Expanding Motor Skills using Relay Networks

Visak CV Kumar
Georgia Institute of Technology
visak3@gatech.edu

Sehoon Ha
Google Brain
sehoonha@google.com

C.Karen Liu
Georgia Institute of Technology
karenliu@cc.gatech.edu

Abstract: While recent advances in deep reinforcement learning have achieved impressive results in learning motor skills, many policies are only capable within a limited set of initial states. We propose an algorithm that sequentially decomposes a complex robotic task into simpler subtasks and trains a local policy for each subtask such that the robot can expand its existing skill set gradually. Our key idea is to build a directed graph of local control policies represented by neural networks, which we refer to as *relay neural networks*. Starting from the first policy that attempts to achieve the task from a small set of initial states, the algorithm iteratively discovers the next subtask with increasingly more difficult initial states until the last subtask matches the initial state distribution of the original task. The policy of each subtask aims to drive the robot to a state where the policy of its preceding subtask is able to handle. By taking advantage of many existing actor-critic style policy search algorithms, we utilize the optimized value function to define "good states" for the next policy to relay to.

Keywords: Model-free Reinforcement Learning, Motor Skill Learning

1 Introduction

The recent advances in deep reinforcement learning have motivated robotic researchers to tackle increasingly difficult control problems. For example, OpenAI RoboSchool [1] challenges the researchers to create robust control policies capable of locomotion while following high-level goals under external perturbation. One obvious approach is to use a powerful learning algorithm and a large amount of computation resources to learn a wide range of situations. While this direct approach might work occasionally, it is difficult to scale up with the ever-increasing challenges in robotics.

Alternatively, we can consider to break down a complex task into a sequence of simpler subtasks and solve each subtask sequentially. The control policy for each subtask can be trained, sequentially as well, to expand existing skill set gradually. While an arbitrary decomposition of a task might lead to suboptimal control policy, a robot performing actions sequentially is often considered a practical strategy to complete a complex task reliably [2, 3], albeit not optimally.

This paper introduces a new technique to expand motor skills by connecting new policies to existing ones. In the similar spirit of hierarchical reinforcement learning (HRL) [4, 5] which decomposes a large-scale Markov Decision Process (MDP) into subtasks, the key idea of this work is to build a directed graph of local policies represented by neural networks, which we refer to as *relay neural networks*. Starting from the first policy that attempts to achieve the task from a small set of initial states, the algorithm gradually expands the set of successful initial states by connecting new policies, one at a time, to the existing graph until the robot can achieve the original complex task. Each policy is trained for a new set of initial states with an objective function that encourages the policy to drive the new states to existing "good" states.

The main challenge of this method is to determine the "good" states from which following the relay networks will eventually achieve the task. Fortunately, a few modern policy learning algorithms,

2nd Conference on Robot Learning (CoRL 2018), Zürich, Switzerland.

such as TRPO-GAE [6], PPO [7], DDPG [8], or A3C [9], provide an estimation of the value function along with the learned policy. Our algorithm utilizes the value function to train a policy that relays to its preceding policy, as well as to transition the policies on the graph during online execution. We demonstrate that the relay networks can solve complex control problems for underactuated systems.

2 Related Works

Hierarchical Reinforcement Learning: HRL has been proven successful in learning policies for large-scale problems. This approach decomposes problems using *temporal abstraction* which views sub-policies as macro actions, or *state abstraction* which focuses on certain aspects of state variables relevant to the task. Prior work [10, 11, 12, 13, 14] that utilizes temporal abstraction applies the idea of parameterized goals and pseudo-rewards to train macro actions, as well as training a meta-controller to produce macro actions. One notable work that utilizes state abstraction is MAXQ value function decomposition which decomposes a large-scale MDP into a hierarchy of smaller MDP's [4, 15, 16]. MAXQ enables individual MDP's to only focus on a subset of state space, leading to better performing policies. Our relay networks can be viewed as a simple case of MAXQ in which the recursive subtasks, once invoked, will directly take the agent to the goal state of the original MDP. That is, in the case of relay networks, the Completion Function that computes the cumulative reward after the subtask is finished always returns zero. As such, our method avoids the need to represent or approximate the Completion Function, leading to an easier RL problem for continuous state and action spaces.

Chaining and Scheduling Control Policies: Many researchers [17] have investigated the idea of chaining a new policy to existing policies. Tedrake [18] proposed the LQR-Tree algorithm that combines locally valid linear quadratic regulator (LQR) controllers into a nonlinear feedback policy to cover a wider region of stability. Borno et al. [19] further improved the sampling efficiency of RRT trees [18] by expanding the trees with progressively larger subsets of initial states. However, the success metric for the controllers is based on Euclidean distance in the state space, which can be inaccurate for high dimensional control problems with discontinuous dynamics. Konidaris et al. [20] proposed to train a chain of controllers with different initial state distributions. The rollouts terminate when they are sufficiently close to the initial states of the parent controller. They discovered an initiation set using a sampling method in a low dimensional state space. In contrast, our algorithm utilizes the value function of the parent controller to modify the reward function and define the terminal condition for policy training. There also exists a large body of work on scheduling existing controllers, such as controllers designed for taking simple steps [21] or tracking short trajectories [22]. Inspired by the prior art, our work demonstrates that the idea of sequencing a set of local controllers can be realized by learning policies represented by the neural networks.

Learning with Structured Initial States: Manipulating the initial state distribution during training has been considered a promising approach to accelerate the learning process. Kakade and Langford [23] studied theoretical foundation of using "exploratory" restart distribution for improving the policy training. Recently, Popov et al. [24] demonstrated that the learning can be accelerated by taking the initial states from successful trajectories. Florensa et al. [25] proposed a reverse curriculum learning algorithm for training a policy to reach a goal using a sequence of initial state distributions increasingly further away from the goal. We also train a policy with reversely-ordered initial states, but we exploited chaining mechanism of multiple controllers rather than training a single policy.

