UNIVERSITY OF TRIESTE



Department of
Mathematics and Geoscience

M. Sc. Program
DATA SCIENCE AND SCIENTIFIC COMPUTING
- Artificial Intelligence Curriculum -

# A Study Of Universal Successor Features With Generalized Policy Improvement

**Supervisor:**
**Prof. Antonio Celani**

**Candidate:**
**Andres Bermeo Marinelli**
**Mat. SM3500444**

**Adjunt Supervisor:**
**Dr. Chris Reinke**

Academic Year 2022-2023

# Contents

# Chapter 1

# Introduction

Reinforcement Learning (RL) is a branch of Artificial Intelligence which deals with solving sequential decision-making tasks in an optimal manner. In the most abstract terms, an agent interacts with an external environment while trying to optimize its behavior through the experience of success and failure.

The advent of Deep Learning has given rise to a new era in RL, where deep neural networks (DNN) are used as function approximators for a wide variety of problems and algorithms, giving way to a new field called Deep Reinforcement Learning (DRL). The incredible performance of DNNs as general approximators of non-linear functions has made DRL very successful in handling high-dimensional and continuous problems, which was previously difficult. This has made DRL useful in a wide variety of applications.

In fact, DRL has achieved human-level performance in various tasks such as playing Atari[14] video games and beating the Go[19, 20] grand master. Furthermore, it has enabled massive improvements in various fields such as: medicine through the use of algorithms that facilitate protein synthesis[7]; computer science with new matrix-matrix multiplication[4] algorithms that could deeply impact the speed of computation in critical applications; robotics[9] and self driving cars[8]; and many

others. It has also proven to be essential to training large language models such Chat-GPT, through Reinforcement Learning from Human Feedback[11].

Despite its success, DRL algorithms require large amounts of data to train, which is prohibitive for many types of applications where data acquisition is costly. Therefore, in many real world scenarios, it is too expensive to train agents from scratch for each new task/scenario. As a consequence, significant effort from the research community has been poured into developing algorithms that foster re-usability of previously learned agents, encourage re-utilization of skills, and increase the speed of training.

One of the most prominent ideas to address this issue is the field of Transfer Reinforcement Learning (TRL). The goal in TRL is to enable the agent to leverage skills learnt in the past to solve new tasks more efficiently and effectively. The inspiration comes from the human ability to incorporate past experiences to solve new challenges in new environments. A plethora of algorithms have been developed[24], however, it still remains an active field of research.

A subcategory of interest in TRL is the case when the environment stays the same, while the purpose of the agent within this environment changes. A promising framework within this formulation is the use of Successor Features (SFs)[1, 10] which encode the outcome of behaviors in an environment in terms of important task features. It allows to re-evaluate how these behaviors work in a new task which in turn helps to select one of the best previous behaviours as the initial behavior in the new task. This last step is done through the use of Generalized Policy Improvement (GPI)[2]. Together, SFs and GPI create a paradigm that is very useful to efficiently enable transfer of past skills to solve new tasks.

New algorithms[3, 13, 16] have been developed which try to take advantage of the SF paradigm to improve the ability of agents to adapt to new tasks compared to classical algorithms. One such attempt is described in the work called Universal

Successor Features (USF) for Transfer Reinforcement Learning by Ma et al.[13] which uses SFs and DNNs to build a single end-to-end architecture that is capable of adapting and generalizing to new tasks at an improved rate compared to a Deep Q-Network (DQN)[14] architecture. Furthermore, it aims at improving SFs by utilizing one single architecture that is able to generalize over different tasks, rather than having to learn the SFs for each new task, as is done in Barreto et al.[1]. However, USFs do not make use of the GPI procedure in their work.

In this work, we extensively study USFs by analyzing a variety of agents. We also try to extend the results of Ma et al.[13] by incorporating the use of the GPI procedure as a tool to enhance the ability of the agent to adapt to new tasks.

The necessity for this extension is primarily inspired by the current work of the INRIA RobotLearn team[6] on the SPRING Project[21]. In particular, the team has been developing a socially-aware robot which must learn to navigate and interact with its environment in a variety of scenarios. One of the goals is to enable a fast adaptation of the robot to new tasks it may encounter without the need to gather large amounts of data that is too costly to obtain. Among the possible solutions to this problem is the proposed framework based on combining USFs and GPI.

The present work is the result of a 6 month internship at INRIA RobotLearn.

## 1.1 Reinforcement Learning

Reinforcement Learning is a field of Artificial Intelligence concerned with optimal sequential decision-making under uncertainty. It is based on the idea of rewarding desired behaviors and punishing those that are undesired.

At a high level, the problem statement is depicted in Fig. 1.1.

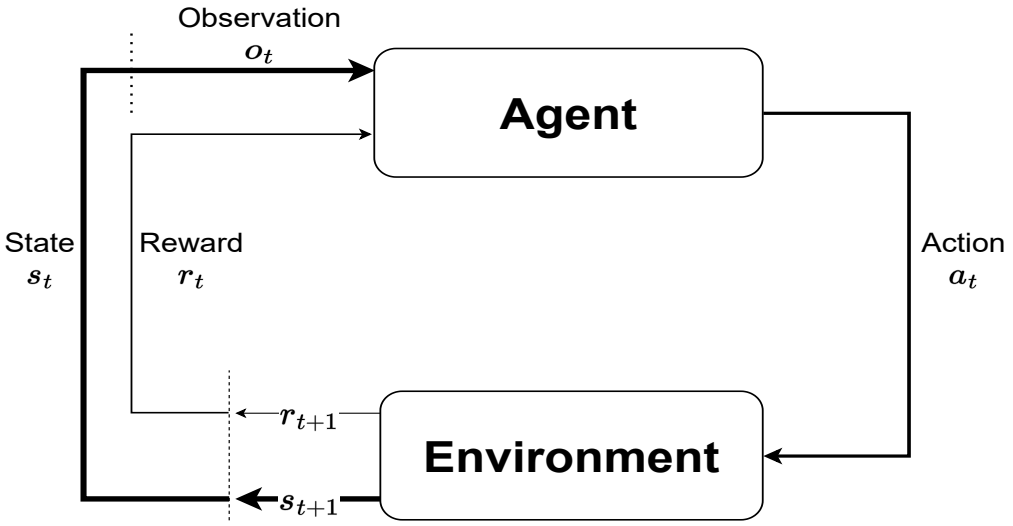


Figure 1.1: A diagram of the typical feedback loop in RL. An agent takes an action $a_t \in \mathcal{A}$ based on the observation $o_t$. In turn, the environment reacts to $a_t$ by returning a reward signal $r_{t+1}$, which indicates the goodness of the action, and a state $s_{t+1}$, which indicates the new state of the agent.

