Improving DQN and TRPO with Hierarchical Meta-controllers

Scott Sun (scottsun) [1] **Aniruddh Shetty (anirudds)** [1] **Nishant Gurunath (ngurunat)** [1]
Raunaq Bhirangi (rbhirang) [1]

Abstract

Complex behaviors are comprised of multiple layers of decision-making. This is naturally at odds with conventional deep RL strategies that only operate at a single temporal scale. Hierarchical adaptations of DQNs and policy gradient methods seek to supplement flat policy agents with a meta-controller that incrementally guides the lower level agent over longer time scales. We compare our own implementations of hierarchical DQN and TRPO agents on a series of environments and explore the design of informative subgoals for these algorithms. We achieve a significant improvement over flat learners in a stochastic MDP and several MuJoCo environments. We also examine the difficulties of constructing suitable subgoals in the Atari environment.

1. Introduction

Deep reinforcement learning has taken some major strides in recent times, enabling autonomous agents to perform a range of tasks, including object manipulation with robotic arms and solving games like Breakout or Space Invaders. A number of these solutions are based on flat policies, which tend to be suitable only for simple tasks with periodic actions. The natural approach to solving larger problems is to decompose the problem into multiple smaller goals, which oftentimes results in easier and faster convergence for the overall system. One method of decomposing these problems is to utilize high level and low level goals when solving a problem, a.k.a. hierarchical reinforcement learning (HRL). As explained in later sections, HRL algorithms have seen great success in solving complicated tasks that involve interplay between achieving an overarching goal (e.g., reaching some high score or location) and mechanical mastery (e.g., jumping over obstacles). Crafting a flat policy learner to tackle such problems is incredibly complicated, as it is difficult to convey the importance of both of these very disparate tasks. Inadequacy in either of these domains results in failure for the overall task.

HRL is promising for solving tasks that require the agent to reason at a higher level to set and achieve intermediate goals over time that require mastery of more primitive actions. While a number of methods have been proposed based on this philosophy, they are often designed in a task-specific manner due to the challenges associated with learning good goals. This makes them difficult to generalize. As a result, major goals of this project will be to explore different HRL algorithms and their approaches to goal-setting in order to compare and contrast their generalizability to a wider range of problems.

The main flavor of HRL this work is proposing is similar to the paradigm of (Vezhnevets et al., 2017) and (D. Kulkarni et al., 2016). Broadly speaking, this involves a higher-level component operating at a broad time scale to supervise a lower-level component operating in the short term. Different HRL algorithms will be tested on tasks of varying complexity and comparisons will be drawn between task-specific performance as well as generalizability.

2. Previous Work

Intelligent exploration continues to be a relatively difficult problem in the area of reinforcement learning, even for simple environments. A comparison between several variants of Proximal Policy Optimization demonstrated that the hierarchical variant, with policies that operate at different temporal scales, is shown to give the best exploration performance as the complexity of the environment increases (Al-Shedivat et al., 2018). This comparison involved a vanilla implementation, one with reward shaping, one with policy sketches, and a hierarchical implementation using a meta-controller to choose intermediate goals for the lower level controller. As a result of the advantages seen with HRL algorithms, it would be interesting to dig deeper and understand these results.

While some approaches to HRL rely upon manually setting goals to be learned, this limits generalizability. Motivated by (Nachum et al., 2018), our work aims to explore a com-

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

10708 Class Project, Spring, 2019.

parison of different goal-setting mechanisms, e.g., hard-coded vs. learned goals. In the paper, for example, the authors make the system task-agnostic by taking state observations in their raw form, without representing goals and rewarded observations within a learned embedded space.

As mentioned before, the HRL paradigm we would like to explore and build upon is the Feudal Network (Vezhnevets et al., 2017) built on the original feudal reinforcement learning framework (Dayan & Hinton, 1993). The core idea behind the Feudal Network is the use of a Manager and a Worker module, whereby the Manager operates on more abstract goals over a longer temporal scale and the Worker performs more primitive actions with fine temporal resolution. Essentially, the higher level components set goals for lower level components that must be achieved, but do not specify the means by which the lower level components must achieve the goal. An important aspect of this work is that the Worker is capable of learning subgoals as opposed to requiring the system designer hard-code them.

