Abstract: A key challenge in multi-goal policy learning for robotic manipulation is scaling up to a large number of qualitatively different manipulation problems within a single domain. For example, very few methods exist (if any) for learning a goal-conditioned policy that can reliably construct any one of hundreds of different block structures. This paper demonstrates that a simple hierarchical approach can work well as long as we are willing to provide the system with a high-level discrete representation of the domain. We compare against standard baselines and show that the method can be used to learn a single goal-conditioned policy that can build any one of more than a hundred different block structures reliably, both in simulation and on a physical system.

Keywords: Multi-goal, Policy Learning, Manipulation

1 Introduction

Multi goal policy learning is important to robotic manipulation because it offers the possibility that an agent could learn to solve a large number of different manipulation problems in a single domain. For example, in a block construction domain, we would like to learn a set of policies that could build any structure shown in Figure 1. Unfortunately, standard methods for multi-goal policy learning do not scale up well to problem domains like that shown in Figure 1. Standard goal conditioned policy learning quickly runs into network capacity limits – it is difficult for a single neural network to encode policies for hundreds of different construction tasks. Moreover, standard methods ignore a key characteristic here – that each of the construction problems shown in Figure 1 shares common substructure with a subset of other tasks within the group. This is a situation where planning methods have the potential to work well. If we can accurately segment the objects in the world and estimate their position and geometry, the a variety of planning methods exist that can be used to find goal-reaching solutions. (We imagine that a new forms of shape completion similar to [1, 2, 3]) could be very helpful here.) Planning methods have an advantage here because they have the ability to solve a large number of tasks in a given domain. Unfortunately, since these methods do not learn end-to-end, they have no ability to improve based on experience.

Submitted to the 5th Conference on Robot Learning (CoRL 2021). Do not distribute.
In this paper, we attempt to strike a balance between multi-goal policy learning and planning with a simple two-level hierarchy that solves discrete planning problems at a high level and executes learned policies at a low level. The high level operates like a standard discrete planner with a hand-coded discrete state space and transition function, not unlike the task planner in standard TAMP methods [4, 5]. Transitions at the high level correspond to goal-conditioned policy learning problems at the low level. Our algorithm finds a path in the high-level discrete state space and then attempts to learn policies (or execute already-learned policies) that traverse the path. Whereas standard multi-goal policy learning methods attempt to learn end-to-end policies for all goals of interest, our method learns a large number of subgoal-reaching policies that are each valid only in a small region of state space (similar to TDM [6]). This helps us learn policies for a large number of subgoals without exceeding network capacity limits. Note that this is not a hierarchical RL method because we are hand-coding the hierarchy instead of learning it. However, in many robotic manipulation problems, we believe it is relatively easy to hand-code this kind of structure compared to solving the challenging perception problems that would otherwise be needed in standard planning.

Our main contribution is to show that a simple hierarchical learning method such as we propose here has the potential to solve what would otherwise be an extremely challenging multi-goal policy learning problem. To our knowledge, no other multi-goal method can learn a policy that can solve the array of 124 block construction problems shown in Figure 1 as we do here using only raw visual inputs.

1.1 Related Work

Goal-conditioned Reinforcement Learning: An RL agent is conditioned on a goal, which can be a particular state (e.g. move a block to an \((x, y)\) position) or a more abstract notion of a goal (e.g. build a tower from four blocks). Methods in this area often adaptively choose goals to explore in order to speed up learning. In the context of model-free RL, Andrychowicz et al. [7] consider every state a goal, and train their agent on goals it has reached online via relabelling. Zhao and Tresp [8], Zhao et al. [9] extended the framework to increase the diversity of explored goals and replayed experiences. In the navigation domain, graph planning over subgoals has been used to solve long-time horizon tasks [10, 11]. Li et al. [12] used an object-factored model to generalize over block stacking tasks involving up to nine blocks. In model-based RL, methods that sample goal states to reach from a previously trained variational autoencoder [13, 14] have been proposed [15, 16].

Automatic curricula for RL agents: Previous works have explored automatic curriculum generation, usually for the purpose of breaking down a difficult task into easier subtasks. Prior work has used a generative model [17] or ensemble of value functions [18] to generate goals that an agent can more readily learn. Florensa et al. [19] manipulated the starting state of an agent so that it is likely to reach the goal with its current abilities. Other works considered a framing with a teacher and a student agent, where a teacher either learns to select tasks for the student, or the student attempts to imitate the teacher [20, 21].