The agent takes an action $a_t \in \mathcal{A}$ based on the observation $o_t$. In turn, the environment reacts to $a_t$ by returning a scalar feedback signal $r_{t+1}$ - the reward - which indicates the goodness of that particular action, and by changing its state to $s_{t+1}$, usually in a stochastic manner. The agent then perceives an observation $o_{t+1}$ from $s_{t+1}$, and the cycle repeats. A single iteration of this procedure defines a *transition*, which is a tuple $(s_t, a_t, r_{t+1}, s_{t+1})$. The differentiation between *state* and *observation* reflects the fact that an agent usually does not have access to a fully informative description of the environment - the state - and makes use of some incomplete representation - the observation.

The ultimate goal of the agent is to maximize the expected sum of discounted rewards $G = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right]$ by taking the best actions. The expectation is over all the possible paths an agent may take with a certain defined behavior, which is also known as a *policy* $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions. This introduces a further complication since this objective means that sometimes it may be beneficial

to sacrifice high immediate reward for a higher long-term cumulative reward.

Reinforcement Learning differs from supervised machine learning in a few key aspects:

1. There is no supervision, instead, it is a trial and error paradigm.

2. It deals with sequential processes. This means that data is not independent and identically distributed (i.i.d). Instead, we have a dynamic system where the agent is moving through the environment and its experiences are intrinsically correlated to those it experienced before.

3. The agent influences the environment and therefore, the data distribution that it encounters.

4. The true feedback for a specific action does not come instantaneously. In other words, the repercussions of a certain action may only be known after many time steps. This means that we do not immediately know which action is the best.

### 1.1.1 Mathematical Formulation

Mathematically speaking, RL is formulated as a Markov Decision Process (MDP)[15]. An MDP is defined as a tuple $\mathcal{M} \doteq (\mathcal{S}, \mathcal{A}, p, r, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space which can be finite or infinite, $p = p(\cdot \mid S_t = s, A_t = a)$ is the probability distribution for the next state $s_{t+1}$ after having taken action $a$ from state $s$ at time step $t$, $r_{t+1} \doteq r(S_t = s, A_t = a, S_{t+1} = s')$ is the one-step reward for a particular transition, and $\gamma \in [0, 1)$ represents the discounting factor which defines the importance of rewards in the future.

The goal of the agent is to find an *optimal* policy $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ which maximizes the expected discounted sum of rewards, also known as the *return*:

$$G_t^{(\pi)} = \mathbb{E}^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right], \tag{1.1}$$

where $\mathbb{E}^{\pi}[\cdot]$ denotes the expectation over the sequences of transitions induced by a certain policy $\pi$ and $p(s_t, a_t, s_{t+1})$.

### 1.1.2 Q-learning

One common way to solve the RL problem is through dynamic programming methods which rely on the concept of a value function. One such method, called Q-learning[23], uses the *action-value function* or Q-function defined in Eq. 1.2.

$$\begin{aligned} Q_t^{\pi}(s,a) &= \mathbb{E}^{\pi}\left[r_{t+1} + \gamma r_{t+2} + \cdots \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}^{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}^{\pi}\left[G_t^{(\pi)} \mid S_t = s, A_t = a\right]. \end{aligned} \tag{1.2}$$

The action-value function encodes the expected returns that the agent would get from taking action $a$ from state $s$ and then following the policy $\pi$ the rest of the time. This quantity encodes all the information needed, since one can derive a policy from it by simply defining $\pi(s) = \max_a Q(s,a)$. Obtaining the Q-function for a given policy is known as *policy evaluation*, and it is an important part of many RL algorithms. The optimal Q-function $Q^{\pi^*}$ satisfies the Bellman Eq. 1.3.

$$Q^{\pi^*}(s,a) = \mathop{\mathbb{E}}_{s_{t+1}\sim p(\cdot|s_t,a_t)}\left[r\left(s_t, a_t, s_{t+1}\right)\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}\sim p(\cdot|s_t,a_t)}\left[\max_a Q^{\pi^*}\left(s_{t+1}, a\right)\right]. \tag{1.3}$$

The Q-learning algorithm uses Eq. 1.3 to compute a one-step estimate for the optimal Q-function, also known as a *temporal difference target*. At each time step, the agent follows a certain policy $\pi$, generating transitions of the form $(s_t, a_t, r_t, s_{t+1})$. Using these quantities, the temporal difference target is computed and used to learn the optimal Q-function through the following update rule:

$$Q^{n+1}(s_t, a_t) \leftarrow \underbrace{Q^n(s_t, a_t)}_{\text{current value}} + \alpha \left( \underbrace{r_{t+1} + \gamma \max_a Q^n(s_{t+1}, a)}_{\text{temporal difference target}} - \underbrace{Q^n(s_t, a_t)}_{\text{current value}} \right) \tag{1.4}$$

where $\alpha$ is a positive learning rate that regulates the magnitude of the update. As $n \to \infty$, this algorithm converges to the optimal Q-function. Furthermore, this introduces the concept of *off-policy* learning since the update above does not impose any restrictions on which policy is used to generate the transitions. The only requirement is that the entire state-action space has to be explored in order to update all of the possibilities. In other words, we can follow a *behavioral* policy $\pi$ while learning the Q-function of the optimal policy $\pi^*$, which is precisely the definition of off-policy learning. A popular choice of behavioral policy is to use an $\epsilon$-greedy strategy where we define $\epsilon \in [0, 1)$ and use Eq. 1.5.

$$\pi(s) = \begin{cases} \arg\max_a Q^n(s, a) & \text{with probability } (1 - \epsilon) \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{1.5}$$

Therefore, $\epsilon$ percentage of the time, the agent is taking random actions to explore the environment and possibly discover additional beneficial states. The rest of the time, it is exploiting the current knowledge it has gathered. The choice of $\epsilon$ sets the balance in the exploration-exploitation trade-off. Low values of $\epsilon$ favor exploitation while high values tend to favor exploration.

