Learning to Walk Using Hierarchical Reinforcement Learning

Tanmaya Dabral (tdabral) [1] **Karthika Nair (knair)** [1] **Abhishek Mohta (amohta)** [1]

Abstract

We explore a hierarchy based meta learning solution to tackle complicated navigation environments, which traditionally trained flat policies find difficult to solve. By decomposing the task down to smaller components which are first learned independently by sub-policies and then using them in conjunction via a master, we show that we are able to successfully navigate to a given goal. Moreover, we present novel regularization techniques that ensure smooth transitions among the sub-policies. We test the proposed methods in various environments wherein a four-legged agent is rewarded for getting to a goal point and penalized for instability. We show that hierarchical policies are able to solve these complicated environments while the flat policies fail. We further show that our proposed regularization techniques produce significantly better performing hierarchical systems.

1. Introduction

While the field of reinforcement learning has made significant advances in the past decade with respect to learning novel tasks, it still suffers from the major setback of having to learn these tasks from scratch. In contrast, humans use prior knowledge gained from other tasks to aid learn the task at hand. The field of meta learning aims to bridge this gap by training the agent on a distribution of related tasks, to then facilitate effortless learning of new tasks from the same distribution.

While there exist various techniques to apply the meta-learning paradigm to RL (Wang et al., 2016)(Gupta et al., 2018), the hierarchical approach (Frans et al., 2017) propose is the most intuitive and interpretable one. In hierarchy-based meta-learning, multiple policies are trained to perform different, simple tasks. On encountering a new related but possibly complicated task, a master policy, which governs which policy to activate given an observation, is trained on top of these sub-policies. Since the sub-policies already contain significant knowledge about the simpler tasks, the master is easily able to learn to combine them. In this work, we employ this hierarchy-based meta-learning approach to move a four-legged agent to a goal. We first train four sub-policies that move the agent in one of the four cardinal directions each. We then train master policies on top of these sub-policies to solve various complicated environments. Our work primarily differs from the work in (Frans et al., 2017) in the following ways:

- **Significantly harder environments:** Apart from rewarding closeness to the target points, our environments also penalize the agent for using too much power and physical instability, which makes movement harder.
- **No joint resets:** In (Frans et al., 2017), the authors manually reset the joints of the robot periodically to prevent it from getting stuck or toppling over. We *do not* use any out-of-policy tweaks to help our agent, but are still able to solve the environments.
- **Regularizing for smooth transitions:** To cope with the previous points, we present novel regularization techniques that help in smooth transitions among the sub-policies, which prevents the agent from getting stuck.
- **Separate training of each sub-policy:** We separately train four sub-policy to learn to walk in four different directions.

In Section 2 we present some relevant work in the reinforcement learning domain, specifically hierarchical and meta-learning. This is followed by Section 3 where we describe the environments we worked with for training the master and the sub policies. In Section 4 we elaborate on the core hierarchical model we used to build our experiments. We describe the policy networks and training procedure for both the master and the slaves. Section 5 describes the baseline with which we compare our results. In section 6, we describe the novel regularization techniques we introduce. We elaborate on the experiments we conducted in section 7. Section 8 contains a thorough analysis of our experiments. Finally, we summarize our findings in section 9.

[1]Carnegie Mellon University, Pittsburgh, PA 15213, USA.

Probabilistic Graphical Models 10-708 Class Project.

2. Literature Review

Owing to the recent advances (Schulman et al., 2015a)(Schulman et al., 2017)(Lillicrap et al., 2015), policy gradient methods (Sutton et al., 2000) have become ubiquitous for continuous control reinforcement learning in the past decade. The development of Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017) was prompted by the need for policy gradient methods capable of taking the largest step to improve performance while placing a constraint on how close the new and old policies should be without having to explicitly solve for the constraint, as done in (Schulman et al., 2015a). The authors propose two flavours of the PPO algorithm - the PPO-Penalty approximately penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that its scaled appropriately. The PPO-Clip eliminates the KL-divergence term in the objective altogether, and instead relies on a clipping in the objective function to incentivize the new policy against going too far from the old policy.

