HIERARCHICAL REINFORCEMENT LEARNING WITH LIMITED POLICIES AND HINDSIGHT

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce a new hierarchical reinforcement learning framework that can accelerate learning in tasks involving long time horizons and sparse rewards. Our approach improves sample efficiency by enabling agents to learn a hierarchy of short policies that operate at different time scales. The policy hierarchies can support an arbitrary number of levels, and all policies within the hierarchy are trained in parallel and end-to-end. Our framework is the first hierarchical reinforcement learning approach that can learn hierarchies with more than two levels of policies in continuous tasks. We demonstrate experimentally in both grid world and simulated robotics domains that our approach can significantly boost sample efficiency. A video illustrating our results is available at `https://www.youtube.com/watch?v=i04QF7Yi50Y`.

1 INTRODUCTION

Reinforcement learning (RL) algorithms struggle in tasks that involve long time horizons and sparse rewards. A major reason for this outcome is that credit assignment, or the process of estimating the long-term value of executing an action in a certain state, is slower. For tasks that require a long sequence of decisions, it takes longer to backup the higher value estimates of the actions that produce the sparse reward to the lower value estimates of the preceding actions that produce no immediate reward. The ability to learn a hierarchy of short policies that operate at different time scales can help alleviate this problem. This idea is illustrated in Figure 1 for a four-rooms grid world task. An agent that could only learn at the time scale of atomic actions would need to learn a policy of around 20 actions to solve this task. However, an agent that learns at two time scales could learn this task using policies consisting of at most five actions. Specifically, the high-level agent could learn to break down the task into the sequence of four subgoal states (i.e., $a \rightarrow b \rightarrow c \rightarrow d$). The low-level agent could then learn the at most 5 atomic actions required to achieve each of these subgoals. Although the ability to learn at different time scales has created more tasks to learn, each of these tasks requires a shorter sequence of actions. If the agent could learn these subtasks in parallel, the agent could potentially learn the full task more quickly.

We introduce a framework that enables agents to learn a hierarchy of short policies operating at different time scales in parallel and end-to-end. The framework contains two components: (i) a particular policy hierarchy to be learned and (ii) a method to train the policy hierarchy. The policy hierarchy to be learned consists of a stack of nested and goal-parameterized policies. Each policy receives as input the current state and goal state and outputs a subgoal state for the next level to achieve, except for the lowest level in the hierarchy, which outputs an atomic action. Another critical feature of the policy hierarchy is a hyperparameter that enforces a limit on the length of the sequence of actions that each policy learns. This policy limit hyperparameter not only ensures that the agent learns short policies, but enables the agent to learn the set of short policies in parallel and end-to-end using Hindsight Experience Replay (Andrychowicz et al. (2017)).

There are other automated hierarchical RL approaches that can work in tasks with continuous state and action spaces, such as Skill-Chaining (Konidaris & Barto (2009)), the Option-Critic Architecture (Bacon et al. (2016)), and FeUdal Networks (FUN) (Vezhnevets et al. (2017)). The key advantage that our framework has over these other methods is that our approach can use policy hierarchies with an arbitrary number of levels, whereas the other methods are limited to just two levels. The additional levels of hierarchy are important because they enable agents to divide the original

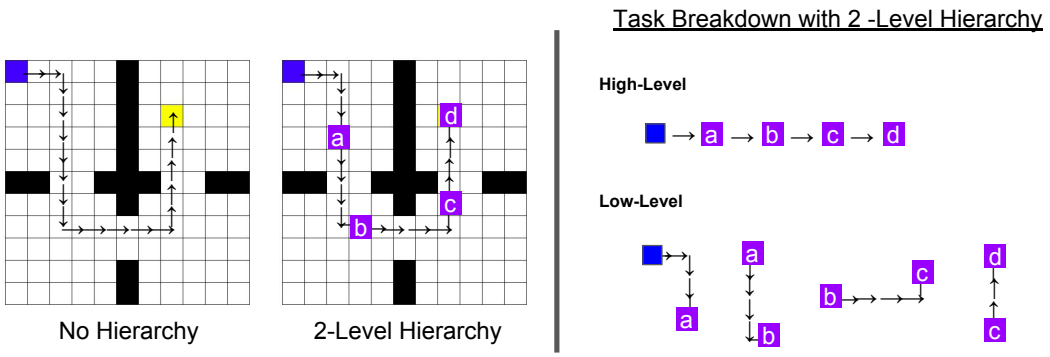

Figure 1: Figure shows a simple grid world example motivating the need for agents to learn short policies that operate at different time scales. If the blue square agent could learn at two time scales, it would only need to learn sequences of at most 5 actions. On the other hand, an agent that only learned at the time scale of atomic actions would need to learn a sequence of 20 actions.

task into even smaller sequences of actions, which can help agents learn the task more quickly. To the best of our knowledge, our framework is the first to enable agents to learn a policy hierarchy with more than two levels in continuous tasks.