Learning Policy and Value Function: Our work builds on the idea of using value functions to provide information for connecting policies. The approach of actor-critic [26, 27] explicitly optimizes both the policy (actor) and the value function (critic) such that the policy update direction can be computed with the help of critic. Although the critic may introduce bias, the policy update can be less subject to variance compared to pure policy gradient methods, such as REINFORCE [28]. Recently, Lillicrap et al. [8] proposed Deep Deterministic Policy Gradient (DDPG) that combines the actor-critic approach [29] with Deep Q Network (DQN) [30] for solving high-dimensional continuous control problems. Schulman et al. [6] proposed the general advantage estimator (GAE) for reducing the variance of the policy gradient while keeping the bias at a reasonable level. Further, Mnih et al. [9] demonstrated a fast and robust learning algorithm for actor-critic networks using an asynchronous update scheme. Using these new policy learning methods, researchers have demonstrated that policies and value functions represented by neural networks can be applied to high-dimensional complex control problems with continuous actions and states [31, 14, 32]. Our algorithm can work with any of

these policy optimization methods, as long as they provide a value function approximately consistent with the policy.

3 Method

Our approach to a complex robotic task is to automatically decompose it to a sequence of subtasks, each of which aims to reach a state where the policy of its preceding subtask is able to handle. The original complex task is formulated as a MDP described by a tuple $\{\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \rho, P\}$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, r is the reward function, $\mathcal{T}$ is the set of termination conditions, $\rho = N(\boldsymbol{\mu}_\rho, \Sigma_\rho)$ is the initial state distribution, and P is the transition probability. Instead of solving for a single policy for the MDP, our algorithm solves for a set of policies and value functions for a sequence of simpler MDP's. A policy, $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$, is represented as a Gaussian distribution of action $\mathbf{a} \in \mathcal{A}$ conditioned on a state $\mathbf{s} \in \mathcal{S}$. The mean of the distribution is represented by a fully connected neural network and the covariance is defined as part of the policy parameters to be optimized. A value function, $V : \mathcal{S} \mapsto R$, is also represented as a neural network whose parameters are optimized by the policy learning algorithm.

We organize the MDP's and their solutions in a directed graph Γ. A node $\mathcal{N}_k$ in Γ stores the initial state distribution ρ_k of the k^{th} MDP, while a directed edge connects an MDP to its parent MDP and stores the solved policy (π_k), the value function (V_k), and the threshold of value function ($\bar{V}_k$) (details in Section 3.2). As the robot expands its skill set to accomplish the original task, a chain of MDP's and policies is developed in Γ. If desired, our algorithm can be extended to explore multiple solutions to achieve the original task. Section 3.4 describes how multiple chains can be discovered and merged in Γ to solve multi-modal problems.

3.1 Learning Relay Networks

The process of learning the relay networks (Algorithm 1) begins with defining a new initial state distribution ρ_0 which reduces the difficulty of the original MDP (Line 3). Although our algorithm requires the user to define ρ_0, it is typically intuitive to find a ρ_0 which leads to a simpler MDP. For example, we can define ρ_0 as a Gaussian whose mean, $\boldsymbol{\mu}_{\rho_0}$, is near the goal state of the problem.

Once ρ_0 is defined, we proceed to solve the first subtask $\{\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \rho_0, P\}$, whose objective function is defined as the expected accumulated discounted rewards along a trajectory:

$$J_0 = \mathbb{E}_{\mathbf{s}_{0:t_f}, \mathbf{a}_{0:t_f}}[\sum_{t=0}^{t_f} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)], \tag{1}$$

where γ is the discount factor, $\mathbf{s}_0$ is the initial state of the trajectory drawn from ρ_0, and t_f is the terminal time step of the trajectory. We can solve the MDP using PPO/TRPO or A3C/DDPG to obtain the policy π_0, which drives the robot from a small set of initial states from ρ_0 to states where the original task is completed, as well as the value function $V_0(\mathbf{s})$, which evaluates the return by following π_0 from state $\mathbf{s}$ (Line 5).

The policy for the subsequent MDP aims to drive the rollouts toward the states which π_0 can successfully handle, instead of solving for a policy that directly completes the original task. To determine whether π_0 can handle a given state $\mathbf{s}$, one can generate a rollout by following π_0 from $\mathbf{s}$ and calculate its return. However, this approach can be too slow for online applications. Fortunately, many of the modern policy gradient methods, such as PPO or A3C, produce a value function, which provides an approximated return from $\mathbf{s}$ without generating a rollout. Our goal is then to determine a threshold $\bar{V}_0$ for $V_0(\mathbf{s})$ above which $\mathbf{s}$ is deemed "good" (Line 6). The details on how to determine such a threshold are described in Section 3.2. We can now create the first node $\mathcal{N}_0 = \{\rho_0\}$ and add it to Γ, as well as an edge $\mathcal{E} = \{\pi_0, V_0, \bar{V}_0\}$ that connects $\mathcal{N}_0$ to the dummy root node $\mathcal{N}$ (Line 7-8).

