Policy Caches with Successor Features

Mark Nemecek [1] **Ronald Parr** [1]

Abstract

Transfer in reinforcement learning is based on the idea that it is possible to use what is learned in one task to improve the learning process in another task. For transfer between tasks which share transition dynamics but differ in reward function, successor features have been shown to be a useful representation which allows for efficient computation of action-value functions for previously-learned policies in new tasks. These functions induce policies in the new tasks, so an agent may not need to learn a new policy for each new task it encounters, especially if it is allowed some amount of suboptimality in those tasks. We present new bounds for the performance of optimal policies in a new task, as well as an approach to use these bounds to decide, when presented with a new task, whether to use cached policies or learn a new policy.

1. Introduction

When an agent learns to perform a set of separate tasks, one might hope that what the agent learns in one task will improve learning in subsequent tasks. This is called *transfer*, and the improvement may take different forms. Ultimately, the benefit of transfer is a reduction in the number of samples or amount of time required for the agent to reach a desired level of performance in the new task. When collecting samples is an expensive process – as is true for many real-world problems – such a reduction can be a large boon. While this impact is often considered during the training of a new policy, we will show that it also applies when an agent can avoid learning a new policy entirely because those it has previously cached are sufficient to meet the performance requirement.

The extent to which transfer is realized in practice or analyzed in theory depends largely upon the relationship between the tasks involved. We consider transfer between tasks which vary in their reward functions, but where the dynamics remain the same. Although limited in scope, this range of changes can induce drastically different optimal behaviors. For example, a change in the reward function for a driving task could change the optimal policy from a safe driving policy that avoids collisions to a demolition derby style policy that seeks them out. While real-world scenarios may not include such a wide range of behaviors, it is quite reasonable to expect that an agent will be confronted with changing costs or rewards that reflect differing priorities or circumstances over the lifetime of the agent. Even if the change in reward function is clearly communicated, the adequacy of previous policies for the new task may not be obvious. The question we address in this paper is: *When presented with a new task (reward function), can policies from previous tasks suffice, or should we learn a new policy?*

To answer this question, we build upon earlier work of Barreto et al. (2017), which assumes that the reward function can be decomposed into a linear combination of features, and uses successor features (SFs), an extension of the successor representation (Dayan, 1993), to learn Q-values that are parameterized by the reward weights. This permits efficient computation of the Q-values for a known policy under the reward function from a new task, regardless of which task was used to learn the policy. Although Barreto et al. used successor features with *Generalized Policy Improvement* (GPI) to provide a lower bound on the value of reusing previous policies in a new task, they did not provide a strong bound on the gap between this and the optimal policy for the new task, leaving little practical guidance on when it would be worth learning a new policy for the new task. Our work addresses this limitation.

Hunt et al. indirectly include an upper bound for the optimal soft Q-function in a new task (Hunt et al., 2019) through their Theorem 3.2 under the assumption that the new reward function is a convex combination of the base reward functions. As soft Q-learning approaches hard-max Q-learning in the limit as the temperature parameter goes to zero, intuition might suggest that their theorem would apply in the same way, but this is not the case. Additionally, their approach requires estimating the function C_b^{∞}, which corrects for the divergence between policies, for each new task. While this can be done off-policy with the same samples used to estimate the base soft Q-functions, this is not neces-

[1]Department of Computer Science, Duke University, Durham, North Carolina, USA. Correspondence to: Mark Nemecek <markn@cs.duke.edu>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

sarily any easier than learning a new Q-function directly and becomes more challenging as the number of base policies increases. They experiment with a heuristic that allows them to learn just one C^∞ function and approximate any other efficiently, but it relies on strong assumptions and doesn't apply when there are more than two base policies.

Our first contribution is to address these limitations by presenting a new upper bound on the hard-max Q-function of the optimal policy for a new task when that task is a conical combination of previous tasks. This result is similar to the bound from Hunt et al., but we prove that it applies to hard-max Q-functions and generalizes to conical combinations. This bound can be calculated efficiently for a given state and action using cached successor features.

Our second contribution is a method by which an agent can decide whether to learn a new policy when presented with a novel task. Based on the assumption that the agent has cached successor features for a set of known reward functions, this method uses our new upper bound to determine if policies from the existing cache suffice based on a given performance threshold, or if it is worth learning a new policy. This method does not require estimating a corrective function. Our experimental results show how the bounds tighten as policies are added to the cache, illustrating the trade-off between using cached policies and growing the cache, and demonstrating that significant storage and computational savings are possible with a given performance requirement.