The paradigm shift towards hierarchy in RL was first introduced in (Dayan & Hinton, 1993), which suggested the use of various degrees of temporal resolution by using top level managers, which assigns tasks to workers, which are in turn, responsible for satisfying these tasks. Feudal RL employs two core principles - information hiding, which suggests that managers only need to know the system state at their own level of hierarchy, and reward hiding, which dictates that a manager must reward a sub-manager when it completes its task irrespective of whether it satisfies the commands of the super-managers. This leads to an abstraction in which the managers are unaware of how the workers satisfy the task assigned to them. However, the feudal Q-learning algorithm was tailored to the grid maze problem and didn't converge to any well-defined optimal policy.

(Sutton et al., 1999) proposed an alternative to model continuous-time discrete-event systems by building on the theory of Semi-Markov Decision Processes (SMDP), which take into account the amount of time passed between decision time instants. In this setting, they define an option as a triple consisting of the initiation set, the option's policy, and a termination condition. The framework consists of two levels - the bottom sub-policy level, which outputs actions and a top-policy level over options, which outputs sub-policies. It was further optimized using intra-option learning, which after each primitive action, updated all the options that could have taken that action, and termination improvement, which interrupted the execution of an option o whenever there is another option o whose expected reward was greater. Unlike feudal learning, if the action space consists of both primitive actions and options, then an algorithm following the options framework is proven to converge to an optimal policy.

Yet another approach in hierarchical learning was proposed by (Dietterich, 1999), which argued for the decomposition of the value function of an MDP into combinations of value functions of smaller constituent MDPs where each sub-task is defined by a termination predicate, a set of actions and a pseudo reward. Formally, this was done by redefining the Q function to take three arguments, $Q(p, s, a) = V(s, a) + C(p, s, a)$ where V represents the classical definition of Q and C is the total reward expected from the performance of the parent-task, p, after taking the action a. This approach, called MAXQ learning, thus learns a recursively optimal context-free ie. each subtask is optimally solved without any reference to the context in which it was executed.

Further, in (Kulkarni et al., 2016), the authors use a DQN framework to implement a gradient-based option learner, in which a top-level value function learns a policy over intrinsic goals, and a lower-level function learns a policy over atomic actions to satisfy the given goals. They contend that this flexibility in goal specifications provides an efficient space for exploration in complicated environments.

While the methods described so far did incorporate hierarchy, they did not provide a framework to perform gradient descent in the policy space without additional feedback in terms of rewards. This was first proposed in (Bacon et al., 2016), in which the managers' output was trained with gradients coming directly from the worker without any additional rewards or subgoals. In contrast to HDQN where descriptions of the subgoals were given as inputs to the option learners, option-critic was lauded as conceptually general as it didn't require intrinsic motivation for learning the options.

In the Meta Learning paradigm, (Finn et al., 2017) introduced a model agnostic algorithm which trains a model on a variety of learning tasks, to generalize to new tasks using only a small sample of training data. This method can be viewed from a feature learning standpoint as building an internal representation that is broadly suitable for many tasks. The model trained via this method is thus shown to be easy to fine tune. In their work in the RL setting, the authors use policy gradient methods to estimate the gradient both for the model gradient updates and the meta-optimization.

(Wang et al., 2016) describe another approach for meta-reinforcement learning. They train a single RNN-based policy over a distribution of MDPs. The recurrent network allows the distribution over the actions at any given time-step to be a function of the entire history seen by the agent allowing the policy to adapt to any new, unseen MDP. We can also interpret this as considering the entire distribution over the MDPs to be a partially observable MDP and using a recurrent network to solve that.

Finally, the work on Meta Learning Shared Hierarchy

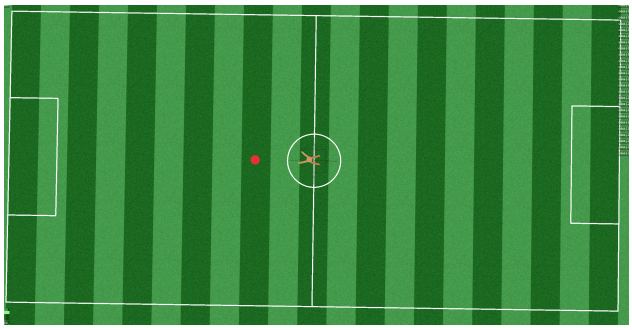

Figure 1. The AntWalker Environment; The red orb just denotes the direction. The real target is 10km away in that direction.

