Expanding Motor Skills through Relay Neural Networks

Visak C.V. Kumar\(^1\), Sehoon Ha\(^2\), C. Karen Liu\(^1\)

**Abstract**—While the recent advances in deep reinforcement learning have achieved impressive results in learning motor skills, many of the trained policies are only capable within a limited set of initial states. We propose a technique to break down a complex robotic task to simpler subtasks and train them sequentially such that the robot can expand its existing skill set gradually. Our key idea is to build a tree of local control policies represented by neural networks, which we refer as relay neural networks. Starting from the root policy that attempts to achieve the task from a small set of initial states, each subsequent policy expands the set of successful initial states by driving the new states to existing “good” states. Our algorithm utilizes the value function of the policy to determine whether a state is “good” under each policy. We take advantage of many existing policy search algorithms that learn the value function simultaneously with the policy, such as those that use actor-critic representations or those that use the advantage function to reduce variance. We demonstrate that the relay networks can solve complex continuous control problems for underactuated dynamic systems.

I. INTRODUCTION

The recent advances in deep reinforcement learning (DRL) have motivated robotic researcher 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 execute each subtask sequentially. The control policy for each subtask can then be trained, sequentially as well, to expand existing skill set gradually. While an arbitrary decomposition of a task might lead to suboptimal control, 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 control policies to existing ones. Inspired by the LQR-Trees method proposed by Tedrake \([4]\), our key idea is to build a tree of local control policies represented by neural networks, which we refer as relay neural networks. Starting from the root 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 tree. Each policy is trained for a new set of initial states with a reward 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, such as TRPO-GAE \([5]\), DDPG \([6]\), or A3C \([7]\), provide an estimation of value function either through actor-critic representations or advantage function or baseline approximation. Our algorithm takes advantage of the learned value function to define the reward function of the new policy, as well as to select the policy on the tree during online execution.

We demonstrate that the relay networks can solve complex control problems for underactuated systems by gradually expanding the existing motor skill set. We show that a hopper first learns hopping forward from a near-upright position and then learns to get up from a horizontal position such that it can use the first policy to hop. We also show that a walker can first learn how to walk in 2D and then learn to walk in 3D by driving non-planar states to the successful planer states.

II. RELATED WORKS

One common approach to building robust controllers is to concatenate multiple controllers that work for a small range of initial conditions \([8]\). Tedrake \([4]\) 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. Konidaris et al. \([9]\) developed a chaining algorithm that breaks a difficult task into easier subtasks and trains an individual policy for each scenario. Glassman et al. \([10]\) proposed a method to estimate the region of attraction for aircraft controllers using the approximated dynamics. Building on Tedrake’s work \([4]\), Borno et al. \([11]\) demonstrated that the expansion of region of attraction can be performed for high-dimensional dynamic systems using random trees that provide denser coverage. There also exists a large body of work on scheduling existing controllers, such as controllers designed for taking simple steps \([12]\) or tracking short trajectories \([13]\). Given the current success in deep reinforcement learning, our work demonstrates that the idea of sequencing a set of local controllers can be realized by learning policies represented by the neural networks.

Our work builds on the idea that the value function of a policy can be used to provide information for connecting

---

1 Visak C.V. Kumar and C. Karen Liu is with the School of Interactive Computing, Georgia Institute of Technology, GA 30308. visak3@gatech.edu, karenliu@cc.gatech.edu

2 Sehoon Ha is with Disney Research, Pittsburgh, PA 15213. sehoon.ha@disneyresearch.com
Our approach to a complex robotic task is to break it down to simpler subtasks. We propose a model-free algorithm that trains the subtasks sequentially such that the robot can expand its existing skill set gradually. The input to our algorithm is a set of small RL problems that share the same state space \( S \). Each RL problem is described by a tuple, \([A, P_{\text{init}}, r, T]\), where \( A \) is the action space, \( P_{\text{init}} \) is the initial state distribution, \( r \) is the reward function, and \( T \) is the set of termination conditions. The dynamic model of each RL problem needs not to be the same, but it is irrelevant to our model-free training process. The RL problems are organized in a tree structure, \( \Gamma \), based on the difficulty of each task. The topology of the tree is a design choice which typically depends on the goal of the original task. However, as the rule of thumb, the RL problem in the parent node should be a prerequisite to that in the child node. As such, the root node should represent the simplest problem that can be easily trained to complete the task.

