IMPROVING LONG-HORIZON DECISION MAKING WITH HIERARCHICAL GOAL-CONDITIONED PLANNING

Kai-Chi Huang, Wei-Jen Ko
kchuang@cs.utexas.edu, wjko@cs.utexas.edu

ABSTRACT

Long-horizon decision making has been a hard challenge for both model-free and model-based reinforcement learning (RL) algorithms. This is especially true for domains that require function approximations, as compounding approximation errors will prevent either implicit (model-free) or explicit (model-based) reasoning into long-term future. However, the recent progress of goal-conditioned reinforcement learning provides a possibility of hierarchical planning that can reduce the propagation of modeling error. In this project, we learn a set of goal-conditioned policies as the local policies between waypoints, and then perform a dynamic-programming type model-based planning on top of that. Our experiments show that the performance of the final plan is sensitive to the quality of goal-conditioned policies, but we are still able to get positive results on a few relatively simple domains.[1]

1 INTRODUCTION

Long-horizon decision making has been a long-lasting challenge for reinforcement learning. Most current model-free methods struggle because the associating value functions or reactive policies could be too complicated to learn with a function approximator. Model-based reinforcement learning methods might perform better since one can perform explicit planning at decision-time. However, when we only have access to a learned approximated model, the compounding error will prevent it from making reasonable long-term prediction (for example, 50 or more steps into the future), and hence, long-term planning.

Actually, long-term prediction with arbitrary action sequences might be infeasible for some kinds of tasks. For example, if the dynamics of an environment is combinatorially complicated (ex: Game of Go) or is unpredictable due to high stochasticity, predicting and planning into long-term future could be inherently infeasible. Fortunately, in some cases it can also be surprisingly easy when executing certain kinds of closed-loop policies, for example, goal-reaching policies that aim to reach a certain goal state. This is illustrated by the following example: Imagine that we want to predict the position of a person after walking forward for 100 steps. This could be a hard task because any noise in step sizes or the direction can result in very different final position. However, if instead of walking forward for 100 steps, he aims to walks toward a door roughly 100 steps away. In this case, the final result will be easy to predict – with high probability he will end up in front of the door, since any minor errors during the trajectory will likely be corrected.

In this project, we designed a hierarchical algorithm that combines goal-conditioned RL and model-based methods to enable long-term planning. By planning at the level of waypoints and using goal-conditioned RL policy as the local policies between waypoints, it partly mitigates the compounding error problem of inaccurate models.

2 RELATED WORK

Multitask RL and Goal-conditional RL Multitask RL is a setting of reinforcement learning where several tasks share the same states and transition functions but have different reward functions.

[1]The source codes and the video link are located at `https://github.com/kevin00036/ut-rl2019`

One recent formulation is UVFA(Schaul et al., 2015), which uses a single function approximator to represent the task-specific value functions $V(s, t|\theta)$ for all tasks t. Successor features(Barreto et al., 2017)(also in (Abbeel & Ng, 2004)) is another way to represent multi-task value functions by calculating the discounted sum of features in the future.

Our project is based on Goal-conditional RL, which is a special case of multi-task RL where each task is defined by reaching a specific goal state. The main benefit of this setting is that the tasks are naturally defined, and often have some extra properties that can be used to accelerate learning. For example, Hindsight Experience Replay(Andrychowicz et al., 2017) is a technique that can use arbitrary off-policy trajectories to learn the value function of many goals at a time. In Temporal Difference Model(Pong et al., 2018), they showed that the shortest distance between two goals can be learned using goal-conditioned RL with $V(s, g) = -\text{distance}(s, g)$. However, one major limitation of current methods are that they only work well on "navigation-like" tasks, i.e. there is a natural distance metric between states, and moving between nearby states is not too hard. The tasks used in (Andrychowicz et al., 2017; Pong et al., 2018; Plappert et al., 2018; Eysenbach et al., 2019) are indeed within this category, for example, 2D/3D navigation, robot arm manipulation where the goals are reaching a specific position/velocity. In environments with irreversible or not easily reversible actions, or where most goals are unreachable, these methods could fail miserably.

One previous work worth noting is (Jaderberg et al., 2017), which learned unsupervised auxiliary subtasks, for example, pixel controlling, to help improve performance of the "main" task. This is particularly of interest because they learned multiple tasks even though there is a "main" task known to the agent, and the pixel-controlling auxiliary subtask also gave us some inspiration about the partial-goal specification. However, they didn't make any explicit use of these auxiliary subtasks other than improving the feature representation (and hence helping the main task). On the contrary, we plan to explicit use the learned goal-conditioned values/policies in high-level model-based planning.

Hierarchical RL / Temporal abstraction Temporal abstraction has been studied for a long time, for example, the option formulation studied in (Sutton et al., 1999) where the main policy chooses options (sub-policies) instead of primitive actions. Feudal RL (Dayan & Hinton, 1992) is a two-level hierarchical RL architecture where the higher level controller assigns a goal, and the lower level controller uses the primitive actions to achieve the goals assigned by the higher level controller. FeUdal Networks (Vezhnevets et al., 2017) is a recent implementation of it that uses neural networks for both controllers and learn them in an end-to-end fashion. (Nachum et al., 2018) is also a deep neural network-based hierarchical RL algorithm, but it improves data efficiency by effectively utilizing off-policy experiences.