2. Background

A *Markov Decision Process* (MDP) is as a 5-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ with state space $\mathcal{S}$; an action space $\mathcal{A}$; a Markovian transition model P, where $P(s'|s,a)$ is the probability of transitioning to state s' after taking action a in state s; a reward function $R(s,a)$ which is the reward for taking action a in state s; and discount factor $\gamma \in [0,1)$, applied to future rewards. We assume WLOG that all rewards are non-negative and bounded since shifting a reward function by a constant does not alter the optimal policies. Policy π maps states to actions, $\pi(s)$ denoting the action to take in state s. For any policy, an MDP is reduced to a Markov Reward Process with transition probabilities corresponding to those for the policy-specified action in the MDP.

A *value function* $V^\pi(s)$ is the expected total discounted reward for starting in state s and performing actions according to π. The *action-value function* $Q^\pi(s,a)$ gives the value of taking action a from state s and following π afterwards. The goal of a learning agent is to learn the optimal policy π^* which maximizes the expected future reward from each state. We can express the optimal value function V^* and action-value function Q^* via the Bellman equation as:

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

In a reinforcement learning setting, the agent does not have access to P or R. Experience is collected by acting in the environment and usually takes the form of a set of tuples, (s,a,r,s'). The agent can then use these samples to learn a policy for the environment by, for example, using a model-based approach where it attempts to learn the underlying MDP or a model-free approach where it learns a value or action-value function directly and extracts a policy from it.

In many useful domains, the state space is too large (or possibly infinite). An exact, tabular representations of the value function may be intractable, and such problems can only be solved approximately. Error may be introduced in several ways, such as the use of a function approximator which is insufficiently expressive, or being unable to train on enough samples. The difference between a value function V and an approximation $\tilde{V}$ is the *approximation error*, ϵ:

$$\epsilon(s) = |V(s) - \tilde{V}(s)|$$

The underlying idea of *transfer learning* for reinforcement learning (Taylor & Stone, 2009) is that through learning to perform well in one or more tasks, the learning process for a new task should be improved in some way, typically by reducing the amount of training in the new task required to reach a given level of performance. As shown in Section 5, an agent may even be able to avoid learning a new policy entirely if it can determine that its cached policies are sufficiently good for the novel task. While there are many notions of transfer, we focus on the case where the difference between tasks lies solely in their reward functions, i.e., the dynamics and other aspects of the environment remain the same. We build upon the previous work of Barreto et al. (2017), described in more detail in the next section. More distantly related work is discussed in section 7.

3. Successor Representation and Successor Features

The successor representation (SR) (Dayan, 1993) encodes the expected, discounted, cumulative sum of future state visitations under a given policy. In the tabular case, the transition function is a matrix P^π, and the reward function is a vector $\mathbf{r}$. The value function for a policy decomposes

into the product of the SR, Λ^π, and the reward vector.

$$\begin{aligned} V^\pi &= \mathbf{r} + \gamma P^\pi \mathbf{r} + (\gamma P^\pi)^2 \mathbf{r} + \cdots \\ &= (I - \gamma P^\pi)^{-1} \mathbf{r} \\ &= \Lambda^\pi \mathbf{r} \end{aligned}$$

Successor features (SFs) (Barreto et al., 2017) have been proposed as a generalization to the successor representation. SFs assume that the reward function can be decomposed into a linear combination of features, $r(s,a,s') = \phi(s,a,s')^T \mathbf{w}$, where $\phi(s,a,s')^T$ is a row vector of features and $\mathbf{w}$ is the weights, changing the representation from one based on state visitation to one based on feature occurrences. Using this assumption, successor features $\psi^\pi(s,a)$ are derived from the action-value function for a given policy π, $Q^\pi(s,a)$, which we can write as the expected sum of discounted rewards starting from timestep t. Here, r_t represents the reward received, while S_t and A_t are random variables representing the state and action taken at t, respectively.

$$\begin{aligned} Q^\pi(s,a) &= E^\pi\big[\sum_{i=t}^{\infty} \gamma^{i-t} r_{i+1} | S_t = s, A_t = a\big] \\ &= E^\pi\big[\sum_{i=t}^{\infty} \gamma^{i-t} \phi_{i+1}^T \mathbf{w} | S_t = s, A_t = a\big] \\ &= \psi^\pi(s,a)^T \mathbf{w} \end{aligned}$$

In the tabular case, i.e., where the reward features $\phi(s,a,s')$ are a one-hot vector in $\mathbb{R}^{|S|^2|A|}$, the SFs become the discounted sum of transition occurrences, thus extending the SR concept from state space to state-action-next-state space. The reward weights $\mathbf{w}$ can also be factored out from the Bellman equation for the action-value function in order to define a new Bellman equation where the reward features take the place of the rewards themselves. This allows us to apply any RL method to learn the SFs. Where $\phi(s,a) = E[\phi(s,a,s')]$, this equation is:

$$\psi^\pi(s,a) = \phi(s,a) + \gamma \sum_{s'} P(s,a,s') \psi^\pi(s', \pi(s'))$$

4. Bounds on Policy Cache Performance

Methods such as GPI (Barreto et al., 2017) give guidance on how to construct a policy for a new task that is no worse, and possibly better, than any previously stored policy. Such composite policies, however, are unlikely to be optimal. Successor features make it easy to compute a lower bound on the performance of an existing policy in a new task and previous work by Hunt et al. (2019) allows for an easily-computed upper bound for soft Q-functions, if the value functions for known policies in the new task are available. However, this does not extend to hard-max Q-learning, which we discuss further in Section 7. In this section, we present a similar upper bound for hard-max Q-functions by making direct use of successor features. Our result applies to combinations that we term *positive conical combinations*, which are conical combinations where the sum of the weights is positive. Since conical combinations only require that the weights are non-negative, these generalize convex combinations.