Dayan and Hinton's paper introduces two principles for the Manager-Worker relationship: reward hiding and information hiding. Reward hiding means that Managers reward or punish their workers based on their performance towards sub-goals set by the Manager irrespective of how it affects the overall goal. This enforces mastery at achieving subgoals at lower levels without regard for whether those have been correctly set. Information hiding means that each level of hierarchy only knows the state of the system at the granularity of their own choice of tasks.

(D. Kulkarni et al., 2016) adopt a similar hierarchical reinforcement learning methodology (h-DQN) that provides some twists on goal formation. They divide the objectives in terms of intrinsic and extrinsic goals. The rewards associated with the extrinsic goals are provided by the environment, however, are very sparse. To address this issue, a meta-controller and controller hierarchy, a la Manager-Worker, is introduced. The meta-controller works on a longer temporal scale, acts by setting goals for the controller and supplies intrinsic rewards to the controller. The controller takes actions at every time step to try and accomplish these goals. The overall system tries to maximize the total expected intrinsic and extrinsic rewards. They test their approach on a complex discrete stochastic process and the ATARI game, "Montezuma's Revenge", in the ALE environment. Unlike Feudal Networks, the h-DQN relies on user-provided lower level subgoals. This makes convergence easier to achieve and can be thought of as a step towards achieving the functionality of the Feudal Network paper. Ultimately, achieving similar performance to this work is the most realistic goal for our project.

3. Methods

3.1. Environments

We perform comprehensive testing on a host of different environments that have continuous and discrete action spaces. These include a stochastic MDP test scenario utilized in (D. Kulkarni et al., 2016); `InvertedDoublePendulum-v2`, `Reacher-v2`, and `FetchReach-v1` OpenAI MuJoCo environments; and the OpenAI Atari environments (Brockman et al., 2016), Montezuma's Revenge and Ms.Pacman. The stochastic MDP and Atari games are discrete action space environments, while the Mujoco environments have a continuous action space.

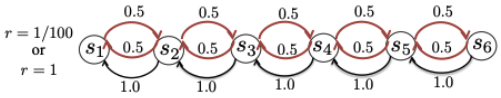

Figure 1. Stochastic MDP environment. Starting at s2 and terminating at s1, collect a reward of 1 if the agent reaches s6, otherwise collect a reward of 0.01. This environment can be extended to an arbitrary length.

The stochastic MDP, shown in Figure 1 is a simple test case to verify the hierarchical model works, as it is difficult to solve optimally without a higher level controller. One can effectively think of it as navigating a long corridor. The double inverted pendulum involves controlling the left-right forces applied to a cart that is balancing a double inverted pendulum. The Reacher environment involves controlling the forces applied to a two-limb arm that must reach and maintain its position at a target. The FetchReach robotics task is similar to the Reacher, but involves controlling more arm segments. The last two environments are the Ms. Pacman and Montezuma's Revenge Atari environments, which are played without direct access to the true game state, instead relying solely on the video stream. Ms. Pacman involves moving a character around a maze, gobbling up all the food pellets scattered around while avoiding ghosts. Montezuma's Revenge is an arcade environment where the character must navigate several rooms to obtain keys while avoiding skulls. The stochastic MDP, double inverted pendulum, and Reacher environments are easier test cases to verify our implementations work before attempting the much harder Fetch robot and Atari environments.

3.2. Baselines

While working towards an eventual hierarchical architecture capable of self-learned lower-level goals, we rely on several simpler baseline designs. The most fundamental building block we implement is the Deep Q Network (DQN). This involves using a neural network to estimate