Most of these hierarchical RL methods are model free, but in RMAX+MAXQ (Jong & Stone, 2008) they combined MAXQ (temporal abstraction) with RMAX (a model-based RL algorithm). Our ultimate goal is to similarly combine temporal abstraction and model-based RL, but with more powerful function approximators and decision-time planning algorithms like MCTS.

Model-based RL Model-based RL methods have the potential to improve both data efficiency and long-horizon reasoning ability upon its model-free counterpart.

The first kind of model-based RL methods is the Dyna(Sutton, 1990)-like methods. They first learn an approximate transition model using trajectories, and then samples transitions from this model to update the value/policy using a model-free RL algorithm. A more recent work(Kaiser et al., 2020) uses deep neural networks for the transition model and showed competitive performance and data efficiency in Atari games. However, these methods only improves data efficiency.

The other series of methods are decision-time planning methods that uses a transition model to reason about possible future events when making actions. The most prominent example recently is the success of AlphaGo(Silver et al., 2016), where Monte-Carlo Tree Search (MCTS, a kind of decision-time planning method) is used in conjunction with neural network-based value approximator. Before AlphaGo, MCTS has already been widely studied(Browne et al., 2012), while not necessarily combined with RL.

(Wang et al., 2020) provides a comprehensive benchmark results on many model-based/model-free RL algorithms and traditional model-based controllers on some OpenAI Gym(Brockman et al.,

2016) robotic manipulation tasks . We may compare our results with it to get some idea about the relative performances.

Deep RL The use of deep neural networks as the function approximator has allowed us to apply RL algorithms on tasks with much higher dimensional state spaces. DQN(Mnih et al., 2013) is probably the first large-scale success, which approximates the Q-value function with convolutional neural networks and achieved near-human skill on 51 Atari games. However, deep RL comes with a bunch of issues such as training instability. Many improvements have been made after that, and an example of combining them is the Rainbow-DQN algorithm(Hessel et al., 2018).

In addition to value-based method like DQN, there are also deep neural network versions of policy gradient methods, like A3C(Mnih et al., 2016) and PPO(Schulman et al., 2017). There are also some algorithms that deals with continuous actions, such as DDPG(Lillicrap et al., 2015) and Soft Actor-Critic(Haarnoja et al., 2018).

We will mostly use these as base learning algorithms when solving high-dimensional tasks.

Combining Goal-conditioned RL with Planning (Eysenbach et al., 2019) is probably the closest work to our project. In their paper, goal-conditioned RL are used as the local policy in the classical planning algorithm. They learned the goal-conditioned value function as the negative distance as in (Pong et al., 2018), but they maintained a distribution of Q-values (Mavrin et al., 2019) instead of a single expected value to increase reliability of the distances. However, the use of classical planning means their method is limited to navigation-like tasks where the ultimate goal is to reach some goal state. We aim to generalize it to RL problems with general reward functions, so model-based planning methods will be needed instead of classical planning algorithms.

3 Our Method

3.1 Broad structure

Our approach of hierarchical planning consists of two parts: (1) Low-level goal-conditioned policies and (2) High-level waypoint-based planning. The waypoint-based planning is used to generate an optimal path through a set of waypoints, and the goal-conditioned policies are used to connect these waypoints along the path (i.e. execute the plan).

3.2 Goal-conditioned policies

Similar to (Eysenbach et al., 2019), we used goal-conditioned reinforcement learning as the local policies. Goal-conditioned reinforcement learning is a kind of multi-task reinforcement learning problem with intrinsic reward functions. Specifically, given an MDP $M = (S, A, P, \gamma, r)$, that is, an MDP without a reward function, and a *goal state* $g \in S$, we can define a goal-specific reward function $r_g : S \to \mathbb{R}$ such that $r_g(s) = \mathbf{1}[s = g]$, that is, a reward of 1 is given upon reaching g, and 0 otherwise. For continuous state spaces, we usually relax this to $r_g(s) = \mathbf{1}[\|s - g\|_2 < \epsilon]$. The reward function is considered *intrinsic* because it doesn't depend on the environment reward. Note that here r_g is only a function of state because the reward is given upon reaching a state.

After adding this reward function, $M_g = (S, A, P, \gamma, r_g)$ becomes a standard MDP. We further define that the episode terminates when either $r_g = 1$ is received or the original MDP terminates. Since this is a standard MDP, we can define the goal-conditioned value functions $V_g^\pi(s)$, the optimal value function $V_g^*(s)$, and optimal policies $\pi_g(a|s)$ in the traditional way and learn them using any RL algorithm.

In particular, the goal-conditioned value function $V_g^\pi(s)$ has an interesting interpretation as the "discounted probability" of reaching g from s following π. Specifically, we can write the value function as

$$V_g^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^{t+1} r_g(s_{t+1}, g) \mid |s_0 = s \right] = \sum_{t=1}^{\infty} \gamma^t \mathbb{P}_\pi[r_g(s_t) = 1 | s_0 = s]. = \mathbb{E}_\pi[\gamma^T],$$

where T is a random variable denoting the number of steps before reaching g (or ∞ if g is never reached). Thus, $V_g^\pi(s)$ can be seen as (roughly) the *expected distance* between s and g.

