Action Priors for Large Action Spaces in Robotics

Ondrej Biza
Northeastern University
Boston, MA, USA
biza.o@northeastern.edu

Dian Wang
Northeastern University
Boston, MA, USA
wang.dian@northeastern.edu

Jan-Willem van de Meent
Northeastern University
Boston, MA, USA
j.vandemeent@northeastern.edu

Robert Platt
Northeastern University
Boston, MA, USA
rplatt@ccs.neu.edu

Lawson L.S. Wong
Northeastern University
Boston, MA, USA
lsw@ccs.neu.edu

ABSTRACT

In robotics, it is often not possible to learn useful policies using pure model-free reinforcement learning without significant reward shaping or curriculum learning. As a consequence, many researchers rely on expert demonstrations to guide learning. However, acquiring expert demonstrations can be expensive. This paper proposes an alternative approach where the solutions of previously solved tasks are used to produce an action prior that can facilitate exploration in future tasks. The action prior is a probability distribution over actions that summarizes the set of policies found solving previous tasks. Our results indicate that this approach can be used to solve robotic manipulation problems that would otherwise be infeasible without expert demonstrations. Source code is available at https://github.com/ondrejba/action_priors.

KEYWORDS

reinforcement learning; deep learning; action prior; robotics; robotic manipulation

ACM Reference Format:


1 INTRODUCTION

Advances in deep learning have made model-free robot control a viable alternative to model-based motion planning [7, 14, 30]. However, the complexity of tasks solvable by these approaches without extra supervision is limited, partly due to sample inefficiency of deep learning. Hand-crafted temporal abstraction of end-to-end motions such as picking, placing and pushing are a compelling alternative, as they allow agents to reason over longer timescales [13, 32, 33]. In particular, Zeng et al. [33] proposed a pixel-wise parameterization of the action space, where each pixel in the observed image of the workspace corresponds to a reaching action to that position followed by a pick or place.

While both low-level action spaces with long time horizons and pixel-wise action spaces are difficult to explore, the pixel-wise parameterization makes this challenge more explicit: the agent is presented with thousands of possible actions, and usually only a handful of them enable the agent to make progress toward its goal. Exploration challenges like this are often addressed using reward shaping, curriculum learning, or imitation learning. However, these methods require additional supervision that maybe difficult to provide. Ideally, our agent could learn new skills without expert supervision.

In this paper, we construct priors over the action space — action priors — that inform the agent of actions that were useful in the context of previously learned tasks. The idea of action priors has existed for some time. Sherstov and Stone [24] considered a single action prior for all states; later, action priors were extended to state-specific priors [1, 6, 21, 22]. However, to date, action priors have not been applied outside of learning in grid-world-like environments with small action spaces [6, 21, 22, 24] and planning in factored models [1].

In contrast, we train action priors in environments with image states and pixel-wise action spaces with thousands of actions. To that end, we represent an action prior as a single fully-convolutional neural network trained to summarize a library of pre-trained policies. We distinguish between a set of training tasks, which we solve using imitation learning, and a held-out set of testing tasks to be solved without expert information. The role of action priors is to bias exploration on the testing tasks toward actions that were found to be useful when solving the training tasks.

We evaluate our approach on 16 robotic block stacking tasks. We proceed in three stages. First, our agent uses imitation learning to find near-optimal solutions to a subset of the 16 tasks. Second, we condense these near-optimal policies onto a state-dependent probability distribution over actions (i.e. the action prior) that gives high probability to any action that was part of one of the near-optimal policies. Finally, we use the action prior to bias exploration when solving a new task. In the block stacking domain, this action prior gives a high probability to picking actions that are likely to lift a block of some type or placing actions that are likely to result in a stable placement. Although we explicitly focus on robotic
manipulation, our approach should generalize well to any problem in robotics with a large action space.

This paper makes two main contributions. First, we show that a state-conditioned action prior is an effective way to transfer knowledge from previously solved tasks to new tasks in a robotic manipulation domain. Our experimental results indicate that this approach can dramatically increase the probability of visiting a goal state during exploration. Second, we introduce a method of learning an action prior in situations where the previously learned policies are valid over different regions of the state/action space. This problem only occurs in large state/action spaces such as in robotic manipulation, and we believe we are the first to address it.

2 RELATED WORK

Action priors bias action selection during the exploration phase of learning towards actions that were previously determined to be viable. This information can either be specified by an expert [1] or extracted from policies previously solved tasks [1, 6, 21, 22, 24]. Note that action priors refer to a different construct than policy priors [5, 31], as action priors do not involve posterior inference of policy parameters.

Sherstov and Stone [24] eliminated actions not optimal for any previous task in a state-agnostic way. Together with their transfer learning algorithm, the state-agnostic action prior increases learning speed in grid-world mazes. Fernández-Rebollo and Veloso [6], Rosman and Ramamoorthy [21, 22] explored state-specific action priors in similar discrete-state-space MDPs. Fernández-Rebollo and Veloso [6] alternated between rolling out the policy being learned and a policy sampled from a library. The probability distribution over the library of policies was updated online to maximize rewards for the current task. Rosman and Ramamoorthy [21, 22] filled in the pseudo-counts of Dirichlet distributions used to select actions in each state by a weighted-sum of actions selected by previously learned policies. Abel et al. [1] combined action priors and hand-crafted object-oriented representations [4] to improve the run time of dynamic programming policy search for a Minecraft environment and a real-world robotic manipulation task.