Let $\mathcal{T}$ be a set of tasks which differ only in their reward functions. For any task $i \in \mathcal{T}$, let r_i be the reward function, π_i be an optimal policy for r_i, and let $V_j^{\pi_i}$ be the value function for policy π_i executed in task j. As π_i is an optimal policy for task i, it follows that the value under π_i in task i is at least as good as under any other policy π_j:

$$V_i^{\pi_i} \geq V_i^{\pi_j} \; \forall i, j \in \mathcal{T}$$

Let Λ^{π_i} be a matrix with each row corresponding to a state $s \in \mathcal{S}$, and each column corresponding to a successor feature. Thus, each row corresponds to a row vector of successor features for a state under π_i: $\Lambda^{\pi_i}(s) = \psi^{\pi_i}(s, \pi_i(s))$ and is analogous to the relationship between V^{π_i} and Q^{π_i}.

Definition 1. *An approximate successor features matrix $\tilde{\Lambda}^{\pi_i}$ differs from the exact matrix Λ^{π_i} by a state-wise error matrix $\Delta^{\pi_i} = \tilde{\Lambda}^{\pi_i} - \Lambda^{\pi_i}$.*

Definition 2. *If $\boldsymbol{w}_j$ is the weight vector for task j, then the approximate value function matrix induced by $\tilde{\Lambda}^{\pi_i}$ in task j is the column vector $\tilde{V}_j^{\pi_i} = \tilde{\Lambda}^{\pi_i} \boldsymbol{w}_j$.*

Definition 3. *The state-wise approximation error of $\tilde{V}_j^{\pi_i}$ is the column vector $\boldsymbol{\epsilon}_j^{\pi_i} = |\tilde{V}_j^{\pi_i} - V_j^{\pi_i}| = |\Delta^{\pi_i} \boldsymbol{w}_j|$ where $|\cdot|$ denotes the element-wise absolute value.*

We now consider a task R with reward function $r_R = \sum_{i \in \mathcal{T}} \alpha_i r_i$ such that $\alpha_i \geq 0$ and $\sum_{i \in \mathcal{T}} \alpha_i > 0$, i.e., r_R is a positive conical combination of the reward functions of the tasks in $\mathcal{T}$. The proof of the theorem below (see appendix) provides an alternate derivation of the lower bound from Barreto et al., but does not improve upon it, while the upper bound is more general than those offered in Hunt et al. (2019) and applies to hard-max Q-learning.

Theorem 1. *If π_R is an optimal policy under r_R and r_R is a positive conical combination, i.e., $\alpha_i \geq 0$ and $\alpha = \sum_{i \in \mathcal{T}} \alpha_i > 0$, then*

$$\begin{aligned} \max_{i \in \mathcal{T}} \big[\tilde{V}_R^{\pi_i}(s) - \boldsymbol{\epsilon}_R^{\pi_i}(s)\big] \leq \\ V_R^{\pi_R}(s) \leq \\ \sum_{i \in \mathcal{T}} \alpha_i \tilde{V}_i^{\pi_i}(s) + \sum_{i \in \mathcal{T}} \alpha_i \boldsymbol{\epsilon}_i^{\pi_i}(s) \end{aligned}$$

Proof. See appendix. □

Theorem 1 turns out to be quite powerful in practice because the optimality of a value function for a particular reward

function significantly constrains the value function for other reward functions with similar weights. Using the information contained in the corresponding value functions allows us to avoid the $\frac{1}{1-\gamma}$ factor that is often found in value function bounds, as this can be quite large for γ close to 1. The bound presented in Theorem 2 from Barreto et al. (2017) is not far from the best that can be done when only the reward functions are known (see appendix) but, as shown in Figure 2(c), it can be quite loose compared to our bound.

5. Policy Cache Construction

Although Barreto et al. (2017) showed that SFs are useful for transfer between tasks, in their experiments a new set of SFs was learned for each task and stored in a cache for continued use. As each set of SFs corresponds to a policy, this can be thought of as storing policies in a policy cache. However, for large SF models or a large number of tasks, this could be space-prohibitive. Instead, Theorem 1 can provide guidance on whether to use existing policies, or create a new one.

