Hierarchical Learning for Adaptive Locomotion Policies

Matt Clark
mcclark@andrew.cmu.edu

Kazuya Otani
kotani@andrew.cmu.edu

Sean Tao
shtao@andrew.cmu.edu

Abstract

In this project, we investigate the use of hierarchical learning for quickly adapting locomotion policies to new environments. The goal was to train a diverse set of locomotion behaviors in a flat obstacle-free environment, then train a high-level policy to navigate an obstacle-filled environment by choosing between low-level policies. Our method uses Deep Deterministic Policy Gradients for low-level policies and Deep Q-Networks for the high-level policy. We demonstrate that our algorithm can sequence appropriate actions to achieve better results with less training episodes than an agent trained from scratch on the specific environment.

1 Introduction

Deep reinforcement learning has been a hot topic recently, especially for tasks in which standard models for control have not been established for various reasons, such as robots in environments too complicated to model. Therefore, aritificial neural networks have been used to approximate various functions, such as in Deep-Q learning or Deep Deterministic Policy Gradient methods. However, due to the complexity of some tasks, training typically takes a substantial amount of time and compute resources, and moreover, due to the random nature of exploration, negative outcomes are common. This makes such methods rather impractical to implement in the real world. Moreover, interpretability of such networks is a large issue. While artificial neural networks are efficient at modeling almost any function, it is hard to interpret exactly how or why decisions are being made. Therefore, learning more complicated tasks can require exponentially more resources and effort, thus resulting in scalability issues.

In particular, while attempts to train tasks for Mujoco environments in OpenAI gym for simple actions, such as running, have produced terrific results, more complicated actions, such as running over a rough terrain with obstacles are less prevalent. The most logical approach to this, then, is to split the larger task into subtasks and use the smaller results to help learn the end goal. This hierarchical approach to learning difficult tasks is the one we eventually adopted. This method has a few benefits. First and foremost, because we have already divided the complicated tasks into smaller sections which we know are needed to solve the end goal, the agent will not have to discover how to accomplish in the required order, speeding up training exponentially. Specifically, if action 2 is only useful after action 1 is taken, then any learning of action 2 before action 1 is completed by the policy is wasted. Second, because different tasks often have repeated substructures, hierarchical learning can save a lot of work when many tasks are required to be learned. For instance, if an agent needs to learn how to walk on sand and how to walk on snow, the general motion of moving one leg up is similar in both tasks. While directly transferring the neural networks does not work well since, as mentioned before, weights are not interpretable, reuse of smaller tasks should make learning easier.

Our plan was as follows: we first trained a baseline agent in Mujoco's HalfCheetah environment to navigate through a terrain with obstacles via direct application of reinforcement learning algorithms. We then broke down this task into two subtasks—running over open terrain and clearing obstacles—and trained networks to do each individually. Afterwards, we constructed a controller to select between the two policies in a hierarchical manner, and compared the results we obtained.

2 Related Work

2.1 Policy learning

The optimal policy for an agent can be approximated using Q-learning where the Bellman equation is used to iteratively find the optimal action-value function of an MDP. In [3], Mnih, Volodymyr, et al. presented the Deep Q-Network (DQN) which demonstrated that Deep Neural Networks can be used as effective action-value function approximators for high dimensional state spaces by training agents to outperform humans in Atari games.

The naive approach to applying Q-learning to continuous action spaces is to discretize the action space which often results in a loss of valuable information and poor performance. In [10], Silver et al. showed that the Stochastic Policy Gradient can be estimated using the average product of the critic network w.r.t. to its' actions and the actor network w.r.t. it's parameters. Lillicrap, Timothy P., et al. [1] expanded on DPG by presenting an actor-critic model-free algorithm using deep neural networks to approximate the target policy. This Deep Deterministic Policy Gradient (DDPG) network is used in this work to train the low-level running and jumping policies of our agent.

2.2 Hierarchical Reinforcement Learning

Our work is most similar to the work by Peng et al [4], in which low-level policies that achieve stepping targets are learned, and a higher-level perceptual policy is trained to provide stepping targets for the policy. One difference with our method is that instead of learning stepping targets for a single locomotion policy, our high-level policy learns to choose from a set of locomotion policies to execute for N steps. We were also inspired by [5], in which a set of motor primitives are learned and executed over a long time horizon by a task-specific policy.

3 Methods

3.1 Environments

To run our experiments, we extended the standard HalfCheetah environment in OpenAI Gym. For training low-level policies, we adjusted the reward function to encode desired behavior for each policy. For training high-level policies, we inserted some block obstacles along the path of the agent, to force the agent to traverse over them. In these environments, we took inspiration from [6] and added a scalar value to the observation space that includes the distance from the agent's center-of-mass to the closest obstacle.