Starting from $\mathcal{N}_0$, the main loop of the algorithm iteratively adds more nodes to Γ by solving subsequent MDP's until the last policy π_k, via relaying to previous policies $\pi_{k-1}, \cdots \pi_0$, can generate successful rollouts from the states drawn from the original initial state distribution ρ (Line 10). At each iteration k, we formulate the $(k+1)^{th}$ subsequent MDP by redefining $\mathcal{T}_{k+1}$, ρ_{k+1}, and the objective function J_{k+1}, while using the shared $\mathcal{S}$, $\mathcal{A}$, and P. First, we define the terminal conditions $\mathcal{T}_{k+1}$ as the original set of termination conditions ($\mathcal{T}$) augmented with another condition, $V_k(\mathbf{s}) > \bar{V}_k$

Algorithm 1: LearnRelayNetworks

1: **Input:** MDP $\{\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \rho, P\}$
2: Add root node, $\bar{\mathcal{N}} = \{\emptyset\}$, to Γ
3: Define a simpler initial state distribution ρ_0
4: Define objective function J_0 according to Equation 1
5: $[\pi_0, V_0] \leftarrow$ PolicySearch($\mathcal{S}, \mathcal{A}, \rho_0, J_0, \mathcal{T}$)
6: $\bar{V}_0 \leftarrow$ ComputeThreshold($\pi_0, V_0, \mathcal{T}, \rho_0, J_0$)
7: Add node, $\mathcal{N}_0 = \{\rho_0\}$, to Γ
8: Add edge, $\mathcal{E} = \{\pi_0, V_0, \bar{V}_0, \}$, from $\mathcal{N}_0$ to $\bar{\mathcal{N}}$
9: $k = 0$
10: **while** π_k does not succeed from ρ **do**
11: $\quad \mathcal{T}_{k+1} = \mathcal{T} \cup (V_k(\mathbf{s}) > \bar{V}_k)$
12: $\quad \rho_{k+1} \leftarrow$ Compute new initial state distribution using Equation 2
13: $\quad$ Define objective function J_{k+1} according to Equation 3
14: $\quad [\pi_{k+1}, V_{k+1}] \leftarrow$ PolicySearch($\mathcal{S}, \mathcal{A}, \rho_{k+1}, J_{k+1}, \mathcal{T}_{k+1}$)
15: $\quad \bar{V}_{k+1} \leftarrow$ ComputeThreshold($\pi_{k+1}, V_{k+1}, \mathcal{T}, \rho_{k+1}, J_{k+1}$)
16: $\quad$ Add node, $\mathcal{N}^i_{k+1} = \{\rho_{k+1}\}$, to Γ
17: $\quad$ Add edge, $\mathcal{E} = \{\pi_{k+1}, V_{k+1}, \bar{V}_{k+1}\}$, from node $\mathcal{N}_{k+1}$ to $\mathcal{N}_k$
18: $\quad k \leftarrow k + 1$
19: **end while**
20: **return** Γ

(Line 11). Next, unlike ρ_0, we define the initial state distribution ρ_{k+1} through an optimization process. The goal of the optimization is to find the mean of the next initial state distribution, $\boldsymbol{\mu}_{\rho_{k+1}}$, that leads to unsuccessful rollouts under the current policy π_k, without making the next MDP too difficult to relay. To balance this trade-off, the optimization moves in a direction that reduces the value function of the most recently solved MDP, V_k, until it reaches the boundary of the "good state zone" defined by $V_k(\mathbf{s}) \geq \bar{V}_k$. In addition, we would like $\boldsymbol{\mu}_{\rho_{k+1}}$ to be closer to the mean of the initial state distribution of the original MDP, $\boldsymbol{\mu}_{\rho}$. Specifically, we compute $\boldsymbol{\mu}_{\rho_{k+1}}$ by minimizing the following objective function subject to constraints:

$$\begin{aligned} \boldsymbol{\mu}_{\rho_{k+1}} = \underset{\mathbf{s}}{\operatorname{argmin}}\, & V_k(\mathbf{s}) + w\|\mathbf{s} - \boldsymbol{\mu}_{\rho}\|^2 \\ \text{subject to} \quad & V_k(\mathbf{s}) \geq \bar{V}_k \\ & C(\mathbf{s}) \geq 0 \end{aligned} \tag{2}$$

where $C(\mathbf{s})$ represents the environment constraints, such as the constraint that enforces collision-free states. Since the value function V_k in Equation 2 is differentiable through back-propagation, we can use any standard gradient-based algorithms to solve this optimization. Procedure for selecting the weighting coefficient $\mathbf{w}$ is explained in the appendix.

In addition, we define the objective function of the $(k+1)^{th}$ MDP as follows:

$$J_{k+1} = \mathbb{E}_{\mathbf{s}_{0:t_f}, \mathbf{a}_{0:t_f}}[\sum_{t=0}^{t_f} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t) + \alpha\gamma^{t_f} g(\mathbf{s}_{t_f})], \tag{3}$$

$$\text{where} \quad g(\mathbf{s}_{t_f}) = \begin{cases} V_k(\mathbf{s}_{t_f}), & V_k(\mathbf{s}_{t_f}) > \bar{V}_k \\ 0 & \text{otherwise.} \end{cases}.$$

Besides the accumulated reward, this cost function has an additional term to encourage "relaying". That is, if the rollout is terminated because it enters the subset of $\mathcal{S}$ where the policy of the parent node is capable of handling (i.e. $V_k(\mathbf{s}_{t_f}) > \bar{V}_k$), it will receive the accumulated reward by following the parent policy from $\mathbf{s}_{t_f}$. This quantity is approximated by $V_k(\mathbf{s}_{t_f})$ because it recursively adds the accumulated reward earned by each policy along the chain from $\mathcal{N}_k$ to $\mathcal{N}_0$. If a rollout terminates due to other terminal conditions (e.g. falling on the ground for a locomotion task), it will receive no relay reward. Using this objective function, we can learn a policy π_{k+1} that drives a rollout towards states deemed good by the parent's value function, as well as a value function V_{k+1} that measures the long-term reward from the current state, following the policies along the chain (Line 14). Finally, we