### A. Learning Relay Networks

Initially, each node in the tree \( \Gamma \) only contains the tuple that describes the RL problem. After learning the relay networks (Algorithm 1), the output of the algorithm is the same tree but each node is augmented with the policy \( \pi : S \times A \rightarrow [0, 1] \), the value function \( V : S \rightarrow \mathbb{R} \), and a threshold for the value function \( V \in \mathbb{R} \) (Section III-B). We represent a policy as a Gaussian distribution of action \( \pi \in A \) conditioned on a state \( s \in S \). 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.

The process of learning the relay networks is shown in Algorithm 1. Starting from the root node, we traverse the tree in the depth first search fashion. When a node is visited, we solve the corresponding RL problem using its tuple and the information passed from the parent node. Specifically, we first solve the RL problem at the root node. We define an objective function that accumulates the discounted reward along a trajectory of \( T \) time steps:

\[
J = \sum_{t=0}^{T} \gamma^{t} r_{0}(s_{t}, a_{t}),
\]

where \( r_{0} \) is the reward function for the root node and \( \gamma \) is the discount factor. This objective function defined, we can solve the RL problem using TRPO or DDPG to obtain the policy \( \pi_{0} \), which drives a small set of initial states to complete the original task, as well as the value function \( V_{0}(s) \), which evaluates the return by following \( \pi_{0} \) from state \( s \).

The next relay network aims to drive the rollouts toward the states where \( \pi_{0} \) can successfully handle, instead of solving from scratch a policy that directly completes the original task. To determine whether \( \pi_{0} \) can handle a particular state \( s \), one can generate a rollout by following \( \pi_{0} \) from \( s \) and calculate the return of the rollout. However, this approach can be too slow for online applications. Fortunately, either TRPO or DDPG produces a value function, which provides an approximated return from \( s \) without generating a rollout. Our goal is then to determine a threshold \( V_{0} \) for \( V_{0}(s) \) above which \( s \) is deemed “good”. The details on how to determine such a threshold are described in Section III-B.

Learning a relay network follows a similar procedure of learning the root network, with a few critical differences. Besides the accumulated reward, the objective function includes an additional reward that encourages “relaying” and the value function of the parent node evaluated at the final state \( s_{T} \):

\[
J = \sum_{t=0}^{T} \gamma^{t} r_{k}(s_{t}, a_{t}) + r_{\text{relay}}(s_{t}) + V_{p}(s_{T}),
\]

where \( r_{\text{relay}}(s) = \begin{cases} b, & V_{p}(s) > V_{p}(k) \\ 0, & \text{otherwise} \end{cases} \).

The first term of the objective function accumulates the reward defined by the reward function \( r_{k} \) of the current node \( k \). The second term, \( r_{\text{relay}} \), gives a large bonus \( b \) to the rollouts that successfully enter the subset of \( S \) where the policy associated with the parent node is capable of handling, i.e. the value function of the parent node returns a value above the threshold \( (V_{p}(k)(s) > V_{p}(k)) \). Note that we define a function \( p(k) \) to return the index of the parent of node \( k \). A rollout that receives such a bonus will terminate. The final term of the objective function recursively adds the accumulated reward from the root node, via relay networks, to the current node. Using this objective function, we can learn a policy \( \pi_{k} \) that drives a rollout toward successful states deemed by the parent value function, as well as a value function \( V_{k} \) that measures the long-term reward from the
Algorithm 1: Learn relay networks

1: **Input:** A tree of RL problems $\Gamma$
2: $\mathcal{L}_{DFS} \leftarrow$ Sort node indices in $\Gamma$ using DFS
3: Define cost function: $J = \sum_t \gamma^t r_t(s_t, a_t)$
4: $[\pi_0, V_0] \leftarrow$ PolicySearch($S, A, P^{init}, J, T_0$)
5: $V_0 \leftarrow$ ComputeThreshold($[\pi_0, V_0, T_0, P^{init}], J$)
6: Append $[\pi_0, V_0, T_0]$ to root node
7: for $k$ in $\mathcal{L}_{DFS}$ do
8: Define relay reward function:
   
   $$r_{relay}(s) = \begin{cases} V, & V_p(k)(s) > V_p(k) \\ 0, & \text{otherwise} \end{cases}$$