We evaluated our approach on both grid world tasks and more complex simulated robotics environments. For each task, we evaluated agents with 1, 2, and 3 levels of hierarchy. In all tasks, agents using 2 and 3 levels of hierarchy significantly outperformed agents that learned a flat policy. Further, in many of the challenging tasks, agents using 3 levels of hierarchy significantly outperformed agents using 2 levels of hierarchy.

2 BACKGROUND

A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R)$ (Sutton & Barto (1998)). $\mathcal{S}$ denotes a set of states. $\mathcal{A}$ represents a discrete set of actions. $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ denotes the transition probability function in which $T(s, a, s')$ is the probability of transitioning to state s' when action a is taken from state s. $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the expected reward from executing action a in state s. The solution to an MDP is a control policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the expected sum of future discounted rewards when acting under the policy.

The Universal Value Function Approximator (UVFA) (Schaul et al. (2015)) extends the MDP framework for the episodic case by introducing a set of goals $\mathcal{G}$ and a goal-dependent reward function $R_g : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, in which $g \in \mathcal{G}$. A goal is typically a state or a set of states. In a standard MDP, the reward function is the same on each episode. In the UVFA scenario, each episode uses a potentially different reward function. Specifically, at the beginning of each episode, we initialize both the state of the agent *and* the goal state. The selected goal state g remains constant for the duration of the episode and defines the reward function r_g in use for that episode. In this framework, a policy maps from both state and goal to an action, $\pi : \mathcal{S} \times \mathcal{G} \rightarrow \mathcal{A}$. As a consequence, the Q-function is also now a function of the goal, $Q(s, a, g)$. UVFAs are helpful because they enable agents to generalize their policy to new goals based on experience achieving similar goals.

Hindsight experience replay (HER) (Andrychowicz et al. (2017)) is a mechanism for improving sample efficiency when learning in a UVFA framework with sparse rewards. During training, experience transitions are created, storing state s_t, action a_t, next state s_{t+1}, and reward r_{t+1} in a replay buffer. However, the UVFA also requires knowledge of the relevant goal state in order to train $Q(s, a, g)$. The most immediate solution is to augment s_t, a_t, s_{t+1}, r_{t+1} with the goal state in effect at the time the experience was encountered. HER supplements this information by storing the same experience multiple times and replacing the original goal with a goal state the agent came across during an episode. This way, a single experience can help the agent learn policies for multiple goals, not just the particular goal in use when the experience was initially stored.

3 APPROACH

We introduce a new framework that enables agents to learn a hierarchy of limited policies that operate at different time scales in parallel and in an automated manner. The framework includes two main components: (i) a policy hierarchy with some certain architectural and operating properties, and (ii) a method for training the policies within the hierarchy.

3.1 POLICY HIERARCHY - ARCHITECTURE

To understand the architecture of the policies within the hierarchy, we first formally define the hierarchy of Universal MDPs (UMDP) from which the policies emerge.

3.1.1 UNIVERSAL MDPS

The UVFA framework (Schaul et al. (2015)) can be viewed as an extension of the MDP framework. We will use the following definition:

Definition 3.1 *A Universal MDP (UMDP) is a tuple $\mathcal{U} = (\mathcal{S}, \mathcal{A}, T, R, \mathcal{G})$ where $\mathcal{S}$, $\mathcal{A}$, and T are defined exactly as they are in the MDP case; $\mathcal{G}$ is a goal set; and $R : \mathcal{S} \times \mathcal{G} \rightarrow \mathbb{R}$ is the reward function where $R(s, g)$ denotes the reward obtained after arriving in state s for a goal g.*

At the beginning of each episode in a UMDP a fixed goal $g \in \mathcal{G}$ is assigned that defines the reward function for the duration of that episode.

The aim of this paper is to develop an efficient method of solving an arbitrary UMDP $\mathcal{U}_{original} = (\mathcal{S}, \mathcal{A}, T, R, \mathcal{G})$ in the standard reinforcement learning scenario where it is assumed that the algorithm can execute actions and observe the results. We will assume that $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{G}$ are known while T and R are unknown.

3.1.2 CONSTRUCTION OF THE HIERARCHY

Given an original UMDP $\mathcal{U}_{original} = (\mathcal{S}, \mathcal{A}, T, R, \mathcal{G})$ to be solved, our approach is to learn a hierarchical policy. We generate this policy by defining and solving a hierarchy of k UMDPs, $\mathcal{U}_0, \ldots, \mathcal{U}_{k-1}$. The hyperparameter k is provided by the user. The hierarchy is constructed as follows:

$\mathcal{U}_0$: This is the lowest level of the hierarchy. It has the same state set and action set as the original UMDP: $\mathcal{S}_0 = \mathcal{S}$ and $\mathcal{A}_0 = \mathcal{A}$. At this level, every state is potentially a goal. Therefore, the goal set $\mathcal{G}_0 = \mathcal{S}$ is equal to the state set. We will use a shortest path reward function for all levels of the hierarchy, in which a reward of 0 is granted if next state, s', maps to the goal: $R_0(s', g) = 0$ if $s' \in g$. Otherwise, a reward of -1 is given to the agent. However, the framework is flexible to any type of reward function.

