Transfer in Reinforcement Learning with Successor Features and Generalised Policy Improvement

André Barreto ANDREBARRETO@GOOGLE.COM
Will Dabney WDABNEY@GOOGLE.COM
Rémi Munos MUNOS@GOOGLE.COM
Jonathan Hunt JJHUNT@GOOGLE.COM
Tom Schaul SCHAUL@GOOGLE.COM
David Silver DAVIDSILVER@GOOGLE.COM
Hado van Hasselt HADO@GOOGLE.COM
Google DeepMind

Abstract

We propose a framework for transfer in reinforcement learning designed for the scenario where the reward function changes between tasks but the environment's dynamics remain the same. Our approach rests on two key ideas: *generalised policy improvement*, a generalisation of dynamic programming's policy improvement step that considers a set of policies rather than a single one, and *successor features*, a representation scheme that makes it possible to compute the value function of a policy under any reward function representable in a given form. Put together, the two ideas lead to a general transfer framework that integrates seamlessly within the reinforcement learning setting.

1. Introduction

In recent years reinforcement learning (RL) has undergone a major change in terms of the scale of its applications: from relatively small and well-controlled benchmarks the field has shifted to problems designed to be challenging for humans—who are now consistently outperformed by artificial agents in domains considered out of reach only a few years ago (Mnih et al., 2015; Bowling et al., 2015; Silver et al., 2016). Although there is still work to be done in the single-task RL setting, the question of how to design agents that can tackle multiple tasks seems now more relevant than ever. This inevitably brings about the subject of *transfer*.

Transfer refers to the notion that an agent should be able to generalise not only within a task but also across tasks (Taylor & Stone, 2009; Lazaric, 2012). This of course raises the question of what exactly a task is; here we adopt the view that a task is any partial accomplishment that will help the agent achieve its objectives in the long run. The specific way in which we make this definition concrete is to assume that the environment has fixed dynamics and each task corresponds to a different reward function.

Ideally, we want a transfer approach to have two important properties. First, the flow of information between tasks should not be dictated by a rigid diagram that reflects the relationship between the tasks themselves, such as hierarchical or temporal dependencies. On the contrary, information should be exchanged between tasks whenever useful. Second, rather than being posed as a separate problem, transfer should be integrated within RL as much as possible, preferably in a way that is transparent to the agent.

In this paper we propose an approach for transfer that has the two properties above. Our method builds on two conceptual pillars that complement each other. The first is a generalisation of Bellman's (1957) classic policy improvement theorem that extends the original result from one to multiple decision policies. The second pillar of our framework is a generalisation of Dayan's (1993) *successor representation* (SR) that makes it possible to compute the value function of a policy under any reward function that can be represented in a given form. Put together, these two ideas lead to a general framework that naturally incorporates transfer into the standard RL setting.

2. Background and Problem Formulation

As usual, we assume that the interaction between agent and environment can be modeled as a *Markov decision process* (MDP, Puterman, 1994). An MDP is defined as a tuple $M \equiv (\mathcal{S}, \mathcal{A}, p, R, \gamma)$. The sets $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, respectively. For each $s \in \mathcal{S}$ and each $a \in \mathcal{A}$ the function $p(\cdot|s, a)$ gives the next-state distribution upon taking action a in state s. We will often refer

Accepted at Lifelong Learning: A Reinforcement Learning Approach Workshop @ICML, Sydney, Australia, 2017. Copyright 2017 by the author(s).

to $p(\cdot|s, a)$ as the *dynamics* of the MDP. The reward received at transition $s \xrightarrow{a} s'$ is given by the random variable $R(s, a, s')$; often one is interested in the expected value of this variable, which we will denote by $r(s, a, s')$ or by $r(s, a) = \mathrm{E}_{S' \sim p(\cdot|s,a)}[r(s, a, S')]$. The discount factor $\gamma \in [0, 1)$ gives smaller weights to future rewards.

The goal of the agent in RL is to find a policy π—a mapping from states to actions—that maximises the expected discounted sum of rewards, also called the *return* $G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$, where $R_t = R(S_t, A_t, S_{t+1})$. One way to address this problem is to use methods derived from *dynamic programming* (DP), which heavily rely on the concept of a *value function* (Puterman, 1994). The *action-value function* of a policy π is defined as

$$Q^{\pi}(s, a) \equiv \mathrm{E}^{\pi}\left[G_t \,|\, S_t = s, A_t = a\right], \tag{1}$$

where $\mathrm{E}^{\pi}[\cdot]$ denotes expected value when following policy π. Once the value function of a policy π is known, we can derive a new policy π' that is *greedy* with respect to $Q^{\pi}(s, a)$, that is, $\pi'(s) \in \mathrm{argmax}_a Q^{\pi}(s, a)$. Policy π' is guaranteed to be at least as good as (if not better than) π. The computation of $Q^{\pi}(s, a)$ and π', called *policy evaluation* and *policy improvement*, define the basic mechanics of RL algorithms based on DP (Sutton & Barto, 1998).