The top part of Eq. 1.5 is part of a procedure that is commonly referred to as *policy improvement* since we use the current Q-function to derive a greedy policy that is guaranteed to be at least as good as the one implied by the current Q-function.

## 1.2 Deep Q-Learning

The variant of Q-learning that uses DNNs is called Deep Q-Learning (DQN). The main difference is that the Q-function is represented as a neural network and the update rule uses a squared loss that is based on the rule in Eq. 1.4:

$$L_Q = \left[ \underbrace{r_{t+1} + \gamma \max_a Q\left(s_{t+1}, a, g; \theta_Q\right)}_{\text{temporal difference target}} - Q\left(s_t, a_t, g; \theta_Q\right) \right]^2, \tag{1.6}$$

where $\theta_Q$ are the parameters of the neural network.

Various techniques are used to stabilize the learning procedure of DQN. For example, it is common to use two networks: a *target* network which is fixed and used to calculate the temporal difference target, and a *policy* network which is updated according to Eq. 1.6. This is reflected in the following loss:

$$L_Q = \left[ \underbrace{r_{t+1} + \gamma \max_a Q^{\text{target}}\left(s_{t+1}, a, g; \bar{\theta}_Q\right)}_{\text{temporal difference target}} - Q^{\text{policy}}\left(s_t, a_t, g; \theta_Q\right) \right]^2, \tag{1.7}$$

where $\bar{\theta}_Q$ are the parameters of the target network which are fixed and not updated by the loss in Eq. 1.7 and $\theta_Q$ are the parameters of the policy network which are updated by the loss in Eq. 1.7.

After a certain number of time steps, the policy network is copied onto the target network to update the current knowledge. This is commonly referred to as a *hard*

update.

Furthermore, in order to train the network with i.i.d. data, it is common practice to maintain a buffer that stores the transitions, also known as a *memory buffer*. Whenever the network is updated, a batch of transitions is randomly sampled from the memory buffer and used to update the network. This is known as *Experience Replay*[12].

An important variant of Experience Replay is Prioritized Experience Replay[17], which enables sampling random transitions from the memory buffer according to a distribution that gives more importance to transitions with high impact on the learning. It uses two hyper-parameters: $\alpha$ which regulates how much to prioritize and $\beta$ which controls how much we correct for the fact that we do not sample data i.i.d when using this distribution.

## 1.3 Successor Features

In this work, we are interested in a reformulation of the RL problem for transfer learning in a multitask scenario. We deal with a particular subset of problems in which the environment configuration remains the same and the tasks are identified by different reward functions $r_i$, indexed by some $i$.

More specifically, we assume that the agent has learned to solve a set of tasks $\mathcal{T}$. The goal is to understand how to use this knowledge to speed up the solution of a new set of tasks $\mathcal{T}'$. One method out of many[18, 22] to solve this problem has been proposed by Barreto et al.[1, 2] through the use of *Successor Features* and *Generalized Policy Improvement*.

The fundamental assumption of Successor Features is that the one-step reward

function $r$ can be decomposed as follows:

$$r_{\mathbf{w}}(s_t, a_t, s_{t+1}) = \boldsymbol{\phi}(s_t, a_t, s_{t+1})^T \mathbf{w}, \tag{1.8}$$

where $\boldsymbol{\phi}(s_t, a_t, s_{t+1}) \in \mathbb{R}^d$ are some features of salient interest for the agent in the environment, and $\mathbf{w} \in \mathbb{R}^d$ are weights that quantify the importance of each feature in the current task. With this scheme, we have a one-to-one correspondence between the weights $\mathbf{w}$, the reward function $r_{\mathbf{w}}$, and the task it identifies. Therefore, we can use $\mathbf{w}$ to refer to the tasks and define $\mathcal{T}, \mathcal{T}'$ as tasks in $\mathbb{R}^d$.

As an example of Eq. 1.8, consider an environment where the agent must pick up objects such as rectangles, squares, and circles. In this case, the features $\boldsymbol{\phi}$ could be a simple three-dimensional boolean vector where each dimension encodes which of the three objects the agent has picked up at a given time step. Therefore, if at time step $t$, the agent has picked up both a rectangle and a square, the feature vector would be $\boldsymbol{\phi}_t = [1, 1, 0]^T$.

The weight vector $\mathbf{w}$ is a three-dimensional vector where each component specifies the importance of picking up that particular object for a given task. So, in a task where picking up rectangles is worth 1.5 and all other shapes are worth 0, the weight vector would be $\mathbf{w} = [1.5, 0, 0]$.

In our example, at time step $t$ the agent would receive a reward given by Eq. 1.9.

$$r_t = [1, 1, 0]^T \cdot [1.5, 0, 0] = 1.5. \tag{1.9}$$

As we can see, we can define various tasks using this framework. In our example, the set of possibilities is defined by $\mathbf{w} \in \mathbb{R}^3$.