$\mathcal{U}_i, 0 < i < k-1$: This is an intermediate level of the hierarchy. It also has the same state set as the original UMDP: $\mathcal{S}_i = \mathcal{S}$. It has an action set equal to the state set: $\mathcal{A}_i = \mathcal{S}$. As on the first level, its goal set is equal to the state set, $\mathcal{G}_i = \mathcal{S}$, and its reward function is defined similarly: $R_i(s', g) = 0$ if $s' \in g$ and $R_i(s', g) = -1$ otherwise.

$\mathcal{U}_{k-1}$: This is the top level of the hierarchy. It has the same state set, goal set, reward function as the original UMDP: $\mathcal{S}_{k-1} = \mathcal{S}$, $\mathcal{G}_{k-1} = \mathcal{G}$, and $R_{k-1} = R$. Its action set is equal to the state space $\mathcal{A}_{k-1} = \mathcal{S}$.

To summarize, the goal of our approach is to learn a hierarchy of universal, or goal-parameterized policies. Each policy will thus try to learn the sequence of actions that bring the agent from the current state to the goal state. The key differences among the policies lie in the time scale of the actions, which is described in the next section, and the action space for each policy. At all levels except for the bottom most level, policies will output subgoal states for the next layer to achieve. The lowest level will output atomic actions like a regular non-hierarchical policy. Figure 2 (Left) shows an example policy architecture of a 2-layered agent that operates in a continuous domain.

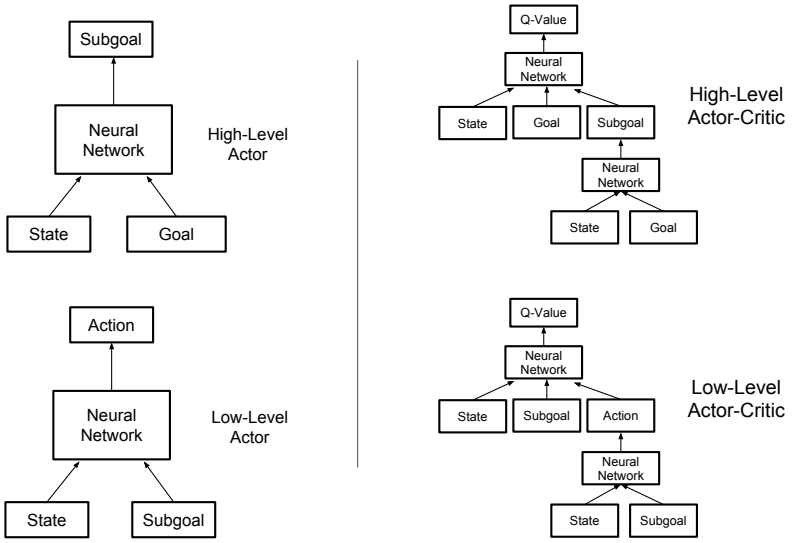

Figure 2: (Left) Policy architecture of an agent with 2 levels of hierarchy that operates in a continuous domain. (Right) Policy, or actor, functions connected to the critic functions used to train each policy.

Figure 2 (Right) shows the policy, or actor networks, together with the critic networks used to train each policy.

3.2 Policy Hierarchy - Action Timing

The combination of the nested structure of the policy hierarchy and the policy limit hyperparameter, $T \in N$, helps the agent learn short policies that operate at different time scales by affecting the timing of actions.

The nested structure of the policies causes action selection for each policy to work in a top-down manner. Let $\pi_i : \mathcal{S}_i \times \mathcal{G}_i \rightarrow \mathcal{A}_i$ denote the current policy at level i. At the beginning of an episode, a start state $s \in \mathcal{S}$ and a goal $g \in \mathcal{G}$ are initialized and a top level action $a_{k-1} \in \mathcal{A}_{k-1}$ is selected by evaluating π_{k-1} for s and g. This action, a_{k-1}, instantiates a goal $g_{k-2} \in \mathcal{G}_{k-2}$ at level $k-2$ and enables the agent to select an action at level $k-2$ by evaluating π_{k-2} for s and g_{k-2}. This process repeats for each level all the way to the bottom hierarchy layer where π_0 is evaluated for s and goal $g_0 = a_1$.

The actual actions that are executed by each policy work bottom up. At the lowest level, once an action a_0 is selected for execution, the policy at that level π_0 continues until a goal state is reached (i.e., until current state $s \in g_0$) or until T a_0 actions have been executed. At this point, control returns to the next level up (level 1 in this case). If π_0 was able to achieve the goal $a_1 = g_0$ from π_1, then we define π_1 as executing action a_1. However, if the base policy π_0 was not able to achieve goal g_0 in T steps, we will define π_1 as executing the subgoal state action, s_0, in which s_0 is the state that was reached after T attempts by π_0. As the next section will show, replacing the original subgoal action with the subgoal state reached in hindsight if the original subgoal was not achieved will enable the use of Hindsight Experience Replay, which will help all policies in the hierarchy learn simultaneously. After the π_1 action has been recorded, a new level 1 action a_1 is selected by π_1 and passed down to π_0. Execution continues in this way at level 1 until a goal state is reached (i.e. until current state $s = g_1$) or until the maximum number of level 1 actions, T, have been executed. This process continues up until the top level.