Algorithm 2: ComputeThreshold

1: **Input:** $\pi, V, \mathcal{T}, \rho = N(\boldsymbol{\mu}_\rho, \Sigma_\rho), J$
2: Initialize buffer $\mathcal{D}$ for training data
3: $[\mathbf{s}_1, \cdots, \mathbf{s}_M] \leftarrow$ Sample states from $N(\boldsymbol{\mu}_\rho, 1.5\Sigma_\rho)$
4: $[\tau_1, \cdots, \tau_M] \leftarrow$ Generate rollouts by following π and $\mathcal{T}$ from $[\mathbf{s}_1, \cdots, \mathbf{s}_M]$
5: Compute returns for rollouts: $R_i = J(\tau_i),\ i \in [1, M]$
6: $\bar{R} \leftarrow$ Compute average of returns for rollouts not terminated by $\mathcal{T}$
7: **for** $i = 1 : M$ **do**
8: **if** $R_i > \bar{R}$ **then**
9: Add $(V(\mathbf{s}_i), 1)$ in $\mathcal{D}$
10: **else**
11: Add $(V(\mathbf{s}_i), 0)$ in $\mathcal{D}$
12: **end if**
13: **end for**
14: $\bar{V} \leftarrow$ Classify($\mathcal{D}$)
15: **return** $\bar{V}$

compute the threshold of the value function (Line 15), add a new node, $\mathcal{N}_{k+1}$, to Γ (Line 16), and add a new edge that connects $\mathcal{N}_{k+1}$ to $\mathcal{N}_k$ (Line 17).

The weighting parameter α determines the importance of "relaying" behavior. If α is set to zero, each MDP will attempt to solve the original task on its own without leveraging previously solved policies. The value of α in all our experiments is set to 30 and we found that the results are not sensitive to α value, as long as it is sufficiently large (Section 4.3).

3.2 Computing Threshold for Value Function

In practice, $V(\mathbf{s})$ provided by the learning algorithm is only an approximation of the true value function. We observe that the scores $V(\mathbf{s})$ assigns to the successful states are relatively higher than the unsuccessful ones, but they are not exactly the same as the true returns. As such, we can use $V(\mathbf{s})$ as a binary classifier to separate "good" states from "bad" ones, but not as a reliable predictor of the true returns.

To use V as a binary classifier of the state space, we first need to select a threshold $\bar{V}$. For a given policy, separating successful states from unsuccessful ones can be done as follows. First, we compute the average of true return, $\bar{R}$, from a set of sampled rollouts that do not terminate due to failure conditions. Second, we compare the true return of a given state to $\bar{R}$ to determine whether it is a successful state (successful if the return is above $\bar{R}$). In practice, however, we can only obtain an approximated return of a state via $V(\mathbf{s})$ during policy learning and execution. Our goal is then to find the optimal $\bar{V}$ such that the separation of approximated returns by $\bar{V}$ is as close as possible to the separation of true returns by $\bar{R}$.

Algorithm 2 summarizes the procedure to compute $\bar{V}$. Given a policy π, an approximated value function V, termination conditions $\mathcal{T}$, a Gaussian initial state distribution $\rho = N(\boldsymbol{\mu}_\rho, \Sigma_\rho)$, and the objective function of the MDP J, we first draw M states from an expanded initial state, $N(\boldsymbol{\mu}_\rho, 1.5\Sigma_\rho)$ (Line 3), generate rollouts from these sampled states using π (Line 4), and compute the true return of each rollout using J (Line 5). Because the initial states are drawn from an inflated distribution, we obtain a mixed set of successful and unsuccessful rollouts. We then compute the average return of successful rollouts $\bar{R}$ that do not terminate due to the terminal conditions $\mathcal{T}$ (Line 6). Next, we generate the training set where each data point is a pair of the predicted value $V(\mathbf{s}_i)$ and a binary classification label, "good" or "bad", according to $\bar{R}$ (i.e. $R_i > \bar{R}$ means $\mathbf{s}_i$ is good) (Line 7-13). We then train a binary classifier represented as a decision tree to find to find $\bar{V}$ (Line 14).

3.3 Applying Relay Networks

Once the graph of relay networks Γ is trained, applying the polices is quite straightforward. For a given initial state $\mathbf{s}$, we select a node c whose $V(\mathbf{s})$ has the highest value among all nodes. We execute the current policy π_c until it reaches a state where the value of the parent node is greater

than its threshold ($V_{p(c)}(\mathbf{s}) > \bar{V}_{p(c)}$), where $p(c)$ indicates the index of the parent node of c. At that point, the parent control policy takes over and the process repeats until we reach the root policy. Alternatively, instead of always switching to the parent policy, we can switch to another policy whose $V(\mathbf{s})$ has the highest value.

3.4 Extending to Multiple Strategies

Optionally, our algorithm can be extended to discover multiple solutions to the original MDP. Take the problem of cartpole swing-up and balance as an example. In Algorithm 1, if we choose the mean of ρ_0 to be slightly off to the left from the balanced goal position, we will learn a chain of relay networks that often swings to the left and approaches the balanced position from the left. If we run Algorithm 1 again with the mean of ρ_0 leaning toward right, we will end up learning a different chain of polices that tends to swing to the right. For a problem with multi-modal solutions, we extend Algorithm 1 to solve for a directed graph with multiple chains and describe an automatic method to merge the current chain into an existing one to improve sample efficiency. Specifically, after the current node $\mathcal{N}_k$ is added to Γ and the next initial state distribution ρ_{k+1} is proposed (Line 12 in Algorithm 1), we compare ρ_{k+1} against the initial state distribution stored in every node on the existing chains (excluding the current chain). If there exists a node $\tilde{\mathcal{N}}$ with a similar initial state distribution, we merge the current chain into $\tilde{\mathcal{N}}$ by learning a policy (and a value function) that relays to $\mathcal{N}_k$ from the initial state distribution of $\tilde{\mathcal{N}}$, essentially adding an edge from $\tilde{\mathcal{N}}$ to $\mathcal{N}_k$ and terminating the current chain. Since now $\tilde{\mathcal{N}}$ has two parents, it can choose which of the two policies to execute based on the value function or by chance. Either path will lead to completion of the task, but will do so using different strategies. The details of the algorithm can be found in Appendix A.