In this paper we are interested in the problem of *transfer* when all components of an MDP are fixed except for the reward function. We now discuss one possible way of formalising this scenario. Suppose that the expected one-step reward associated with transition (s, a, s') is given by

$$r(s, a, s') = \boldsymbol{\phi}(s, a, s')^{\top}\mathbf{w}, \tag{2}$$

where $\boldsymbol{\phi}(s, a, s') \in \mathbb{R}^d$ are features of (s, a, s') and $\mathbf{w} \in \mathbb{R}^d$ are weights. Supposing that (2) is true is not restrictive because we are not making any assumptions about $\boldsymbol{\phi}(s, a, s')$: if we have $\phi_i(s, a, s') = r(s, a, s')$ for some i, for example, we can clearly recover any reward function.

If we fix the function $\boldsymbol{\phi}(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}^d$, any $\mathbf{w} \in \mathbb{R}^d$ gives rise to a new MDP. Based on this observation, we define

$$\mathcal{M}^{\phi}(\mathcal{S}, \mathcal{A}, p, \gamma) \equiv \{M(\mathcal{S}, \mathcal{A}, p, r, \gamma) | r(s, a, s') = \boldsymbol{\phi}(s, a, s')^{\top}\mathbf{w}\}, \tag{3}$$

that is, $\mathcal{M}^{\phi}$ is the set of MDPs induced by $\boldsymbol{\phi}$ through all possible instantiations of $\mathbf{w}$. Since what differentiates the MDPs in $\mathcal{M}^{\phi}$ is only the agent's goal, we will refer to $M \in \mathcal{M}^{\phi}$ as a *task*. Let M_i be a task in $\mathcal{M}^{\phi}$ defined by $\mathbf{w}_i \in \mathbb{R}^d$. We will use π_i^* to refer to an optimal policy of MDP M_i and use $Q_i^{\pi_i^*}$ to refer to its value function. The value function of π_i^* when executed in $M_j \in \mathcal{M}^{\phi}$ will be denoted by $Q_j^{\pi_i^*}$.

Our goal is to solve (a subset of) the tasks in the environment $\mathcal{M}^{\phi}$. Since the tasks are related, we would like to transfer knowledge accumulated in previous tasks to speed up learning in a new task. More specifically, we formulate the problem of transfer as follows. Let $\mathcal{M}, \mathcal{M}' \subset \mathcal{M}^{\phi}$ be two sets of tasks such that $\mathcal{M}' \subset \mathcal{M}$, and let M be any task. Then we say there is transfer if, after training on $\mathcal{M}$, the agent always performs as well or better on task M than if only trained on $\mathcal{M}'$. Note that $\mathcal{M}'$ can be the empty set.

3. Generalised Policy Improvement

One of the key results in DP is Bellman's (1957) policy improvement theorem. Basically, the theorem states that acting greedily with respect to a policy's value function gives rise to another policy whose performance is no worse than the former's. This is the driving force behind DP, and most RL algorithms that use the notion of a value function are exploiting Bellman's result in one way or another.

In this section we extend the policy improvement theorem to the scenario where the new policy is to be computed based on the value functions of a *set* of policies. We show that this extension can be done in a very natural way, by simply acting greedily with respect to the maximum over the value functions available. Our result is summarized in the theorem below.

Theorem 1. (Generalised Policy Improvement) *Let π_1, π_2, ..., π_n be n decision policies and let $\tilde{Q}^{\pi_1}$, $\tilde{Q}^{\pi_2}$, ..., $\tilde{Q}^{\pi_n}$ be approximations of their respective action-value functions such that $|Q^{\pi_i}(s, a) - \tilde{Q}^{\pi_i}(s, a)| \le \epsilon$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, and $i \in \{1, 2, ..., n\}$. Define $\pi(s) \in \mathrm{argmax}_a \max_i \tilde{Q}^{\pi_i}(s, a)$. Then,*

$$Q^{\pi}(s, a) \ge \max_i Q^{\pi_i}(s, a) - \frac{2}{1-\gamma}\epsilon \tag{4}$$

for any $s \in \mathcal{S}$ and any $a \in \mathcal{A}$, where Q^{π} is the action-value function of π.