Learning and planning with symbols: Similarly to our structured (symbolic) goal space, Zhang et al. [22] considered a setting where each state has a ground-truth symbolic description. Their symbols do not pertain to goals specifically, but to other characteristics of states, such as two blocks being next to each other in a grid-world. The goal of their work is to learn low-level policies and a transition function for a high-level planner, whereas we are specifically interested in solving a large number of goals. Symbolic task descriptions have also been considered in the context of learning of composable skills [23, 24]. Other work has investigated learning symbolic representations to facilitate transfer to new tasks [25].

2 Approach

2.1 Manipulation as a Goal-Conditioned MDP in a Spatial Action Space

We formulate the manipulation problem as a deterministic goal-conditioned MDP in a spatial action space. Typically, when formulating a robotics problem as an MDP, the action space is over end effector or joint velocities [26, 27]. However, in a spatial action space, the action space includes a subset of \(SE(2)\) (or \(SE(3)\)) where the spatial variables denote the target pose for an arm motion.
A goal-conditioned Markov decision process has a state space $S$, an action space $A$, a family of goal sets $G \subset 2^S$, an unknown transition function $T : S \times A \rightarrow Pr(S)$, a goal conditioned reward function $R_g : S \rightarrow \{-1, 0\}$, $g \in G$, a distribution of initial states $\rho$, and the discount factor $\gamma$. In this work, the reward function is sparse with $R_g(s, a) = -1$ if $s \notin g$ and $R_g(s, a) = 0$ otherwise.

Visual State Space: The state space $S = S_{world} \times S_{robot}$ is the cross product of the state of the world and the state of the robot (robot state includes objects held by the robot). The state of the world is expressed as a single $n$-channel image $I \in S_{world} = \mathbb{R}^{n \times h \times w}$ that contains all relevant objects in a scene. The state of the robot could be arbitrary, but in this paper it is encoded by the “in-hand” image $H \in S_{robot} = \mathbb{R}^{n \times h' \times w'}$ that shows objects currently grasped by the robot, and a boolean flag $b$ indicating if the robot is holding an object (i.e. was gripper able to fully close on last PICK). If the robot is not holding anything, then the pixels in $H$ are zero. $H$ is generated by storing an $h' \times w'$ image crop from $I$ just prior to executing the last pick action. An example of $I$ and $H$ is shown in Figure 4.

Spatial Action Space: The action space $A = A_{sp} \times A_{arb}$ is the cross product of the spatial component of action $a_{sp} \in A_{sp}$ and additional arbitrary action variables $a_{arb} \in A_{arb}$. Component $a_{sp}$ encodes the position to which the robot is to move its hand while $a_{arb}$ encodes information about how the robot is to move or what will happen after the move is complete. In this paper, $A_{sp}$ is the space of $x, y$ positions where the hand is moved to and $A_{arb} = \{\text{PICK, PLACE}\}$ denotes whether this is a pick action or a place action. If $a_{arb} = \text{PICK}$, then the robot closes the gripper after moving to the position $a_{sp}$. If $a_{arb} = \text{PLACE}$, then the robot opens its fingers after completing the move.

Learning Objective: Given the goal-conditioned MDP formulated above, our objective is to find a goal-conditioned policy that achieves optimal expected returns with respect to a probability distribution over goals, $\Lambda : G \mapsto \mathbb{R}$. $\Lambda$ can be arbitrary, for example allocating all probability mass to a single goal or evenly over a subset of goals.

2.2 Abstract Structure

We structure learning using a hand-coded discrete graph representation of the problem that we call the goal graph. First, our agent solves a discrete planning problem over the goal graph. Then, it executes a sequence of goal-conditioned policies that move the agent along a planned path.