4 Results

We evaluate our algorithm on motor skill control problems in simulated environments. We use DART physics engine [33] to create five learning environments similar to Cartpole, Hopper, 2D Walker, and Humanoid environments in Open-AI Gym [34]. To demonstrate the advantages of relay networks, our tasks are designed to be more difficult than those in Open-AI Gym. Implementation details can be found in the supplementary document. We compare our algorithm to three baselines:

- A single policy (ONE): ONE is a policy trained to maximize the objective function of the original task from the initial state distribution ρ. For fair comparison, we train ONE with the same number of samples used to train the entire relay networks graph. ONE also has the same number of neurons as the sum of neurons used by all relay policies.
- No relay (NR): NR validates the importance of relaying which amounts to the second term of the objective function in Equation 3 and the terminal condition, $V_k(\mathbf{s}) > \bar{V}_k$. NR removes these two treatments, but otherwise identical to relay networks. To ensure fairness, we use the same network architectures and the same amount of training samples as those used for relay networks. Due to the absence of the relay reward and the terminal condition $V_k(\mathbf{s} > \bar{V}_k$, each policy in NR attempts to achieve the original task on its own without relaying. During execution, we evaluate every value function at the current state and execute the policy that corresponds to the highest value.
- Curriculum learning (CL): We compare with curriculum learning which trains a single policy with increasingly more difficult curriculum. We use the initial state distributions computed by Algorithm 1 to define different curriculum. That is, we train a policy to solve a sequence of MDP's defined as $\{\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \rho_k, P\}$, $k \in [0, K]$, where K is the index of the last node on the chain. When training the next MDP, we use previously solved π and V to "warm-start" the learning.

4.1 Tasks

We will briefly describe each task in this section. Please see Appendix B in the supplementary document for detailed description of the state space, action space, reward function, termination conditions, and initial state distribution for each problem.

- **Cartpole:** Combining the classic cartpole balance problem with the pendulum swing-up problem, this example trains a cartpole to swing up and balance at the upright position. The mean of initial

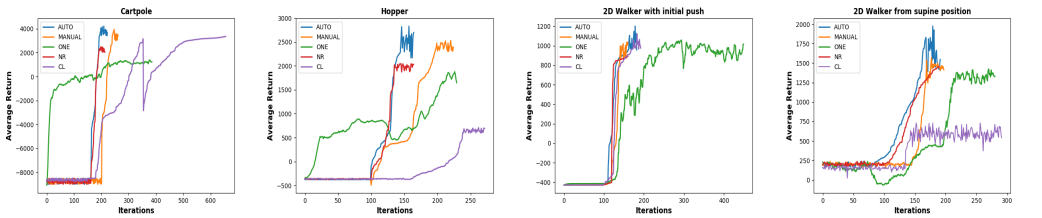


Figure 1: Testing curve comparisons.

state distribution, $\boldsymbol{\mu}_\rho$, is a state in which the pole points straight down and the cart is stationary. Our algorithm learns three relay policies to solve the problem.

- **Hopper:** This example trains a 2D one-legged hopper to get up from a supine position and hop forward as fast as possible. We use the same hopper model described in Open AI Gym. The main difference is that $\boldsymbol{\mu}_\rho$ is a state in which the hopper lies flat on the ground with zero velocity, making the problem more challenging than the one described in OpenAI Gym. Our algorithm learns three relay policies to solve the problem.
- **2D walker with initial push:** The goal of this example is to train a 2D walker to overcome an initial backward push and walk forward as fast as possible. We use the same 2D walker environment from Open AI Gym, but modify $\boldsymbol{\mu}_\rho$ to have a negative horizontal velocity. Our algorithm learns two relay policies to solve the problem.
- **2D walker from supine position:** We train the same 2D walker to get up from a supine position and walk as fast as it can. $\boldsymbol{\mu}_\rho$ is a state in which the walker lies flat on the ground with zero velocity. Our algorithm learns three relay policies to solve the problem.
- **Humanoid walks:** This example differs from the rest in that the subtasks are manually designed and the environment constraints are modified during training. We train a 3D humanoid to walk forward by first training the policy on the sagittal plane and then training in the full 3D space. As a result, the first policy is capable of walking forward while the second policy tries to brings the humanoid back to the sagittal plane when it starts to deviate in the lateral direction. For this example, we allow the policy to switch to non-parent node. This is necessary because while walking forward the humanoid deviates from the sagittal plane many times.

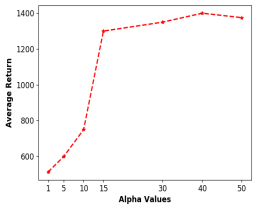
4.2 Baselines Comparisons

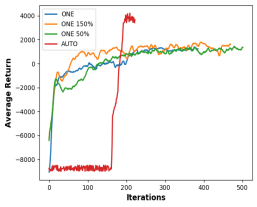
We compare two versions of our algorithm to the three baselines mentioned above. The first version (AUTO) is exactly the one described in Algorithm 1. The second version (MANUAL) requires the user to determine the subtasks and the initial state distributions associated with them. While AUTO presents a cleaner algorithm with less user intervention, MANUAL offers the flexibility to break down the problem in a specific way to incorporate domain knowledge about the problem.

Figure 1 shows the *testing curves* during task training. The learning curves are not informative for comparison because the initial state distributions and/or objective functions vary as the training progresses for AUTO, MANUAL, NR, and CL. The testing curves, on the other hand, always computes the average return on the original MDP. That is, the average objective value (Equation 1) of rollouts drawn from the original initial state distribution ρ. Figure 1 indicates that while both AUTO and MANUAL can reach higher returns than the baselines, AUTO is in general more sample efficient than MANUAL. Further, training the policy to relay to the "good states" is important as demonstrated by the comparison between AUTO and NR. The results of CL vary task by task, indicating that relying learning from a warm-started policy is not necessarily helpful.