We start with a set of base tasks defined by a set of reward feature weights, and assume all subsequent tasks have rewards within the conical hull of the initial set. Algorithm 1 uses Theorem 1 to decide if the current policy cache is sufficient, given a performance threshold. If the gap between the upper and lower bounds is small enough for a given state, then we know that the value of the best policy in the cache must be within some small factor of the value for the optimal policy. If the agent is given the starting state distribution or can approximate it based on previous experience, it can calculate the expected size of the gap for that state distribution. With that information, the agent decides if the performance of the existing policy cache is sufficient.

Algorithm 1 LearnNewPolicy?

Input: policy cache Π, set of old reward weights W, new reward weights w_{new}, start state s, threshold η
$ub = \text{CalcUpperBound}(\Pi, W, w_{new}, s)$
$lb = \max_{\psi^\pi \in \Pi}[\max_a \psi^\pi(s,a)^T w_{new}]$
$ratio = (ub - lb)/|lb|$
return $(ratio > \eta)$

Algorithm 2 CalcUpperBound

Input: policy cache Π, set of old reward weights W, new reward weights w_{new}, start state s
$V = \{W_i^T \psi^\pi(s, \pi(s)) \mid \psi^\pi \in \Pi\}$
$vars, objectiveValue = SolveLP(V, W, w_{new})$
return $objectiveValue$

Once an additional policy has been added to the cache beyond those for the base tasks, any subsequent task may not be a *unique* conical combination of previous tasks for which policies are stored. Therefore, we use a linear program in Algorithm 2 to find the valid combination which minimizes the upper bound. For this LP, the variables are the coefficients α_i, W is the set of reward weight vectors for the tasks in which the policies in the cache were learned, w_{new} is vector of reward weights for the new task, m is the number of policies in the cache, and s is the start state:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} \alpha_i V_i^{\pi_i}(s) \\ \text{subject to} \quad & \sum_{i=1}^{m} \alpha_i W_i = w_{new} \\ & \alpha_i \geq 0, \qquad\qquad i = 1, ..., m \end{aligned}$$

6. Experimental Results

We demonstrate this new method in three environments. Two of them, Gridworld and Reacher, involving the agent interacting with objects. The former is a discrete gridworld which involves collecting objects before reaching a goal, and the latter is a robotic arm simulation which requires touching targets scattered within reach of the arm. The remaining environment, Terrainworld, is based around a grid where each cell contains "terrain" which regulates the reward for moving to that cell.

In our experiments, the reward features are provided to the agent. We train the agent on each task independently in order to avoid the confounding effect of training using an ensemble approach such as GPI.

As described in Algorithm 1, the agent starts with a policy cache containing policies for each of the base tasks for the given environment and when presented with each subsequent task, calculates the upper and lower bounds at the start state using the current policy cache and compares them. If the size of the gap exceeds threshold η, then a new policy is added to the cache.

As a matter of experimental convenience for the results below, the policies for all tasks were learned ahead of time, allowing us to compute these graphs quickly as "thought experiments" for different values of η. In practice, an agent would need to compute the policies as needed when a new task is presented, a potentially costly step that our method would help the agent avoid in many cases. By pre-computing the policies we are able to consider different values of η and approximate expectations over many different permutations of tasks in a computationally efficient manner. If we did not pre-compute these policies for our experiments, the graphs would look the same; it would just take much longer to produce them.

If the max weight assigned to any base task in the conical combination is known in advance, it may be desirable to

normalize the weights so that all subsequent tasks can be expressed as convex combinations of the base tasks. While not required by the theory, this upper bounds the suboptimality of any subsequent task by bounding the right hand side of Theorem 1. It also helps normalize results across domains by establishing a common scale. For these reasons, we focus on convex combinations of base tasks in our experiments.

6.1. Gridworld

In our first environment, inspired by Barreto et al. (2017),the underlying navigation problem is a discrete grid with an absorbing goal state. Although the grid used in our experiments, shown in Figure 1(a), is not large, the state space grows exponentially with the number of objects. The reward function for a task is defined by assigning a value to each object. The agent has four actions available: moving left, down, right, and up. For these experiments, all actions are deterministic and attempting to move into a wall (or the edge of the grid) causes the agent to stay in place.

The grid is constructed with four sections, each of which contains three paths of equal length which can be traversed in a forward direction (closer to the goal cell) and on each path lies an object which can be picked up. This structure means that there is a region around each object which is rarely traversed by an optimal policy for a task unless that object has a positive reward in that task. This allows for a large number of different optimal policies to be induced by the selection of reward weights.

For the 12 base tasks, a reward of 1.0 is received for reaching the goal and a reward of 2.0 is received for picking up one of the objects. Subsequent tasks were generated by creating a convex combination of the base task reward functions. In particular, we used the 1573 equal-weight convex combinations which include two to five (inclusive) of the base tasks. We computed the SFs for each of these tasks and then simulated what would happen when an agent used our method to choose whether to learn a new policy to add to its cache when presented with a new task given different values of η. Our results are averaged over 10000 different permutations of the generated tasks.