Goal Graph: The goal graph is a graph $\bar{\Gamma} = (\bar{S}, \bar{T})$, where $\bar{S}$ is a set of abstract states and $\bar{T}$ is a set of edges. Each underlying state $s \in S$ maps onto an abstract state $\bar{s} \in \bar{S}$ via the abstraction function $f : S \rightarrow \bar{S}$, which we assume is known to the agent, i.e. $f$ is provided by the system designer. The graph must be such that each goal set $g \in G$ in the goal-conditioned MDP $M$ corresponds to the preimage of an abstract state. That is:

$\forall g \in G, \exists \bar{s} \in \bar{S}$ such that $f^{-1}(\bar{s}) = g$

(Where $f^{-1}(\bar{s}) = \{ s \in S | f(s) = \bar{s} \}$ denotes the preimage of $\bar{s}$ under $f$). The connectivity of $\bar{\Gamma}$ encodes how our agent will attempt to reach end goals and should reflect the connectivity of the underlying problem. If an edge $(\bar{s}_1, \bar{s}_2) \in \bar{T}$ exists in the graph, then our agent may attempt to learn a policy that traverses that edge. We assume that each edge in the graph can be traversed in at most $k$ time steps. Figure 2 illustrates the goal graph for the block construction domain.

Figure 2: Goal graph for block construction domain.
At the top level, we use graph search to find a path through the goal graph that reaches the end-goal abstract state. This abstract state maps to a set of underlying states in the MDP via the preimage of \( f, f^{-1}(\hat{s}) \subset S \). Paths through the goal graph from the start state (the blue dot in the center of Figure 2) to end-goal abstract states near the edge of the Figure are sequences of subgoals that our agent will attempt to follow.

### 2.3 Plan Execution

Control happens at two levels in our framework.

Planning at the high level: At the top level, we use graph search to find a path through the goal graph that reaches the end-goal abstract state. This is illustrated in Figure 3 by the orange line from \( \hat{s}_b \) to \( \hat{s}_e \). Each successive vertex on the path denotes an abstract state (qualitative block structure) that can be reached from the last in at most \( k \) time steps \((k = 2\) for the block construction domain).

Policy execution at the low level: At the low level, we learn and execute goal conditioned policies using DQN. We learn a Q function \( Q(s, a, \hat{s}) \), where \( \hat{s} \) is the abstract subgoal state. Let \( \pi_Q(\hat{s}) \) denote the policy induced by \( Q \) with \( \hat{s} \) as a goal – call this the subgoal policy for \( \hat{s} \). Subgoal policies are executed for the sequence of abstract subgoal states on the path found using graph search. In Figure 3 for example, the agent is initialized in a state \( s \in f^{-1}(\hat{s}_b) \) and parameterized with \( \hat{s}_b \) as a goal. The plan generated at the high level is a sequence of abstract states \( \hat{s}_b, \hat{s}_a, \hat{s}_d, \hat{s}_c, \hat{s}_e \). Starting from \( s \in f^{-1}(\hat{s}_b) \), the agent executes \( \pi_Q(\hat{s}_b) \). If the agent reaches a state \( s \in f^{-1}(\hat{s}_e) \), \( \pi_Q(\hat{s}_e) \) terminates and the agent begins executing \( \pi_Q(\hat{s}_e) \). This continues until reaching a state \( s \in f^{-1}(\hat{s}_e) \).

![Figure 3: Subgraph of the full goal graph for the block construction.](image)

**Algorithm 1 Execute Plan**

```plaintext
Require: \( \text{env} \): environment, \( \Gamma \): goal graph, \( \Lambda \): distribution over end goals, \( f \): abstraction function.
1: \( Q_\theta.\text{initialize}() \)
2: for each episode do
3: \( \text{env}.\text{reset}() \)
4: \( \hat{s}_{\text{end}} \sim \Lambda \)
5: \( \text{while } f(s) \neq \hat{s}_{\text{end}} \) do
6: \( s \leftarrow \text{env}.\text{get state}() \)
7: \( \hat{s}_{\text{next}} \leftarrow \Gamma.\text{get next subgoal}(f(s), \hat{s}_{\text{end}}) \)
8: \( s \leftarrow Q_\theta.\text{execute policy}(\text{env}, \hat{s}_{\text{next}}) \)
9: \( \text{if } f(s) \neq \hat{s}_{\text{next}} \) then
10: continue
11: end if
12: end while
13: end for
```

### 2.4 Policy Learning

Subgoal policies are learned on-line during execution of Algorithm 1 inside of \( Q_\theta.\text{execute policy}(\text{env, } \hat{s}_{\text{next}}) \) in Line 8. This is standard Double DQN [28] with \( \gamma = 1 \), and a hard target update performed every 1000 optimization steps. The subgoal policy executes for at most \( m \) time steps \((m = 4 \) in our experiments) and terminates early as soon as the subgoal state is reached. During training, the agent follows an \( \epsilon \)-greedy exploration strategy with a linear annealing schedule. The value of \( \epsilon \) controls both the probability of a random action and the amount of random Gaussian noise added to the output of the Q-network.