4.3 Analyses

Relay reward: One important parameter in our algorithm is the weight for relay reward, i.e. α in Equation 3. Figure 2(a) shows that the policy performance is not sensitive to α as long as it is sufficiently large.





true return

predicted return	P	N	T
P	65	8	73
N	4	23	27
T	69	31	100

true return

predicted return	P	N	T
P	75	1	76
N	2	22	24
T	77	23	100

Figure 2: (a) The experiment with α. (b) Comparison of ONE with different numbers of neurons. (c) Confusion matrix of value function binary classifier (d) Confusion matrix after additional regression.

Accuracy of value function: Our algorithm relies on the value function to make accurate binary prediction. To evaluate the accuracy of the value function, we generate 100 rollouts using a learned policy and label them negative if they are terminated by the termination conditions $\mathcal{T}$. Otherwise, we label them positive. We then predict the rollouts positive if they satisfy the condition, $V(\mathbf{s}) > \bar{V}$. Otherwise, they are negative. Figure 2(c) shows the confusion matrix of the prediction. In practice, we run an additional regression on the value function after the policy training is completed to further improve the consistency between the value function and the final policy. This additional step can further improve the accuracy of the value function as a binary predictor (Figure 2(d)).

Sizes of neural networks: The size of the neural network for ONE affects the fairness of our baseline comparison. Choosing a network that is too small can restrict the ability of ONE to solve challenging problems. Choosing a network that is equal or greater in size than the sum of all relay polices might make training too difficult. We compare the performance of ONE with three different network sizes: N, $150\%N$, $50\%N$, where N is the number of neurons used for the entire relay networks graph. Figure 2(b) shows that the average return for these three polices are comparable. We thus choose N to be the one for our baseline.

4.4 Multiple chains

We test the extended algorithm for multiple chains (Appendix A) on two tasks: Cartpole and Hopper. For the first chain of the Cartpole example, the mean of initial state distribution for the first node is slightly off to the right from the balanced position (Figure 3 in Appendix A). The algorithm learns the task using three nodes. When training the second chain, we choose the mean of the initial state distribution for the first node to be slightly leaning to the left. The algorithm ends up learning a very different solution to swing up and balance. After learning two subtasks, the second chain identifies that the next subtask is very similar to the existing third subtask on the first chain and thus merges with the first chain. For Hopper, we make the mean of initial state distribution leaning slightly backward for the first chain, and slightly forward for the second chain. The algorithm creates three nodes for the first chain and merges the second chain into the first one after only one policy is learned on the second chain (Figure 3 in Appendix A).

To validate our method for automatic merging nodes, we also learn a graph that does not merge nodes (Γ'). The average return of taking the second chain after the junction in Γ is comparable to the average return of the second chain in Γ'. However, Γ requires less training samples than Γ', especially when the merge happens closer to the root. In the two examples we test, merging saves 4K training samples for Cartpole and 50K for Hopper.

5 DICUSSION

We propose a technique to learn a robust policy capable of controlling a wide range of state space by breaking down a complex task to simpler subtasks. Our algorithm has a few limitations. The value function is approximated based on the visited states during training. For a state that is far away from the visited states, the value function can be very inaccurate. Thus, the initial state distribution of the child node cannot be too far from the parent's initial state distribution. In addition, as mentioned in the introduction, the relay networks are built on locally optimal policies, resulting globally suboptimal solutions to the original task. The theoretical bounds of the optimality of relay networks can be an interesting future direction.

Acknowledgments

We want to thank the reviewers for their feedback. This work was supported by NSF award IIS-1514258, Samsung Global Research Outreach grant and AWS Cloud Credits for Research.

References

[1] OpenAI. *Open-source software for robot simulation, https://github.com/openai/roboschool*.

[2] M. Vukobratovic, A. Frank, and D. Juricic. On the stability of biped locomotion. *IEEE Transactions on Biomedical Engineering*, (1):25–36, 1970.

[3] J. Stückler, J. Schwenk, and S. Behnke. Getting back on two feet: Reliable standing-up routines for a humanoid robot. In *IAS*, pages 676–685, 2006.

[4] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Int. Res.*, 13(1):227–303, Nov. 2000. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622262.1622268.

[5] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, Jan. 2003. ISSN 0924-6703. doi:10.1023/A:1022140919877. URL https://doi.org/10.1023/A:1022140919877.

[6] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[10] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[11] C. Daniel, G. Neumann, and J. Peters. Hierarchical relative entropy policy search. In *Artificial Intelligence and Statistics*, pages 273–281, 2012.

[12] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems*, pages 3675–3683, 2016.

[13] N. Heess, G. Wayne, Y. Tassa, T. Lillicrap, M. Riedmiller, and D. Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

[14] X. B. Peng, G. Berseth, K. Yin, and M. van de Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*, 36(4), 2017.

[15] A. Bai, F. Wu, and X. Chen. Online planning for large mdps with maxq decomposition. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '12, pages 1215–1216, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9817381-3-3, 978-0-9817381-3-0. URL http://dl.acm.org/citation.cfm?id=2343896.2343928.

[16] K. Gräve and S. Behnke. Bayesian exploration and interactive demonstration in continuous state maxq-learning. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 3323–3330, 2014. doi:10.1109/ICRA.2014.6907337. URL https://doi.org/10.1109/ICRA.2014.6907337.

[17] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *The International Journal of Robotics Research*, 18(6):534–555, 1999.

[18] R. Tedrake. Lqr-trees: Feedback motion planning on sparse randomized trees. 2009.

[19] M. A. Borno, M. V. D. Panne, and E. Fiume. Domain of attraction expansion for physics-based character control. *ACM Transactions on Graphics (TOG)*, 36(2):17, 2017.

[20] G. Konidaris and A. G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in neural information processing systems*, pages 1015–1023, 2009.

[21] S. Coros, P. Beaudoin, and M. Van de Panne. Robust task-based control policies for physics-based characters. In *ACM Transactions on Graphics (TOG)*, volume 28, page 170. ACM, 2009.