We used a tabular representation for the SFs and computed them using modified policy iteration (MPI). This approach was chosen to focus on the policy cache construction problem rather than the SF learning problem, as the approximation error is essentially zero. Results for additional experiments, one with an image-based representation and one with engineered features, are included in the appendix. These experiments involve training a neural network from samples to approximate the SFs, but demonstrate the benefits of our method even when there is some approximation error.

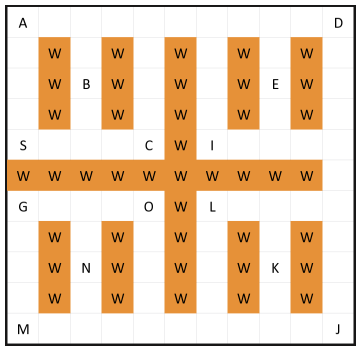
Figure 2(a), shows the values of the start state for each task under the learned optimal policy as well as the upper and lower bounds for several chosen thresholds, including the case where the agent learns a policy for every task, which is equivalent to $\eta = 0.0$. The upper bound shows little movement for $\eta > 0.2$, while the lower bound shows most of its movement for $\eta > 0.2$. The different sensitivity to η arises because different factors are driving the upper and lower bounds. The lower bound is driven empirically by the diversity of policies required to achieve good performance in the range of tasks for the domain. Once a critical mass of good policies is achieved, the lower bound may not improve much by adding more policies. In contrast, the upper bound is driven more geometrically. It is influenced by the slope of the value function (magnitude of the successor features) and the distance in weight space of the new tasks from the weights of tasks with corresponding cached policies. Thus, the upper bounds can be more sensitive to sparse coverage of weight space by the cached policies.

To quantify the reduction in policy cache size due to the chosen threshold, in Figure 2(b) we show the number of policies added to the cache as a function of the number of tasks experienced for several different thresholds. The "all policies" line represents the case where a new policy is cached for all tasks. The $\eta = 0.1$ line demonstrates that an agent that generates new policies only when there is a chance of doing 10% better than an existing policy avoids computing new policies about $\frac{2}{3}$ of the time.

Figure 2(c) compares our upper bound to that which one can calculate using Theorem 2 of Barreto et al. (2017). For our bound, we use the curve for the policy cache built with $\eta = 0.1$ and the Barreto et al. curve is calculated using the reward weights from all previously-encountered tasks. Our bound is significantly tighter and the gap between them dwarfs the gap between our upper and lower bounds.

6.2. Terrainworld

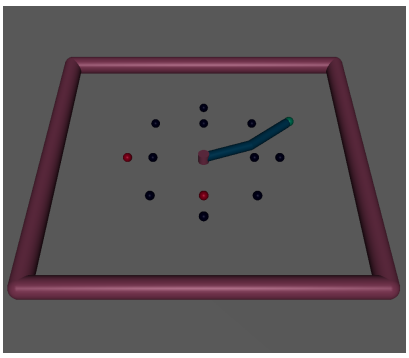
Our second environment, Terrainworld, is related to Gridworld, but instead of picking up objects, the agent traverses different types of "terrain" that have been assigned to each cell. Each cell contains a combination of one or more of the available terrain types and there is a reward feature for each terrain type. Since every cell has terrain, these reward features are very dense, unlike those in Gridworld. In Figure 1(b), the numbers are bitmasks where each bit corresponds to one of the seven terrain types. The reward features for each cell are the coefficients of the equal-weight convex combination of the terrain types indicated by the bits set to one in the bitmask. As a metaphor, we can think of the reward features as indicating the terrain present in each cell and the reward weights as indicating the performance of a vehicle in each terrain type. The start cell is marked with X and the goal cell with Y.



(a) GridWorld with start (S), walls (W), goal (G), objects (other letters)

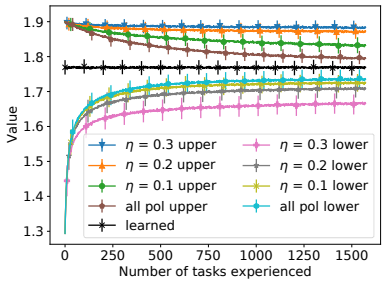
0:X	65	33	17	9	5	3	1
2	66	34	18	10	6	2	3
4	68	36	20	12	4	6	5
8	72	40	24	8	12	10	9
16	80	48	16	24	20	18	17
32	96	32	48	40	36	34	33
64	64	96	80	72	68	66	0:Y