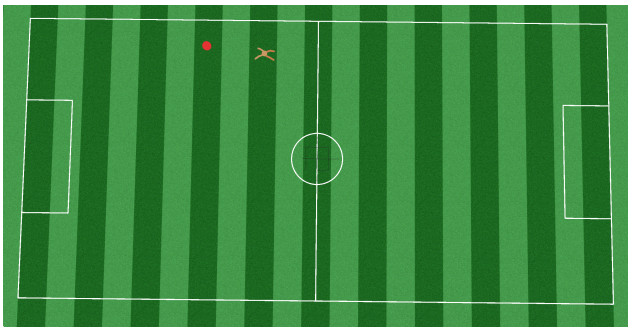

Figure 2. The AntBandit Environment

(MLSH) by (Frans et al., 2017) builds upon this framework by learning hierarchical policies consisting of sub-policies that are executed for a fixed number of time steps and a master policy whose action is to chose the sub policy to execute for the next N time steps. The authors train such hierarchies on a distribution over MDPs and are able to solve complicated tasks.

3. Environments

We train our hierarchical model in two phases: sub-policy training and master policy training. For the sub-policy training we define the AntWalker environment, and for the master policy training we define AntBandit and AntMaze, as explained below.

3.1. AntWalker

Four sub-policies are trained for each of the four cardinal directions. The details of the environment are as follows -

- **Agent**: The agent is a four-legged ant-like robot.
- **Target**: Point along the axis 10km away for the corresponding sub-policy in context, for example, for right, the target is (10000, 0).
- **Observation space**: The agent observes the relative position of the joints (16-dim), boolean information about which feet are in contact with the ground (4-dim), the relative change in height of the ant (1-dim), the velocities of the agent from the point of view of the agent (3-dim), the roll of the torso of the ant (1-dim), the pitch of the torso (1-dim) and the sin and cos of the roll, pitch and the yaw of the torso (6-dim), making the observation space 32 dimensional.
- **Action space**: The torque applied to the different joints of the ant (8-dim).
- **Reward**: The agent is rewarded for moving closer to the target point with the reward for a step proportional to the progress made in that particular step, and for staying alive. The agent is penalized for colliding feet, stuck joints and applying too much torque, incentivizing stability.

3.2. AntBandit

The AntBandit environment samples the goal point from four potential points, and the agent must learn to get to the chosen point.

- **Agent**: The agent is a four-legged ant-like robot.
- **Target**: The four potential target points are the four corners of a square. At each reset of the environment, one of these four points is randomly chosen as the target.
- **Observation space**: The observation space contains the 32-dimensional observation space of the AntWalker, and the current x-y coordinates of the ant (2-dim) and the x-y coordinates of the target (2-dim), making the observation 36 dimensional.
- **Action space**: From the point of view of the sub-policies or a flat policy, the action space is the same as the previous environment. Of course, it is different from the point of the view of the master policy.
- **Rewards**: As previously, the agent is rewarded for moving closer to the target point with the reward proportional to the progress per step in the potential field. We test this environment both with and without the stability penalties.

3.3. AntMaze

Apart from solving the AntBandit task, we try to solve a more complicated task using the learned sub-policies. The

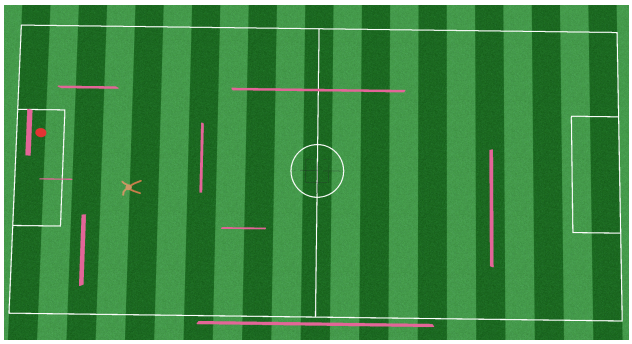

Figure 3. The AntMaze environment

ant is placed in a maze and it is trained to reach a pre-determined fixed point.