The key idea here is that the policy hierarchy described above has the same qualities as the hierarchical policy we seek. The goal is to learn a hierarchy of short policies that act over different time scales in order to accelerate credit assignment. In the policy hierarchy we have introduced, the hyperparameter T ensures that each policy only needs to learn short sequences of actions. Further, the combination of the nested structure and T enable each policy to act over different time scales. Each π_0 action will have the timescale of an atomic action. Each π_1 action will have the time scale

of at most T atomic actions. Each π_2 action will have the time scale of at most T level 1 actions or T^2 level 0 actions and so on.

3.3 LEARNING THE POLICY HIERARCHY

Agents learn this hierarchy of limited policies in parallel and end-to-end using two different types of transitions: (i) hindsight transitions and (ii) subgoal testing transitions.

Agents are able to learn goal-parameterized policies that operate at different time scales primarily as a result of Hindsight Experience Replay (Andrychowicz et al. (2017)). In our approach, HER is implemented at each level of the hierarchy in nearly the same way as it was originally defined. Copies are made of the transitions that were produced by the original sequence of actions. The original goal state in each transition is replaced by a state that was actually reached during that action sequence. The original reward is also replaced with the appropriate value given the updated goal state. The key difference lies in the action component used for all subgoal layer (i.e., levels $i > 0$) hindsight transitions. As described in the previous section, if level $i - 1$ is able to achieve the subgoal state a_i proposed by level i, then the action contained in the hindsight transition for level i is also a_i. However, if a_i is not achieved within T level $i - 1$ actions, then in the hindisght transition, the action component a_i is replaced with the subgoal state s that was actually achieved by level $i - 1$ after T steps. These hindsight transitons are critical for learning in an end-to-end manner because they provide the higher level policies with examples of subgoal state actions that both have the appropriate time scale and are helpful in achieving more distant goal states. The hindsight transitions also help all policies learn in parallel because a policy does not need to wait for the lower level policies to learn how to reach a subgoal state before it can evaluate the effectiveness of that subgoal state. The example provided in the next section should provide more clarity.

Subgoal testing transitions play an important role in ensuring that subgoal policies do not propose subgoals that are too ambitious (i.e., subgoal states that cannot be achieved by the succeeding layer in T attempts). A key issue caused by the nested structure of the hierarchy and the policy limit parameter is that the agent can only move so far in T attempts. As a result, hindsight transitions will only expose a subgoal layer to relatively nearby subgoal state actions. Yet, the action space for a subgoal layer is the full state space, and the agent needs to be able to assign a likely low value to these too distant subgoal state actions. Otherwise, problems can arise, particularly in continuous domains when function approximation is used. It is possible that the critic function approximator will assign a relatively high value to these distant subgoal actions because it has received no transitions indicating otherwise. This may cause the actor function to output an unrealistic subgoal, which is more problematic for a hierarchical agent using our framework than for a flat agent because the next lower level of the hierarchical agent is only trained to reach relatively close goals.

In order to overcome this issue, we perform a process we refer to as subgoal testing, which works as follows. After policy π_i proposes a subgoal state a_i, a certain percentage of the time we will turn off all exploration noise that is added to the actions from the policies below level i. Then if it turns out that policy π_{i-1} is unable to achieve $a_i = g_{i-1}$ in T attempts, layer i will be penalized for proposing that subgoal via a transition that contains a low reward. The framework is flexible to the value of the penalty reward. However, the penalty should be lower than the reward for proposing a subgoal action that is achieved but that does not achieve the higher level goal in order to incent the agent to not propose too ambitious subgoals in the future. For instance, in our implementation, penalty transitions included a reward of $-T$, which is lower than the -1 reward for proposing a subgoal that is reached but does not achieve the goal provided by the layer above. Also, it is important to note that at the beginning of training this will cause the agent to penalize useful subgoal states that can be reached with T actions. In the short term, we have seen that the UVFA will average the subgoal penalty with the higher rewards granted by the hindsight transitions for these useful subgoal states. In the longer term as the lower level policies learn how to achieve reachable subgoal states, the penalty transitions will be diluted away.

Each level of the hierarchy will use the transitions described above and the generalization capability of the UVFA to learn from experience. For the lowest level layer, the hindsight transitions will show which sequences of atomic actions are and are not capable of achieving which subgoal states. For all subgoal layers, the hindsight and subgoal testing transitions will show subgoal actions that (i)

belong to the appropriate time scale and are useful towards achieving more distant goal states but also (ii) achievable within T steps by the level below in the hierarchy.