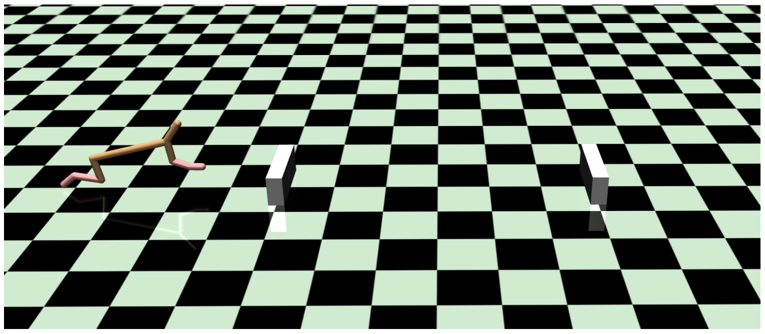

Figure 1: An instance of our obstacle environment.

3.2 Policy learning

The HalfCheetah environment has a continuous action space, which means that methods designed for discrete action spaces have to be tweaked to work. We tried two popular actor-critic policy gradient methods. For both algorithms, we extended open-source Pytorch implementations from [8].

We first tried the Advantage Actor-Critic (A2C) algorithm [9], in which we replaced the standard discrete action space with a network which output μ, σ values, describing the probability distribution of the action. The probability distribution was assumed to be Gaussian, and an action was sampled from the designated distribution at each timestep in training (for exploration), according to the mean and a standard deviation of μ and σ, respectively. At test time, the agent simply used the μ value as the action.

We then tried the Deterministic Policy Gradient algorithm. The DDPG network only outputs a μ value (hence the "deterministic" in the name), and exploration was added by adding noise to the policy. This can be done in two ways: adding noise to the action, or adding noise (random, or OU noise) to the policy network parameters. The former is the traditional method for exploration, but recent work has shown that the latter may lead to better results [2].