- **Agent**: The agent is a four-legged ant-like robot.
- **Target**: The target is a pre-determined fixed reachable point in the maze.
- **Observation space**: The observation space contains the 32-dimensional observation space of the AntWalker, and the current x-y coordinates of the ant (2-dim), making the observation space 34 dimensional.
- **Action space**: Same as AntBandit.
- **Rewards**: Same as AntBandit.

3.4. Implementation of the environments

Due to constraints on the use of Mujoco (Todorov et al., 2012) on the AWS, the above environments were all implemented from scratch in PyBullet (Erwin Coumans). The vanilla Ant in PyBullet is heavier than the one in Mujoco, encouraging it to typically have two or more legs on the ground, making the problem a bit more difficult. Also note that the walls in Figure 5 are actually connected, but a bug in PyBullet halves the size of the rendering of the "Box" objects under certain circumstances.

4. The Hierarchical Policies

We use a two-level hierarchy of policies to solve the AntBandit and the AntMaze environments. This master-slave architecture is presented in 4. The four slave policies move the ant in one of the four cardinal directions each. After every k time steps, the master policy decides which policy to run next, where k is a hyperparameter.

As mentioned previously, the slave policies are trained independently on the AntWalker environment. Therefore, the observation space that the slaves are trained on is different from that offered by the AntMaze and the AntBandit environments. However, note that the observation space in the AntWalker environment is a proper subset of that in the other two. Therefore, we can simply strip away the extra information when running the sub-policies. The master, however, looks at the entire observation space when making the decision on the sub-policies. Every time a sub-policy is chosen, the reward observed by the master is the sum of the rewards obtained throughout the k-step run of the chosen sub-policy.

The action distributions given a state are diagonal-Gaussians for all policies, with the mean and variance of the Gaussian being learned functions of the state. The learned function is modeled as a neural network with two $\tanh$-activated hidden layers with 64 units each. All policies are trained in an actor-critic framework using PPO, with the critic also modeled as a neural network with two $\tanh$-activated hidden layers with 64 units each.

5. The Flat Policies

For comparison, we also train non-hierarchical policies on the AntMaze and the AntBandit environments. The policies, unlike the slave policies in the hierarchical case, observe the entire observation space, to make them comparable to the hierarchical policies. Once again, the policies model actions as diagonal-Gaussians given a state, with the mean and variance of the Gaussian being learned functions of the state. Moreover, similar to the previous case, the policies are trained in an actor-critic framework with both the policy and the value functions modeled as neural networks with two $\tanh$-activated hidden layers with 64 units each.

6. Regularization Techniques

Once the four policies from the AntWalker environment are trained, using them together under a master can lead to some unseen initial states for the sub-policies, leading to the ant failing to stand stably. In our experiments, we found the agent consistently failing to transition to an orthogonal policy. In (Frans et al., 2017), the authors circumvent this issue by periodically resetting the state of the ant. To handle the issue without manual out-of-policy manipulations, we propose the following regularization techniques. All of these regularization techniques are employed while training the four sub-policies in the AntWalker environment.

- **Random Start Regularization** - Perform random actions for the first t timesteps to randomize the initial state before training, so the agent learns to recover from unseen states.
- **Random Repeated Regularization** - Perform random actions every t timesteps, so the randomization is inter-