This is similar to the goal-conditioned value function in (Eysenbach et al., 2019), but they used a reward of -1 for each time step before reaching g, so their goal-conditioned value corresponds to exactly the expected distance. Compared to this, our formulation has several advantages: (1) It naturally combines the effects of distance and reaching probability. For example, if $\gamma = 0.5$, then a value of 0.25 could either mean (a) π will reach g in exactly 2 steps with certainty, or (b) π will reach g in 1 step with probability 0.5 or will never reach g otherwise, or (c) other possible mixtures. (2) The exponential discounting implicitly emphasizes closer events. For example, reaching the goal with 1 or 10 steps could have a large differences in the values, but reaching in 100 or 200 steps might both yields values close to 0. This is a desirable behavior in goal-conditioned RL since we usually care most about nearby goals. (3) The value function is always bounded within $[0, 1]$. In contrast, the value function in (Eysenbach et al., 2019) is unbounded below and required a distributional function approximator to represent properly. (4) As we will see later, it fits naturally into the higher-level planning algorithm.

The whole goal-conditioned RL problem is a multi-task consists of a whole sets of goal-conditioned MDPs, i.e. $\{M_g : g \in S\}$. To enable generalization across different goals, we use UVFA (Schaul et al., 2015) style function approximator, that is, to approximate all Q-value functions for different g with a single function approximator:

$$Q_g(s, a) \approx Q^\theta(s, a, g).$$

In our experiments, we used 1-step Double DQN (Mnih et al., 2013) with a technique from TD3 (Fujimoto et al., 2018) that further suppresses the maximization bias (see Section 3.4).

The training trajectories consists of two parts: the on-policy part and the off-policy part. For the on-policy part, when each episode starts with initial state s_0, a goal state g is randomly sampled, and then we execute our current goal-conditioned policy $\pi_g(s, a)$ until either the goal is reached or the environment terminates the episode. Let the resulting trajectory be $(s_0, a_0, s_1, a_1, \dots, s_n)$, then we calculate the goal-conditioned reward $r_i = r_g(s_{i+1}, g)$ and store $(s_i, a_i, r_i, s_{i+1}, g)$ into the replay buffer, and update later as (informally, will state the exact update rule in Section 3.4)

$$Q(s_i, a_i, g) \leftarrow Q(s_i, a_i, g) + \alpha(r_i + \gamma \max_a Q(s_{i+1}, a, g) - Q(s_i, a_i, g)).$$

For the off-policy parts, we follow the sampling procedure in Hindsight Experience Replay (HER) (Andrychowicz et al., 2017). Specifically, for each s_i we randomly sample a number k from $[1, t]$ and set the virtual goal g' as the future state s_{i+k} in the same trajectory, and then store $(s_i, a_i, r_{g'}(s_{i+1}, g'), g')$ in to the replay buffer. This off-policy update assures sufficient positive examples even when the goal-conditioned policy fails to reach the originally proposed goal g. However, relying solely on off-policy updates will make the state distribution of updates too far away from the on-policy distribution, and as a result the learning will become biased and unstable. In our experiments, we sample on-policy and off-policy transitions with equal probability.

3.3 WAYPOINT-BASED PLANNING

Similar to (Eysenbach et al., 2019), we perform waypoint-based planning on top of the goal-conditioned policies. However, (Eysenbach et al., 2019) focused on navigation problems where the goal is to reach some ultimate goal state, so the used classical planning algorithms (for example, Dijkstra's shortest path algorithm) to find the optimal path. On the other hand, out goal is to maximize the external reward function r given by the environment, as in any classical RL problems.

To accomplish this, we perform model-based planning on a higher level. That is, we plan a path of waypoints instead of primitive transitions.

Specifically, for a start state s_0, we want to find a (probably infinite) path $p = (s_0, s_1, s_2, \dots)$ with $s_0 \in W$ such that its expected external return is maximized. $W \subseteq S$ is a set of waypoints states, and in our experiments we randomly sample a number of previously seen states from the replay buffer as in (Eysenbach et al., 2019).

The plan $p = (s_0, s_1, s_2, \dots)$ is executed with goal-conditioned policies. That is, we first execute π_{s_1} until reaching s_1, and then execute π_{s_2} until reaching s_2, and so on. If π_{s_i} will never reach s_i, π_{s_i} will be run indefinitely. In this way, the expected external return $R(p)$ is well defined and can be calculated as

$$R(p) = \sum_{i=0}^{\infty} \left(\prod_{j=0}^{i-1} V_{s_{j+1}(s_j)} \right) R_{s_{i+1}}(s_i) = R_{s_1}(s_0) + V_{s_1}(s_0) R(p'), \tag{1}$$

where $p' = (s_1, s_2, \dots)$ is the sub-plan starting from s_1, and $R_g(s)$ is the expected (discounted) return when executing π_g from s until either reaching g or when the episode is terminated by the environment.

To avoid confusion, the notations with states as subscripts (ex: $V_g(s), R_g(s), \pi_g$) corresponds to goal-conditioned quantities, while those without subscripts (ex: $V(s), R(s)$) are quantities related to external reward functions.