The proofs of our theoretical results are available in an extended version of this paper (Barreto et al., 2016). As one can see, our theorem covers the case in which the policies' value functions are not computed exactly, either because function approximation is used or because some exact algorithm has not be run to completion. This error is captured by ϵ, which of course re-appears as a "penalty" term in the lower bound (4). Such a penalty is inherent to the presence of approximation in RL (Bertsekas & Tsitsiklis, 1996).

In order to contextualize generalised policy improvement (GPI) within the broader scenario of DP, suppose for a moment that $\epsilon = 0$. In this case Theorem 1 states that π will perform no worse than *all* of the policies π_1, $\pi_2, ..., \pi_n$ starting from *any* state. It is not difficult to see that π will be strictly better than all previous policies if no single policy dominates all other policies, that is, if $\mathrm{argmax}_i \max_a \tilde{Q}^{\pi_i}(s, a) \cap \mathrm{argmax}_i \max_a \tilde{Q}^{\pi_i}(s', a) = \emptyset$ for any $s, s' \in \mathcal{S}$. If one policy does dominate all others,

GPI reduces to the original policy improvement theorem.

GPI provides a principled way of combining multiple policies into a single policy whose performance is generally better than that of its constituents. In the context of transfer, this makes it possible to leverage knowledge accumulated over many tasks to learn a new task faster. Suppose that an agent has computed optimal policies for tasks $M_1, M_2, ..., M_n \in \mathcal{M}^\phi$. Suppose also that when exposed to a new task M_{n+1} the agent computes $Q^{\pi_i^*}_{n+1}$—the value functions of the policies π_i^* under the new reward function induced by $\mathbf{w}_{n+1}$. In this case, applying GPI to the newly-computed set of value functions $\{Q^{\pi_1^*}_{n+1}, Q^{\pi_2^*}_{n+1}, ..., Q^{\pi_n^*}_{n+1}\}$ will give rise to a policy that performs at least as well as a policy computed based on any subset of the set above, including the empty set. Therefore, GPI satisfies our definition of successful transfer.

There is a caveat, though. Why would one waste time computing the value functions of $\pi_1^*, \pi_2^*, ..., \pi_n^*$, whose performance in M_{n+1} may be mediocre, if the same amount of resources can be allocated to compute a sequence of n policies with increasing performance? This is where successor features come into play, as we discuss next.

4. Successor Features

We start this section by simplifying our notation slightly with the definition $\boldsymbol{\phi}_t = \boldsymbol{\phi}(s_t, a_t, s_{t+1})$. Now, plugging (1) into (2) we have

$$\begin{aligned} Q^\pi(s,a) &= \mathrm{E}^\pi\left[\boldsymbol{\phi}_{t+1}^\top \mathbf{w} + \gamma \boldsymbol{\phi}_{t+2}^\top \mathbf{w} + ... \,|\, S_t = s, A_t = a\right] \\ &= \mathrm{E}^\pi\left[\textstyle\sum_{i=t}^\infty \gamma^{i-t} \boldsymbol{\phi}_{i+1} \,|\, S_t = s, A_t = a\right]^\top \mathbf{w} \\ &= \boldsymbol{\psi}^\pi(s,a)^\top \mathbf{w}. \end{aligned} \tag{5}$$

The decomposition (5) has appeared before in the literature under different names and interpretations (Barreto et al., 2016). Since here we see (5) as an extension of Dayan's (1993) SR, we call $\boldsymbol{\psi}^\pi(s,a)$ the *successor features* (SFs) of (s,a) under policy π.

The ith component of $\boldsymbol{\psi}^\pi(s,a)$ gives the discounted sum of ϕ_i when following policy π starting from (s,a). In the particular case where $\mathcal{S}$ and $\mathcal{A}$ are finite and $\boldsymbol{\phi}$ is a tabular representation of $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ the vector $\boldsymbol{\psi}^\pi(s,a)$ is the discounted sum of occurrences, under π, of each possible transition. This is essentially the concept of SR extended from the space $\mathcal{S}$ to the set $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ (Dayan, 1993). SFs also extend SR in two other ways. First, the concept readily applies to continuous state and action spaces without any modification. Second, by explicitly casting (2) and (5) as inner products involving feature vectors, SFs make it evident how to incorporate function approximation: as will be shown, these vectors can be learned from data.

The SFs $\boldsymbol{\psi}^\pi$ summarize the dynamics induced by π in a given environment. As shown in (5), this allows for a modular representation of Q^π in which the MDP's dynamics are decoupled from its rewards, which are captured by the weights $\mathbf{w}$. One potential benefit of having such a decoupled representation is that only the relevant module must be relearned when either the dynamics or the reward changes.