If Eq. 1.8 is valid, then we can derive a fundamental quantity by rewriting the

definition of the Q-function as shown in Eq. 1.10.

$$\begin{aligned} Q_t^\pi(s,a) &= \mathbb{E}^\pi \left[ r_{t+1} + \gamma r_{t+2} + \cdots \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}^\pi \left[ \boldsymbol{\phi}_{t+1}^T \cdot \mathbf{w} + \gamma \boldsymbol{\phi}_{t+2}^T \cdot \mathbf{w} + \cdots \mid S_t = s, A_t = a \right] \\ &= \mathbb{E}^\pi \left[ \boldsymbol{\phi}_{t+1} + \gamma \boldsymbol{\phi}_{t+2} + \cdots \mid S_t = s, A_t = a \right]^T \cdot \mathbf{w} \\ &= \mathbb{E}^\pi \left[ \sum_{i=t}^{\infty} \gamma^{i-t} \boldsymbol{\phi}_{i+1} \mid S_t = s, A_t = a \right]^T \cdot \mathbf{w} \\ &= \boldsymbol{\psi}_t^\pi(s,a)^T \cdot \mathbf{w} \end{aligned} \tag{1.10}$$

The term $\boldsymbol{\psi}_t^\pi(s,a)$ is called the *Successor Features* (SFs) at time $t$ of policy $\pi$ for the state-action pair $(s,a)$. The $i^{th}$ component of $\boldsymbol{\psi}_t^\pi(s,a)$ represents the expected discounted sum of the $i^{th}$ feature $\boldsymbol{\phi}_i$ if the agent takes action $a$ from state $s$ at time step $t$ and then follows policy $\pi$.

A key observation is that with this framework, we have decoupled the dynamics of the environment, encoded by $\boldsymbol{\psi}$, from the rewards of the task, represented by $\mathbf{w}$. Therefore, once we have learned the SFs of a certain policy $\pi$, we can evaluate the action-value function of that policy for any other task identified by $\mathbf{w}'$. In other words, we can have a fast *policy evaluation* of any policy on any task we choose.

The remaining question is how to learn $\boldsymbol{\psi}^\pi(s,a)$, $\mathbf{w}$, and $\boldsymbol{\phi}$ for a problem. To learn $\mathbf{w}$ and $\boldsymbol{\phi}$, we can use Eq. 1.8 which is essentially a supervised learning problem. More specifically, if we consider $\boldsymbol{\phi}(s_t, a_t, s_{t+1}; \boldsymbol{\theta}_{\boldsymbol{\phi}})$ and $\mathbf{w}(\boldsymbol{\theta}_{\mathbf{w}})$ as functions of some parameters $\boldsymbol{\theta}_{\boldsymbol{\phi}}$ and $\boldsymbol{\theta}_{\mathbf{w}}$, respectively, then, as the agent experiences the environment, it can learn these two quantities by using the loss in Eq. 1.11.

$$L = \left[ r_{\mathbf{w}}(s_t, a_t, s_{t+1}) - \boldsymbol{\phi}(s_t, a_t, s_{t+1}; \boldsymbol{\theta}_{\boldsymbol{\phi}})^T \cdot \mathbf{w}(\boldsymbol{\theta}_{\mathbf{w}}) \right]^2 \tag{1.11}$$

After this regression has converged, we will have a model for $r_{\mathbf{w}}$ which satisfies Eq. 1.8.

To learn the SFs, we use the fact that they obey the Bellman Eq. 1.12.

$$\boldsymbol{\psi}_t^{\pi^*}(s,a) = \mathbb{E}^{\pi}\left[\boldsymbol{\phi}_{t+1} + \gamma \max_a \boldsymbol{\psi}_t^{\pi^*}(s_{t+1},a) \mid S_t = s, A_t = a\right]. \tag{1.12}$$

Therefore, in a similar fashion to Deep Q-learning, we can use the following loss to learn this quantity:

$$L_\psi = \left\| \underbrace{\boldsymbol{\phi}\left(s_t, a_t, s_{t+1}\right) + \gamma \max_a \boldsymbol{\psi}^{\text{target}}\left(s_{t+1}, a, g; \bar{\theta}_\psi\right)}_{\text{temporal difference target}} - \boldsymbol{\psi}^{\text{policy}}\left(s_t, a_t, g; \theta_\psi\right) \right\|_2^2, \tag{1.13}$$

where $\bar{\theta}_\psi$ are the parameters of the target $\psi$ network which are fixed and not updated by the loss in Eq. 1.13 and $\theta_\psi$ are the parameters of the policy network which are updated by the loss in Eq. 1.13.

## 1.4 Generalized Policy Improvement

The framework of Successor Features enables fast policy evaluation on any task. However, on its own, it does not help with the transfer learning problem.

As we mentioned previously, we are interested in a multitask learning scenario where all components of the MDP are fixed except for the reward function, which defines the task. We assume that the agent has learned a set policies that solve a set of tasks $\mathcal{T}$ and we want to transfer this accumulated knowledge to speed up the solution to a different set of tasks $\mathcal{T}'$. One way to achieve this is by identifying helpful policies from the previous tasks $\mathcal{T}$ that could be re-used to solve a new task in $\mathcal{T}'$. To do this, we need to re-evaluate how the previous policies behave when solving the new task and select the best one.

We can re-formulate this problem in the framework of Successor Features by

defining the set of tasks $\mathcal{T}$ for a fixed $\boldsymbol{\phi} \in \mathbb{R}^d$ as follows:

$$\mathcal{T} \equiv \left\{ \mathbf{w} \in \mathcal{T} \subset \mathbb{R}^d \mid r_{\mathbf{w}}(s_t, a_t, s_{t+1}) = \boldsymbol{\phi}(s_t, a_t, s_{t+1})^T \cdot \mathbf{w} \right\}. \tag{1.14}$$

We assume that the agent has learned the optimal policy $\pi^*_{\mathbf{w}}$, $\forall\ \mathbf{w} \in \mathcal{T}$ which is implied by $Q^{\pi^*_{\mathbf{w}}}(s,a) = \boldsymbol{\psi}^{\pi^*_{\mathbf{w}}}(s,a)^T \cdot \mathbf{w}$. In other words, for each policy $\pi^*_{\mathbf{w}}$, the Successor Features $\boldsymbol{\psi}^{\pi^*_{\mathbf{w}}}(s,a)$ are known.