In contrast to pre-defined factored representations in Abel et al. [1], we learn the action prior and model-free policies from pixels. The Dirichlet prior [21, 22] is not easily extensible to continuous state spaces; instead, we learn the action prior as a convolutional network. Compared to Fernández-Rebollo and Veloso [6], we cannot keep a library of policies loaded in memory, as each policy is parameterized by a large convolutional network—we distill all policies into a single action prior network.

In concurrent work, Ajay and Agrawal [2], Pertsch et al. [19] learned action priors over fixed-length sequences of actions (also called skill priors). Both approaches use variational autoencoders to learn representations for action sequences, and can be used to solve composite robotic manipulation tasks. Singh et al. [25] studied action priors (here called behavior priors) in a setting where training and testing tasks differ in terms of the objects being manipulated, but are otherwise the same.

The topic of efficient exploration is closely related to action prior. Methods in this category often do not use additional information, such as prior policies. Instead, they use a notion of surprise or information content of a visited state. These quantities can be measured by counting the number of times states were visited [26] or by model-based approaches [11, 17]. Our problem statement is incomparable with these approaches, as we exploit additional information from previously learned tasks, which facilitates much more targeted exploration compared to the notion of surprise alone.

Transfer learning has been studied extensively both in classical reinforcement learning [27] and in deep reinforcement learning [9, 16, 28]. Goyal et al. [9], Teh et al. [28] learned a so-called default policy while learning multiple specialized policies in a multi-task or a multi-goal RL. To transfer to new tasks, Teh et al. [28] used the KL-divergence between the default policy and a new policy as regularization, and Goyal et al. [9] used their default policy to quantify the notion of a “decision state”: a state in which we need make a decision based on the task we want to solve (e.g., a crossroads in a maze). Their agent is then encouraged to explore decision states by adding an intrinsic reward. Both Goyal et al. [9], Teh et al. [28] focuses their experimental evaluation on navigation tasks, with the latter transfer method only being applicable to discrete-state-space domains (due to them using count-based exploration). Parisotto et al. [16] distill policies from training tasks into a single student, which is then used to initialize the testing policy.

3 BACKGROUND

We model the pick and place robotics tasks in this paper as Markov Decision Processes (MDPs, [3]) $\mathcal{M} = \langle S, A, P, R, \rho_0, \gamma \rangle$. $S$ and $A$ represent the sets of state and actions, $P : S \times A \rightarrow Pr(S)$ is a
Algorithm 1 Action prior learning

**Input:** Set of training tasks \( T = \{ T_1, T_2, ..., T_N \} \).
**Output:** Action prior network \( f_{AP} \).

1: **procedure** LEARNAP
2: for \( T_i \) in \( T \) do
3: Train an expert policy \( \pi_i \) for task \( T_i \).
4: Collect \( K \) transitions by rolling out \( \pi_i \). Store visited states in \( D_i \).
5: end for
6: Concatenate datasets \( \{ D_1, D_2, ..., D_N \} \) into \( D \).
7: Train task classifier \( f_c : S \rightarrow \Delta^{N-1} \) on \( D \) (Section 5.1).
8: Collect optimal action sets for \( \pi_1, \ldots, \pi_N \) on \( D \).
9: Merge optimal action sets using \( f_c \) and add the union set to \( D \) (Section 5.2).
10: Train action prior network \( f_{AP} \) on \( D \) (Section 5.3).
11: Return \( f_{AP} \).
12: **end procedure**

Algorithm 2 Action prior exploration

**Input:** Reinforcement learning agent \( f_{RL} \), action prior network \( f_{AP} \), action prior probability threshold \( \sigma \), task \( T \).

1: **procedure** EXPLOREAP
2: while Stopping condition not reached do
3: Get environment state \( s \).
4: if Explore then
5: \( A^* \leftarrow \{ a | a \in A \land f_{AP}(s, a) > \sigma \} \).
6: Randomly sample \( a \sim \text{Uniform}(A^*) \).
7: else
8: Choose \( a \) according to \( f_{RL} \) (e.g. \( a \) with maximum Q-value in DQN).
9: end if
10: Execute action \( a \) in the environment.
11: Observe reward \( r \) and next state \( s' \).
12: Add tuple \((s, a, r, s')\) into the replay buffer.
13: Perform a learning step of \( f_{RL} \).
14: end while
15: **end procedure**

transition function that returns a probability mass/density over states and the reward function \( R : \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R} \) maps state-action pairs to their expected rewards. We consider MDPs with an initial state distribution \( \rho_0 \) and a discount factor \( \gamma \).

A policy \( \pi : \mathbb{S} \times \mathbb{A} \rightarrow [0, 1] \) captures the decision making process of an agent as the probability distribution over actions for each state. Each policy has an associated state-action value function \( Q_\pi(s, a) = R(s, a) + \gamma \mathbb{E}_{s', a'} \phi \left[ Q_\pi(s', a') \right] \), the discounted return when executing action \( a \) in state \( s \) and following policy \( \pi \) thereafter.