The representation in (5) requires two components to be learned, $\mathbf{w}$ and $\boldsymbol{\psi}^\pi$. Since the latter is the expected discounted sum of $\boldsymbol{\phi}$ under π, we must either be given $\boldsymbol{\phi}$ or learn it as well. Note that approximating $r(s,a,s') \approx \boldsymbol{\phi}(s,a,s')^\top \tilde{\mathbf{w}}$ is a supervised learning problem, allowing the use of well-understood techniques to learn $\tilde{\mathbf{w}}$ (and potentially $\tilde{\boldsymbol{\phi}}$, too) (Hastie et al., 2002). As for $\boldsymbol{\psi}^\pi$, we note that

$$\boldsymbol{\psi}^\pi(s,a) = \mathrm{E}^\pi[\boldsymbol{\phi}_{t+1} + \gamma \boldsymbol{\psi}^\pi(S_{t+1}, \pi(S_{t+1})) \,|\, S_t = s, A_t = a], \tag{6}$$

that is, SFs satisfy a Bellman equation in which ϕ_i play the role of rewards. Therefore, in principle *any* RL method can be used to compute $\boldsymbol{\psi}^\pi$ (Szepesvári, 2010).

5. GPI with SFs

Now that we have presented GPI and SFs we are ready to describe our framework for transfer. In order to do so, we go back to the scenario discussed in the end of Section 3, in which we have learned policies π_i^* associated with tasks M_i induced by weight vectors $\mathbf{w}_i$.

Suppose that we have learned the functions $Q_i^{\pi_i^*}$ using the representation scheme shown in (5). Now, if the reward changes to $r_{n+1}(s,a,s') = \boldsymbol{\phi}(s,a,s')^\top \mathbf{w}_{n+1}$, as long as we have $\mathbf{w}_{n+1}$ we can compute the new value function of π_i^* by simply making $Q^{\pi_i^*}_{n+1}(s,a) = \boldsymbol{\psi}^{\pi_i^*}(s,a)^\top \mathbf{w}_{n+1}$. This reduces the computation of all $Q^{\pi_i^*}_{n+1}$ to the problem of defining $\mathbf{w}_{n+1}$. In some cases we can assume that $\mathbf{w}_{n+1}$ is provided by the environment or can be inferred from the observations. However, even when the only signal indicating that the task has changed is the reward itself, $\mathbf{w}_{n+1}$ can be computed via supervised learning in order to minimise some loss derived from (2) (see experiments below).

Once $Q^{\pi_i^*}_{n+1}$ have been computed, we can apply GPI to derive a policy π whose performance on M_{n+1} is no worse than the performance of $\pi_1^*, \pi_2^*, ..., \pi_n^*$ on the same task. A question that arises in this case is whether we can provide stronger guarantees on the performance of π by exploiting the structure shared by the tasks in $\mathcal{M}^\phi$. The following theorem answers this question in the affirmative.

Theorem 2. *Let $M_i \in \mathcal{M}^\phi$ and let $Q_i^{\pi_j^*}$ be the value function of an optimal policy of $M_j \in \mathcal{M}^\phi$ when executed in M_i. Given approximations $\{\tilde{Q}_i^{\pi_1^*}, \tilde{Q}_i^{\pi_2^*}, ..., \tilde{Q}_i^{\pi_n^*}\}$ such that $\left|Q_i^{\pi_j^*}(s,a) - \tilde{Q}_i^{\pi_j^*}(s,a)\right| \leq \epsilon$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$, and $j \in \{1,2,...,n\}$, let $\pi(s) \in \mathrm{argmax}_a \max_j \tilde{Q}_i^{\pi_j^*}(s,a)$. Fi-*

nally, let $\phi_{\max} = \max_{s,a} ||\phi(s,a)||$, *where* $||\cdot||$ *is the norm induced by the inner product adopted. Then,*

$$Q_i^{\pi_i^*}(s,a) - Q_i^{\pi}(s,a) \leq \frac{2}{1-\gamma}\left(\phi_{\max} \min_j ||\mathbf{w}_i - \mathbf{w}_j|| + \epsilon\right). \tag{7}$$

Note that we used "M_i" rather than "M_{n+1}" in the theorem's statement to remove any suggestion of order among the tasks. As shown in (7), the loss $Q_i^{\pi_i^*}(s,a) - Q_i^{\pi}(s,a)$ is upper-bounded by two terms. The term $2\phi_{\max}\min_j||\mathbf{w}_i - \mathbf{w}_j||/(1-\gamma)$ is of more interest here because it reflects the structure of $\mathcal{M}^\phi$. This term is a multiple of the distance between $\mathbf{w}_i$, the task we are currently interested in, and the closest $\mathbf{w}_j$ for which we have computed a policy. This formalises the intuition that the agent should perform well in task $\mathbf{w}_i$ if it has solved a similar task before.