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    end for
end for
```

In addition to the standard running policy, we trained a set of jumping policies through reward shaping. We trained two different methods of jumping/hopping locomotion by rewarding the agent for vertical position and velocity, while penalizing it for excessive tilt to avoid unnatural behaviors.

3.3 Hierarchical learning

Once we trained a set of motor primitives, we set up a Deep Q-network for choosing which locomotion policy to execute. The DQN's observation space was a concatenation of the standard robot state and a scalar value indicating the distance from the robot's center-of-mass to the closest obstacle. The action space of the DQN was the set of low-level policies. Each "step" for the high-level policy was comprised of N low-level steps; we set $N = 10$ for our experiments. By treating the high-level policy choosing problem as its own Markov Decision Process, the DQN is able to learn the appropriate policies to use in different situations.

4 Results

4.1 Locomotion policy learning

Despite a significant amount of hyperparameter tuning, we found that A2C usually converged to a gait that was unnatural and involved either flipping onto its back or wiggling forward on its head.

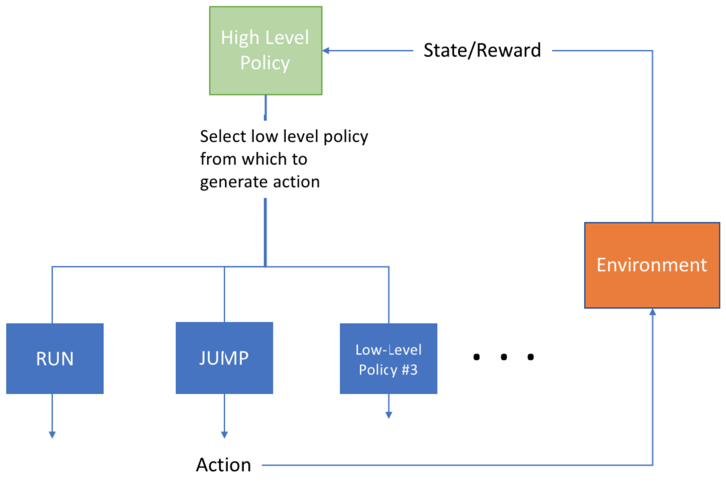

Figure 2: Hierarchical learning structure

The original DDPG implementation we used added OU noise to the actions; we found that this resulted in an unnatural "headstand wiggle" gait, although it still performed better than the A2C policy. We modified the architecture to use layer-norm and parameter noise instead, to match the OpenAI Baselines implementation. We found that this resulted in a robust high-performing running policy. These findings are in line with the reported results in [2] and [8].

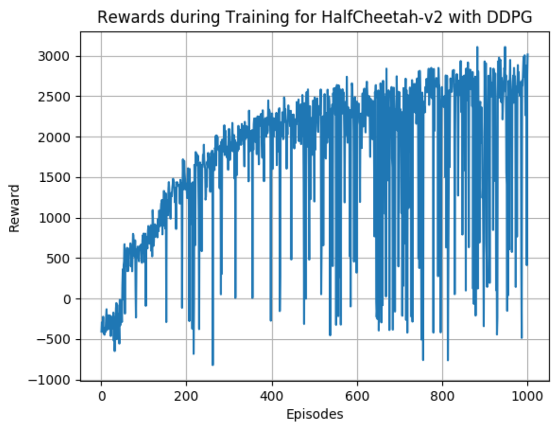

Figure 3: Training plot for HalfCheetah running policy with DDPG

To train a jumping behavior, we used the following reward function:

$$r(x, u) = c_1 v_x + c_2 x_z^2 + c_3 v_z - c_4\, abs(\theta) - c_5 |u|^2$$

Where x_z, v_x, v_z, θ, u are vertical position, horizontal velocity, vertical velocity, torso orientation, and joint torque commands, respectively. We generated two different jumping behaviors by varying the weighting terms c_i. The resulting policies are illustrated below.

We then trained a DQN meta-controller to choose between the set of low-level locomotion policies, and compared its performance to that of a single policy network trained from scratch on the environment.

Our main observation was that because our DQN meta-controller has access to more robust locomotion policies from the start, it is less likely to get stuck in locally optimal policies.

For example, we found that our baseline policy often converged to a policy which favored "scooting" over obstacles on its back, then wiggling forward in between obstacles. This worked well for the specific setup of the environment that the policy was trained on, but it did not generalize well to environments in which the obstacles were placed in different locations. It also does not move as fast in between obstacles. On the other hand, our algorithm quickly found a policy that only used the jumping policy when it got close to obstacles, and ran quickly in between. This allowed it to gain momentum and hop entirely over obstacles, even though the jumping policy alone would not have generated enough horizontal velocity to clear the entire obstacle. These differences are reflected in the average rewards, shown in Table 1.