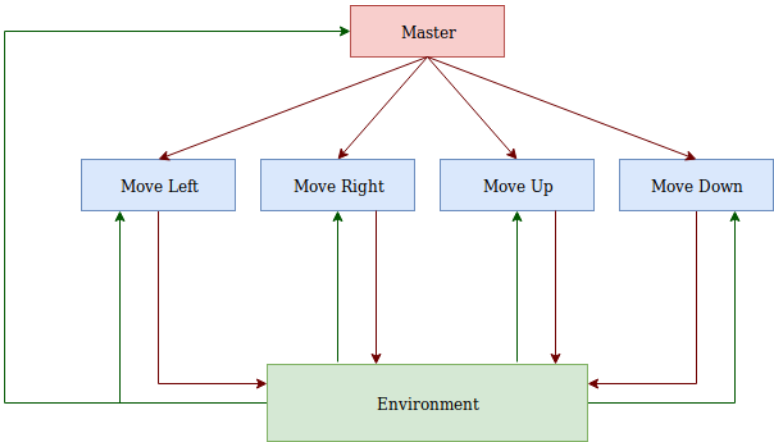

Figure 4. The two-level hierarchical model

spersed across the training iteration, helping the agent learn better recovery techniques.

- **Appropriate Bias Regularization** - Based on the prior knowledge that most of the transitions will occur between orthogonal directions, the appropriate bias can be added while training the model for the sub-policies. For the first t timesteps, the learnt policy for a randomly sampled orthogonal direction is run and then training is started for the direction in consideration. As handling transitions is our main goal, this method seems to work best, with the transitions taking place successfully almost always, once the model is trained.

In Section 8 we show the impact regularization creates in the hierarchical models we develop.

7. Experiments

We conduct the following experiments:

- We first train four sub-policies on the AntWalker environment with and without the aforementioned regularization techniques. The rewards obtained by the agent include penalties for applying too much torque and instability.
- We train a master policy on the AntMaze environment on top of the regularized and non-regularized sub-policies trained in the previous experiment.
- We train two flat policies on the AntMaze environment. One of the flat policies is trained with the stability penalties, while the other is trained without.
- We train a master policy on the AntBandit environment on top of the regularized and non-regularized sub-policies trained in the first experiment.

Hyperparameter	Value
Learning rate	3e-4
Discount factor	0.99
GAE λ	0.95
PPO clip ratio	0.2
Value loss coeff.	0.5
I	10
B	32

Table 1. The hyperparameters shared across all experiments

- We train two flat policies on the AntBandit environment. One of the flat policies is trained with the stability penalties, while the other is trained without.

All policies are trained using the Proximal Policy Optimization algorithm with a clipped surrogate objective(Schulman et al., 2017) with Generalized Advantage Estimation (Schulman et al., 2015b) under an actor-critic framework. Both the networks are trained using the Adam optimizer(Kingma & Ba, 2014). The training schedule is characterized by three integers S, I, and B:

1. Take S steps in the environment, resetting when necessary and store the experience in a new storage.
2. Create B batches from the experience.
3. Iterate over the B batches I times and update the actor and the critic by minimizing a weighted sum of the PPO surrogate loss and the value loss.
4. Repeat

For all the experiments, the hyperparameters in Table 1 were the same. The hyperparameter S was different for