Sequenced subgoal policy learning: Notice that since Algorithm 1 does not do any goal relabeling, it does not learn all subgoal policies at the same time. Instead, it learns subgoal policies in the same order as they appear in plans generated in Step 7. In the example shown in Figure 3 where the graph planner generates the subgoal sequence \( \hat{s}_e, \hat{s}_o, \hat{s}_c, \hat{s}_d, \hat{s}_e \), our agent does not begin learning the policy for subgoal \( \hat{s}_e \) until the \( \hat{s}_o \) policy is learned sufficiently well to enable the agent to reach \( \hat{s}_o \). If the
agent never reaches $s_0$, then the subgoal policy for $s_0$ never executes and $\pi_Q(s_0)$ is never trained.

This approach has a couple of important effects. First, as we describe below, the policies that are learned for early subgoals condition the distribution of states over which we learn later subgoals.

Second, the approach does not waste any time learning policies that are not relevant to any task, i.e. it does not learn policies for subgoals that are not on the planned path for any end-goals that have positive probability under $\Lambda$.

Learning subgoal policies over a limited radius: Perhaps the most important outcome of sequencing subgoal execution as described in Algorithm 1 is that learning is focused on states from which the subgoal can be reached in a small number of steps. This constraint is not explicitly enforced.

However, since each subgoal in the goal graph $G$ can be reached in at most $k$ steps from adjacent subgoals, most learning will occur in the neighborhood of this $k$ step between the two subgoals.

This is important because it reduces the network capacity that the $Q$ function would otherwise need. For example, we can learn subgoals policies for most of the 124 subgoal states shown in Figure 1 without exceeding the capacity of a standard $U$-net model architecture.

Learning the abstraction function $f$: In Section 2.1, we assumed that an abstraction function $f : S \rightarrow \hat{S}$ was given to us. However, since world state is encoded as an image, hand coding $f$ could be difficult. Therefore, instead of assuming $f$ is given directly, we assume that we are given a mechanism for generating training data, $(s, \hat{s}) \in \mathcal{D}$, that can be used to train $f$ as a multi-class classifier. Specifically, we hand-code a function that generates structure instances in simulation given abstract state $\hat{s}$. Such a generative function is easy to encode in many manipulation domains and can be used to create arbitrary amounts of simulated training data.

$Q$ Network Model Architecture: Figure 4 shows the model architecture used to learn the goal-conditioned policies. At the upper left, state is input to the network in the form of the world image $I$ and the in-hand image $H$ (Section 2.1). At the lower left, we input the abstract goal state $\hat{s}_{goal}$ as well as the current abstract state $f(s)$. The output is a two-channel feature map that encodes the $Q$ value of a spatial action to any pixel position. The two $Q$-maps correspond to the PICK or PLACE actions and is indexed using the boolean flag, $b$ (the full model architecture is described in Appendix 5.2). Note that this model takes the current abstract state $f(s)$ as input, which is analogous to the “achieved goal” input used in Hindsight Experience Replay [7].

3 Experiments

3.1 Block Construction Domain

We explore our approach in a domain where the agent learns to build block structures. We automatically generate the set of subgoal structures shown in Figure 1 to be those that can be created by combining layers of a single brick, a single roof, or two adjacent cubes for up to five layers. The size of this set is exponential in height (see Figure 6). An edge exists between two structures if a single block can be added to move between them. We denote any structure that cannot be further built upon, due to a roof block or height limit, as an end-goal structure (i.e. a dead-end node in goal graph). At the start of each episode, the workspace is reset such that all necessary blocks for a given structure are present, and the platform is free of blocks. The block construction domain is simulated in Pybullet [29]. Abstract states are represented as a one-hot vector over all possible structures plus a boolean flag that indicates that the platform is clear. The state space is $S = S_{world} \times S_{robot}$, where $S_{world} = \mathbb{R}^{1 \times 90 \times 90}$ is an image that describes the state of the world. $S_{robot} = \mathbb{R}^{1 \times 24 \times 24} \times \{0,1\}$ describes the state of the robot in terms of an “in-hand” image that describes the object grasped in the hand and a binary flag that denotes whether the gripper is holding an object.