Now, suppose we have to solve a new task $\mathbf{w}' \in \mathcal{T}'$. Using the framework of Successor Features, we can evaluate the Q-function of all the policies $\pi^*_{\mathbf{w}}$ on the task $\mathbf{w}'$ by computing the following quantity:

$$Q^{\pi^*_{\mathbf{w}}}_{\mathbf{w}'}(s,a) = \boldsymbol{\psi}^{\pi^*_{\mathbf{w}}}(s,a)^T \cdot \mathbf{w}', \quad \forall \mathbf{w} \in \mathcal{T} \tag{1.15}$$

We expect that if the tasks $\mathcal{T}$ are sufficiently similar to the new task $\mathbf{w}'$, then, one of the learned policies will perform well on the new task.

Moreover, we extend the policy improvement step to this case by doing a *Generalized Policy Improvement* (GPI) procedure as follows:

$$\begin{aligned} \pi_{\mathbf{w}'}(s) &= \arg\max_a \max_{\mathbf{w}} Q^{\pi^*_{\mathbf{w}}}(s,a) \\ &= \arg\max_a \max_{\mathbf{w}} \boldsymbol{\psi}^{\pi^*_{\mathbf{w}}}(s,a)^T \cdot \mathbf{w}', \quad \forall a \in \mathcal{A}, \quad \forall \mathbf{w} \in \mathcal{T} \end{aligned} \tag{1.16}$$

This operation corresponds to greedily improving a policy by selecting the best action over a set of available policies. This will generate a policy that performs at least as well as the whole set. There is the implicit assumption that the usefulness of this procedure derives from some similarity between tasks $\mathcal{T}$ and $\mathcal{T}'$ so that the policies $\pi^*_{\mathbf{w}}$ are able to perform well in the second set.

Barreto et al.[1] prove that the degree to which this procedure is able to find the optimal policy for task $\mathbf{w}'$ depends on the distance between $\mathcal{T}$ and $\mathbf{w}'$ as well as the

error in the approximations $\tilde{\pi}^*_{\mathbf{w}}$ of the true policies $\pi^*_{\mathbf{w}}$.

## 1.5 Universal Successor Features

Many works[3, 13] have tried to combine the general value functions[18, 22], which aim to generalize across the space of tasks, with the framework of Successor Features, which exploits the structure of the RL problem itself to do that.

In this work, we focus particularly on the results presented by Ma et al.[13], which designs an end-to-end architecture named *Universal Successor Features* (USF) that learns the features $\boldsymbol{\phi}$, the Successor Features $\boldsymbol{\psi}$, the goal weights $\mathbf{w}$, and the Q-function for a set of navigation tasks where the agent has to learn to reach a specific goal. More importantly, USFs are able to generalize over different goals/tasks by learning the optimal SFs for various goals. This is in contrast to the original SF framework where we have to separately learn the corresponding $\psi$ for each new task, making USFs more general and more powerful as they can also be used to solve tasks that are not seen during training. However, contrary to the original SF framework, USFs do not take advantage of the GPI procedure since their main purpose is to be able to generalize the SFs over various goals.

In particular, they use the architecture shown in Fig. 1.2. The architecture is composed of fully connected feedforward layers $\theta_{\psi}^{(1,2,3)}$ and $\theta_w$. The inputs are the current state $s$ of the agent and the goal $g$ that the agent has to reach, which corresponds to the task. In turn, the neural network will have to learn the corresponding features $\boldsymbol{\phi}(s)$, which are independent of the goal, the successor features $\boldsymbol{\psi}(s, a, g)$ that identify the optimal policy to reach goal $g$, the goal weights $\mathbf{w_g}$, and the Q-function, which is computed through the relation $Q_g(s, a) = \boldsymbol{\psi}(s, a, g)^T \cdot \mathbf{w_g}$. The network is trained over various goals so that it may learn to generalize the SFs over the goals.

The training procedure is subdivided into two phases: a first phase where the agent learns to navigate to a set of goals which constitute the base tasks, and a second phase where the agent is trained to reach a second set of goals. During the second phase, they test whether their architecture allows the agent to learn the new tasks at an improved rate compared to a goal-conditioned DQN architecture. The results in Ma et al.[13] show that USFs significantly outperform a goal-conditioned DQN architecture in its ability to learn new tasks efficiently and generalize to unseen tasks.

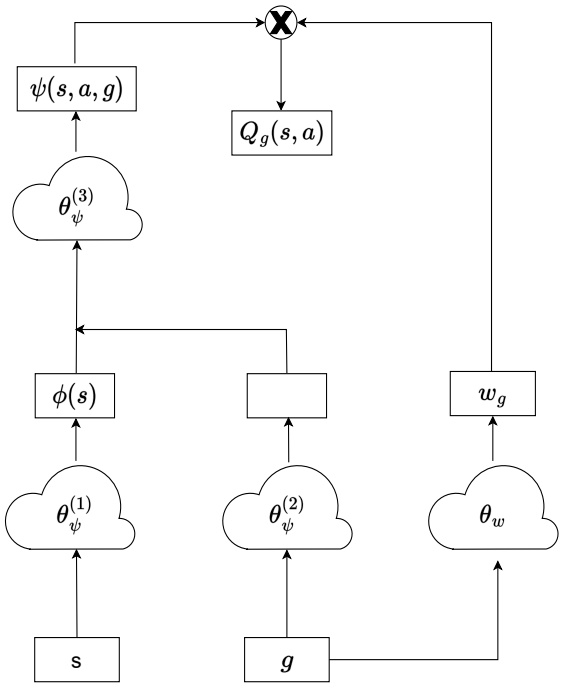


Figure 1.2: General USF architecture. The inputs are the state $s$ of the agent and the goal $g$ which identifies the task. The goal $g$ identifies both the Successor Features $\boldsymbol{\psi}(s,a,g)$ as well as the goal weights $\mathbf{w}_g$. The Q-function is computed through the relation $Q_g(s,a) = \boldsymbol{\psi}(s,a,g)^T \cdot \mathbf{w_g}$.