9: Define cost function:
   
   $$J = \sum_t \gamma^t r_k(s_t, a_t) + r_{relay}(s_t) + V_p(k)(s_T)$$

10: $T_k = T_k \cup (V_p(k)(s) > V_p(k))$
11: $[\pi_k, V_k] \leftarrow$ PolicySearch($S, A_k, P^{init}, J, T_k$)
12: $V_k \leftarrow$ ComputeThreshold($[\pi_k, V_k, T_k, P^{init}], J$)
13: Append $[\pi_k, V_k, T_k]$ to node $k$
14: end for
15: return $\Gamma$

Algorithm 2: Compute threshold for value function

1: **Input:** $\pi, V, T, P^{init}, J$
2: Initialize buffer $D$ for training data
3: $[s_1, \cdots, s_N] \leftarrow$ Sample states from $P^{init}$
4: $[\tau_1, \cdots, \tau_N] \leftarrow$ Generate rollouts by following $\pi$ and $T$ from sampled states;
5: Compute returns for rollouts: $R_i = J(\tau_i), \ i \in [1, N]$
6: Sort $R_i$ and determine a threshold $\bar{R}$
7: for $i = 1 : N$ do
8: if $R_i > \bar{R}$ then
9: Add $(V(s_i), 1)$ in $D$
10: else
11: Add $(V(s_i), 0)$ in $D$
12: end if
13: end for
14: $\bar{V} \leftarrow$ Classify($D$)
15: return $\bar{V}$

B. Computing Threshold for Value Function

In the ideal case, the learned value function $V(s)$ reflects the true return of the policy $\pi$. In practice, $V(s)$ provided by DDPG or TRPO is only an approximation of the true value function. We observe that the scores $V(s)$ 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 $V(s)$ as a binary classifier to separate “good” states from “bad” ones, but not as a reliable predictor of the true returns of the policy (See more analysis on the value function in Section IV-E.

Selecting a threshold that separates successful states from unsuccessful ones can be done by heuristically determining a return value (e.g. 90% of the highest possible return). However, applying this threshold to the value function $V(s)$ might end up separating different sets of states due to the differences between predicted returns by $V(s)$ and the true returns. Therefore, we obtain the threshold $\bar{V}$ for $V(s)$ by training a binary classifier using sampled data shown in Algorithm 2.

Given a policy $\pi$, an approximated value function $V$, termination conditions $T$, initial state distribution $P^{init}$, and the objective function of the RL problem $J$, we begin the training by sampling the initial states $[s_1, \cdots, s_N]$. By generating rollouts from these initial states using $\pi$, we can compute their true returns $[R_1, \cdots, R_N]$ of $\pi$. We then sort the return values and heuristically determine a threshold $\bar{R}$ by observing the rollouts or simply choosing it to be the 90% of the maximum return. Note that this threshold $\bar{R}$ is in the scale of the return $R_i$’s, which is accurate but expensive to compute.

Instead, our goal is to find the threshold $\bar{V}$ in the scale of the value function $V(s)$ by training a classifier. We generate the training set where each data point is a pair of the predicted value $V(s_i)$ and a binary classification label, “good” or “bad”, according to $\bar{R}$ (i.e. $R_i > \bar{R}$ means $s_i$ is good). We then train a binary classifier $\bar{V}$ using the simple expectation-maximization (EM) approach.

C. Applying Relay Networks

Once the tree of relay networks $\Gamma$ is trained, applying the policies is quite straightforward (Algorithm 3). For the given initial state $s$, we select a node $c$ whose $\bar{V}(s)$ has the highest value among all nodes. We execute the current policy $\pi_c$ until it reaches the maximal rollout length $T$ or a state where the value of the parent node is greater than its threshold $V_p(c)(s) > V_p(c)$. 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 $\bar{V}(s)$
has the highest value.