Given a set of possible waypoints W, the value (i.e. expected external return) $V(s)$ of the optimal plan starting from s can be calculated by a formula similar to Bellman's equation:

$$V(s) = \max_{s' \in W} \left(R_{s'}(s) + V_{s'}(s) V(s') \right) \tag{2}$$

As in classical dynamic programming, by iterating through the following update rule, $\hat{V}(s)$ will converge to the optimal value $V(s)$.

$$\hat{V}(s) \leftarrow \max_{s' \in W} \left(R_{s'}(s) + V_{s'}(s) \hat{V}(s') \right) \tag{3}$$

As shown in the derivation (Appendix B), the goal-conditioned value function naturally fits into the formula as the multi-step discount factor, as long as we use the external (i.e. provided by the environment) discounting factor γ when learning goal-conditioned values.

Interestingly, this update rule resembles the dynamic programming approach of model-based planning. The $R_g(s)$ corresponds to *immediate reward model*, and the goal-conditioned value $V_g(s)$ plays the role of (transition-dependent) *discount factor*. The *transition probability* doesn't have an analogical variable here; all transitions $(s \to s')$ are possible, but unlikely transitions won't make much difference due to small $V_g(s)$. Thus, the Q-network and R-network combined can be seen as a temporally-extended dynamic model, and this waypoint-based planning algorithm is essentially performing dynamic programming on that model.

Just like that $V_g(s)$ can be learned with goal-conditioned RL, $R_g(s)$ can also be learned with standard policy evaluation algorithms. In particular, we used 1-step TD evaluation with on-policy transition (s, a, r, s'):

$$\hat{R}_g(s) \leftarrow \begin{cases} \hat{R}_g(s) + \alpha(r - \hat{R}_g(s)), & s' \text{ is a termination state} \\ \hat{R}_g(s) + \alpha(r + \gamma \hat{R}_g(s') - \hat{R}_g(s)), & \text{otherwise} \end{cases} \tag{4}$$

Note that for the purpose of learning R_g, termination means either g is reached or the episode is terminated by the environment. Also, r here is the *external* reward. In our experiments, $\hat{R}_g(s)$ is also approximated by a neural network.

It's worth noting that if $R_g(s), V_g(s)$ are learned exactly, $\pi_g(s)$ is optimal, the environment is deterministic, no two actions will lead to the same state, and with appropriate waypoint sets W, this planning algorithm will guarantee to yield the optimal value function in the traditional sense. This is because when the environment is deterministic, an optimal policy π^* starting from s_0 will always generate the unique trajectory $(s_0, a_0, s_1, a_1, \dots)$. If all the states $s_0, s_1, \dots$ are in W, then our planning algorithm will find the best path $p^* = (s_0, s_1, \dots)$. The optimal goal-conditioned policies will be deterministically $\pi^*_{s_{i+1}}(s_i) = a_i$ and guaranteed to reach s_{i+1} in exactly 1 time step. This means the overall algorithm will follow the optimal trajectory $(s_0, a_0, s_1, a_1, \dots)$ and then get the optimal expected return.

In general stochastic environments, this guarantee might not hold, but we assume that it will still achieve reasonable performance when the environments are structured.

3.4 MAXIMIZATION BIAS

As pointed out in (Van Hasselt et al., 2016; Fujimoto et al., 2018), most value-based RL algorithms with bootstrapping are susceptible to the problem of *maximization bias* i.e. they tend to over-estimate the value. For example, in the Q-learning algorithm, the update target is

$$Q_{\text{target}}(s, a) = r + \gamma \max_{a'} \hat{Q}(s', a') \tag{5}$$

The maximization over a' will tend to select actions whose estimated action-value is incorrectly high due to approximation errors.

For usual RL settings, this might only affects its performance to some limited extent, because the absolute error in Q-value might not severely impact the quality of resulting policy as long as the *relative order* Q-value of different actions is accurate. However, our algorithm is very sensitive to this maximization bias because any *absolute* error in goal-conditioned values $V_g(s)$ will cause biases to the planning phase.

To mitigate this, we applied a correction suggested by (Fujimoto et al., 2018):

$$Q_{\text{target}}(s, a) = r + \gamma \cdot \min(Q(s', a'), Q'(s', a')),\ a' = \arg\max_b Q(s', b) \tag{6}$$

where Q' is a separate slowly-updating *target network*.

The maximization bias can also happen in the planning phase. Thus, we added a similar correction that separates the selection of optimal s' and its value estimation:

$$\hat{V}(s) \leftarrow \tilde{R}_{s'}(s) + \tilde{V}_{s'}(s)\hat{V}(s') \tag{7}$$

where

$$\begin{aligned} s' &= \arg\max_u \left(R_{s'}(s) + V_{s'}(s)\hat{V}(s') \right) \\ \tilde{R}_{s'}(s) &= \min(R_{s'}(s), R'_{s'}(s)) \\ \tilde{V}_{s'}(s) &= \min(V_{s'}(s), V'_{s'}(s)), \end{aligned}$$

and $R'_{s'}, V'_{s'}$ are again corresponding target networks.