# Chapter 2

# Methodology

The aim of the present work is to combine the architecture presented in Ma et al.[13] with the GPI procedure to further enhance the ability of the agents to quickly adapt to new tasks. In fact, Ma et al.[13] show that their USF outperforms a goal-conditioned DQN through the use of an architecture that uses the framework of Successor Features. They claim that the improvements come from the disentanglement of the dynamics of the environment from the particular task.

However, they do not utilize the benefits of the GPI procedure in their framework. Therefore, we would like to combine their results with this technique, in a similar fashion to the work done by Borsa et al.[3] where they design an architecture that benefits from the generalization power of general value functions[18] with the ability to take advantage of the structure of the environment dynamics to transfer knowledge through SF and GPI[1].

The difference is that we will make use of the simpler architecture used by Ma et al.[13]. Furthermore, we will explicitly analyze the ability to transfer to new tasks by employing the two stage training procedure described in Ma et al.[13]. Another important difference is that in the work of Borsa et al.[3] one of the inputs of the network are the goal weights, rather than the goal itself as in our case. We argue

that the latter representation is more natural in a large variety of scenarios while the former is usually unknown ahead of time.

Additionally, in contrast to both works, we will extensively analyze a variety of agents in order to understand the true implications of using USFs.

The methodology of the present work will consist in the following steps:

1. Replication of the work by Ma et al.[13] using the grid world environment with rooms.

2. The use of the GPI procedure in the second phase of training. In particular, it will be used in two places: to help the agent in selecting better actions to discover new goals faster, referred to as *action selection*, and during the evaluation of the agent's capability to solve the new tasks, referred to as *evaluation*.

The first step of the methodology will enable us to obtain the basic building blocks which we wish to extend. It will also enable us to instantiate a variety of agents not studied previously in order to understand the benefits of the proposed framework.

The second step will enable us to further improve the framework of Ma et al.[13]. Theoretically speaking, we expect that the GPI procedure will improve the speed with which the agent can adapt to new tasks.

## 2.1 Implementation Details

### 2.1.1 Environment

The environment is a $9 \times 9$ grid world with 4 rooms. In particular, the agent can take 4 actions: go up, down, right, left. The environment is fully deterministic, meaning that if the agent goes in a certain direction, it will travel one cell in that direction with probability $p = 1$, unless it is a wall, in which case it stays put.

At the beginning of an episode, the agent starts at a random state $s$ and it must reach a certain goal $g$, which represents the current task. Different goals correspond to different tasks, as discussed in section 1.3. The reward scheme used is: $-0.1$ per time step and 0 upon reaching the goal position. This is used to stimulate the agent to reach the goal as soon as possible.

In this environment we can represent $\boldsymbol{\phi}(s_t, a_t, s_{t+1})$ with $\boldsymbol{\phi}(s_{t+1})$. For example, we can use a one-hot vector of length $9 \times 9$ which indicates where the agent is in the grid world. Similarly, the goal weights are one-dimensional vectors of length $9 \times 9$ filled with $-0.1$ everywhere and 0 at the cell that corresponds to the goal position in the grid world.

Both $s$ and $g$ are represented with $(i, j)$ in matrix index notation. Therefore, the upper left corner corresponds to position $(0, 0)$. The environment is depicted in Fig. 2.1. We also implemented a regular grid world environment without rooms in order to be able to execute some experiments on a simplified version of our environment.

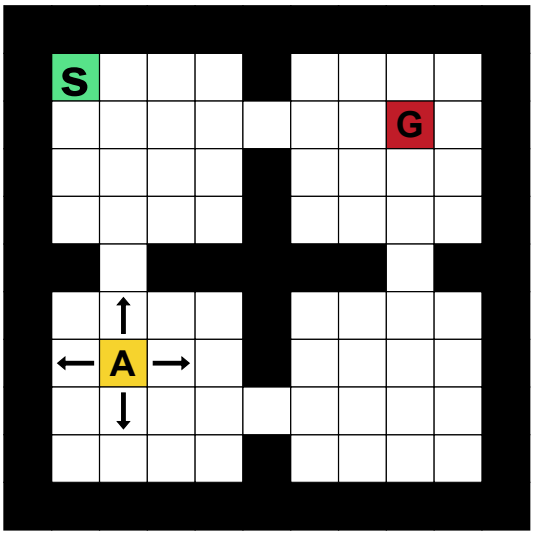


Figure 2.1: The grid world environment with 4 rooms. The agent starts from a random position `s` and must learn to navigate to the goal `g` which is randomly selected. At any step, the agent `A` can move up, down, right, or left to reach the goal. The reward is $-0.1$ per step, 0 upon reaching the goal position.

### 2.1.2 Training Procedure and Evaluation

As mentioned in section 1.5 the training procedure is divided into two parts: the first phase and the second phase.

To begin, we sample three disjoint sets of 12 random goals, with 3 goals per room. The first set of goals - referred to as *primary* tasks or goals - constitutes the base tasks $\mathcal{T}$ that the agent must learn and then transfer. The second set of goals - referred to as *secondary* tasks or goals - constitutes the tasks $\mathcal{T}'$ that the agent must solve by transferring knowledge from $\mathcal{T}$, previously learned. The third set - referred to as *tertiary* goals or tasks - represents a set of goals used for evaluating the zero-shot generalization to unseen goals.

During the first phase of training, the agent learns to reach the set of primary goals. At the beginning of each episode, a random goal is selected from the primary set. The agent tries to reach the goal within an allotted number of steps, which is

set to 31. If the agent reaches the goal or it exceeds the allotted steps, the episode terminates and the procedure restarts. The duration of the first phase is $48,000$ steps.