IV. EVALUATION

We evaluate our algorithm using four classic control tasks in simulation: Cartpole balance and swing-up, 2D Hopper hopping and getting-up, 2D Biped walking and 3D Biped walking. We use DartEnv [22], which is an Open-AI environment that uses PyDart [23] as a physics simulator. PyDart is an open source python binding for Dynamic Animation and Robotics Toolkit (DART) [24]. The time step of all the simulations are 0.002s. To train the policy, we use rllab implementation of TRPO-GAE.

We also present the comparison of our algorithm to the baseline policy, which is trained to minimize the objective function of the original task (Equation 4) from the initial state distribution covers by the entire relay networks (i.e. $\cup_i P_{\text{init}}^i$). For fair comparison, we train the baseline policy with the same number of samples used to train the relay networks.

A. 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 $\theta$ and $\dot{\theta}$ are the angle and velocity of the pendulum and $x$ and $\dot{x}$ are the position and velocity of the cart. We construct a relay network tree consisting of five nodes (Figure 1 Left) and divide the initial state space to five regions based on $\theta$ (Figure 1 Right). Note that the other dimensions of the state space ($\theta, x, \dot{x}$) cover the same range in all nodes. We only need to train $C_0$, $C_1$, and $C_2$ by exploiting the symmetry in the problem. The reward function of the problem is to keep the pole upright and the cart close to the origin:

$$r_i = \cos(\theta) - x^2, \text{ where } i = 0, \ldots, 4. \quad (3)$$

We also use certain termination conditions. While training $C_0$, we terminate the rollout when the pole falls below an angle threshold. For the relay policies, the rollout is terminated when the value function threshold condition is met or the rollout horizon length has been achieved. Note that all the nodes in the cartpole have the same reward function but each of the relay network has its own $r_{\text{relay}}$ term. Each policy is represented by a fully connected neural network with two hidden layers, composed of 32 and 32 hidden units respectively. We use a batch size of 10,000 for 500 training iterations.

Figure 3 shows the comparison of the performance between the relay networks and the baseline. We randomly sample 20 states from the combined initial state distribution, $\cup_{i=0}^{4} P_{\text{init}}^i$, and generate rollouts from these 20 states. The returns of the rollouts following the relay networks are shown in blue, while those following the baseline policy are shown in red. We also label the samples with different symbols based on which initial state distribution they are drawn from. The baseline policy also learns to swing up and balance but can only balance very briefly given the same amount of training samples used to train the relay networks.

B. 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 $[y, \theta, q_{\text{leg}}, \dot{x}, \dot{y}, \dot{\theta}, \dot{q}_{\text{leg}}]$, where $x, y$, and $\theta$ are the torso position and orientation in the global frame. The action is a vector of torques $\tau_{\text{leg}}$ for leg joints. Note that the policy does not depend on the global horizontal position of the hopper.

We construct a relay network tree that has two nodes, $C_0$ and $C_1$. The root policy $\pi_0$ is trained for initial states near the upright position, while its child policy $\pi_1$ is initialized to a lying down position. Both $C_0$ and $C_1$ have the same reward function defined as

$$r_i = \dot{x} - 0.001\|\tau\|^2, \text{ where } i = \{0, 1\}. \quad (4)$$

Equation 4 encourages fast horizontal velocity and low torque usage. While training $\pi_0$, we terminate the rollout if the hopper leans beyond a certain angle in either the forward or backward direction. For $\pi_1$, the termination condition is when the state is higher than the set threshold or the rollout length is reached. For this problem we use a neural network with two hidden layers, each of which has 32 hidden neurons. The batch size is 20,000 and total training iterations are 800. Figure 3 shows the comparison of the performance between the relay networks and the baseline policy. Bottom row of Figure 2 shows the hopper able to get up and continue hopping. When initialized to a lying down position, policy $\pi_1$ is active. When it reaches a state where $V_0(s) > \tilde{V}_0$, we switch to the root policy $\pi_0$ which controls the hopper to continue hopping forward.

C. 2D Walker

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 in Section IV-B, the 17D state vector includes $[y, \theta, q_{\text{leg}}, \dot{x}, \dot{y}, \dot{\theta}, \dot{q}_{\text{leg}}]$ without the global horizontal position $x$.