3.4 HIERARCHICAL ACTOR-CRITIC ALGORITHM

Algorithm 1 shows the procedure used to train the policy hierarchy in continuous domains, which we refer to as $Hierarchical\ Actor - Critic\ (HAC)$. For simplicity, the pseudocode assumes an agent that uses 2 levels of hierarchy. However, the algorithm can support a hierarchy with an arbitrary number of levels by simply adding more inner for loops. Further, the algorithm can be easily adapted to the discrete domain setting by replacing the off-policy actor-critic algorithm with Q-learning, for instance.

Algorithm 1 Hierarchical Actor-Critic (HAC)

Input:
- Key agent parameters: number of levels in hierarchy k, policy limit T, subgoal testing frequency and penalty.
- Task design: $S_{init} \leftarrow$ Initial state space, $G_{k-1} \leftarrow$ task goal space

Output:
- k trained actor and critic functions $\pi_0, ..., \pi_{k-1}, Q_0, ..., Q_{k-1}$

```
for M episodes do                                          ▷ Train for M episodes
    s ← S_init, g ← G_{k-1}                                ▷ Sample initial state and task goal
    train − level(k − 1, s, g)                             ▷ Begin training
    Update all actor and critic networks
end for

function TRAIN-LEVEL(i, s, g)
    s_i ← s, g_i ← g                                       ▷ Set current state and goal for level i
    for T attempts or until g_n, i ≤ n < k achieved do
        a_i ← π_i(s_i, g_i) + noise (if not testing)       ▷ Sample (noisy) action from policy
        if i > 0 then
            Determine whether to test subgoal a_i
            s_i' ← train − level(i − 1, s_i, a_i)          ▷ Train level i − 1 using subgoal a_i
        else
            Execute atomic action a_0 and observe next state s_0'
        end if
                                                           ▷ Create replay transitions
        if i > 0 and a_i missed then
            if a_i was tested then                         ▷ Penalize subgoal a_i
                Replay_Buffer_i ← [s = s_i, a = a_i, r = Penalty, s' = s_i', g = g_i]
            end if
            a_i ← s_i'                                     ▷ Replace original action with action executed in hindsight
        end if
                                                           ▷ Evaluate executed action on current goal and hindsight goals
        Replay_Buffer_i ← [s = s_i, a = a_i, r = {−1, 0}, s' = s_i', g = g_i]
        HER_Storage_i ← [s = s_i, a = a_i, r = TBD, s' = s_i', g = TBD]
        s_i ← s_i'
    end for
    Replay_Buffer_i ← Perform HER using HER_Storage_i transitions
    return s_i'                                            ▷ Output current state
end function
```

3.5 EXAMPLE

This section walks through a simple toy example showing how hindsight and subgoal transitions are generated and how they help agents learn multiple limited policies in parallel. The agent in this task

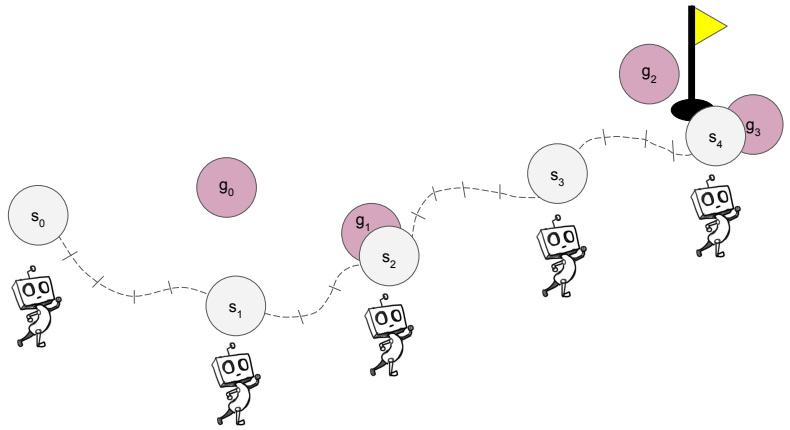

Figure 3: Figure shows an example episode trajectory for a simple toy example. The tic marks along the trajectory show the next states for the robot after each atomic action is executed. The gray spheres show the subgoal states reached in hindsight after at most T actions. The pink spheres show the original subgoal actions.

is a robot and its goal is to reach the yellow flag. The agent has two levels in its hierarchy and a policy limit parameter $T = 5$. Thus, the agent's goal is to learn both a low and high-level policy that focus on action sequences that contain no more than 5 actions.

Figure 3 shows an example episode trajectory. Please note that all subscript numbers in the figure refer to the iteration number of the high-level policy. The episode begins with the robot in state s_0 when the high-level policy π_1 proposes its first subgoal state g_0. The robot decides that during this iteration, subgoal g_0 will not be tested. As a result, the low-level policy, π_0, is free to add exploration noise to the atomic actions it proposes. Level 0 then has at most $T = 5$ atomic actions (i.e., joint torques in this case) to achieve g_0. After each action, a transition will be created showing how the action from π_0 performed given the goal state it was provided. For instance, the first transition in this example may look like $[s = s_0, a = Joint\ Torques, r = -1, s' = first\ tick\ mark, g = g_0]$. In addition, a separate transition will be created and temporarily stored in order to be used for level 0's Hindsight Experience Replay. This transition will be of the form $[s = s_0, a = Joint\ Torques, r = TBD, s' = first\ tick\ mark, g = TBD]$. Per the example in Figure 3, the robot fails to achieve state g_0 after 5 attempts and lands in state s_1 instead. After this sequence of 5 atomic actions, HER is performed for layer 0. The state s_1 can replace the TBD goal component in the extra hindsight transitions that were created and the reward would then be replaced with its appropriate value.