As the agent is being trained, it is evaluated through a metric called *done rate* which is the fraction of completed goals out of the set. During the first phase we evaluate the done rate of the primary goals every 100 steps. This is done to test the ability of the agent to learn to solve these goals. We expect that as the number of steps increases, the agent is able to achieve a done rate of 100% on the primary goals.

At the same time, the done rate of the tertiary goals is recorded every 100 steps to test the ability of the agent to generalize to unseen goals it is not being trained on. It is important to highlight that during the first phase, the agent is never trained on the tertiary goals. The expectation is that as the agent is better able to solve the primary goals, it also learns to generalize to unseen goals.

During the second phase of training the agent is trained on the secondary goals. In this phase, the memory buffer from the first phase is carried on, while a second new buffer is maintained. At each step of training, one of the buffers is chosen with equal probability to update the agent. In this phase, the done rate of the secondary goals is used to observe the speed with which the agent learns to solve the secondary goals. The second phase lasts $12,000$ steps and we measure the done rate on the secondary goals every 100 steps.

All the experiments were run 10 times with different seeds and averaged to give single measures.

We list all the hyper-parameters used in the Table 2.1.

Table 2.1: The default hyper-parameters used in the grid world rooms environment.

| Hyper parameters | DQN | USF |
|---|---|---|
| Learning Rate | $5e^{-4}$ | $5e^{-4}$ |
| $\epsilon$ for $\epsilon$-greedy | 0.25 | 0.25 |
| $W_\psi$ | N/A | 0.01 |
| Batch Size | 32 | 32 |
| Discount Factor $\gamma$ | 0.99 | 0.99 |
| Target Network Update Frequency | 10 | 10 |

### 2.1.3 Architecture and Loss

To combine GPI with the USF architecture, we modify the architecture shown in Fig. 1.2 so that the neural network uses two goals as input rather than one. This is illustrated in Fig. 2.2.

The first goal on the left, $g_p$, selects the policy. In other words, it generates the Successor Features $\boldsymbol{\psi}(s, a, g_p)$ for the policy $\pi^{g_p}$ that has learned to navigate to $g_p$. On the other hand, the second goal on the right, $g_e$, represents the current goal the agent is trying to reach.

In this way, during the second phase of training, we can select the policies learned during the first phase by using the primary goals as the input $g_p$ and we can evaluate their performance on the current task represented by one of the secondary goals, identified by $g_e$.

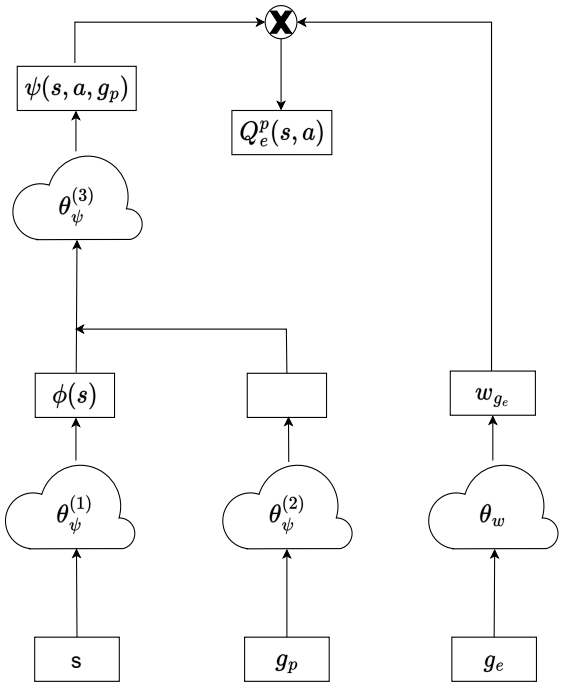


Figure 2.2: The modified USF architecture in order to combine GPI with the USF framework. All USF agents use two goals as input: $g_p$ which selects the policy that learned to navigate to $g_p$ by generating the SFs $\boldsymbol{\psi}(s, a, g_p)$ and $g_e$ which represents the current goal the agent is trying to reach.

With this modification, we can use GPI by simply specifying the set of policies of interest. In our case, this is the union between the 12 policies identified by the primary goals and the policy identified by the current goal the agent wants to reach. Ideally, by including the current policy in the set of policies of interest, the GPI procedure should eventually keep selecting policy $\pi^{g_e}$ when trying to reach goal $g_e$.

To avoid confusion, it is important to stress that we can only use the GPI procedure in two places:

1. When evaluating the done rate of the agent on a set of goals: in this case, for each goal we want to reach, we allow the agent to also use the policies it learned during the first phase.

2. When selecting the next action to take: at each step, the agent will use GPI to evaluate what action to take to reach the current goal. This changes the transitions that the agent sees compared to not using GPI for action selection.

In the first case, we expect a better done rate since the agent can use other policies when trying to reach a goal. In the second case, we expect that the agent will see transitions of higher quality. Therefore it will need to explore less to reach the new goals which means that we will have faster convergence.

When updating the agent, we use $g_e = g_p$ to update only the policy for that goal. The goal the agent wants to reach is included the transitions that are stored in the memory buffer.

In order to learn these components, Ma et al.[13] use a weighted loss of $L_Q$ and $L_\psi$:

$$L = W_Q \cdot L_Q + W_\psi \cdot L_\psi, \tag{2.1}$$

with $W_Q = 1.0$ and $W_\psi = 0.01$.

It is useful to note that Eq. 1.7 for $L_Q$ can be rewritten in terms of $\boldsymbol{\psi}(s, a, g; \boldsymbol{\theta}_\psi)$ and $\mathbf{w}(\boldsymbol{\theta}_w)$ as follows:

$$L_Q = \left[ r_{t+1} + \gamma \max_a \boldsymbol{\psi}^{\text{target}} \left(s_{t+1}, a, g; \bar{\boldsymbol{\theta}}_\psi\right)^T \cdot \mathbf{w}(g; \bar{\boldsymbol{\theta}}_w) - \boldsymbol{\psi}^{\text{policy}} \left(s_t, a_t, g; \boldsymbol{\theta}_\psi\right)^T \cdot \mathbf{w}(g; \boldsymbol{\theta}_w) \right]^2. \tag{2.2}$$

It is important to highlight that this loss is a scalar loss since we have the dot product between $\boldsymbol{\psi}$ and $\mathbf{w}$. On the other hand, the loss $L_\psi$ shown in Eq. 1.13 is a vectorial loss since it relies on the $\ell^2$-norm to compute the distance between vectors. The nature of both of these losses will be important to better understand this framework later.