4 EXPERIMENTS

4.1 GOAL-CONDITIONED RL

Environments We mainly did our experiments on OpenAI Gym (Brockman et al., 2016) environments `CartPole-v1` and `MountainCar-v0`. Both of them are discrete-action environments. Actually, we also implemented a continuous-action version of goal-conditioned RL algorithm based on TD3 and tried on higher-dimensional manipulation tasks (ex: `HalfCheetah-v3`), but it fails to reach the goals with significant probability, so we do not include them in the report.

Following the traditional evaluation scheme of RL algorithms, we trained our models using discounting factor $\gamma = 0.99$, but evaluate the final (external) performance with undiscounted return. We will make clear of this discrepancy in the results.

Also, OpenAI gym environments has an automatic timeout (ex: in 200 or 500 steps). To maintain the Markovity, when making updates we treat terminating transitions due to timeout as non-terminating transitions.

Model details Following [TD3], for the Q-network, R-network, and their target networks, we used a two-layer feed-forward neural networks with 400, 300 neurons in each hidden layer, respectively. ReLU activation functions are applied after each layer except for the final output.

The neural networks are optimized with Adam optimizer with a fixed learning rate of 10^{-3}. We make one TD-update to both Q and R networks with a minibatch of 32 after each environment step. Target networks are updated after each TD-update with rate $\tau = 5 \times 10^{-3}$.

Trajectory sampling When collecting trajectory, we start each episode with initial state s_0 given by the environment (i.e. sampled from the initial state distribution of the environment). The goal state g is sampled as:

$$g_i = s_i + d \cdot \delta_i z_i$$

with each component sampled independently, where $d \in \mathrm{Unif}[1, 10]$ is the shared step-size parameter, δ_i is the average magnitude of change on that component caused by random actions, and $z_i \in \mathcal{N}(0, 1)$ is the standard Gaussian noise. Intuitively, this corresponds to sampling a goal roughly d steps away from s.

After g is decided, we run the policy π_g with ϵ-greedy exploration ($\epsilon = 0.05$) until either g is reached or the episode is terminated by the environment. Goal g is considered reached when

$$\|(s - g)/\delta\|_2 = \sum_{i=1}^{d} \left(\frac{s_i - g_i}{\delta_i} \right)^2 \leq 1,$$

that is, when s is within roughly 1 step away from g.

When testing the performance of the policies, we sample s and g in the same way, but we run the greedy policy $\pi_g(s) = \arg\max_a \hat{Q}_g(s, a)$ *without* ϵ-greedy exploration.

Results We ran the algorithm on `CartPole-v1` and `MountainCar-v0`. Each experiment is repeated 5 times with different random seeds. The solid line in each figure represent the mean over 5 runs, and the shaded area indicates the ± 1 standard deviation range. Note that in all of the experiments we update the neural networks after every environment step, so the x-axis can be interpreted as both training updates and training environment steps.

Figure 1 and 2 shows the average *goal-conditioned discounted* return during the learning process. We can see that for both environments, the return increases steadily, from about 0.04 to 0.12 for `CartPole-v1` and about 0.2 to 0.6 in `MountainCar-v0`. Although the final return value is still not very high, this is mainly because we randomly sampled goals roughly 1 to 10 steps away, and the farther goals are extremely hard or impossible to reach.

4.2 WAYPOINT-BASED PLANNING

During the learning of goal-conditioned policies, we periodically test the performance of waypoint-based planning.

In each episode, we first sample $N = 1000$ random states from the replay buffer to form the waypoint set W, and perform the pre-planning with current Q-networks and R-networks.

After the pre-planning is completed, we start to interact with the environment from the initial state s_0. Every time we get a state s, we perform an incremental planning step to get the action a according to Algorithm 2. That is, we are effectively re-planning at every step.

Results Figure 3 and 4 shows the average *undiscounted external* return for both environments. A comparison with a standard RL algorithm is also included. Here we used TD3-optimized Deep Q-learning as our standard RL algorithm. We used exactly the same network architecture as the Q-network in our goal-conditioned RL implementation, except that we always feed the all-zero vector as the goal.

We can see that both algorithms successfully solved `CartPole-v1`. The planning method learns roughly as fast as the standard RL algorithm. This is expected since model-free RL is known to work well on `CartPole-v1`. On the other hand, it is a little bit surprising that both algorithms failed completely on `MountainCar-v0`. As explained in the following section, this is mainly due to the difficulty to reach the goal even once with near-random policies.

While this preliminary result shows that out planning method has the potential to perform at least as well as standard RL algorithms, at this stage its applicability is still severely limited by the difficulty to learn good goal-conditioned policies.