The completion of level 0's for loop represents a single action by level 1's policy π_1. Because its original subgoal g_0 was not achieved, π_1 will then record the hindsight state s_1 as the actual action that was taken. The transition $[s = s_0, a = s_1, r = -1, s' = s_1, g = Yellow\ Flag]$ will be added to level 1's replay buffer. The additional transition $[s = s_0, a = s_1, r = TBD, s' = s_1, g = TBD]$ will be stored until HER is performed for level 1.

Skipping ahead to when the robot is in state s_2, the high-level policy proposes subgoal g_2 and then decides to test the subgoal. The low-level policy then has $T = 5$ attempts without exploration noise to try to achieve g_2. In this scenario, the robot fails to achieve g_2. Indeed, state g_2 is too far away from s_2 to ever be achieved in 5 attempts. Because the subgoal was tested and the robot failed to achieve it, a penalty transition is created and passed to the replay buffer for level 1. Assuming the penalty is $-T = -5$, the transition would be of the form $[s = s_2, a = g_2, r = -5, s' = s_3, g = Yellow\ Flag]$. The robot will also create the regular and hindsight transitions showing that state s_3 is a possible subgoal state from s_2. The regular transition is again of the form $[s = s_2, a = s_3, r = -1, s' = s_3, g = Yellow\ Flag]$. The episode then finishes with the agent achieving its goal. After the episode, HER is performed for level 1 and then all actor-critic networks are updated using an off-policy RL algorithm, such as DDPG (Lillicrap et al. (2015)).

From this single episode, both levels have learned important information. Using hindsight, level 0 has learned the joint torques that can achieve a variety of states. Similarly, level 1 has learned a sequence of subgoal states $(s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4)$ that fits its time scale and can lead the robot to

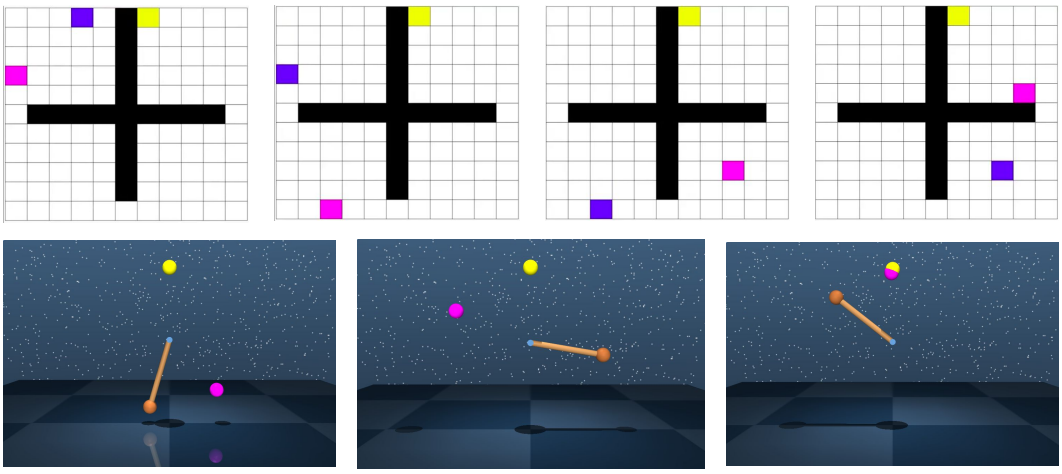

Figure 4: Episode sequences from the four rooms (Top) and inverted pendulum tasks (Bottom). In the four rooms task, the agent is the blue square, the goal is the yellow square, and the learned subgoal is the purple square. In the inverted pendulum task, the goal is the yellow sphere and the subgoal is the purple sphere.

a certain area in the goal space. The sequences of actions that each level is provided is short and thus can potentially be learned more quickly than a flat policy trying to learn a long sequence of atomic actions. Moreover, through subgoal testing, level 1 has also learned about subgoals that may be too ambitious. Using its UVFA, each level should be able to use this new information to better generalize to different goal states.

4 Experiments

4.1 Environments

We evaluated our framework in a variety of discrete and continuous tasks. The discrete tasks consisted of grid world environments. The continuous tasks consisted of simulated robotics environments developed in Mujoco (Todorov et al. (2012)). Figure 4 shows sample episode sequences from a four rooms grid world environment (Top) and the inverted pendulum environment (Bottom). A video showing our experiments can is available at `https://www.youtube.com/watch?v=i04QF7Yi50Y`.