In this paper, we consider MDPs with the following properties:
- States are represented as images.
- The action space is large, usually one action per state pixel.
- The time horizon is short, around 10 time steps, and rewards are sparse.

Learning an optimal policy for this class of MDPs without additional information is extremely difficult because there is only a handful of optimal actions in each state. Hence, the probability of getting a reward for a sequence of random actions is minuscule.

4 PROBLEM STATEMENT

Let \( T_{train} = \{ T_1, T_2, ..., T_N \} \) be a set of training tasks expressed as MDPs. A task \( T_i = (S, A, P_i, R_i, \rho_0, \gamma) \) shares its definition with all other tasks except for its reward and transition function. For example, a task of building a tower is built from blocks of height two and a task of building a tower of height three is clear a different reward function. Even though the dynamics of picking and placing objects are the same for both tasks, the former task terminates when a tower of two is built, whereas the latter does not. Therefore, there are small variations in the transition function between the tasks related to terminal states.

We assume we have access to an expert policy \( \pi_i \) for each training task \( i \) together with a dataset of on-policy transitions \( D_i \). Given a testing task \( T_{test} \) (different in its transition and reward dynamics from training tasks), our goal is to learn the best possible policy. We formalize this as summarizing experience from previous tasks \( (D_i, \pi_i)_{i=1}^N \) in some function (such as an action prior) parameterized by \( \phi \). The parameters are then used in some training process

\[ \pi(\phi) \] resulting in a policy for the testing task. We then indirectly maximize the success of the testing policy by manipulating \( \phi \).

\[ \arg \max_{\phi} \mathbb{E}_{(\pi(\phi), \rho_0, P_{N+1})} \left[ \sum_{t=0}^{\infty} \gamma^t R_{N+1}(s_t, a_t) \right]. \tag{1} \]

5 ACTION PRIORS

Given a set of expert policies \( \pi_1, \ldots, \pi_N \) for training tasks \( T_i \in T_{train} \), we define the action prior to be a policy

\[ \pi_{AP}(s, a) = \eta \cdot \max_{i \in [1, N]} \pi_i(s, a), \]

where \( \eta \) is normalizes \( \pi_{AP} \). For example, if \( \pi_1, \ldots, \pi_N \) are deterministic, then \( \pi_{AP} \) assigns equal probability to each action \( \pi_1(s) \ldots \pi_N(s) \).

Algorithm 1 outlines the procedure we use to train the action prior. First, we train expert policies for the \( N \) training tasks. Action priors are invariant to the method used to train them (Appendix Section 8). Then in Step 4, for each task \( i \) and policy \( \pi_i \), we obtain a sample of on-policy states by rolling out \( \pi_i \).

Next, in Step 7, we train a classifier \( f_c \) that predicts the task that is most likely to have caused the agent to visit a state. This is important because the policies \( \pi_i \) are not all valid over the entire state space. For example, a policy trained to assemble the structure in Figure 1a (tower from two cubes and a small roof) has never seen the state shown in Figure 1b (a house built from two blocks, a brick and a large roof). To determine the set of policies applicable in a given state, we train a task classifier \( f_c : S \rightarrow \Delta^{N-1} \) (Section 5.1), which predicts the tasks in which a state is most frequently encountered. This allows the action prior to ignore policies for tasks that are not relevant to a particular state.

Then, in Step 8 of Algorithm 1, after training the policies \( \pi_i \) and the task classifier \( f_c \), we collect the training dataset for the action prior (Section 5.2). The dataset contains an equal number of states for each task, which are obtained by rolling out the learned policies \( \pi_i \). For each state, we compute a binary mask that represents the union of actions that are optimal (in any task) given a set
of policies. The set of applicable policies is predicted by the task classifier. Step 10 of Algorithm 1 trains the action prior network \( f_{\text{AP}} : S \times A \rightarrow [0, 1] \) to predict the probability of an action being optimal for any task in a given state (Section 5.3). Finally, we create an action prior policy \( \pi_{\text{AP}}(s, a) \) based on the action prior network \( f_{\text{AP}} \). By thresholding the probabilities predicted by \( f_{\text{AP}} \), we get a set of proposed actions \( A^*(s) \) for state \( s \). We set \( \pi_{\text{AP}}(s, a) \) to be a uniform distribution over \( A^*(s) \) with actions outside of the optimal set being assigned zero probability.