Although Theorem 2 is inexorably related to the characterization of $\mathcal{M}^\phi$ in (3), it does not depend on the definition of SFs in any way. Here SFs are the *mechanism* used to efficiently apply the protocol suggested by Theorem 2. When SFs are used the value function approximations are given by $\tilde{Q}_i^{\pi_j^*}(s,a) = \tilde{\psi}^{\pi_j^*}(s,a)^\top \tilde{\mathbf{w}}_i$. The modules $\tilde{\psi}^{\pi_j^*}$ are computed and stored when the agent is learning the tasks M_j, which can be done sequentially or in parallel. When faced with a new task M_i the agent computes an approximation of $\mathbf{w}_i$, which is a supervised learning problem, and then follows the GPI policy π defined in Theorem 2. Note that we do not assume that either $\psi^{\pi_j^*}$ or $\mathbf{w}_i$ is computed exactly: the effect of errors in $\tilde{\psi}^{\pi_j^*}$ and $\tilde{\mathbf{w}}_i$ in the approximation of $Q_i^{\pi_j^*}(s,a)$ is accounted for by the term ϵ. As shown in (7), if ϵ is small and the agent has seen enough tasks the performance of π on M_i should already be good, which means that the agent should be able to perform well without the need to compute a new policy. Of course, one can use the data collected by π to learn a new $\tilde{\psi}^{\pi_i^*}$, which can then be added to our set of SFs, restarting the loop.

Interestingly, Theorem 2 also provides guidance for some practical algorithmic choices. Since in an actual implementation one wants to limit the number of SFs $\tilde{\psi}^{\pi_j^*}$ stored in memory, the corresponding vectors $\tilde{\mathbf{w}}_j$ can be used to decide which ones to keep. For example, if the maximum number of SFs is reached at task $\tilde{\mathbf{w}}_i$, the new $\tilde{\psi}^{\pi_i^*}$ can replace $\tilde{\psi}^{\pi_k^*}$, where $k = \text{argmin}_j||\tilde{\mathbf{w}}_i - \tilde{\mathbf{w}}_j||$.

6. Experiments

We now present empirical results to provide some intuition on how the proposed framework works in practice. The environment we consider involves navigation tasks defined over a two-dimensional continuous space composed of four "rooms" (Fig. 1). The agent starts in one of the rooms and must reach a goal region located in the farthest room. The environment has objects that can be picked up by the agent

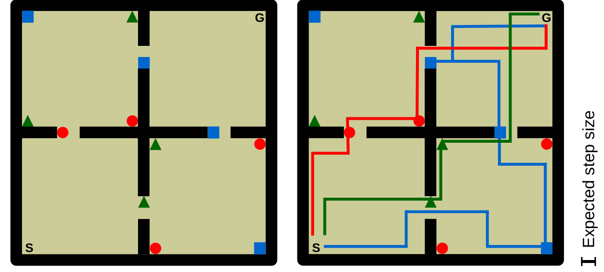


Figure 1. Environment layout and some examples of optimal trajectories associated with specific tasks. The shapes of the objects represent their classes; 'S' is the start state and 'G' is the goal.

by passing over them. There is a total of n_o objects, each belonging to one of $n_c \leq n_o$ *classes*. The class of an object determines the reward r_c associated with it—that is, two objects of class c always lead to the same reward r_c. The rewards r_c can be positive, negative, or zero. There is also a positive reward r_g associated with the goal. The objective of the agent is to pick up the "good" objects and navigate to the goal while avoiding "bad" objects. An episode ends when the agent reaches the goal, upon which all the objects re-appear. Figure 1 shows the specific environment layout used, in which $n_o = 12$ and $n_c = 3$.

We assume that r_g is always 1 but r_c may vary: a specific instantiation of the rewards r_c defines a *task*. In our experiments each task lasts for 2×10^4 transitions, and when a new task starts the rewards r_c are sampled from a uniform distribution over $[-1, 1]$. Our environment is thus presented as an infinite stream of tasks. Looking at Figure 1 one can see that different tasks may require quite distinct behaviours: the goal of the agent is to maximise the sum of rewards accumulated over a sequence of 250 tasks.