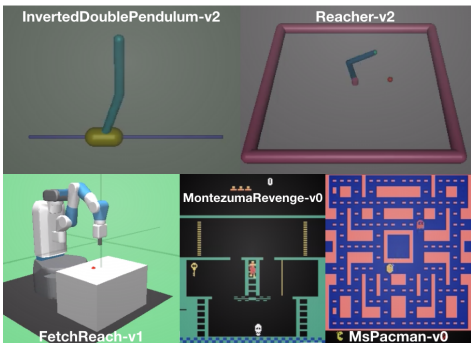

Figure 2. OpenAI Gym Environments

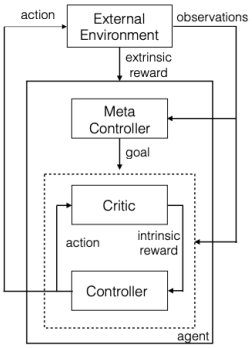

Figure 3. Control Sequence Describing the Meta-Controller and Controller Process

the Q values for each action. We also rely upon a separate target network that fixes the Q value and updates it every number of episodes to stabilize training. The loss function is Huber loss. This is essentially a flat policy variant of the h-DQN implemented by (D. Kulkarni et al., 2016). The DQN architecture lends itself naturally to discrete action space environments due to parameterizing Q based on actions; however, it can be modified to work on continuous action space environments by discretizing the action space. To produce better results for the MuJoCo continuous action space environments, we rely on a flat and hierarchical policy gradient methods. Policy gradient methods directly model the action probabilities (which allows it work better in continuous action spaces) instead of Q, and they find parameters of the policy network to maximize the total future expected rewards. Unlike DQNs, these are on-policy methods. At each iteration, the policy network defines the action probabilities according to the current state and the parameters are updated to get a slightly better policy than the current one according to the rewards obtained.

3.3. Proposed Model

3.3.1. HIERARCHICAL-DQN

Our proposed strategy is derived from the h-DQN framework presented in (D. Kulkarni et al., 2016). We first reproduce the model implementation they have presented in both tabular q-learning and h-DQN forms. As discussed above, the h-DQN framework consists of meta-controller and controller relationship. The meta-controller feeds intrinsic subgoals to the controller to better deal with sparse rewards. The Figure 3 shows the control sequence of the h-DQN.

We applied the h-DQN method to the MDP environment shown in Figure 1. As mentioned in the paper, we use the states as the subgoals for the controller. The meta-controller learns which state to guide the controller towards next. This approach involves crafting the subgoal space for each environment explicitly as opposed to in Feudal Networks, where the model also learns the space of the subgoals.

For hierarchical tabular q-learning we maintain two q-tables, one for meta-controller (state(6) $\times$ state(6)) and one for controller (state-subgoal(36) $\times$ action(2)). To generate the sample space we sample a subgoal from the the meta-controller table using ϵ-greedy method. This subgoal-state pair is then used to sample an action from the controller table via ϵ-greedy method. Then, we perform this action in the environment to obtain the next state. Both of these entries are stored in the respective replay buffers of the meta-controller and controller. The entries used are of the form:

$$entry_{\text{con}} = (c_s, a, n_s, r) \tag{1}$$

where c_s is current state, a is action, n_s is the next state and r is the reward.

$$entry_{\text{meta-con}} = (c_s, sg, n_s, t_r) \tag{2}$$

where sg is the subgoal and t_r is the total reward for the episode.

During training, we randomly sample a batch from each of the replay buffers and use them to update the two tables. The predicted q-value for meta controller is the entry corresponding to the state-subgoal pair in the meta-controller table and target q-value is computed using the Bellman equation:

$$q_{t_{target}} = R_t + \gamma max(q_{t+1}) \tag{3}$$

where R_t is the reward value in the replay buffer entry, q_{t+1} is the maximum value among all possible subgoals corresponding to the next_state entry in the replay buffer sample, and γ is the discount rate. The table entries are updated using:

$$q_{state} = \alpha q_{state} + (1 - \alpha) q_{target} \tag{4}$$

The training for the controller is done in a similar fashion. For our implementation we maintain a replay buffer of size 1M and $\gamma = 1$.

For hierarchical DQN, we use two neural networks to parameterize the meta-controller and controller. The input to the meta controller is the current state of the environment; its output is an n-dimensional vector corresponding to the subgoal space (where there are n subgoals). The input to the controller is the state concatenated with the one-hot encoded subgoal. The output of the network is a m-dimensional vector representing Q values corresponding to the action space of size m. The neural networks consist of 2-layer MLPs with ReLU activations, with an optional CNN prior to the MLP when the input states are images. The sampling process and replay buffer entry generation is similar to the tabular method except that now we sample from the two neural networks. During training, we update the parameters of the two neural networks by minimizing the Huber loss between the predicted and target Q-values. The target value is still obtained using the Bellman equation. Huber loss is defined as:

$$L = \begin{cases} \frac{1}{2}(q_t - q_p)^2 & |q_t - q_p| \leq \delta| \\ \\ \delta(q_t - q_p) - \frac{1}{2}\delta^2 & otherwise \end{cases} \tag{5}$$

where q_t is the target q-value and q_p is the predicted q-value.

The predicted value for meta-controller is sampled from the meta-controller neural network and the target q-value for the next state is sampled from a target neural network, which is a copy of the meta-controller neural network and updated every few timesteps. This delayed update ensures the model has a fixed target.