5.1 Learning the Task Classifier

The task classifier \( f_c : S \rightarrow \Lambda^{N-1} \) determines which of the expert policies are relevant in the context of a particular state. To train this classifier, we use a dataset of states and categorical labels \( \{(s_i, y_i)\}_{i=1}^M \). We construct this dataset by generating policy rollouts for each task. If a state \( s \) was visited during a rollout of a policy for a task \( y \in \{1, \ldots, N\} \), we include the pair \((s, y)\) in the dataset. Note that this results in a dataset where class labels for each state are not unique, as each state could have been encountered during rollouts for multiple tasks (e.g., the initial state with all objects placed on the ground appears in all tasks). We can interpret the data as samples for multiple tasks (e.g., the initial state with all objects placed on the ground appears in all tasks). We train this classifier, using a dataset of states and categorical labels \( \{(s_i, y_i)\}_{i=1}^M \). We collect training data by rolling out the learned policies for each task for a fixed number of time steps. For each state, we compute the optimal mask, which for each action \( a \) contains a bit that indicates whether this action is part of the approximate optimal action set

\[
\tilde{A}^*(s) = \{ a \in \tilde{A}^*(s_i) \}.\tag{6}
\]

We train this multi-task classifier using a standard logistic loss

\[
L_{\text{AP}} = -\frac{1}{M} \sum_{i=1}^M \sum_{j=1}^{|A|} m_{i,j} \log f_{\text{AP}}(s_i, a_j) + (1 - m_{i,j}) \log (1 - f_{\text{AP}}(s_i, a_j)).\tag{7}
\]

As discussed above, the training masks contain only the subset of the union of optimal action sets for all tasks. We can view this problem from the perspective of precision-recall trade-off. A model that achieves a lower \( L_{\text{AP}} \) value will have high precision (i.e., it will avoid false-positive optimal actions), but it might have low recall because not every optimal action is represented in the training data. This trade-off can be controlled by moving the decision boundary \( \sum_{i=1}^M m_{i,j} \geq \sigma \) that determines if an action is deemed optimal. Experimentally, we found that setting \( \sigma \) to a low value (i.e., increasing recall and decreasing precision) results in a large increase in the success rate of the action prior exploration policy.

5.4 Action prior exploration

Algorithm 2 summarizes how we use the action prior for exploration when training a reinforcement learning agent on a new task. We follow the standard recipe of alternating between selecting random actions (exploration) and action according to the policy being learned (exploitation). The most common and one of the simplest approaches to controlling this trade-off is an \( \varepsilon \)-greedy exploration policy with the value of \( \varepsilon \) decaying over time (e.g., this approach was used in the original deep Q-network paper [15]). We use an \( \varepsilon \)-greedy strategy in which, during exploration, we select actions uniformly at random from the set actions for which the action prior exceeds a threshold \( \sigma \)

\[
\tilde{A}^*(s) = \{ a \in A \land f_{\text{AP}}(s, a) > \sigma \}.\tag{8}
\]

This results in a more focused exploration compared to selecting actions uniformly at random from the full set \( A \).
We demonstrate the effectiveness of action priors in two domains: (Section 6.3). Policies learned in PyBullet can be deployed on a prior over $\mathcal{N}$ by leave-one-out cross-validation, where we learn an action ple 20 tasks for fruit sequences. The results were obtained for picking up between one and four of them. Picking can be done with five distinct fruits in the environment and the agent should choose to actions that were not necessarily optimal for any previous action during the exploration step. But, we find it to decrease the learning rate of the action prior exploration policy. We hypothesize that by setting $\sigma$ to a low value (around 0.1) the action prior generalizes to actions that were not necessarily optimal for any previous task but are nevertheless plausible.

6 EXPERIMENTS

We demonstrate the effectiveness of action priors in two domains: a proof-of-concept Fruits World (Section 6.2) and a block stacking robotic manipulation experiment in the PyBullet physics simulator (Section 6.3). Policies learned in PyBullet can be deployed on a real-world UR5 robotic arm (Section 6.4). The key question we aim to answer is if action priors enable a deep Q-network (DQN) to learn tasks that were previously out of reach.

6.1 Domains

Fruits World is a grid-world-like domain. The world is a $5 \times 5$ grid with five distinct fruits placed in random positions (the positions change at the start of each episode). There are 25 actions, one for each position. The agent must pick a subset of fruits either in a particular order (sequences) or in any order (combinations); each subset constitutes a different task. When the agent thinks it finished the task, it must execute the 25th action to get a reward and reset the environment. A reward of 1 is given for successfully picking up the target fruits. Due to its combinatorial nature, this environment is surprisingly difficult to solve with model-free deep reinforcement learning. Hence, we also give the agent a reward of -0.1 for picking up the wrong fruit and only allow it to put the right fruits in the basket.

The state is a $5 \times 5 \times 5$ tensor with the first two dimensions corresponding to the grid. If fruit is present in a grid cell, its ID is one-hot encoded in the third dimension. Otherwise, the values in the grid cell are all zero. In combinations, a sixth channel is added to the observation: the positions corresponding to fruits the agent has added to its basket are set to one. In sequences, the agent is given an additional sequence of one-hot encoded IDs of fruits it has picked.

We use a deep Q-network as a model-free agent. All neural networks are a multilayer perceptron with two hidden layers (see Appendix Section C.1).

PyBullet block stacking is a set of simulated tasks involving stacking blocks of various shapes and sizes with a robotic arm (Figure 1). The robot observes the workspace with a depth camera from above (Figure 1 (b)), receiving a 90x90 image with pixel values corresponding to heights. It can execute a top-down pick or place action at a specified coordinate with fixed hand rotation. Before executing a pick action, the robot will take a 24x24 picture centered at the coordinates, where it is executing the pick action. If it successfully picks up an object, it uses the in-hand image to decide where to place it.

