IEEE, 2016.

[32] D Kuvayev and Richard S Sutton. Model-based reinforcement learning. Technical report, Tech. rept. university of massachusetts, Dept of computer science, 1997.

[33] 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.

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

[35] Weiwei Li and Emanuel Todorov. Iterative linear quadratic regulator design for nonlinear biological movement systems. In *ICINCO (1)*, pages 222–229, 2004.

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

[37] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[38] 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.

[39] 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.

[40] Luke Metz, Julian Ibarz, Navdeep Jaitly, and James Davidson. Discrete sequential prediction of continuous actions for deep rl. *arXiv preprint arXiv:1705.05035*, 2017.

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

[42] 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.

[43] Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*, 2017.

[44] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[45] 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*, pages 5199–5208, 2017.

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

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

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

[49] 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)*, pages 1889–1897, 2015.

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

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

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

[53] 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.

[54] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[55] 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.

[56] 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)*, pages 129–136, 2011.

[57] 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*, pages 1057–1063, 2000.

[58] 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*, pages 1057–1063, 2000.

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

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

[61] Yee Teh, Victor Bapst, Razvan Pascanu, Nicolas Heess, John Quan, James Kirkpatrick, Wojciech M Czarnecki, and Raia Hadsell. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4497–4507, 2017.

[62] 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*, pages 5026–5033. IEEE, 2012.

[63] Emanuel Todorov and Weiwei Li. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *American Control Conference, 2005. Proceedings of the 2005*, pages 300–306. IEEE, 2005.

[64] 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.

[65] Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *Proceedings of the 24th international conference on Machine learning*, pages 1015–1022. ACM, 2007.

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

[67] Zhaoyang Yang, Kathryn Merrick, Hussein Abbass, and Lianwen Jin. Multi-task deep reinforcement learning for continuous action control. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3301–3307, 2017.