[22] L. Liu and J. Hodgins. Learning to schedule control fragments for physics-based characters using deep q-learning. *ACM Transactions on Graphics (TOG)*, 36(3):29, 2017.

[23] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002.

[24] I. Popov, N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017.

[25] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel. Reverse curriculum generation for reinforcement learning. In S. Levine, V. Vanhoucke, and K. Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 482–495. PMLR, 13–15 Nov 2017. URL http://proceedings.mlr.press/v78/florensa17a.html.

[26] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[27] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[28] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[29] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[31] X. B. Peng, G. Berseth, and M. van de Panne. Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 35(4):81, 2016.

[32] V. C. Kumar, S. Ha, and C. K. Liu. Learning a unified control policy for safe falling. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[33] DART: Dynamic Animation and Robotics Toolkit. URL http://dartsim.github.io/.

[34] Openai gym. URL https://gym.openai.com/.

[35] DartEnv: Openai gym environments transferred to the DART simulator. URL http://github.com/DartEnv/dart-env/wiki/OpenAI-Gym-Environments.

[36] PyDART: A Python Binding of Dynamic Animation and Robotics Toolkit. URL http://pydart2.readthedocs.io.

[37] Openai baselines. URL https://github.com/openai/baselines.

Appendix A: Multiple Strategies Algorithm

This appendix provides additional information on finding multiple solutions to the original MDP (Section 3.4). Algorithm 3 is the pseudo-code of the described algorithm and Fig. 3 shows the directed graphs generated for the Cartpole and the Hopper tasks.

Algorithm 3: LearnMultiRelayNetworks

Input: MDP $\{\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \rho, P\}$
$\Gamma = \emptyset, \mathcal{D} = \emptyset$
Add root node, $\bar{\mathcal{N}} = \{\emptyset\}$, to Γ
for $i = 1 : nSolutions$ **do**
  Define a simpler initial state distribution ρ_0
  Define objective function J_0 according to Equation 1
  $[\pi_0, V_0] \leftarrow$ PolicySearch$(\mathcal{S}, \mathcal{A}, \rho_0, J_0, \mathcal{T})$
  $\bar{V}_0 \leftarrow$ ComputeThreshold$(\pi_0, V_0, \mathcal{T}, \rho_0, J_0)$
  Add node, $\mathcal{N}_0^i = \{\rho_0\}$, to Γ
  Add edge, $\mathcal{E} = \{\pi_0, V_0, \bar{V}_0, \}$, from $\mathcal{N}_0^i$ to $\bar{\mathcal{N}}$
  $k = 0$
  while π_k does not succeed from ρ **do**
    $\mathcal{T}_{k+1} = \mathcal{T} \cup (V_k(\mathbf{s}) > \bar{V}_k)$
    $\rho_{k+1} \leftarrow$ Compute new initial state distribution using Equation 2
    Define objective function J_{k+1} according to Equation 3
    if there exists $\tilde{\mathcal{N}} \in \mathcal{D}$ such that $D_{KL}(\tilde{\mathcal{N}}.\boldsymbol{\mu}, \boldsymbol{\mu}_{\rho_{k+1}}) < \epsilon$ **then**
      $[\pi_{k+1}, V_{k+1}] \leftarrow$ PolicySearch$(\mathcal{S}, \mathcal{A}, \tilde{\mathcal{N}}.\boldsymbol{\mu}, J_{k+1}, \mathcal{T}_{k+1})$
      $\bar{V}_{k+1} \leftarrow$ ComputeThreshold$(\pi_{k+1}, V_{k+1}, \mathcal{T}, \tilde{\mathcal{N}}.\boldsymbol{\mu}, J_{k+1})$
      Add edge $\mathcal{E} = \{\pi_{k+1}, V_{k+1}, \bar{V}_{k+1}\}$ from node $\tilde{\mathcal{N}}$ to $\mathcal{N}_k^i$
      break
    end if
    $[\pi_{k+1}, V_{k+1}] \leftarrow$ PolicySearch$(\mathcal{S}, \mathcal{A}, \rho_{k+1}, J_{k+1}, \mathcal{T}_{k+1})$
    $\bar{V}_{k+1} \leftarrow$ ComputeThreshold$(\pi_{k+1}, V_{k+1}, \mathcal{T}, \rho_{k+1}, J_{k+1})$
    Add node, $\mathcal{N}_{k+1}^i = \{\rho_{k+1}\}$, to Γ
    Add edge, $\mathcal{E} = \{\pi_{k+1}, V_{k+1}, \bar{V}_{k+1}\}$, from node $\mathcal{N}_{k+1}^i$ to $\mathcal{N}_k^i$
    $k \leftarrow k + 1$
  end while
  Add $[\mathcal{N}_0^i, \cdots, \mathcal{N}_k^i]$ to $\mathcal{D}$
end for
return Γ

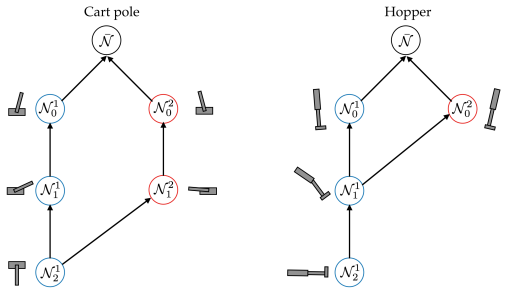


Figure 3: The generated directed graph for the Cartpole (Left) and the Hopper (Right) tasks. Each small image illustrates the mean of the initial state distribution of the corresponding node.