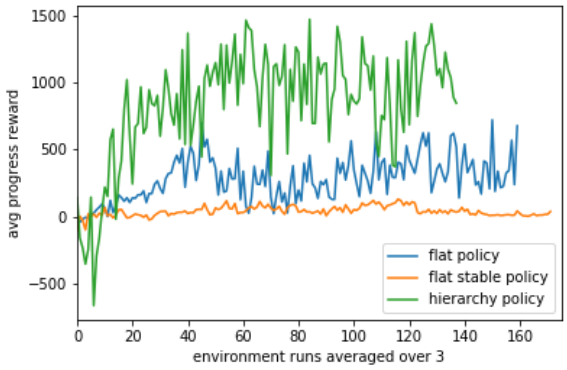

Figure 5. Reward plot for the AntMaze environment

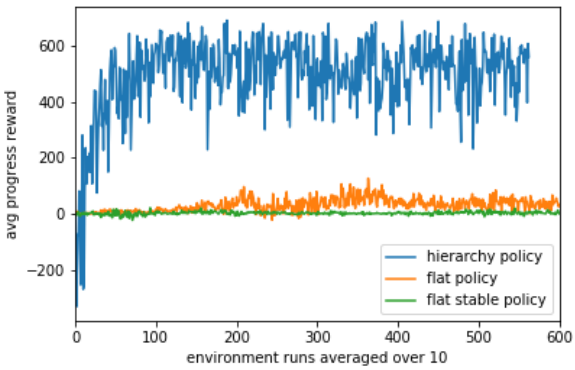

Figure 6. Reward plot for the AntBandit environment

different experiments, since a single step from the point-of-view of the master in case of a hierarchical policy is equivalent to many steps from the point-of-view of the flat policy. The S values are given in Table 2. The results on

Experiment	S
Sub-policies	2048
AntMaze HRL	128
AntBandit HRL	40
AntMaze Flat	2048
AntBandit Flat	2048

Table 2. The number of environment steps between epochs for each experiment. Note that for the hierarchical policies, the steps are from the POV of the master.

the three environments are shown in Figures 6 to 10. To get a cleaner plot, we average the reward over multiple consecutive environment runs.

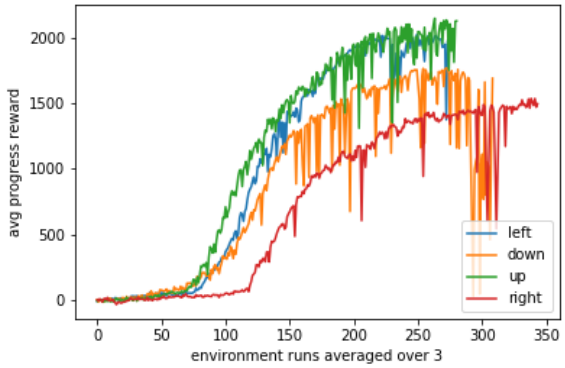

Figure 7. Unregularized sub-policies training (AntWalker)

8. Analysis

The graphs in Figures 6 to 10 provide some interesting insights. Note that for the AntMaze and the AntBandit environments, even in the cases where stability penalties are involved, only the reward corresponding to the distance travelled is graphed. Also note that since the sub-policies for a hierarchical system are already trained with stability penalties, adding those penalties while training the master does not make a difference.

For both the AntBandit (Figure 6) and the AntMaze (Figure 5) environments, we find that the regularized hierarchical policy outperforms the flat policies by a significant margin. As noted in Section 3, the observation space in both the environments is enough for them to be solved by an ideal reinforcement learning algorithm. Therefore, this is a fair

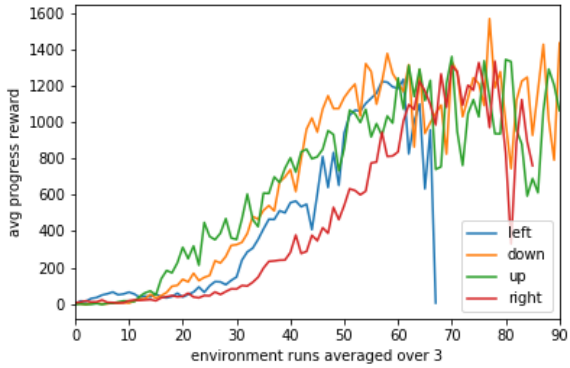

Figure 8. Regularized sub-policies training (AntWalker)

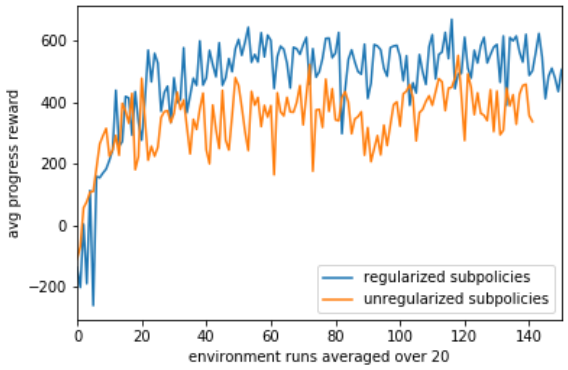

Figure 9. Reward comparisons for HRL with regularized vs unregularized sub-policies for AntBandit