All the following models are derived from the hierarchical DQN approach. These are essentially extensions of the h-DQN model for continuous action spaces.

3.3.2. Hierarchical Policy gradient

For continuous observation and action spaces we extend the idea of hierarchical DQNs to a hierarchical policy gradient method. The architecture of meta-controller and controller remains the same; however, they learn a policy (instead of Q-value) over $subgoals$ and $actions$, respectively. After evaluating a few policy gradient methods like A2C, we eventually settle on using Trust Region Policy Optimization (TRPO) (Schulman et al., 2015), which uses an adaptive step size for gradient descent to learn both meta-controller and controller policies. TRPO has found to be effective for optimizing neural network based non-linear policies. TRPO also results in a more monotonic improvement during training ac compared to A2C. It solves the following optimization problem:

$$\begin{aligned} &\text{maximize}_\theta \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A_t \right] \\ &\text{subject to} \quad \mathbb{E}_t \left[KL[\pi_{\theta old}(.|s_t)|\pi_\theta(.|s_t)] \right] \leq \delta \end{aligned}$$

where π_θ and $\pi_{\theta old}$ denote the current policy and policy from previous update cycle respectively and A_t is the advantage term same as advantage actor critic (A2C).

The second condition in the above equation helps to determine the appropriate step size for policy update by restricting the updated policy to be not very different from the policy at previous time step.

Similarly to the hDQN, the meta-controller and controller work in different time horizons. The meta-controller feeds a subgoal(s) to the controller and waits while the controller tries to take actions to achieve the subgoal.

3.3.3. Feudal Network

This model is derived from the 2017 paper (Vezhnevets et al., 2017). Currently, no existing implementation is available online that is capable of reproducing the results in this paper, so we foresee difficulty matching their results.

We get the observation x_t from the environment and apply a perceptual module to generate an intermediate representation z_t shared by the Manager and the Worker. The Manager internally computes a latent space representation s_t by passing z_t through another module which is then passed through a dilated LSTM to get the goal vector g_t.

The Worker takes the shared embedding z_t and feeds it through an LSTM to get the output U_t which is combined with w_t using a dot product to get the policy π, vector of probabilities over primitive actions. w_t is obtained using a linear transformation of the goal vectors g_t.

The Worker can be trained using a standard policy gradient algorithm with the reward from the environment combined with the intrinsic reward given by the Manager. However, the Manager is trained independently to predict advantageous transitions in the state space and intrinsically rewards the Worker to follow this direction. We make the assumption that the Worker will closely follow the Manager's goals

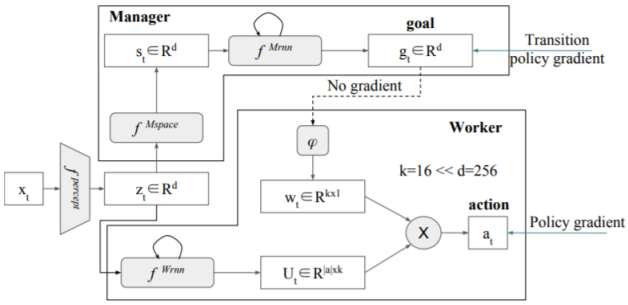

Figure 4. Feudal Network Architecture

eventually and thus we update the gradients as follows:

$$\nabla g_t = A_t^M \nabla_\theta d_{cos}(s_{t+c} - s_t, g_t(\theta)) \tag{6}$$

where A_t^M is the Manager's advantage function, and d_{cos} is the cosine distance. Thus g_t now has acquired a semantic meaning as an advantageous direction in latent space s_t at a temporal resolution of c.

4. Results and Analysis

4.1. Long Corridor Stochastic MDP

In order to first verify the validity of our hierarchical implementations, we begin with the stochastic MDP environment, where a flat policy learner should struggle to approach the optimal reward of 1. The subgoal space for the hierarchical agent is defined as the space of all states in the MDP. The meta-controller converges towards selecting increasing states until the controller has reached the end before then transitioning backwards towards the s_1 state.

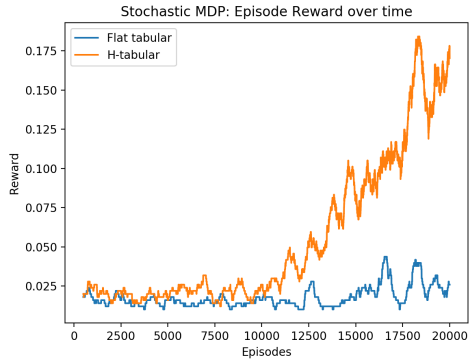

Figure 5. Comparison of Q Learning performance on stochastic MDP environment with 6 states smoothed with moving average over every 500 episodes.