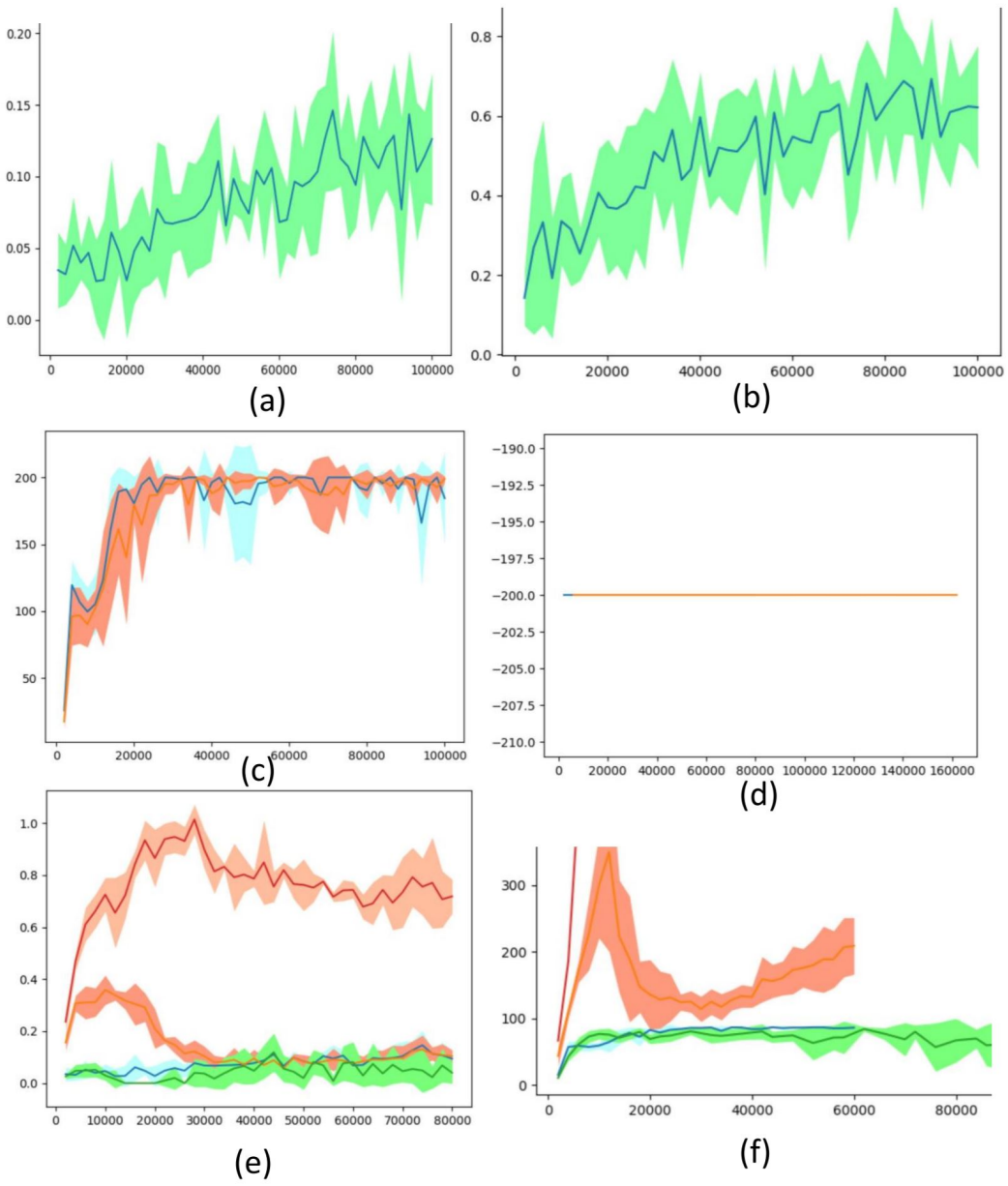

Figure 1: In all the figures X-axis is the number of training steps. (a) Cartpole. (b) Mountaincar. For (a) and (b), Y-axis is the discounted goal-conditioned return (b) Mountaincar. Y-axis is the discounted goal-conditioned return (c) Cartpole. (d) Mountaincar. For (c) and (d), Y-axis is the undiscounted external return. Blue line is Standard RL, Orange line is Waypoint-based planning. (e) Discounted goal-conditioned returns for Carpole. UVFA with TD3 (Blue line is True. Orange line is Estimated.) and without (Green line is True. Red line is Estimate.)(f) Discounted external returns Waypoint-based planning with TD3(Blue line is True. Orange line is Estimated.) and without (Green line is True. Red line is Estimated.)

4.3 EFFECTS OF MAXIMIZATION BIAS

We investigate the effect of maximization bias on the quality of both goal-conditioned and final planning-based policies.

To see this, we ran our experiments both with and without the TD3-styled optimization described in Section 3.4. We evaluate the *true* and *estimated* values for both goal-conditioned and external value function. The *true* values are evaluated with repeated Monte-Carlo simulation, and the *estimated* values are calculated as (1) For goal-conditioned value, the output of function approximator $\hat{V} = \max_a Q_\theta(s_0, a, g)$, and (2) for external value function, the final value function of the wapoint-based planning algorithm $\hat{V} = V(s_0)$.

From Figure 5 and 6, we can see that the *true* values of both goal-conditioned and external value functions are higher and more stable with TD3-styled optimization enabled. On the other hand, the *estimated* values are much higher without TD3-styled optimization.

After a closer look at Figure 5, we can also discover that in the case of TD3-styled optimization enabled, the gap between true and estmated goal-conditioned values are very small, i.e. within statistical error range, while the gap is very large without the optimization. This means the TD3-styled optimization indeed overcomes the maximization bias problem for the goal-conditioned RL.