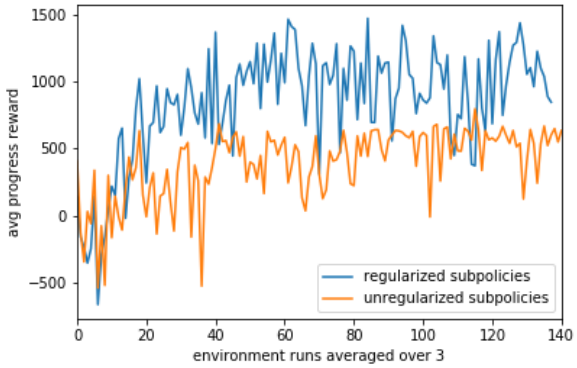

Figure 10. Reward comparisons for HRL with regularized vs unregularized sub-policies for AntMaze

comparison. Again, this is expected, as the search space for a master policy is much smaller than the search space for a flat policy, as a single action of the master changes the state of the environment by a large amount in a potentially useful way. Another interesting thing to note is that the flat policies without the stability penalties perform better than the ones with the penalties. This is again expected as in the environment without the penalties, the robot only focuses on optimizing the reward obtained by moving closer to the target.

Comparing the unregularized and regularized sub-policy training graphs, it is apparent that the unregularized sub-policies show a considerably smaller variance. This is expected as the regularization procedure described in Section 6 involves taking random steps at the beginning or in the middle of a run, thereby increasing the overall stochasticity. The unregularized policies do achieve higher rewards eventually, but they do not allow smooth transitions among the policies.

Our hypothesis is validated when we compare the rewards reaped by the unregularized and the regularized sub-policies when a master is trained over them (Figure 9 and Figure 10). The regularized sub-policies serve as much better slaves. Furthermore, the difference is much more pronounced in the AntMaze environment as it is significantly more complicated and requires multiple transitions among the policies, while the AntBandit requires only one in the ideal case.

9. Conclusion

We have presented an attempt to solve complicated environments using a hierarchy of policies. Even though the environments created are MDPs, traditional flat policies find it difficult to solve them and get stuck in local minima. For example, in the AntMaze environment, if the agent tries to move in a straight line toward the target, it will get stuck in a wall, which is what happens in the case of (non-penalized) flat policies. Setting up a hierarchy shrinks the space for the master, and allows it to explore much more efficiently, allowing it to avoid these local minima. Furthermore, we present novel regularization techniques that help in learning robust sub-policies in a hierarchical settings by training for smoother transitions among them. Our results show that these techniques induce significant gains in the performance.

References

Bacon, P., Harb, J., and Precup, D. The option-critic architecture. *CoRR*, abs/1609.05140, 2016. URL http://arxiv.org/abs/1609.05140.

Dayan, P. and Hinton, G. E. Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., and Giles, C. L. (eds.), *Advances in Neural Information Processing Systems 5*, pp. 271–278. Morgan-Kaufmann, 1993. URL http://papers.nips.cc/paper/714-feudal-reinforcement-learning.pdf.

Dietterich, T. G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *CoRR*, cs.LG/9905014, 1999. URL http://arxiv.org/abs/cs.LG/9905014.

Erwin Coumans, Yunfei Bai, J. H. Pybullet. URL https://pypi.org/project/pybullet/.

Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017.

Frans, K., Ho, J., Chen, X., Abbeel, P., and Schulman, J. Meta learning shared hierarchies. *CoRR*, abs/1710.09767, 2017.

Gupta, A., Eysenbach, B., Finn, C., and Levine, S. Unsupervised meta-learning for reinforcement learning. *arXiv preprint arXiv:1806.04640*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. B. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057, 2016. URL http://arxiv.org/abs/1604.06057.

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

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. In *International Conference on Machine Learning*, pp. 1889–1897, 2015a.

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

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Sutton, R. S., Precup, D., and Singh, S. P. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211, 1999. URL http://webdocs.cs.ualberta.ca/~sutton/papers/SPS-aij.pdf.

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

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.