Since the MDP environment has discrete state and action spaces, we implemented a basic tabular Q learning algorithm and modified it to be hierarchical for comparison. In a 6-state MDP, as shown in Figure 5, the hierarchical agent clearly demonstrates the capacity to learn, unlike the flat agent. With this working, we can confidently use this environment as a test for our h-DQN implementation.

Since a DQN relies on more expressive neural networks than tabular Q learning, it is capable of learning on the 6-state MDP. Here, comparisons between flat and hierarchical variants on longer corridors are much more informative. The results between a flat and hierarchical DQN on 8 and 10 state MDPs are shown in Figure 6.

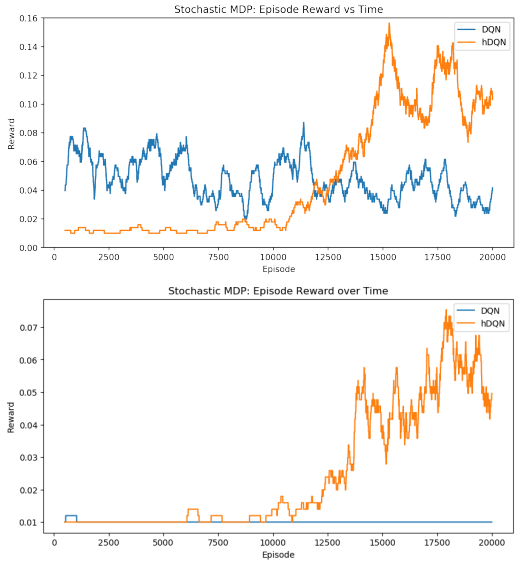

Figure 6. Comparison of DQN performance on stochastic MDP environment with 8 (top) vs 10 (bottom) states smoothed with moving average over every 500 episodes.

As the number of states in the MDP is increased, the performance difference between the flat and hierarchical becomes increasingly apparent. For 8 states, the DQN is able to obtain some rewards, although the hDQN eventually surpasses it. For 10 states, the DQN virtually never obtains the full rewards even after 20,000 episodes. This makes sense because for a longer corridor, it is increasingly unlikely for the flat learner to stumble upon the correct policy by randomly exploring until it reaches the end of the corridor and transitions all the way back. The hierarchical agent, on the other hand, has a meta-controller that learns to select the end of the corridor as its subgoal, which the controller is able to fulfill before moving back to the first state, thereby granting it the full reward for the environment.

4.2. Inverted Double Pendulum

Moving on to more difficult MuJoCo environments, their continuous action spaces require binning the DQN's action space. The results for the DQN on the Inverted Double

Pendulum are shown in Figure 7. Rewards are cumulative across each episode. While the DQN is learning quite

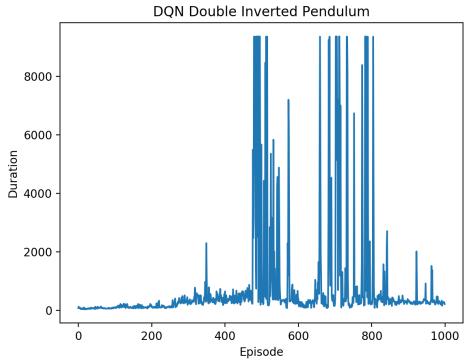

Figure 7. DQN training plot on inverted double pendulum

well (a score of 8000 corresponds roughly to the max duration of each episode), it exhibits inconsistent behavior despite heavily cooling the exploration ϵ. As a result of discretizing the action space, a fine movement resolution is difficult to regress. Before proceeding to the remaining MuJoCo environments, we investigate the performance of A2C policy gradient method, as shown in Figure 8. Since it performs much better than the DQN, it became clear that achieving meaningful results on the MuJoCo environment necessitates the use of a policy gradient method. Because the double inverted pendulum has such a simplistic objective, it is difficult to structure as a hierarchical problem, so comparisons with hierarchical models will be reserved for the Reacher and FetchReach environments.