4.2 States, Actions, and Rewards

For the grid world tasks, the state space was an integer value representing which square the blue square agent was in. The atomic actions available to the agent were $a_0 = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. Both the subgoal and end goal spaces were the same as the state space. The rewards at all levels of the hierarchy were either {-1,0} depending on whether a policy's action achieved its goal. For the grid world tasks, instead of testing subgoals, we simply used pessimistic Q-value initializations so subgoal state actions that are too far will always have poor Q-values.

For the simulated robotics tasks, the state space was the concatenation of all joint angles and joint velocities. The atomic actions available to the agent were vectors of joint torques. The subgoal action space was equal to the state space. The end goal space was either equal to the state space or a lower dimensional projection of the state space. The rewards granted at all levels were either $\{-T$,-1,0$\}$.

4.3 Results

For each task we compared the performance of agents using 1 and 2 subgoal layers with the flat agent. The flat agents used Q-learning with HER in the discrete tasks and DDPG with HER in the continuous tasks.

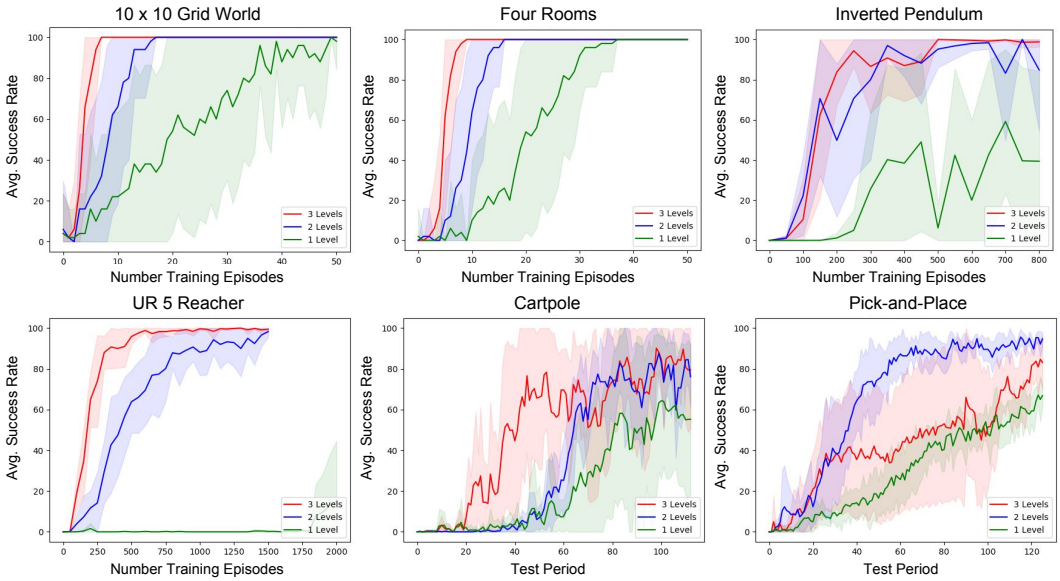

Figure 5: Figure presents results of our experiments. For each task, the average success rate is shown over time for a 2 subgoal layer agent (Red), a 1 subgoal layer agent (Blue), and a flat agent (Green). The error bars represent 1 standard deviation.

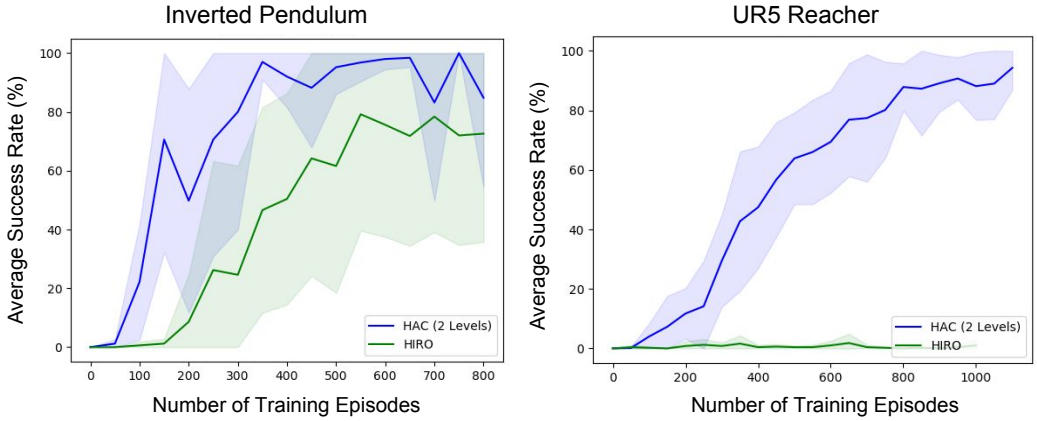

Figure 6: Figure compares the performance of HAC (2 Levels) and HIRO. The charts show the average success rate over 5 trials for each algorithm. The error bars represent 1 standard deviation.