Appendix B: Task Descriptions

B.1 Cartpole

This example combines the classic cartpole balance problem with the pendulum swing-up problem. Our goal is to train a cartpole to be able to swing up and balance by applying only horizontal forces to the cart. The state space of the problem is $[\theta, \dot{\theta}, x, \dot{x}]$, where θ and $\dot{\theta}$ are the angle and velocity of the pendulum and x and $\dot{x}$ are the position and velocity of the the cart. The reward function of the problem is to keep the pole upright and the cart close to the origin:

$$r_i = cos(\theta) - x^2 - 0.01\|\boldsymbol{\tau}\|^2 + \kappa. \tag{4}$$

Here κ ($= 2.0$) is an alive bonus and $\boldsymbol{\tau}$ is the control cost. The termination conditions $\mathcal{T}$ in the MDP's before augmented by Algorithm 1 include the timeout condition at the end of rollout horizon T_{max}. For the root note, $\mathcal{T}_0$ also includes the condition that terminates the rollout if the pendulum falls below the horizontal plane.

B.2 Hopper

In this problem, our goal is to train a 2D one-legged hopper to get up from a supine position and hop forward as fast as possible. The hopper has a floating base with three unactuated DOFs and a single leg with hip, knee, and ankle joints. The 11D state vector consists of $[y, \theta, \mathbf{q}_{leg}, \dot{x}, \dot{y}, \dot{\theta}, \dot{\mathbf{q}}_{leg}]$, where x, y, and θ are the torso position and orientation in the global frame. The action is a vector of torques $\boldsymbol{\tau}$ for leg joints. Note that the policy does not dependent on the global horizontal position of the hopper. The reward function is defined as:

$$r_i = \dot{x} - 0.001\|\boldsymbol{\tau}\|^2 + \frac{1}{1+|\theta|} + \kappa. \tag{5}$$

Equation 5 encourages fast horizontal velocity, low torque usage and maintaining an upright orientation. The termination conditions for the root policy $\mathcal{T}_0$ include the timeout condition and the failing condition that terminates the hopper when it leans beyond a certain angle in either the forward or backward direction.

B.3 2D walker under initial push

This task involves a 2D walker moving forward as fast as possible without losing its balance. The 2D walker has two legs with three DOFs in each leg. Similar to the previous problem, the 17D state vector includes $[y, \theta, \mathbf{q}_{leg}, \dot{x}, \dot{y}, \dot{\theta}, \dot{\mathbf{q}}_{leg}]$ without the global horizontal position x. The action is the joint torques $\boldsymbol{\tau}$. The reward function encourages the positive horizontal velocity, penalizes excessive joint torques and also encourages maintaining an upright orientation. :

$$r_i = \dot{x} - 0.001\|\boldsymbol{\tau}\|^2 + \frac{1}{1+|\theta|} + \kappa. \tag{6}$$

While training the root policy, we terminate the rollout when the orientation of the 2D-walker exceeds a certain angle or if the timeout condition is met.

B.4 2D walker from supine position

This tasks uses the same 2D Walker robot with the same state space as described in the previous section. It also uses the same reward function described in Equation 6. The termination condition for the root policy $\mathcal{T}_0$ is met if either the rollout satisfies the timeout condition or if the robot exceeds a certain orientation.

B.5 Humanoid walks

The 3D Walker problem has the similar goal of moving forward without falling. The robot has a total of 21 DOFs with six unactuated DOFs for the floating base, three DOFs for the torso, and six DOFs for each leg. Therefore, the 41D state vector includes $[y, z, \mathbf{r}, \mathbf{q}_{torso}, \mathbf{q}_{leg}, \dot{x}, \dot{y}, \dot{z}, \dot{\mathbf{r}}, \dot{\mathbf{q}}_{torso}, \dot{\mathbf{q}}_{leg}]$

where x, y, z are the global position and $\mathbf{r}$ is the global orientation of the robot. The action is the joint torques $\boldsymbol{\tau}$. The reward function for this problem is:

$$r_i = \dot{x} - 0.001\|\boldsymbol{\tau}\|^2 \ - 0.2|c_y| + \kappa \text{ , where } i = \{0, 1\}, \tag{7}$$

where c_y is the deviation of center of mass. While training the root policy, we terminate the rollout when the orientation of the walker exceeds a certain angle or if the timeout condition is met.

Appendix C: Implementation Details

We use DartEnv [35], which is an Open-AI environment that uses PyDart [36] as a physics simulator. PyDart is an open source python binding for Dynamic Animation and Robotics Toolkit (DART) [33]. The time step of all the simulations are 0.002s. To train the policy, we use *baselines* [37] implementation of PPO. We provide the hyperparameters of neural networks in Figure 4.

TASK	AUTO	ONE	CL	NR
	{Neural Net Layer Architecture} (Number of Policies)			
Cartpole	{32,32}(3)	{128,64}(1)	{128,64}(1)	{32,32}(3)
Hopper	{64,64}(3)	{256,128}(1)	{256,128}(1)	{64,64}(3)
2D Walker initial push	{64,64}(2)	{128,128}(1)	{128,128}(1)	{64,64}(2)
2D Walker supine position	{64,64}(3)	{256,128}(1)	{256,128}(1)	{64,64}(3)
3D Walker	{64,64}(3)	{256,128}(1)	{256,128}(1)	{64,64}(3)

Figure 4: Different neural network architectures used in the training for each task.

Appendix D: Weighting Coefficient

As described in section 3.1, we generate new nodes in the relay by defining an optimization problem. The objective function of this is described in Equation 2. Just like many optimization problems, the weights of the objective terms are often determined empirically. We determine the value of w in Equation 2 based on the following procedure. For example, after the root policy π_0 is trained, we look at the magnitude of the value function $V_0(\mathbf{s})$ for a few successful states sampled near ρ_0. Then the value of w is set such that the magnitude of the two terms $V_0(\mathbf{s})$ and $\|\mathbf{s} - \boldsymbol{\mu}_\rho\|^2$ of the objective function are comparable. In our experiments, we found that training the relay networks is not sensitive to the value of w across sub-tasks. However, the values can be different for different tasks (e.g. hopping vs walking).