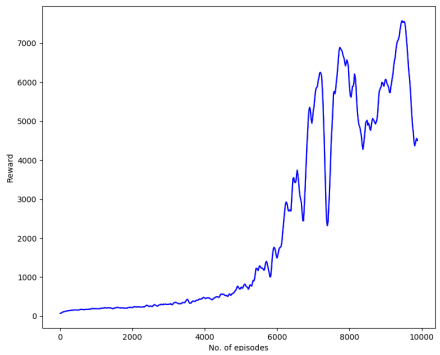

Figure 8. A2C training plot on inverted double pendulum.

4.3. Reacher

For the Reacher environment, we compare a DQN, flat TRPO policy gradient, and hierarchical-TRPO method. We switched to TRPO from A2C due to its superior performance on this more difficult task. Since Reacher works in a 2D environment, the subgoal space has 5 elements: $\{x^+(\text{right}), x^-(\text{left}), y^+(\text{up}), y^-(\text{down}), \text{stay}\}$. The subgoal target is defined as

$$\text{target}_s = \begin{cases} \text{curr}_{\text{pos}} + (\alpha, 0) & \text{dir} = x^+ \\ \text{curr}_{\text{pos}} + (-\alpha, 0) & \text{dir} = x^- \\ \text{curr}_{\text{pos}} + (0, \alpha) & \text{dir} = y^+ \\ \text{curr}_{\text{pos}} + (0, -\alpha) & \text{dir} = y^- \\ \text{curr}_{\text{pos}} & \text{dir} = stay \end{cases} \tag{7}$$

where curr_{pos} is the current position of the Reacher's end-effector.

We ran Reacher in both dense and sparse meta-controller (r_{ext}) reward settings. For dense rewards,

$$r_{ext} = r_{env} \tag{8}$$

where environmental reward r_{env} is the negative Euclidean distance from the Reacher's end-effector to the target. For sparse rewards, the extrinsic reward is defined as

$$r_{ext} = \begin{cases} -1 & \|\text{curr}_{\text{pos}} - \text{target}\|_2 > \delta \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

This penalizes the Reacher if its end-effector is not within a certain radius of the environment target. The intrinsic reward for the controller is defined in the same manner, except the distance is to the subgoal target, not the environment target.

$$r_{int} = \begin{cases} -1 & \|\text{curr}_{\text{pos}} - \text{target}_s\|_2 > \beta \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

For our experiments we used $\alpha = 0.25$, $\beta = 0.15$ and $\delta = 0.05$ based on the observation space values of the environment. The Reacher results with dense rewards are shown in Figure 9.

Clearly, the policy gradient methods outperform the DQN, which is consistent with results from the inverted double pendulum. Furthermore, the hierarchical policy gradient method converges to a higher reward than the flat policy gradient method. For the DQN, the arm often overshoots or misses the target. The DQN is especially troublesome with regards to state discretization, as binning the two joints into 10 bins each results in a 100-dim output from the policy network. This is unsustainable and produces worse results than policy gradient.

The difference between flat and hierarchical TRPO becomes more apparent when the environment reward is restructured to be sparse, as shown in Figure 10. While the hierarchical learner only provided marginal benefits in the dense reward case from Figure 9, it provides a significant improvement under sparse rewards.

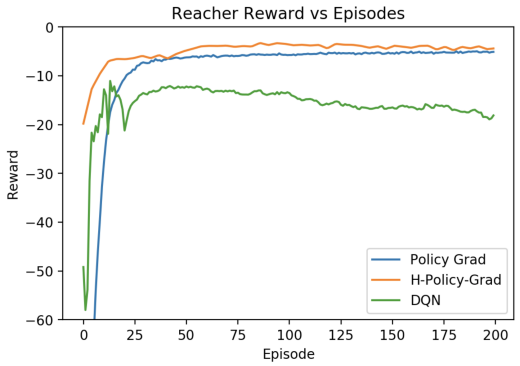

Figure 9. Comparison of reward plots on Reacher-v2 environments with standard dense rewards smoothed using a 20-episode moving average.