However, in Figure 6, we can see that for the case of external values with waypoint-based planning, the gap with TD3-styled optimization is smaller but still exists. One possible explanation is that in the dynamic programming iteration will magnify existing errors and make the maximization bias problem worse.

We believe that to further suppress this bias, the most effective way is to start with more accurate Q_θ and R_ϕ approximators. One possible way as suggested in (Eysenbach et al., 2019) is to use distributional neural networks or use an ensemble of networks.

5 DISCUSSIONS & FUTURE WORKS

It turned out that our experiment results are less successful than we anticipated. We believe that the main reason is the difficulty to learn a reasonably good goal-conditioned value function on high-dimensional environments and the still existing problem of maximization bias in the planning phase.

Goal-selection In high-dimensional environments, generating a reachable goal is a nontrivial task. Currently, we only use goal states from existing trajectories in off-policy experience replay, but not when evaluating the goal-conditioned policies. In high-dimensional environments, randomly sampling a goal – even just a random deviation from the current state – can result in invalid or unreachable goal states with very high probability. To deal with this problem, we can either train a generative model to generate plausible goal states, or design a strategy to retrieve reachable goal states from previously visited states (replay buffer).

Exploration problem In `MountainCar-v0`, we are able to learn reasonably good goal-conditioned policies, but failed to optimize the external return because it didn't manage to obtain any meaningful external reward (i.e. reach the top-right goal). Actually, MountainCar is indeed a hard-exploration problem, but traditional RL algorithms (ex: Q-learning with tile coding and ϵ-greedy exploration) can accidentally reach the goal because their immature policies usually tend to repeat a same actions for multiple time steps.

There are existing works (ex: count-based exploration) that deal with extremely sparse-reward tasks, one possible direction is to incorporate these methods into goal-conditioned RL. Also, although most of the existing work in goal-conditioned RL assume the reward-agnostic setting (i.e. operate on MDP\R), in our case the external reward is known in advance and is actually our final objective for planning. It would be interesting if we can use the external reward to guide the goal-conditioned learning, for example, focus on the goals related to high external reward.

Partial-goal specification Intuitively, it should be easier to reach a *partial goal* than a fully-specified goal state. For example, In MountainCar, a state (x, v) consists of the position x and

velocity v of the car. A fully-specified goal could then be $g = (0.2, 0.6)$. However, sometimes we only care about the position but not the velocity, so in this case we can set the goal as $g' = (0.2, \cdot)$, where $\cdot$ means it can be any value.

Although the original UVFA (Schaul et al., 2015) setting admits arbitrary goal functions, and the HER (Andrychowicz et al., 2017) also uses a different goal space than the full observation, but they both require the goal space to be manually defined. Most environments (ex: CartPole and MountainCar) do not provide this information, so the algorithm will need to figure out the goals by itself. Extending our algorithm to partially-specified goals will require at least substantial modification because it will involve planning in the space of partially-observable belief states.

REFERENCES

Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *ICML*, 2004.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *NIPS*, 2017.

André Barreto, Will Dabney, Rémi Munos, Jonathan J. Hunt, Tom Schaul, Hado van Hasselt, and David Silver. Successor features for transfer in reinforcement learning. In *NIPS*, 2017.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Benchmarking model-based reinforcement learning. In *arXiv*, 2016.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, and Philipp R. A survey of monte carlo tree search methods. In *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, 2012.

Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *NIPS*, 1992.

Benjamin Eysenbach, Ruslan Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. In *NIPS*, 2019.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *AAAI*, 2018.

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *ICLR*, 2017.

Nicholas K. Jong and Peter Stone. Hierarchical model-based reinforcement learning: Rmax + maxq. In *ICML*, 2008.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, and Henryk Michalewski George Tucker. Model-based reinforcement learning for atari. In *ICLR*, 2020.

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. In *ICLR*, 2015.

Borislav Mavrin, Shangtong Zhang, Hengshuai Yao, Linglong Kong, Kaiwen Wu, and Yaoliang Yu. Distributional reinforcement learning for efficient exploration. In *ICML*, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS DL workshop*, 2013.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *NIPS*, 2018.

Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research. In *arXiv*, 2018.

Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control. In *ICLR*, 2018.

Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pp. 1312–1320, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *arxiv*, 2017.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. In *Nature*, 2016.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICM:*, 1990.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learnin. In *Artificial Intelligence*, 1999.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *ICML*, 2017.

Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. In *ICLR submission*, 2020.

A ALGORITHMS

Initialize UVFA networks $Q_\theta, Q_{\theta'}, R_\phi, R_{\phi'}$
Initialize replay buffer $\mathcal{B}$
for episode $= 1$ *to* M **do**
  Sample initial state s_0 from the environment
  Sample a goal state g
  while *not terminated and* g *not reached* **do**
    Sample an action $a_t \sim \pi_g(s_t)$ with ϵ-greedy exploration
    Execute a_t and observe environmental reward r_t and a new state s_{t+1}
    Make an update step
  end
  for $t = 0$ *to* $T - 1$ **do**
    With probability 0.5, let $g_t = g$
    Else, let $t' = \mathrm{Unif}[t + 1, \min(t + K, T)]$, and $g_t = s_{t'}$
    Calculate the goal-conditioned reward $r_{g_t,t} = \mathbf{1}[s_{t+1}$ reached $g_t]$
    Store transition $(s_t, a_t, r_t, r_{g_t,t}, s_{t+1}, g_t)$ into $\mathcal{B}$
  end
end
return $Q_\theta, Q_{\theta'}, R_\phi, R_{\phi'}$