We discretize the action space as a 90x90 grid, each cell corresponding to one pixel of the observation. We instantiate 16 different block stacking tasks: each one builds a structure of a width of one or two small blocks and a height of two or three blocks; each structure has a roof on top. Figure 1 (c) and (d) show goal states of building a tower from two small blocks and a small roof and of building a structure from two small blocks followed by one long block and a large roof respectively. Each task is represented as a string (e.g. a small roof on top of a small block is "1b1r") and we use a context-free grammar to generate all possible tasks with particular parameters (Appendix Section A).
<table>
<thead>
<tr>
<th>Method</th>
<th>1b1r</th>
<th>2b1r</th>
<th>2b2r</th>
<th>1l1r</th>
<th>1l2r</th>
<th>1b1b1r</th>
<th>2b1b1r</th>
<th>2b2b1r</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN RS</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>DQN RS, WS</td>
<td>98%</td>
<td>0%</td>
<td>0%</td>
<td>99%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>DQN HS</td>
<td>97%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>DQN HS, WS</td>
<td>97%</td>
<td>0%</td>
<td>2%</td>
<td>100%</td>
<td>99%</td>
<td>3%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>DQN AP</td>
<td>100%</td>
<td>100%</td>
<td>98%</td>
<td>99%</td>
<td>100%</td>
<td>100%</td>
<td>93%</td>
<td>0%</td>
</tr>
<tr>
<td>DQN AP, WS</td>
<td>100%</td>
<td>96%</td>
<td>97%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>92%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 1: Transfer experiments in the block stacking domain. We consider “DQN AP, WS” the main contribution of this paper. Each column reports the final success rate averaged over 100 episodes after training. The baselines and ablations of our method we consider are random action selection (RS), heuristic action selection (HS), weight sharing (WS) and action prior (AP).

Both the 90×90 observation and the 24×24 in-hand image are encoded using a modified version of U-Net [20, 29]. We chose this model because it can produce detailed segmentation maps. The task our DQN, which we use for model-free learning, and action prior models solve is comparable to image segmentation: we make a prediction for each pixel of the input image. In the DQN case, the U-Net predicts a single state-action value for each pixel of the input image. In the action prior case, the model produces a logit for each pixel of the observation. The exact architecture of our U-Net is depicted in Figure 7 in the appendix.

6.2 Fruits World experiments

We sample 20 and 30 tasks for fruits sequences and combinations tasks respectively (Section 6.1). Action priors are tested in a leave-one-out transfer experiment: an action prior is trained on 19 and 29 tasks respectively; it is then used for exploration on the held-out task. This process is repeated for each task. Expert policies for the training tasks use imitation learning (Appendix Section B).

Our comparison in Figure 2 involves action priors (AP), a randomly initialized deep Q-network (DQN) and three baselines based on Actor-Mimic [16] (AP-share, AP-freeze and AP-prog). Actor-Mimic first trains a student to mimic the policies of all experts. The student weights are then used to initialize the agent when learning the testing task. We followed the original implementation except for us having a student with N heads, one for each training task.

This deviation from the original paper is due to the state of our environment not containing any information about which task the agent is solving. In contrast, Parisotto et al. [16] test their method on several Atari games, each with a distinct state space.

In all cases, we remove the N student heads and transfer the weights of the hidden layers. In AM-share, all weights are trainable; AM-freeze only learns the final fully-connected layer. AM-prog is based on Rusu et al. [23], where each hidden layer of the agent network receives both features from the previous agent layer and from the previous student layer. Hence, there are two networks, student and agent, but only the agent is trained. We do not use an adaptation layer suggested by [23] because we only have one student network.

Action priors significantly outperform all baseline in fruits combinations (Figure 2a) and are the only method to solve any fruits sequences task (Figure 2b). In the former, AM-prog and AM-freeze have similar performance, whereas random initialization and AM-share train much slower. We hypothesize that AM-share overwrites the student weights at the start of training, causing it to perform similarly to random initialization. The negative rewards at the start of training received by action priors in Figure 2b can be explained by AP picking up the wrong fruits (small negative reward), and the other methods failing to pick up any fruit (zero reward).

1URL: https://github.com/eparisotto/ActorMimic, visited 21/02/08.
We perform two experiments in this domain: one focused on model-free learning with action priors and the other solely on exploration using action priors. The first experiment matches the setup in Section 6.2: we use 15 out of the 16 block stacking tasks to learn an action prior, which we subsequently test on the 16th task (Table 1). An imitation learning method called SDQfD facilitates the expert policies for the training tasks (Appendix Section B). The second experiment evaluates the success rate of action prior policies on all tasks without additional training (Figure 5 shows two example tasks, Figure 9 (Appendix) contains the results for all tasks). We measure success rate as a function of the action prior probability threshold $\sigma$ and the presence or absence of a task classifier.

Experiment 1 compares our method with two baselines. DQN AP is a deep Q-network with action prior exploration. We test random and heuristic action selection. As our observations are depth images taken from the top of the workspace, the heuristic only allows selecting actions that act on parts of the observation with non-zero height. Therefore, it forces the agent to interact with objects. But, not all interactions necessarily lead to desirable outcomes—the agent often ends up pushing objects out of the workspace. We call the two methods DQN RS and DQN HS respectively.