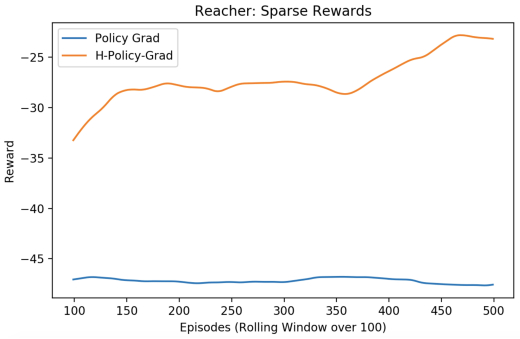

Figure 10. Hierarchical TRPO versus vanilla TRPO policy gradient on Reacher-v2 environment using sparse rewards

4.4. Fetch Reach

In the Fetch Robotics MuJoCo environment, we compared the performance of a vanilla TRPO and hierarchical TRPO policy gradient agent for the FetchReach task. The agent's meta-controller uses the dense reward function from the environment, where a negative reward is provided proportional to the Euclidean distance between the robot's end effector and the target. The subgoals are defined as the six cardinal directions in the xyz-space along which the end-effector of the Fetch robot can move. The controller relies on the same dense reward as used with the Reacher. We used the parameter values $\alpha = 0.1$, $\beta = 0.05$.

Initially we tried giving the subgoal target directly without using direction. In this approach, the meta-controller used a continuous policy network rather than the discrete one (used when we specify the cardinal directions) to give the subgoal target for the controller. This method failed, as the meta-controller output space was too large and lacked constraints. To solve this problem, we first thought of adding another reward term in the meta-controller to encourage the meta-controller to give a subgoal not far from the current position, but this also did not work. And now we could set the hyperparameter α to constrain the subgoal the controller has to reach. The final results are shown in Figure 11.

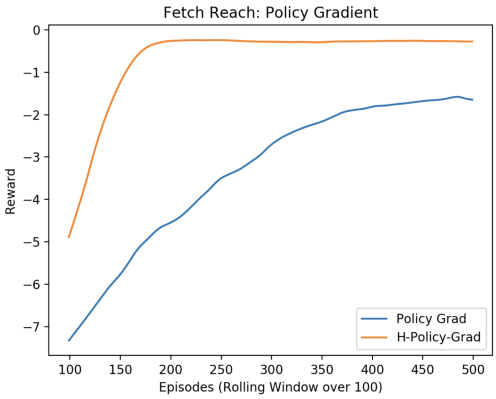

Figure 11. Reward vs Episodes for the Fetch Robot Reach Environment

The hierarchical learner not only learns at a faster rate, but also achieves a much higher reward value at convergence. Visually, the flat policy gradient method has difficulty stopping at the target point and oftentimes overshoots. The hierarchical method, on the other hand, consistently reaches and stops at the target point. The behavior of the hierarchical learner is as expected, as the controller is slowly able to reach more and more of the subgoals it has been provided (Figure 12).

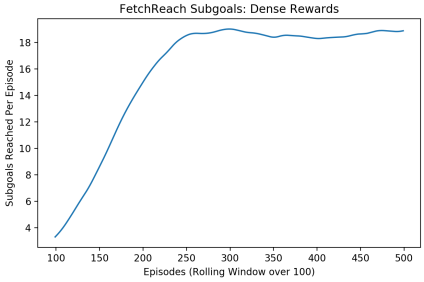

Figure 12. Progression of controller performance over time. The controller learns to reach the subgoals better over time.

4.5. Atari

Lastly, we attempt to design flat and hierarchical DQNs that are able to solve Atari environments.

Since we are dealing with video streams, the agents require the addition of CNNs to encode the images as feature vectors. To provide user-defined subgoals also requires knowing where the character is in the game. Thus, a custom object detector is required to localize the agent-controlled character in the environment. We utilize an exemplar image blob of the character and perform a search over the

image for an object of the same color. This proves to be a very robust and time-efficient strategy, as our previous attempt with a normalized sum-of-squared differences algorithm was much more computationally involved. We also tried convolving the environment image with the template image to produce a filtered image response where we could simply take the maxima. However, this failed to be lighting-invariant, as brighter objects naturally result in a higher response regardless of how well they match the template.

The results for Ms. Pacman are shown in Figure 13. The flat learner demonstrates the capacity to learn over time, albeit slowly. A random agent is capable of achieving rewards of around 200. For the h-DQN, we made a number of attempts at creating a good reward structure and our efforts are described as follows. Ultimately, the hDQN converges poorly as a result of improper subgoal space formulation. While initially it appears to perform better than DQN, it quickly converges suboptimally. This demonstrates that while the controller is learning to achieve the subgoal, the intrinsic rewards are not well aligned with the extrinsic rewards.