The relay network tree $\Gamma$ consists of two nodes, $C_0$ and $C_1$. We first learn a policy $\pi_0$ with less challenging initial states where the robot already has the positive velocity moving forward ($[2, 3]$ m/s). Subsequently, we learn the second policy
\( \pi_1 \) where the robot starts with a negative initial velocity \((-3, -2) \text{ m/s}\) as if it is being pushed backward. The reward function encourages the positive horizontal velocity and penalizes excessive joint torques:

\[
    r_i = \dot{x} - 0.001\|\tau\|^2, \quad i = \{0, 1\}. \tag{5}
\]

We use the orientation of the robot as a termination condition. During the training of \( \pi_1 \), the relay reward \( r_{\text{relay}} \) will encourage the robot to accelerate forward and build up the momentum to the point that \( \pi_0 \) can take over. Figure 3 shows the comparison of the performance between the relay networks and the baseline policy. We use a neural network with two hidden layers, each of which has 64 hidden neurons, and train it with a batch size of 50,000 for a total of 700 iterations. Top row of Figure 2 illustrates the sequence of motions where the 2D walker can recover from an initial negative velocity and continue walking.

\section*{D. 3D Walker}

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 \([y, z, r, q_{\text{torso}}, q_{\text{leg}}, \dot{x}, \dot{y}, \dot{z}, \dot{r}, \dot{q}_{\text{torso}}, \dot{q}_{\text{leg}}]\) where \(x, y, z\) are the global position and \(r\) is the global orientation of the robot. To tackle this high-dimensional problem, we again build a relay network tree \( C \) with two nodes \( C_0 \) and \( C_1 \). The RL problem in \( C_0 \) focuses on training a planar walking skill, where the lateral movement is constrained within \( \pm 10 \text{ cm} \) in Y-axis. This problem design simplifies the walking problem so that the robot can focus on learning to move forward without worrying about lateral balance. However, we do not completely limit the lateral movements so that the neighboring non-planar states will be visited during training, increasing the accuracy of \( V_0 \) for training the relay network later. The reward function for this problem is:

\[
    r_i = \dot{x} - 0.001\|\tau\|^2 - 0.2|c_y|, \quad i = \{0, 1\}. \tag{6}
\]

where \( c_y \) is the deviation of center of mass in the y-axis. Again, in this task as well, the termination conditions are dependent on the orientation of the robot, if the robot falls beyond a threshold to the side or in the forward direction, we terminate the rollout. Next, we train \( \pi_1 \) which aims to achieve the same reward function as \( r_{\text{0}} \) while connecting the non-planar states to the planar ones via \( r_{\text{relay}} \). During the training of \( \pi_1 \), the constraints on the lateral movement is removed. Figure 3 shows the comparison of the performance between the relay networks and the baseline policy.

We use a neural network with two hidden layers, each of which has 64 hidden neurons, and train it with a batch size of 50,000 for a total of 1,500 iterations.

\section*{E. Accuracy of the value function}

The value function approximated by TRPO-GAE is in general accurate around the visited states during training. It provides valuable information to distinguish successful states under the corresponding policy. Figure 4 shows the comparison between the true return and the return predicted by the value function from 50 randomly sampled states using the Cartpole example. The confusion matrix is shown in Table I. The columns of the confusion matrix represent the true returns. The positive column denotes rollouts whose true returns are higher than the threshold \( (R_{\text{t}} > \bar{R}) \). The positive row denotes the rollouts whose predicted returns are higher than the threshold of the trained value function \( (V_{\text{t}} > \bar{V}) \). Ideally, the classification by value function threshold \( \bar{V} \) should be identical to the classification using the threshold of true return \( \bar{R} \). In this example, out of 50 states, there are only 3 false positives.

| True Return | Positive | Negative | Total |
|-------------|----------|----------|-------|
| Prediction  | Positive | Negative | 23    |
| Negative    | 3        | 24       | 27    |
| Total       | 26       | 24       | 50    |

Fig. 2. **Top:** An initial velocity of \(-2\text{m/sec}\) is applied to the 2D biped, and the motion sequence illustrates the recovery to a state where it can continue to walk forward. **Bottom:** Hopper learns to get up and continue hopping.
We also label the samples with different symbols based on which initial state distribution they are drawn from. Although the values predicted by the value function are not the same as the true returns, they suggest that the value function is sufficiently accurate to serve as a binary classifier.