(b) Terrainworld with numbers indicating terrain types

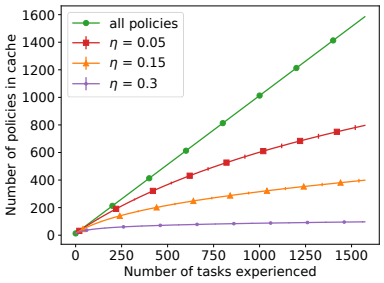


(c) Reacher with red active and black inactive targets

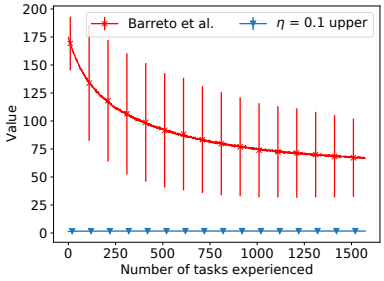
Figure 1. Environment Visualizations



(a) Bounds for different thresholds, mean and variance



(b) Cache size for different thresholds, mean and std. dev.



(c) Our upper bound and Barreto at al. Theorem 2, mean and std. dev.

Figure 2. Data for the GridWorld environment, 10000 runs

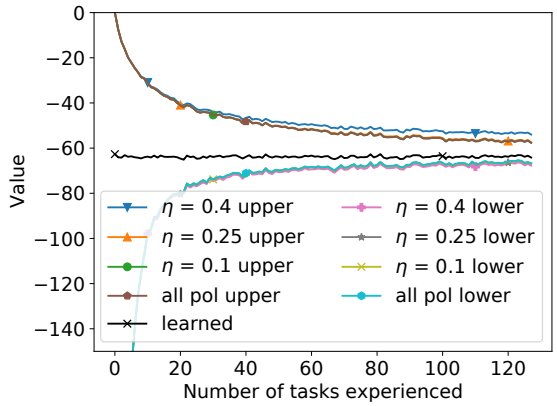
The seven base tasks each assign a weight of -1 to one of the seven terrain types and a weight of 0 to the goal. Subsequent tasks assign some combination of the coefficients 1 and 30 to each base task. Exhausting the possible combinations of this form results in 128 generated tasks. Our results are averaged over 5000 permutations of these generated tasks. Like Gridworld, we used a tabular representation for the SFs computed with MPI.

As before, we show the bounds in Figure 3(a) as a function of the number of experienced tasks. In this environment, the bounds tighten at a much greater rate and by a much greater magnitude as more tasks are experienced than occurred in Gridworld. This can be attributed to a larger gap between the optimal policy and the worst performing cached policies in a new task as well as the smaller number of optimal policies across all tasks (relative to the total number of tasks) – an agent is much more likely to have already cached a policy that is nearly-optimal for a new task. These characteristics make it appear as if there is little change in the bounds as η increases, but the change is still large beyond $\eta = 0.25$ compared to the gap between the bounds.

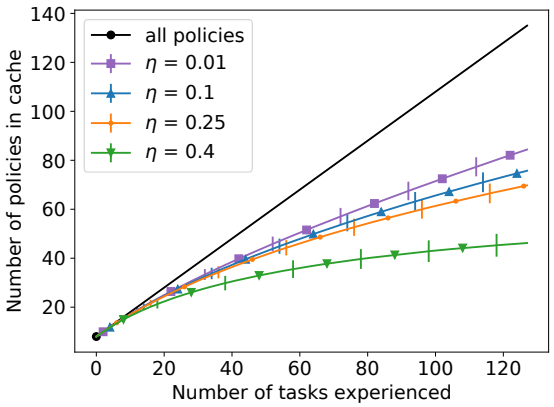
The results for cache growth shown in Figure 3(b) show that there is a drastic reduction in the number of cached policies even when little suboptimality is allowed via small values of η. Allowing a factor of just 0.01 reduces the number of cached policies by about $\frac{1}{3}$, and a greater effect occurs with larger η. This tells us that there is a lot of redundancy in the set of optimal policies for all tasks in this environment and we can significantly reduce an agent's computation and storage needs by weakening the guarantee on an agent's performance by a small amount.

6.3. Reacher

Our last environment, shown in Figure 1(c), is a modification of "Reacher-v2" from OpenAI Gym (Brockman et al., 2016), which is similar to the environment used by Barreto et al. (2017). The agent controls a two-part arm by applying torque to the two joints, and reward values can be assigned to "tapping" each of the 12 targets for the first time, which requires the end effector to be moving slowly when it comes in contact with the target.