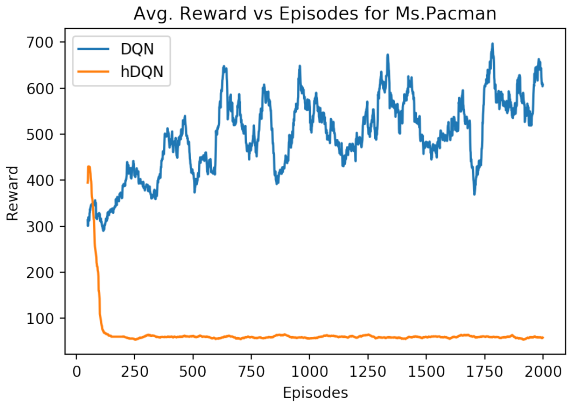

Figure 13. Reward vs episodes for the Atari Ms.Pacman Environment, smoothed with moving average over every 50 episodes. The hDQN is an example of poor convergence behavior we were seeing.

We started off with specifying the subgoals to be the power pellets that allow Pacman to eat the ghosts. The idea was that these subgoals would result in Pacman getting bonus points for eating ghosts, and the meta-controller would eventually learn to point Pacman to the nearest power pellet. While this would not necessarily result in the game being solved completely, it would give us a good understanding of the subgoals and reward strucutre, as we moved to a feudal network where the goal space is learned as well.

The intrinsic reward was initially defined to be 1 when Pacman reached the subgoal specified by the meta-controller, and 0 otherwise. With this reward structure, Pacman was seen to essentially stay in the same spot and not move very far from its starting position. We determined that the reward structure was too sparse for it to learn to move, and changed the intrinsic reward to be the negative of the Euclidean distance between the subgoal and Pacman's current position. Pacman now learns to move towards the subgoal, but is seen to get stuck in corners while moving in this direction. These corners correspond to local minima of the intrinsic reward distribution.

We also crafted another dense intrinsic reward structure, which was a weighted average of the extrinsic reward and a binary reward for reaching the subgoal. This vastly improved Pacman's ability to traverse the environment, but it would not learn to move towards the specified subgoal. We believed this was due to the subgoals being sparsely-distributed, and added more subgoals at multiple junctions. This failed likely because although the controller would learn to reach the subgoals that weren't power pellets, the extrinsic reward received by the meta-controller relates very poorly to the actions it can take (choosing the next subgoal). The meta-controller in this case operates in a sparse environment, since it can only learn a policy based on the rewards obtained from Pacman eating the ghosts soon after the power pellet.

We also implemented an hDQN for the Montezuma's Revenge environment, with the subgoals being different objects in the environment, like the ladders, ropes, doors and keys. The extrinsic rewards are very sparse in this case, with points only awarded for collecting the key and passing through the door after that. Since performance was similarly poor, as the environment was much more challenging than Ms.Pacman, we omitted the results for this environment.

5. Conclusion

Ultimately, we learned several lessons from applying hierarchical models to the flat DQN and TRPO agents. While hierarchical models can be more performant, as demonstrated in the MuJoCo environments, it took several attempts at subgoal formulation to achieve reasonable results (and this had to be repeated separately for each environment). This difficulty was very evident as we transitioned to Atari environments, where the state space was no longer as well-defined. We notice that while HRL performs marginally better in dense-reward environments, its performance is far superior (given correct convergence) in sparse reward environments. In essence, the strength of the hierarchical network is most evident when it is sufficiently straightforward to construct subgoals. While we were unable to achieve the ultimate goal of Feudal Networks' fully-learned subgoals, we obtain excellent results in solving the MuJoCo environments with our own h-TRPO algorithm.

References

Al-Shedivat, M., Lee, L., Salakhutdinov, R., and Xing, E. On the Complexity of Exploration in Goal-Driven Navigation. *arXiv e-prints*, art. arXiv:1811.06889, Nov 2018.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

D. Kulkarni, T., Rajagopal Narasimhan, K., Saeedi, A., and B. Tenenbaum, J. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. 04 2016.

Dayan, P. and Hinton, G. Feudal reinforcement learning. 1993. URL http://www.cs.toronto.edu/~fritz/absps/dh93.pdf.

Nachum, O., Gu, S. S., Lee, H., and Levine, S. Data-efficient hierarchical reinforcement learning. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31*, pp. 3303–3313. Curran Associates, Inc., 2018.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. *CoRR*, abs/1703.01161, 2017. URL http://arxiv.org/abs/1703.01161.