We focus on the online RL scenario where the agent must learn while interacting with the environment $\mathcal{M}$. As shown in Algorithm 1, we defined a straightforward instantiation of our approach in which both $\tilde{\mathbf{w}}$ and $\tilde{\psi}^\pi$ are computed incrementally in order to minimise losses induced by (2) and (6). Every time the task changes the current $\tilde{\psi}^{\pi_i}$ is stored and a new $\tilde{\psi}^{\pi_{i+1}}$ is created. We call this method "SFQL" as a reference to the fact that SFs are learned through an algorithm analogous to Watkins & Dayan's (1992) Q-learning (QL). Also due to this similarity we use QL itself as a baseline for our comparisons. As a more challenging reference point we report results for a transfer method called *probabilistic policy reuse* (Fernández et al., 2010). The version of the algorithm we adopt builds on QL and reuses all policies learned. The resulting method, PRQL, is directly comparable to SFQL.

We assume that the agents know their position $\{s_x, s_y\} \in [0,1]^2$ and also have an "object detector" $\mathbf{o} \in \{0,1\}^{n_o}$ whose i^{th} component is 1 if and only if the agent is over

Algorithm 1 SFQL

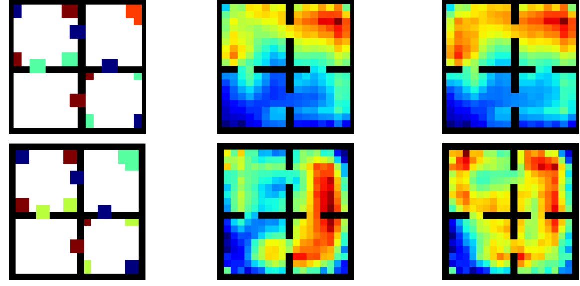
Require:
- ϕ features to be predicted by SFs
- ϵ parameter for ϵ-greedy strategy
- α_z, α_w learning rates

1: **for** $t \leftarrow 1, 2, ..., \text{num_tasks}$ **do**
2: $\quad \mathbf{w}_t \leftarrow$ small random initial values
3: $\quad \mathbf{z}_t \leftarrow$ random values // $\mathbf{z}_t$ are the parameters of $\tilde{\boldsymbol{\psi}}_t$
4: $\quad \text{new_eps} \leftarrow \text{true}$
5: $\quad$ **for** $i \leftarrow 1, 2, ..., \text{num_steps}$ **do**
6: $\quad\quad$ **if** new_eps **then**
7: $\quad\quad\quad \text{new_eps} \leftarrow \text{false}$
8: $\quad\quad\quad s \leftarrow$ initial state
9: $\quad\quad \text{sel_rand_a} \sim \text{Bernoulli}(\epsilon)$ // true with probability ϵ
10: $\quad\quad$ **if** sel_rand_a **then** $a \sim \text{Uniform}(\{1, 2, ..., |A|\})$
11: $\quad\quad$ **else** $a \leftarrow \text{argmax}_b \max_{k \in \{1,2,...,t\}} \tilde{\boldsymbol{\psi}}_k(s, b)^\top \mathbf{w}_t$
12: $\quad\quad$ Execute a and observe reward r and next state s'
13: $\quad\quad$ **if** s' is terminal **then** $\gamma \leftarrow 0$ **and** $\text{new_eps} \leftarrow \text{true}$
14: $\quad\quad$ **else** $a' \leftarrow \text{argmax}_b \max_{k \in \{1,2,...,t\}} \tilde{\boldsymbol{\psi}}_k(s', b)^\top \mathbf{w}_t$
15: $\quad\quad \mathbf{w}_t \leftarrow \mathbf{w}_t + \alpha_w \left[r - \boldsymbol{\phi}(s, a, s')^\top \mathbf{w} \right] \boldsymbol{\phi}(s, a, s')$
16: $\quad\quad$ **for** $k \leftarrow 1, 2, ..., d$ **do**
17: $\quad\quad\quad \delta_k \leftarrow \phi_k(s, a, s') + \gamma \tilde{\psi}_{tk}(s', a') - \tilde{\psi}_{tk}(s, a)$
18: $\quad\quad\quad \mathbf{z}_{tk} \leftarrow \mathbf{z}_{tk} + \alpha_z \delta_k \nabla_{\mathbf{z}} \tilde{\psi}_{tk}(s, a)$
19: $\quad\quad s \leftarrow s'$

object i. Using this information the agents build two vectors of features. The vector $\boldsymbol{\varphi}_p(s) \in \mathbb{R}^{100}$ is composed of the activations of a regular 10×10 grid of Gaussian functions at the point $\{s_x, s_y\}$. In addition, using $\mathbf{o}$ the agents build an "inventory" $\boldsymbol{\varphi}_i(s) \in \{0, 1\}^{n_o}$ whose i^{th} component indicates whether the ith object has been picked up or not. The concatenation of $\boldsymbol{\varphi}_i(s)$ and $\boldsymbol{\varphi}_p(s)$ plus a constant term gives rise to the feature vector $\boldsymbol{\varphi}(s) \in \mathbb{R}^D$ used by all the agents to represent the value function: $\tilde{Q}^\pi(s, a) = \boldsymbol{\varphi}(s)^\top \mathbf{z}_a^\pi$, where $\mathbf{z}_a^\pi \in \mathbb{R}^D$ are learned weights.