Update step:
Sample a minibatch of transitions (s, a, r, r_g, s', g) from $\mathcal{B}$
$a' = \arg\max_b Q_\theta(s', b, g)$
$\tilde{Q}(s', g, a') = \min(Q_\theta(s', g, a'), Q_{\theta'}(s', g, a'))$
$\tilde{R}(s', g, a') = \min(R_\theta(s', g, a'), R_{\theta'}(s', g, a'))$
Calculate the termination signal $d = \mathbf{1}[$episode terminated $\vee\, r_g = 1]$
Calculate the loss:
$L_Q = \sum(r_g + \gamma\tilde{Q}(s', g, a') \cdot (1 - d) - Q_\theta(s, g, a))^2$
$L_R = \sum(r + \gamma\tilde{R}(s', g, a') \cdot (1 - d) - R_\theta(s, g, a))^2 \cdot \mathbf{1}[a = \pi_g(s)]$
Update $\theta \leftarrow \theta + \alpha\nabla_\theta L_Q$
Update $\phi \leftarrow \phi + \alpha\nabla_\phi L_R$
Update target networks $\theta' \leftarrow \tau\theta + (1 - \tau)\theta', \phi' \leftarrow \tau\phi + (1 - \tau)\phi'$

Algorithm 1: Goal-conditioned Reinforcement Learning

Pre-plan:
Given a set of waypoint states $W \subseteq S$
Given Q-networks $Q_\theta, Q_{\theta'}$ and R-networks $R_\phi, R_{\phi'}$
Initialize the value functions $V(s) = 0, \forall s \in W$
For $s, g \in W$, calculate:
$a_{s,g} = \arg\max_{a'} Q_\theta(s, a', g)$, $R_g(s) = R_\phi(s, a_{s,g}, g)$, and $V_g(s) = Q_\theta(s, a_{s,g}, g)$
for iter $= 1$ *to* M **do**
  for *each* $s \in W$ **do**
    Let $s_m = \arg\max_{s'}[R_{s'}(s) + V_{s'}(s)V(s')]$
    Let $V_{\text{new}} = \tilde{R}_{s_m}(s) + \tilde{V}_{s_m}(s)V(s_m)$
  end
  Update $V \leftarrow V_{\text{new}}$
end

Get an action:
Given a start state $s \in S$
For $g \in W$, calculate:
$a_{s,g} = \arg\max_{a'} Q_\theta(s, a', g)$, $R_g(s) = R_\phi(s, a_{s,g}, g)$, and $V_g(s) = Q_\theta(s, a_{s,g}, g)$
Let $s_{\text{next}} = \arg\max_{s'}[R_{s'}(s) + V_{s'}(s)V(s')]$
return $\pi_{s_{\text{next}}}(s)$

Algorithm 2: Waypoint-based Planning

B DERIVATION OF THE PLANNING UPDATE RULE

Lemma 1 Let $p = (s_0, s_1, s_2, \dots)$ be a plan with $s_i \in W$, and we execute the goal-conditioned policy $\pi_{s_{i+1}}$ to reach s_{i+1} from s_i. Then the expected discounted external return of this plan is given by

$$R(p) = R_{s_1}(s_0) + V_{s_1}(s_0)R(p'),$$

where

- $R_g(s)$ is the expected discounted *external* return by executing π_g from s until either g is reached or the episode is terminated.
- $V_g(s)$ is the goal-conditioned value function, i.e. the expected discounted *internal* return by executing π_g from s until either g is reached or the episode is terminated.
- $p' = (s_1, s_2, \dots)$ is the sub-plan starting from s_1.

Proof. Case 1: Assume π_{s_1} reaches s_1 in exactly T steps. In this case, the external return obtained before reaching s_1 is exactly

$$\sum_{t=0}^{T-1} \gamma^t r_t.$$

After reaching s_1, the execution just follows $p' = (s_1, s_2, \dots)$ and is independent of the events before reaching s_1. Hence, the expected discounted return is $\gamma^T R(p')$ since every reward obtained is T time steps later than if we started in s_1 at $t = 0$. Thus, the total expected reward conditioned on that π_{s_1} reaches s_1 in exactly T steps is

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right] + \gamma^T R(p')$$

Case 2: Assume π_{s_1} is terminated by the environment at time step T without ever reaching s_1.

In this case, the total expected reward is

$$\mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right]$$

If we define $\gamma^T = 0$ in Case 2 and take the expectation over all cases, then we will have the overall expected return being

$$R(p) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right] + \mathbb{E}[\gamma^T] R(p'),$$

with T now being a random variable.

Now, observe that by definition,

$$R_{s_1}(s_0) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right],$$

and

$$V_{s_1}(s_0) = \mathbb{E}[\gamma^t]$$

, so we arrived at the conclusion

$$R(p) = R_{s_1}(s_0) + V_{s_1}(s_0) R(p'),$$

□