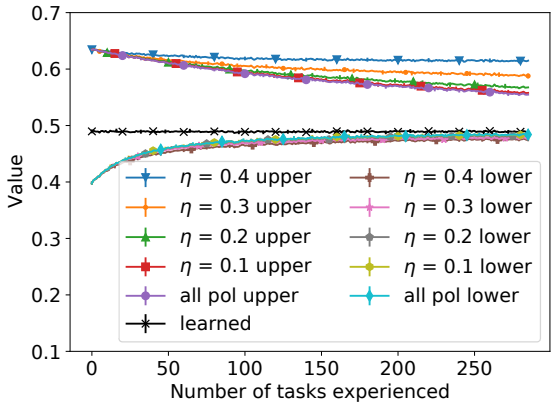


(a) Bounds for different thresholds, mean and variance

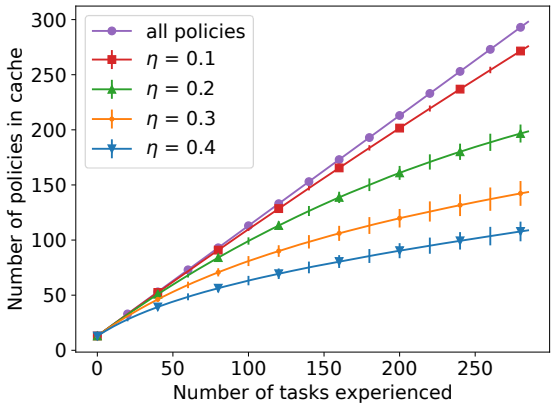


(b) Cache size for different thresholds, mean and std. dev.

Figure 3. Data for the Terrainworld environment, 5000 runs



(a) Bounds for different thresholds, mean and variance



(b) Cache size for different thresholds, mean and std. dev.

Figure 4. Data for the Reacher environment, 5000 runs

The two-dimensional action space was discretized into nine actions such that the agent can apply full clockwise, full counterclockwise, or no torque to each joint. The state representation consists of the sine and cosine of the two joint angles, the velocity of the end effector along the x- and y-axes, and the target inventory, which tracks which targets have already been tapped. The inventory is discrete, but adds an exponential factor in the number of objects to the size of the state space. The reward features are 12 delta functions indicating a tap of the corresponding target to which potential-based shaping (Ng et al., 1999) is applied based on the negative distances to the targets. Additionally, one feature for the norm of the control values is included, as an energy cost is incurred based on this norm.

We used a neural network to approximate the SFs, which were trained using stochastic gradient decent in a version of SFQL (Barreto et al., 2017) which was modified to train arbitrary neural network models and only operates with one set of SFs, i.e., GPI was not used in training. More detail on hyperparameters, network structures, and training procedures are in the appendix.

For this domain, there are 12 base tasks, which each assign a reward of 1 to a single target, plus 286 additional tasks, which were convex combinations of the base tasks. These were generated in a combinatorial manner by generating a convex combination with equal weights on a subset of the base tasks. We limited the generated tasks to those using two or three base tasks, which corresponds to two or three "active" targets, so 286 tasks are enough to exhaust the set of valid combinations of these sizes. This ensured that many different optimal policies were induced across tasks. The results were averaged over 5000 different permutations of the tasks.

The results in the Reacher domain show a somewhat different pattern to the Gridworld results. Like in Gridworld, the upper bounds in Figure 4(a) are greatly affected by the choice of η and loosen significantly as η increases, but in

contrast the lower bounds remain close to the same. This difference is largely the result of the gap between the best base task policy and the optimal policy being much smaller, something that is visible in hindsight that but that is not visible to an agent, which has no way of knowing where the next policy will lie in the gap between the upper and lower bounds. While this shows less room to loosen the lower bound with increasing η, the upper bound remains sensitive to η in this case, demonstrating a tradeoff between the number of policies and the guaranteed performance of the policy cache. The learned policy value curve is nearly horizontal, but this does not imply that all tasks have equal value; it is the result of taking the mean over many different permutations of the same set of tasks. In Figure 4(b), we can again see how the policy cache grows based on the choice of performance threshold.

7. Related Work

Hunt et al. consider a similar form of novel task, and their results indirectly include an upper bound in their Theorem A.1 (and 3.2) (Hunt et al., 2019) which is very similar to ours, if one drops the C term. However, while soft Q-learning approaches hard-max Q-learning in the limit as the temperature parameter goes to 0, their theorem cannot be extended to hard-max Q-functions in the same way. We can show this with a simple counterexample: Suppose the value of $\lim_{\alpha \to 0^+} C^{\infty}_{\mathbf{w}}(s, a)$ could be calculated for a set of hard-max Q-function-based policies with substantially different values, and suppose that the equality in Theorem A.1 held, which would mean that the value of the C term is not 0. Now suppose that the reward functions for all of the tasks were scaled by a positive constant, k. In that case, the corresponding optimal Q-functions would also be scaled by k. However, since the optimal hard-max policies do not change due to such a scaling, and the C term is based only on the policies themselves and not the reward functions, the C term would not change with the scaling and thus the equality could not continue to hold. While it is intuitive to think that one could still drop the C term to get an upper bound on the hard-max Q-function, this is not supported by the Hunt et al. proof. Our Theorem 1 uses a different proof technique that covers the common hard-max case, as well as positive conical combinations.