In all tasks, our approach significantly outperformed the flat agent. Figure 6 shows the performance graphs for each agent in each task. Each chart plots the average episode success rate for a given level of training for the 2 subgoal layer agent (Red), 1 subgoal layer agent (Blue), and the flat agent (Green). For the discrete tasks, inverted pendulum, and UR Reacher task, the level of training is expressed in terms of the actual number of training episodes that have taken place. For the remaining tasks, the level of training is expressed in terms of test periods, in which each test period is separated by around 250 training episodes. For the discrete tasks, the charts averages data from 50 trial runs. The continuous tasks use data from 5-10 trial runs. Each chart also plots the areas that are 1 standard deviation from the mean success rate.

Our empirical results also support our claim that additional layers of hierarchy can improve sample efficiency because they can shorten the action sequence length that each policy needs to learn. In all of the discrete tasks and in the more challenging continuous tasks, such as UR5 Reacher and Cartpole, the agent using 2 subgoal layers outperformed the agent using 1 subgoal layer.

5 RELATED WORK

There have been a wide range of hierarchical reinforcement learning approaches that have been implemented. Most of these approaches either only work in discrete domains, require low-level controllers, or require a model of the environment [$Suttonet\,al.$ (1999), $Dietterich$ (1998), $McGovern\&Barto$ (2001), $Kulkarniet\,al.$ (2016), $Menacheet\,al.$ (2002), $Simseket\,al.$ (2005)]

There are a few other automated hierarchical RL techniques that can work in continuous domains. Konidaris & Barto (2009) proposed Skill-Chaining, a method that iteratively chains options from the end goal state to the start state. A key advantage that our framework has relative to Skill-Chaining is that our approach learns the subgoal options needed to reach a more distant goal state in parallel rather than incrementally, which can lead to improved sample efficiency. For instance, in the toy robot example provided above, agents using Skill-Chaining may use the experience to only learn an option to move from state s_3 to s_4. On the other hand, our approach can immediately begin to learn all of the low-level subgoal policies (i.e., $s_0 \rightarrow s_1, s_1 \rightarrow s_2, s_2 \rightarrow s_3, s_3 \rightarrow s_4$) in parallel. The Option-Critic architecture (Bacon et al. (2016)) and FeUdal Networks (FUN) (Vezhnevets et al. (2017)) do learn the policies within the hierarchy in parallel. However, these methods are limited to hierarchies with only two levels, whereas our framework can support an arbitrary number of levels. This enables agents using our approach to divide the original task into a parallel set of even shorter action sequences.

The approach taken by Feudal Reinforcement Learning (Dayan & Hinton (1993)) is probably the hierarchical RL algorithm most similar to ours. The method learns a top-down hierarchical policy, in which a "manager" proposes subgoals for a "worker" to achieve. Like our approach, Feudal RL thus divides the original RL problem into a parallel set of simpler RL problems. One key difference is how each method enforces policy length limits. Our framework uses an extra hyperparameter that limits the number of actions a policy can take to achieve a goal. Feudal RL chooses to break down the state space into increasingly fine regions and have each level in the hierarchy become an expert at that granularity of the state space. It is unclear how this strategy can be scaled to continuous state spaces that have several dimensions. Feudal RL also does not take advantage of learning from hindsight to boost sample efficiency.

There are also non-hierarchical methods that can accelerate backwards credit assignment, such as n-step TD or eligility traces (Sutton & Barto (1998)). The advantage that our algorithm has over these techniques is that they do not shorten the length of the policies that need to be learned. Further, our method does not exclude the use of these value estimation techniques. We only presented the 1-step TD version of our method, but it can be combined with n-step TD or eligibility traces.

6 CONCLUSION

We propose a new framework for effectively scaling reinforcement learning to long time horizon tasks involving sparse rewards. Instead of learning a single lengthy policy, our approach learns a hierarchy of limited policies that operate at different time scales. The policies are trained in parallel and end-to-end using Hindsight Experience Replay. Our results show that our approach can significantly improve sample efficiency in both discrete and continuous domains.

REFERENCES

Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5055–5065. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7090-hindsight-experience-replay.pdf`.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. *CoRR*, abs/1609.05140, 2016. URL `http://arxiv.org/abs/1609.05140`.

Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In S. J. Hanson, J. D. Cowan, and C. L. Giles (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.

Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *In Proceedings of the Fifteenth International Conference on Machine Learning*, pp. 118–126. Morgan Kaufmann, 1998.

George Dimitri Konidaris and Andrew G. Barto. Skill chaining : Skill discovery in continuous domains. 2009.

Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 29*, pp. 3675–3683. Curran Associates, Inc., 2016.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML*, volume 1, pp. 361–368, 2001.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *ECML*, 2002.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1312–1320, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/schaul15.html.

Özgür Simsek, Alicia P. Wolfe, and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, pp. 816–823, 2005. doi: 10.1145/1102351.1102454. URL http://doi.acm.org/10.1145/1102351.1102454.

R. S. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. *Artificial Intelligence Journal*, 112:181–211, 1999.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

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

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *CoRR*, abs/1703.01161, 2017. URL http://arxiv.org/abs/1703.01161.