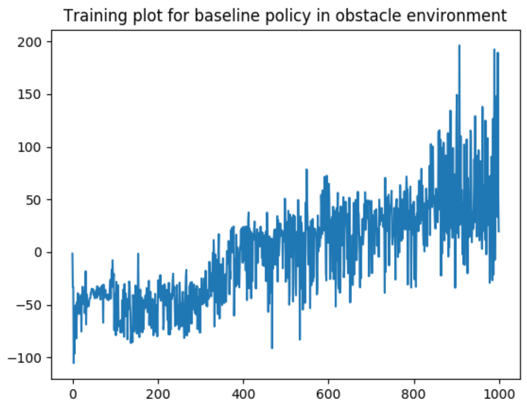

Figure 4: Training plot for our baseline policy, which was trained from scratch in the obstacle environment

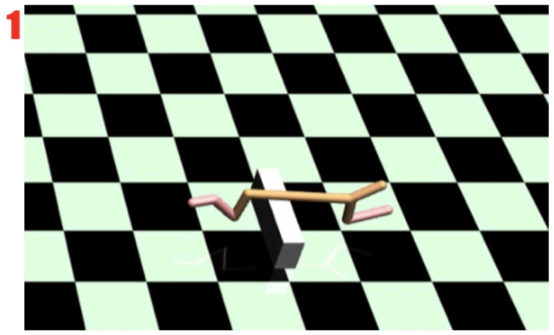

Figure 5: For some episodes, the hierarchical policy would get stuck on top of an obstacle and struggle to get off. This explains the high standard deviation in the rewards.

5 Conclusions

In this project, we showed the benefits of pre-training a diverse set of locomotion policies, then training a high-level controller to use these policies to quickly adapt to new environments. This hierarchical control scheme resulted in more robust locomotion over a wider range of environments.

It's important to note that while our method succeeded in solving the obstacle environment that we constructed, there was some domain knowledge (e.g. jumping policy for getting over obstacles) encoded in our choice of primitive policies. To generalize this method to a larger set of environments, it will be necessary to either train a much more diverse set of locomotion policies, or leave room for low-level policies to be fine-tuned to the environment while the high-level policy is being trained.

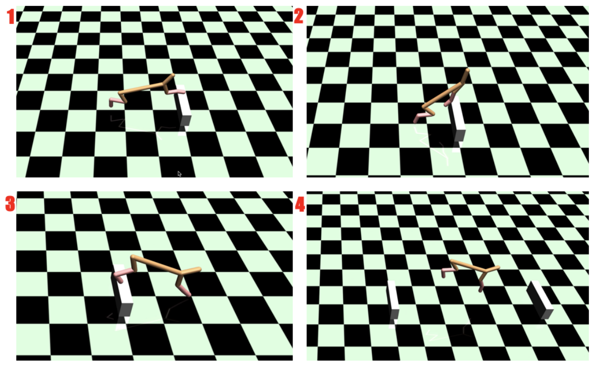

Figure 6: Example of successful traversal behavior for our hierarchical policy.

Table 1: Summary of performance for the baseline (training an agent from scratch) and our hierarchical method (500 episodes).

	Average reward	Std reward
Baseline	954.3	327.6
Hierarchical (ours)	3602.7	2627.5

Table 2: Generalization performance of the baseline and hierarchical policies in an environment with different obstacle locations from the one that they were trained on (500 episodes).

	Average reward	Std reward
Baseline	0.4	1.1
Hierarchical (ours)	795.7	765.4

References

[1] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).

[2] Plappert, Matthias, et al. "Parameter space noise for exploration." arXiv preprint arXiv:1706.01905 (2017).

[3] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[4] Peng, Xue Bin, et al. "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning." ACM Transactions on Graphics (TOG) 36.4 (2017): 41.

[5] Frans, Kevin, et al. "Meta learning shared hierarchies." arXiv preprint arXiv:1710.09767 (2017).

[6] Heess, Nicolas, et al. "Emergence of locomotion behaviours in rich environments." arXiv preprint arXiv:1707.02286 (2017).

[7] Henderson, Peter, et al. "Deep reinforcement learning that matters." arXiv preprint arXiv:1709.06560 (2017).

[8] Kostrikov, Ilya. "PyTorch Implementations of Reinforcement Learning Algorithms" https://github.com/ikostrikov

[9] Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International Conference on Machine Learning. 2016.

[10] Silver, Lever, et al. "Deterministic policy gradient algorithms." International Conference on Machine Learning. (2014)

[11] Sutton, R. and Barto, A. (1998). Reinforcement Learning: an Introduction. MIT Press.

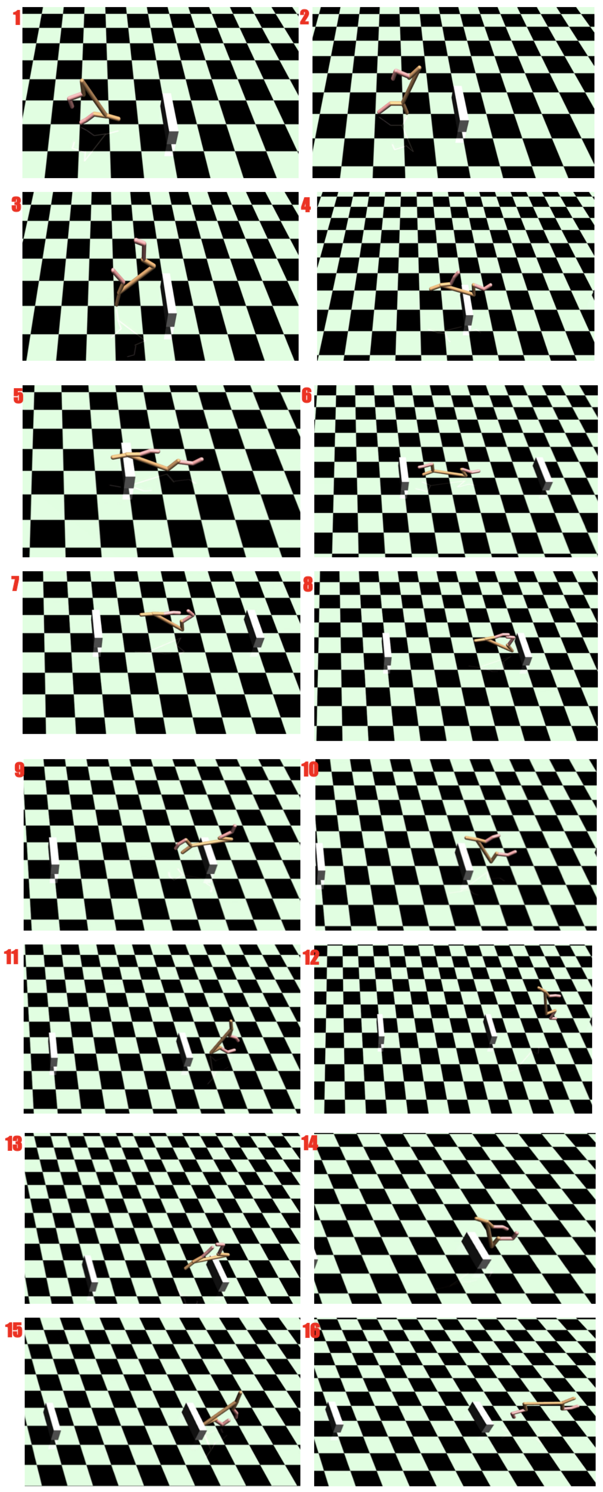

Figure 7: Example of behavior for the baseline policy, rolling over the obstacles on its back