As a second mechanism for speeding up learning, we employ weight sharing in the unseen task. Because an action prior summarizes the information contained in experts for the 15 training tasks, we initialize the DQN for the 16th testing task with the action prior weights. To mitigate catastrophic forgetting [8], we include a $L^2$ penalty between the original action prior weights and the current DQN weight; it is added to the DQN loss function with a weight of $\omega_{WS}$. We test weight sharing both with random and heuristic baselines (DQN RS, WS and DQN HS, WS) and action priors (DQN AP, WS).

A DQN with an action prior can solve all tasks except for “2b2b1r” and “2b1l2r” (Table 1). As shown in Figure 9 (Appendix), the success rates of action priors on these tasks without training is between 1 and 4%. Hence, a more targeted action prior might be needed.

The DQN sharing weights with the action prior does not lead to noticeable benefits. In fact, it fails to learn “2b2b2r” unlike the non-weight-sharing variant. Since the purpose of weight sharing is to improve training speed of the network, it only provides minor benefits due to exploration being the main challenge in our domains.

As we expected, both the heuristic and random action selection baselines are unable to solve most of the tasks. Random actions succeed on the three simplest tasks (“1b1r”, “2b1r” and “2b2r”), and heuristic selection increases the number to four (plus “1l1l2r”). In the rest of the tasks, the success rates of the baseline exploration policies are close to zero; therefore, no learning method including weight sharing can succeed.

Experiment 2 evaluates two variants of action prior each again trained in leave-one-out fashion on 15 tasks and tested on the 16th task. In this experiment, we only log the success rate of the action prior policy without learning on the testing task.

Figure 5 shows examples of action prior success rates for tasks “1b1r” and “2b1l1r”. We plot the success rates as a function of the action prediction threshold $\sigma$ and the presence or absence of a task classifier. The success rates are measured by rolling out episodes in

![Figure 4: Visualizing actions proposed by an action prior (red crosses) with the probability threshold $\sigma$ set to 0.1 (top row) and 0.5 (bottom row). In the first column, the robot is supposed to pick up an object, in the second column, it is holding a cube in its hand and wants to place it. The images shown are pictures from a depth camera mounted at the top of the workspace in simulation.](image)

![Figure 5: Success rates for “1b1r” and “2b1l1r” using exploration with an action prior as a function of $\sigma$. We compare action priors trained with and without task classifier.](image)
the test task (depicted in the top-left corner) with actions selected by the action prior policy $\pi_{\text{AP}}$. We show results for all tasks in Appendix Figure 9.

The main trends in Figure 5 (and the Appendix Figure 9) are that (a) more complicated tasks benefit more from the task classifier and that (b) almost all action priors that use a task classifier benefit from a decrease in the prediction probability threshold. The increase in success rates for action priors with task classifiers in more complex tasks can be explained by them needing a targeted and precise exploration policy: a task classifier ensures that the training data for the action prior does not mark actions from irrelevant tasks as optimal. On the other hand, we are unsure as to why action priors without task classifiers perform well on simple tasks. See Figure 4 for example of sets of actions proposed by action priors.

6.4 Real-world robot experiments

In theory, policies trained in our PyBullet simulation can be transferred to a real-world robot, as both setups use a depth camera and a robotic arm with the same action spaces. However, in practice noise in the real-world observations often confuses the policies trained with perfect depth images.

To make our models more robust, we re-train the action prior DQNs with weight sharing (DQN AP, WS) in the transfer learning experiment in Section 6.3 with two modifications to the environment. First, we add Perlin noise, a type of correlated gradient noise, to the simulated depth images [12, 18]. The noise models blobs of lower and higher estimated depth values that are present in the real-world images. Next, we initialize all objects in the environment with small rotations compared to their default orientations. In our previous experiments, all objects had the same orientation, as the agent cannot control the rotation of the robot hand. However, the initialization of the real-world scene will inevitably be less precise.

Our setup is depicted in Figure 6 and described in details in Appendix Section C.3. We pick 8 of the 16 trained policies for testing. Each such policy is run for ten episodes and we report the fraction of successful episodes (Table 2). Most policies reach around 80% – 90% success rate. Two common failure modes are the robot failing to find the small roof because of noise in the depth image and the robot accidentally knocking down a structure when placing a block. The latter cannot always be attributed to a bad policy–our gripper sometimes releases objects off-center. Since the real-world gripper is significantly larger than the simulated one, it fails if objects get pushed too close to each other.

<table>
<thead>
<tr>
<th>Task</th>
<th>Successes</th>
<th># Pick Failures</th>
<th># Place Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1b1r</td>
<td>9/10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1l1r</td>
<td>8/10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1l2r</td>
<td>9/10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1b1l1r</td>
<td>9/10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1l1b1r</td>
<td>10/10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1l2b2r</td>
<td>8/10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1l1l1r</td>
<td>7/10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1l1l2r</td>
<td>8/10</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Real-world experiment with a UR5 robotic arm (Figure 6). The environment was set up in the same way as the simulated workspace and we evaluated policies trained in simulation in 10 trials. Each trial had a budget of 20 time steps. We break down failures into two components: pick failures and place failures. Pick failures mean that the robot could not find the desired object or it could not pick it with sufficient precision. Place failures occur when the robot fails to place an object on the appropriate structure or it topples the structure over.