V. CONCLUSIONS AND DISCUSSION

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 builds a tree of relay networks to gradually expand the motor skills of the robot. The algorithm takes the advantage of many existing policy search algorithms that learn the value function simultaneously with the policy and uses the value function to determine the boundary of the state space each relay network can successfully handle. We demonstrate that our algorithm can solve complex continuous control problems for underactuated dynamic systems.

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. This limitation also increases the challenge in designing the topology of the relay network tree as dividing the initial state space can be unintuitive for certain tasks. With a poor division, during training a control policy might visit many states which are intended to be the initial states of the child node. This might result in a poor value function for the child node due to under exploration; most rollouts terminate very quickly because the initial states already have high value according to the parent’s value function. Finally, 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.

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, no. 1, pp. 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 IJAS, 2006, pp. 676–685.
[4] R. Tedrake, “Lqr-trees: Feedback motion planning on sparse randomized trees,” 2009.
[5] 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.
[6] 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.
[7] 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, 2016, pp. 1928–1937.
[8] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, “Sequential composition of dynamically dexterous robot behaviors,” The International Journal of Robotics Research, vol. 18, no. 6, pp. 534–555, 1999.
[9] G. Konidaris and A. G. Barto, “Skill discovery in continuous reinforcement learning domains using skill chaining,” in Advances in neural information processing systems, 2009, pp. 1015–1023.
[10] F. Glassman, A. L. Desbiens, M. Tobenkin, M. Cutkosky, and R. Tedrake, “Region of attraction estimation for a perching aircraft: A lyapunov method exploiting barrier certificates,” in Robotics and Automation (ICRA), 2012 IEEE International Conference on. IEEE, 2012, pp. 2235–2242.
[11] M. A. Borno, M. V. D. Panne, and E. Fiume, “Domain of attraction expansion for physics-based character control,” ACM Transactions on Graphics (TOG), vol. 36, no. 2, p. 17, 2017.
[12] S. Coros, P. Beaudoin, and M. Van de Panne, “Robust task-based control policies for physics-based characters,” in ACM Transactions on Graphics (TOG), vol. 28, no. 5. ACM, 2009, p. 170.
[13] L. Liu and J. Hodgins, “Learning to schedule control fragments for physics-based characters using deep q-learning,” ACM Transactions on Graphics (TOG), vol. 36, no. 3, p. 29, 2017.
[14] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronalike adaptive elements that can solve difficult learning control problems,” IEEE transactions on systems, man, and cybernetics, no. 5, pp. 834–846, 1983.
[15] X. B. Peng, G. Bersett, and M. van de Panne, “Terrain-adaptive locomotion skills using deep reinforcement learning,” ACM Transactions on Graphics (TOG), vol. 35, no. 4, p. 81, 2016.
[16] X. B. Peng, G. Bersett, K. Yin, and M. van de Panne, “Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning,” ACM Transactions on Graphics (Proc. SIGGRAPH 2017), vol. 36, no. 4, 2017.
[17] 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.

[18] V. R. Konda and J. N. Tsitsiklis, “Actor-critic algorithms,” in *Advances in neural information processing systems*, 2000, pp. 1008–1014.

[19] P. Wawrzyński, “Real-time reinforcement learning by sequential actor–critics and experience replay,” *Neural Networks*, vol. 22, no. 10, pp. 1484–1497, 2009.

[20] R. Hafner and M. Riedmiller, “Reinforcement learning in feedback control,” *Machine learning*, vol. 84, no. 1, pp. 137–169, 2011.

[21] N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa, “Learning continuous control policies by stochastic value gradients,” in *Advances in Neural Information Processing Systems*, 2015, pp. 2944–2952.

[22] “DartEnv: Openai gym environments transferred to the DART simulator.” [Online]. Available: [http://github.com/DartEnv/dart-env/wiki/OpenAI-Gym-Environments](http://github.com/DartEnv/dart-env/wiki/OpenAI-Gym-Environments)

[23] “PyDART: A Python Binding of Dynamic Animation and Robotics Toolkit.” [Online]. Available: [http://pydart2.readthedocs.io](http://pydart2.readthedocs.io)

[24] “DART: Dynamic Animation and Robotics Toolkit.” [Online]. Available: [http://dartsim.github.io](http://dartsim.github.io)