Our bounds in Theorem 1 are similar to those shown by Parr (1998) where the value of a state is dependent on a policy and a value vector which gives values for some subset of the states in the MDP. Here, however, we perform this analysis in a different setting in the context of SFs.

Bayesian Policy Reuse (Rosman et al., 2016), is superficially similar to our work in that it maintains a collection of stored policies that can be reused on similar tasks. The focus of that work, however, is efficient evaluation of the stored policies on new tasks that differ in unknown ways from known tasks. Our approach assumes that the new task differs in a known way, and establishes suboptimality bounds that influence when to add new policies.

Abel et al. (2018) consider how to use previous policies or value functions to jumpstart learning in a new problem. Their focus is on how to select or combine previous policies to improve initial performance and potentially reduce the number of samples needed to achieve optimal performance on a new problem. Unlike our work, they do not provide bounds on the suboptimality with respect to the optimal policy for the new task.

In terms of the types of tasks, the -AND- composition of Van Niekerk et al. (2019) is most similar to our experiments. In that case, they consider composition of base tasks which differ in their absorbing sets, and the composed task terminates after all objects are collected. However, they do not calculate their bounds in practice, which would require an expensive expectation to get the C function (a measure of divergence between policies), nor do they selectively learn new policies. Our approach uses the actual approximate value function which is calculated efficiently using the stored successor feature approximators and the reward weights for the new task.

Lipschitz Lifelong RL from Lecarpentier et al. (2021) focuses on reducing sample complexity and involves a related upper bound, but solving the necessary sample-based fixed point equations to approximate their pseudometric for a novel task requires an amount of computation similar to solving the MDP. Our approach does not focus on sample complexity, but leverages successor features to provide easily computable bounds for a novel task with no samples from that task (or a small number if the start state distribution is unknown).

8. Discussion and Future Work

We have demonstrated empirically that our approach can be combined with function approximation. Although our bounds include approximation error, our experiments assume low approximation error and don't include approximation error when deciding whether to grow the cache. One approach to estimating this could be through Monte Carlo estimates of the successor features of the initial state distribution. This would be done once for each policy by executing each in the task for which it was learned.

Universal Successor Feature Approximators (USFAs) (Borsa et al., 2019) offer the potential to have a parameterized set of successor features generalize across tasks. If the USFAs generalize well, this could obviate policy caches, but a question remains about whether such generalization can be guaranteed. It may be possible to extend our work to

USFAs to create a method for determining when the USFAs are sufficient to guarantee a certain level of performance in a new task or if they need to be trained on the new task.

Inverse RL (IRL) methods typically make the same assumption that the reward function can be decomposed into a linear combination of features. Some of the insights gained in this line of work may also apply to IRL.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1815300. We would like to thank Kate O'Hanlon, Barrett Ames, Ajinkya Kokandakar, and Lesia Semenova for their valuable feedback and proofreading efforts as well as our anonymous reviewers for their suggestions for improving clarity.

References

Abel, D., Jinnai, Y., Guo, S. Y., Konidaris, G., and Littman, M. Policy and value transfer in lifelong reinforcement learning. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 20–29. PMLR, 2018.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. P., and Silver, D. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems 30*, pp. 4055–4065. Curran Associates, Inc., 2017.

Borsa, D., Barreto, A., Quan, J., Mankowitz, D. J., van Hasselt, H., Munos, R., Silver, D., and Schaul, T. Universal successor features approximators. In *International Conference on Learning Representations*, 2019.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

Dayan, P. Improving Generalization for Temporal Difference Learning: The Successor Representation. *Neural Computation*, 5(4):613–624, jul 1993. ISSN 0899-7667. doi: 10.1162/neco.1993.5.4.613.

Hunt, J., Barreto, A., Lillicrap, T., and Heess, N. Composing entropic policies using divergence correction. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2911–2920. PMLR, 2019.

Lecarpentier, E., Abel, D., Asadi, K., Jinnai, Y., Rachelson, E., and Littman, M. L. Lipschitz Lifelong Reinforcement Learning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, AAAI-21, pp. 8270–8278. AAAI Press, 2021.

Ng, A. Y., Harada, D., and Russell, S. J. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pp. 278–287. Morgan Kaufmann Publishers Inc., 1999. ISBN 1558606122.

Parr, R. Flexible Decomposition Algorithms for Weakly Coupled Markov Decision Problems. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 422–430, 1998.

Rosman, B., Hawasly, M., and Ramamoorthy, S. Bayesian policy reuse. *Machine Learning*, 104(1):99–127, 2016.

Taylor, M. E. and Stone, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685, 2009.

Van Niekerk, B., James, S., Earle, A., and Rosman, B. Composing value functions in reinforcement learning. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6401–6409. PMLR, 2019.