7 DISCUSSION AND CONCLUSION

In this work, we proposed a method for efficient exploration using action priors. Our approach to learning action priors involves solving a set of training tasks with imitation learning and summarizing the learned policies in a single action prior neural network. This network is trained on sets of optimal actions predicted by the policies with an addition of a task classifier that determines which policies are relevant in each state.

In contrast to prior work on action priors, which has predominantly considered grid-world-like environments, our method is applicable to domains with image states and thousands of actions. In addition to a proof-of-concept Fruits World domain, where it always finds near-optimal policies, we demonstrate performance on a simulated robotic block stacking task. A deep Q-network augmented with an action prior can solve 14 out of the 16 block stacking tasks within 100k episodes. Moreover, the policies trained in simulation can also be deployed on the real robotic arm with only a small drop in their success rates.

A future direction of our work is to develop action priors that are suitable to learning online during training on a new task. In this way, the action prior would progressively increase the chance of hitting the goal state while exploring. A natural extension of our manipulation task is to consider a full SE(3) action space (i.e. grasping from any angle).
A  STACKING GRAMMAR

Instead of arbitrarily choosing tasks, we characterize the whole family of possible tasks. Here, we define a simple grammar of stacking tasks. It only captures the notion of a stack, so it cannot represent tasks such as two blocks in a row. The only notion of two blocks being near one another is if a long block or a long roof can be put on top of them.

We have the following terminals: one block (1b), two blocks next to each other (2b), one brick (1l), short roof (1r) and long roof (2r). The non-terminals are ground (G), wildcard (W), short (S) and long (L). The starting symbol is the ground.

The rules allow for stacking of long objects (two blocks next to each other or one long block) on long objects, short on short and short on long. We separate the different possible outcomes of a rule by commas.

- G $\rightarrow$ 1bS, 2bW, 1lW
- W $\rightarrow$ S, L
- S $\rightarrow$ 1bS, 1b, 1r
- L $\rightarrow$ 1lW, 2bW, 1l, 2b, 2r

If we restrict the maximum height to three and only select structures that end with a roof, we get the 16 tasks used in our experiments.

B  EXPERT POLICIES

Action priors as well as our weight sharing and Actor-Mimic baselines require expert policies for training tasks. Next, we describe the methods we used to train expert policies in Fruits World and Block Stacking, but action priors are invariant to these design decisions. In both cases, we start with a replay buffer pre-populated with expert demonstrations and learn a near-optimal policy, which we also call an expert.

This might seem redundant, but, as you will see, a method that is used to generate expert demonstrations is not always able to select the optimal action in an arbitrary state. Moreover, the expert demonstrations contain optimal actions, not distributions over all actions, which are required by Actor-Mimic [16].

**Fruits World:** We use the same DQN both for the experts and for training on the testing task (Section C.1), but the expert version is given a pre-populated replay buffer. The buffer contains 50k transitions generated by a policy that executes the optimal action with 50% probability, and a random action otherwise. The optimal action can be easily computed given the full state of the environment. The expert is trained for 50k steps offline on this buffer; then, it is trained online for another 50k steps with new experiences mixed into the pre-populated buffer.

**Simulated Block Stacking:** We use an imitation learning method called SDQfD [31] to learn expert policies. Similarly to above, it requires a replay buffer pre-populated with expert demonstrations. Following Wang et al. [31], we start in the goal state (i.e., with a simulator initialized so that a particular goal structure is build), pick blocks from the top of the structure, and place them in empty spots on the ground. Since all actions are reversible, a deconstruction episode played backward looks like the agent is building the goal structure. We consider the reversed episodes to be expert demonstrations. Programming an initialization function for each structure and a deconstruction policy is much easier than having a custom planner for each task. Note that the initialization function and deconstruction policy is used only for training tasks, not the testing task.

The objective of SDQfD is a weighted sum of a temporal-difference loss $L_{TD}$ (as in DQN [15]) and a term $L_{SLM}$ (with weight $\omega$) that penalizes action not selected by an expert with high predicted values

$$L_{SDQfD} = L_{TD} + \omega L_{SLM}. \quad (9)$$

During training, the method identifies the set of non-expert actions $A_{s,a_e}$ with values higher than the value of an expert action $a_e$ minus a margin $l(a_e, a)$ (positive constant for $a_e \neq a$; zero otherwise)

$$A_{s,a_e} = \{ a \in A \mid Q(s, a) > Q(s, a_e) - l(a_e, a) \}. \quad (10)$$

The penalty called a "strict large margin" is then applied to the values of all actions in $A_{s,a_e}$

$$L_{SLM} = \frac{1}{|A_{s,a_e}|} \sum_{a \in A_{s,a_e}} Q(s, a) + l(a_e, a) - Q(s, a_e). \quad (11)$$

We pre-train SDQfD on a replay buffer with expert demonstrations for 10k steps. After pre-training, we alternate between taking one environment step according to the current policy and performing a training step. We maintain two separate buffers: one for expert and one for on-policy transitions. Each batch of training data contains an equal number of samples from both buffers. The strict large margin is only computed for the expert data.