An important caveat to note in all the experiments below is that we limit the spatial action space to $x, y$—the orientation of the gripper remains fixed throughout. In other words, while we defined $a_{sp} \in SE(2)$ in Section 2.1, in these experiments we restrict $a_{sp} \in \mathbb{R}^2$. This limitation made it easier for us to run the experiments below, but we do not view it as a theoretical limitation because the feasibility of policy learning over a $SE(2)$ spatial action space is already well established [30, 31].
3.2 Comparison With Multi-Goal Baselines

Here, we evaluate the ability of our approach to learn a set of subgoal policies that can reach any one of a set of end-goal structures. Recall from Section 2.1 that $\Lambda$ is the probability distribution over end-goal abstract states that defines the learning objective. We evaluate our method over five different distributions $\Lambda$ with successively larger support. For evaluation purposes, we created five sets of end-goal block structures, as shown in Figure 6. For each of these five classes, we define $\Lambda_i$ (the probability distribution over end-goals of maximum height $i$) to be the uniform distribution over the structures in that set. We use $Q$ learning with the version of $\epsilon$-greedy exploration described in Section 2.4. $\epsilon$ varies linearly between 1 and 0 over the entire training process (50k steps for $\Lambda_1$, 100k steps for $\Lambda_2$, 150k steps for $\Lambda_3$, 200k steps for $\Lambda_4$, 250k steps for $\Lambda_5$).

Baselines: We compare our method against two baselines, UVFA and HER. In UVFA, we simply instantiate the same goal conditioned $Q$ network we use for subgoal learning (Figure 4), but train that network directly without goal relabeling [32]. End-goals are sampled in exactly the same way they are in Algorithm 1, however the graph structure and subgoals are not used. HER is the same as UVFA except that the method incorporates the standard goal relabeling strategy proposed in [7].

Results and Discussion: Our method outperforms the baselines in all scenarios, and the difference increases with the maximum height of the structures (Figures 5 and 7). Neither HER nor UVFA can fit a large fraction of the 63 height-5 structures, whereas our method reaches 86% success rate. The degree to which our method outperforms the HER and UVFA baselines is striking. Probably the main reason for this is related to exploration. The sequenced policy learning described in Section 2.4...
essentially guides our agent through a series of learning tasks. This is not exactly the same as curriculum learning [33] because each learning task has a different subgoal and we are not relying on weight sharing to transfer learned knowledge. However, it does help guide exploration through appropriate high level subtasks. Without this guidance, the chances of solving any of the height 5 structures is nearly zero.

3.3 Comparison With Shaped Reward Function

A key reason why our method outperforms HER and UVFA in Section 3.2 is exploration. By presenting the agent with sequential subgoal learning problems, our method guides the agent to experience more complex structures than it otherwise see. In this section, we compare this approach with a shaped reward approach where the agent obtains incremental reward for reaching each subsequent subgoal along a path towards the goal. We perform this comparison for the same set of height 5 end-goals we explored in Section 3.2 (Figure 6e). At the start of each episode, we sample an end-goal from $\Lambda_5$.

Comparison With Shaped Reward Function Architecture: Our shaped reward function architecture is the same as the UVFA architecture used in Section 3.2 (no goal relabeling, same network architecture as in Figure 4) except that we provide a shaped reward signal based on the high level plan. At the start of each episode, this agent samples an abstract end-goal state $s_{end}$ from $\Lambda_5$, and plans a path in the goal graph to reach it, $(\bar{s}_1, \bar{s}_2, \cdots, \bar{s}_N)$ where $\bar{s}_N = s_{end}$. The agent executes the UVFA policy conditioned on $s_{end}$ with a discount factor of $\gamma = 0.9$. Each time the agent reaches subgoal state on the path, i.e. each time it reaches a state $s \in f^{-1}(\bar{s}_i)$, it receives a reward of $+i$. Essentially, the agent’s incremental rewards increase with each subsequent subgoal achieved. Execution stops after reaching an end-goal state or after 4N time steps, whichever comes first.