It is instructive to take a closer look at how exactly SFQL represents the value function. Note that, even though our algorithm also represents $\tilde{Q}^\pi$ as a linear combination of the features $\boldsymbol{\varphi}(s)$, it never explicitly computes $\mathbf{z}_a^\pi$. Specifically, SFQL represent SFs as $\tilde{\boldsymbol{\psi}}^\pi(s, a) = \boldsymbol{\varphi}(s)^\top \mathbf{Z}_a^\pi$, where $\mathbf{Z}_a^\pi \in \mathbb{R}^{D \times d}$, and the value function as $\tilde{Q}^\pi(s, a) = \tilde{\boldsymbol{\psi}}^\pi(s, a)^\top \tilde{\mathbf{w}} = \boldsymbol{\varphi}(s)^\top \mathbf{Z}_a^\pi \tilde{\mathbf{w}}$. By making $\mathbf{z}_a^\pi = \mathbf{Z}_a^\pi \tilde{\mathbf{w}}$, it becomes clear that SFQL unfolds the problem of learning $\mathbf{z}_a^\pi$ into the sub-problems of learning $\mathbf{Z}_a^\pi$ and $\tilde{\mathbf{w}}$. These parameters are learned via gradient descent in order to minimise losses induced by (6) and (2), respectively (see Alg. 1).

By associating each object in $\mathbf{o}$ with its class, the SFQL agent can easily construct features $\boldsymbol{\phi} \in \mathbb{R}^{n_c+1}$ that perfectly predicts the reward for all $M \in \mathcal{M}^\phi$, as in (2) and (3). In our experiments we compared two versions of SFQL. In the first one, called SFQL-$\boldsymbol{\phi}$, we assume that the agent knows $\boldsymbol{\phi}$. The second version of our agent



$\tilde{\boldsymbol{\phi}}(s, a, s')^\top \tilde{\mathbf{w}}_t \qquad \max_{a, i<t} \tilde{\boldsymbol{\psi}}_i(s, a)^\top \tilde{\mathbf{w}}_t \qquad \max_{a, i \le t} \tilde{\boldsymbol{\psi}}_i(s, a)^\top \tilde{\mathbf{w}}_t$

Figure 3. Functions computed by SFQL after 200 transitions into two randomly selected tasks (all objects present).

had to learn the mapping above directly from data, replacing the handcrafted $\boldsymbol{\phi}$ with an approximation $\tilde{\boldsymbol{\phi}} \in \mathbb{R}^h$. Note that h may not coincide with the "real" d, which in this case is $n_c + 1 = 4$. The process of learning $\tilde{\boldsymbol{\phi}}$ followed the multi-task learning protocol proposed by Caruana (1997) and Baxter (2000). In summary, SFQL used plain QL to collect data for 20 tasks, and then used this data to compute an approximation $\tilde{\boldsymbol{\phi}}(s, a, s')^\top \tilde{\mathbf{w}}_t = \varsigma \left(\boldsymbol{\varphi}(s, s')^\top \mathbf{H} \right) \tilde{\mathbf{w}}_t \approx r_t(s, a, s')$ for $t = 1, 2, ..., 20$, where $\boldsymbol{\varphi}(s, s') = [\boldsymbol{\varphi}(s), \boldsymbol{\varphi}(s')]$, $\varsigma(\cdot)$ is a sigmoid function applied element-wise, $\mathbf{H} \in \mathbb{R}^{D \times h}$ are the weights defining $\tilde{\boldsymbol{\phi}}$, and $r_t(\cdot)$ is the reward function of task t. Since we used different values for h, we refer to the corresponding instances of our algorithm as SFQL-h.

The results of our experiments are shown in Figure 2. As shown, all versions of SFQL significantly outperform the other two methods, with an improvement on the average return of more than 100% when compared to PRQL, which itself improves on QL by around 100%. Interestingly, SFQL-h seems to achieve good overall performance *faster* than SFQL-ϕ, even though the latter uses features that allow for an exact representation of the rewards. One possible explanation is that, unlike their counterparts ϕ_i, the features $\tilde{\phi}_i$ are activated over most of the set $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$, which results in a dense pseudo-reward signal that facilitates learning.