C  EXPERIMENT DETAILS

C.1  Fruits World

We use the same neural network for both the DQN and the action prior: it has two hidden layers with 256 neurons and we use the ReLU non-linearity in-between. The DQN predicts a Q-value for each action and the action prior predicts the log probability of an action being optimal for each action separately.

Baseline DQNs are trained for 100k steps with an $\epsilon$-greedy policy that linearly decreases from 1.0 to 0.1 over 80k steps. The learning rate was set to $5 \times 10^{-4}$ and we use prioritized replay with default parameters, dueling network and double Q-learning [24, 30, 32].
Each action prior network is trained for 10k steps with a learning rate of 0.01. For the transfer experiments, we train a DQN with the same parameters as in the training tasks, except it uses an action prior for exploration, or it is initialized with Actor-Mimic weights (see Section 6.2).

C.2 Simulated Block Stacking

We use a modified version of the U-Net architecture [20] for all our networks in this experiment. The schema of the fully-convolutional network is included in Figure 7.

To train the SDQfD, we collect 50k expert trajectories obtained by reversing deconstruction experience [35]. We pre-train model on this experience for 10k steps. Then we train the model while it interacts with the simulator for 40k episode. Each episode has a maximum length of 20. The learning rate is set to $5 \times 10^{-5}$ and both the large margin weight and the margin coefficients are set to 0.1. There is no exploration, the batch size is set to 32 and the discount to 0.9. We run five simulated environments in parallel—take one step in each environment, collect the transitions, take one training step of the SDQfD and repeat.

For the training datasets, we collect 20k steps for each of the 15 tasks used to train an action prior and concatenate the experience. The task classifier is trained on this data for 20k steps with a learning rate of $10^{-3}$, weight decay of $10^{-5}$ and batch size set to 32. We use a probability threshold for relevant tasks $\theta$ of 0.05. Action priors are trained with the same settings except for a batch size of 50 for 10k training steps.

In the transfer experiment, we use an action prior probability threshold $\sigma$ of 0.1. During exploration, the model only selects actions proposed by the action prior. We train a DQN for 100k episodes with the modified $\epsilon$-greedy policy. $\epsilon$ linearly decays from 1.0 to 0.01 for 80k episodes. The learning rate is set to $10^{-4}$, batch size to 32 and the discount factor to 0.9. We use prioritized experience replay with default parameters, but we do not use the weighting of the sampled transitions in the loss function.

In the weight sharing experiments, the weights $L^2$ penalty $\omega_{WS}$ is set to 0.1.

C.3 Real-World Block Stacking

We tested the trained model on a Universal Robots UR5 arm with a Robotiq 2F-85 Gripper. The observation is provided by an Occipital Structure sensor pointing to the workspace from top-down. Figure 6 shows the robot experiment setup. All task parameters mirror the simulation. We run 10 trials for each task, and the maximum number of steps for each trial is 20. Figure 8 shows an example run of task 1l2b2r.

D ADDITIONAL RESULTS

Table 9 shows results for experiment #2 (Section 6.3) for all 16 tasks. As we stated in the main text, we found that task classifier helps in complex tasks and decreasing the probability threshold $\sigma$ tends to increase the success rates of the action priors that use the task classifier.

Figure 7: A modified U-Net network architecture. The inputs are a 90×90 depth image of the environment $I$ and a 24×24 zoomed-in image of an object the robot picked up. If the robot is not holding anything, all pixel values are set to zero. 3×3 conv, 32 means a convolutional layer with 32 filters of size 3. All max pooling layers use a kernel of size 2 and a stride of 2. We omit the ReLU activations in the diagram.
Figure 8: One example run of the 1l2b2r task.

Figure 9: We show the same results as in Figure 5 in the main test, but for all 16 tasks.
## Table 3: Additional Fruits World results accompanying Figure 2. We report the average reward over 10 runs in the middle of training (Mid.) and at the end (Fin.). Num. fruits refers to the number of fruits the agent is supposed to pick—it determines the difficulty of the task. Each row is an average over all tasks involving picking up the specified number of fruits.

<table>
<thead>
<tr>
<th>Num. fruits</th>
<th>AP (ours)</th>
<th>DQN</th>
<th>AM-share</th>
<th>AM-freeze</th>
<th>AM-prog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruit combinations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0.98</td>
<td>1</td>
<td>0.98</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>1</td>
<td>0.02</td>
<td>0.68</td>
<td>0.05</td>
</tr>
<tr>
<td>Mean</td>
<td>0.99</td>
<td>1</td>
<td>0.75</td>
<td>0.92</td>
<td>0.76</td>
</tr>
<tr>
<td>Fruit sequences</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>3</td>
<td>0.94</td>
<td>0.99</td>
<td>0.54</td>
<td>0.96</td>
<td>0.59</td>
</tr>
<tr>
<td>4</td>
<td>0.77</td>
<td>0.93</td>
<td>-0.02</td>
<td>0</td>
<td>-0.02</td>
</tr>
<tr>
<td>Mean</td>
<td>0.93</td>
<td>0.98</td>
<td>0.63</td>
<td>0.74</td>
<td>0.64</td>
</tr>
</tbody>
</table>