Comparisons: Figure 8 shows our comparison. The blue line is our method. The green line is HER. The red line is the shaped reward function. Notice that the vertical axis in Figure 8 measures the average number of blocks stacked successfully, rather than success rate as in Figure 5 and 7. This enables us to measure performance more precisely. We can measure how much of a structure on average was completed successfully. Since the height-5 structures from which we sample in this experiment are composed of 6.63 blocks on average, 6.63 is the optimum value here.

Results and Discussion: Figure 8 shows that the shaped reward function helps the UVFA method a lot in comparison to HER (red vs green). However, our method (blue) still outperforms by a significant margin: at 400k time steps, our method is able to stack 6.1 blocks correctly on average versus the shaped reward function which only stacks an average of 4.7 blocks. Note that this difference in averaged blocks stacked has a significant affect on average success rate since the agent must stack all blocks correctly in order to succeed. While our method has better exploration than undirected multi-goal Q learning, this is not its only advantage. We hypothesise that our method outperforms because it does not need to learn a global multi-goal policy over the entire state space – it can instead execute a series of locally-performant policies.
3.4 Robot Experiments

Setup: We performed experiments on a UR5 robot equipped with a Robotiq two-finger gripper which was mounted on a flat table top. Depth sensor measurements were provided by two Occipital Structure sensors – one mounted above the table pointing down and the other mounted at the wrist. We trained the agent for 500k environment steps in the simulator before evaluating on the real robotic system. Figure 9 shows two examples of our agent building structures with the real robot.

Sim-to-real transfer: Sim-to-real transfer was not a significant problem because we used depth images rather than RGB. We augmented the depth images created during simulation by adding a small amount of Perlin noise (magnitude 0.05) to the images and then blurring the images slightly using a Gaussian kernel (sigma sampled in the range from 0.5 to 1). Additionally, to account for the fact that real life blocks were not perfectly axis aligned, we added a small amount of orientation noise in simulator resets (our system learns policies over position only, not orientation).

Results and Discussion: Success rates from our real robot runs are shown in Table 1. We performed 10 runs on each of the 10 different structures shown in the table (100 attempts at block construction in total). The top row (Success) shows the average number of times the robot successfully constructed each of the ten structures. The bottom row (Progress) shows the average number of blocks placed correctly. The results suggest that our method can be very successful on a real robotic system. It is important to emphasize that these structures were created using a single goal-conditioned policy – we did not learn a separate policy for each structure. Notice that the success rate for the “two towers” structure on the far right has only a 50% success rate, stacking an average 8.3 blocks out of 10. This reflects the difficulty of balancing two towers of five blocks each.

4 Conclusions

Discussion: This paper demonstrates that a simple approach to hierarchical policy learning involving hand-coding high level structure can enable large scale multi-goal policy learning. As far as we know, there is no other method that can learn a single policy that can construct more than a hundred different block structures. One concern is that this method could only be relevant to block construction tasks and not to more practical manipulation applications. However, in contemporaneous work, we demonstrate that policy learning in a spatial action space can indeed be an effective way to solve practical robotics manipulation applications [34] and we believe that this would could be extended similarly.

Limitations: An important failure mode we observed is shown in Figure 10 where the agent is attempting to construct the two towers shown on the right side of Table 1 but the towers have collapsed because the agent did not align successive blocks sufficiently accurately on top of one another. The problem here is that our method does not back up reward through multiple subgoal policies. Since policy learning only reasons about the stability of the next successive block, it fails to recognize that a misplaced block early in the structure could cause failure that only occurs much later during construction. Fundamentally, this results from a defect in how abstract states are defined – had they been more restrictive in terms of block alignment, this failure would not occur. However, anticipating all possible failure modes of a hand-coded abstraction is infeasible for complex domains; a possible avenue for handling this problem is to enable the abstraction to adapt online as a result of end-to-end feedback.
References