More generally, the good performance of SFQL seems to be a direct consequence of the transfer promoted by the combination of GPI and SFs, as illustrated in Figure 3. Note how after only 200 transitions into a new task SFQL already has a good approximation of the reward function, which, combined with the set of previously computed $\tilde{\boldsymbol{\psi}}^{\pi_i}$, with $i < t$, provide a very informative value function even without the current $\tilde{\boldsymbol{\psi}}^{\pi_t}$.

7. Conclusion

This paper builds on two concepts, GPI and SFs. GPI can be seen as a principled way of turning a collection of poli-

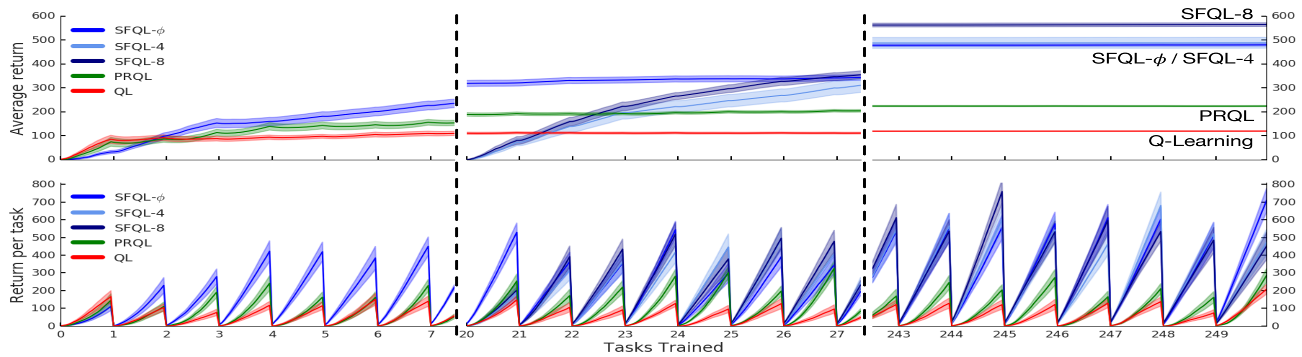


Figure 2. Average and cumulative return per task in the four-room domain. SFQL-h received no reward during the first 20 tasks while learning $\tilde{\phi}$. Error-bands show one standard error over 30 runs.

cies into a policy whose performance is generally better than that of its precursors. Following a fundamental principle in RL, we would like the individual policies to be learned in a goal-oriented way—that is, by trying to maximise a given reward signal. However, if each policy is induced by a different signal, the resulting value functions will reflect their performance under distinct criteria, and thus GPI no longer applies. More generally, the difficulty in handling value functions computed under different reward signals represents an obstacle for using the DP machinery in multi-task scenarios, which may be one of the reasons why transfer is often posed as a separate problem outside of the RL setting. SFs resolve this issue by allowing one to efficiently compute the value function of a policy under any reward function that can be represented in a given form.

By combining GPI and SFs, one is able to leverage knowledge that has been acquired in a purposeful, goal-oriented, way. How precisely this takes place will reflect the specificities of the scenario of interest, and may give rise to different transfer approaches. For example: the individual policies can be learned in sequence or in parallel, with the corresponding tasks reflecting some structure of the problem or organized by level of difficulty, by some notion of similarity, or even in a hierarchical way. All these instantiations can be naturally accommodated under our framework, which we believe to be an elegant extension of DP's basic setting that provides a solid foundation for transfer in RL.

References

Barreto, André, Munos, Rémi, Schaul, Tom, and Silver, David. Successor features for transfer in reinforcement learning. *CoRR*, abs/1606.05312, 2016.

Baxter, Jonathan. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

Bellman, Richard E. *Dynamic Programming*. Princeton University Press, 1957.

Bertsekas, Dimitri P. and Tsitsiklis, John N. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

Bowling, Michael, Burch, Neil, Johanson, Michael, and Tammelin, Oskari. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

Caruana, Rich. Multitask learning. *Machine Learning*, 28(1): 41–75, 1997.

Dayan, Peter. Improving generalisation for temporal difference learning: The successor representation. *Neural Computation*, 5(4):613–624, 1993.

Fernández, Fernando, García, Javier, and Veloso, Manuela. Probabilistic policy reuse for inter-task transfer learning. *Robotics and Aut. Systems*, 58(7):866–871, 2010.

Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2002.

Lazaric, Alessandro. *Transfer in Reinforcement Learning: A Framework and a Survey*, in *Reinforcement Learning: State-of-the-Art*. pp. 143–173. 2012.

Mnih *et al.* Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Puterman, Martin L. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

Silver *et al.* Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

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

Szepesvári, Csaba. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Mach. Learning. Morgan & Claypool Publishers, 2010.

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

Watkins, Christopher and Dayan, Peter. Q-learning. *Machine Learning*, 8:279–292, 1992.