The goal-conditioned DQN agent is trained only with $L_Q$ shown in Eq. 1.7.

### 2.1.4 Agents

Since we are interested mainly in a scenario where the features are known, we implemented various agents, each with a different configuration compared to the original agents in Ma et al[13]. We also implemented the goal-conditioned DQN agents used as baselines in Ma et al.[13].

We first list the agents with a small description for quick reference. Afterwards, we explain each architecture in depth, as well as the intuition behind each agent.

- **State-Goal USF (SG USF)**: An agent based on the USF architecture. As input, it takes the agent's position and the goal position. It learns the goal weights $\mathbf{w_g}$, the features $\boldsymbol{\phi}(s)$, and the SFs $\boldsymbol{\psi}(s, a, g)$.

- **State-Goal-Weight USF (SGW USF)**: An agent based on the USF architecture. As input, it takes the agent's position, the goal position, and the goal weights. It learns the features $\boldsymbol{\phi}(s)$, and the SFs $\boldsymbol{\psi}(s, a, g)$.

- **Feature-Goal USF (FG USF)**: An agent based on the USF architecture. As input, it takes the agent's features $\boldsymbol{\phi}(s)$ and the goal position. It learns the goal weights $\mathbf{w_g}$ and the SFs $\boldsymbol{\psi}(s, a, g)$.

- **Feature-Goal-Weight USF (FGW USF)**: An agent based on the USF architecture. As input it, takes the agent's features $\boldsymbol{\phi}(s)$, the goal position, and the goal weights $\mathbf{w_g}$. It learns the SFs $\boldsymbol{\psi}(s, a, g)$.

- **State-Goal DQN (SG DQN)**: An agent based on the DQN architecture. As input, it takes the agent's position and the goal position. It learns a general Q-function which is a function of the state, action, and goal - $Q(s, a, g)$.

- **State-Goal DQN Augmented (SG DQNA)**: This is the same as the agent above, except it accounts for the extra network weights available in the corresponding USF.

- **Feature-Goal DQN (FG DQN)**: An agent based on the DQN architecture. As input, it takes the agent's features $\boldsymbol{\phi}(s)$ and the goal position. It learns a general Q-function which is a function of the features $\boldsymbol{\phi}(s)$, action, and goal - $Q(\boldsymbol{\phi}(s), a, g)$.

- **Feature-Goal DQN Augmented (FG DQNA)**: This is the same as the agent above, except it accounts for the extra network weights available in the corresponding USF.

The SG USF agent corresponds to the "learned representation" agent described in Ma et al.[13]. It learns the SFs end-to-end by experiencing the environment, learning the features, the goal weights, and the SFs. We use the architecture described Fig. 2.2. The architectures for all the other agents can be derived from Fig. 2.2 by eliminating some sub-branches. The exact details are discussed in Appendix A.1.

The SGW USF agent learns both the features and the SFs. This task is simpler than the full SG USF agent since it does not need to learn the goal weights. This agent is useful to decouple the complexity of learning the features and SFs from learning the goal weights.

The FG USF agent corresponds to the "one-hot encoding" agent described in Ma et al.[13]. It uses a one-hot vector as features: a vector of length $9 \times 9$ filled with zeros everywhere and 1 at the cell that corresponds to the agent's position in the grid world. This agent only needs to learn the goal weights and the SFs. It requires knowledge of the features in advance. In robotics tasks, these are usually

known and controlled. Furthermore, considering the SPRING project, this was the main use-case of interest.

The FGW USF represents an agent that only has to learn the SFs. This agent has to learn the minimal amount of information compared to the other ones. It is helpful to analyze the performance of this agent to understand the difficulty of learning the SFs without the added difficulty of learning the features and the goal weights.

Finally, there are different DQN versions that act as baseline references. The logic behind having an augmented DQN version for each case is to ensure that the superior performance of the USFs is not due to the increased representation power deriving from its additional network weights. In fact, in Ma et al.[13] the architecture used for a DQN agent essentially corresponds to Fig. 1.2 without the right part of the network which learns the goal weights. Therefore, it is not immediate that the superior performance is due to some fundamental improvement or an increased representation power. Here, we aim to investigate this issue further. The exact architectural details for the DQN agents are discussed in Appendix A.1.

# Chapter 3

# USF with GPI

## 3.1 Replicating USFs

The first step towards mixing USFs and GPI is to replicate the results obtained by Ma et al.[13]. Fig. 3.1 shows the results of training the agents on the primary goals during the first phase. Fig. 3.2 shows the performance of the agents on the unseen tertiary goals during the first phase of training. Fig. 3.3 show the results of training all the agents on the secondary goals during the second phase.

The results match those obtained by Ma et al.[13], except for the SG USF agent which does not converge. This is problematic since according to them, this should be the agent that performs the best. The SGW USF agent also isn't able to converge. We see that the FG USF agent performs better than the SG DQN agent both during the first phase with the primary goals and during the second phase with the secondary goals, which is the behavior seen in Ma et al.[13].

We also observe that the augmented versions of DQN perform the best in all scenarios except in the first phase training with the primary goals, where the FG USF outperforms the SG DQNA. However, the difference is very small and, considering that the SG DQNA is more general as it does not need knowledge of features, this

difference could be meaningless in real applications. In fact, the SG DQNA is able to generalize better on unseen goals and learn faster in the second phases of training.

These results beg the question whether the USF architecture truly provides an advantage or if the results obtained by Ma et al.[13] were due to an incorrect comparison between networks of different representation power.

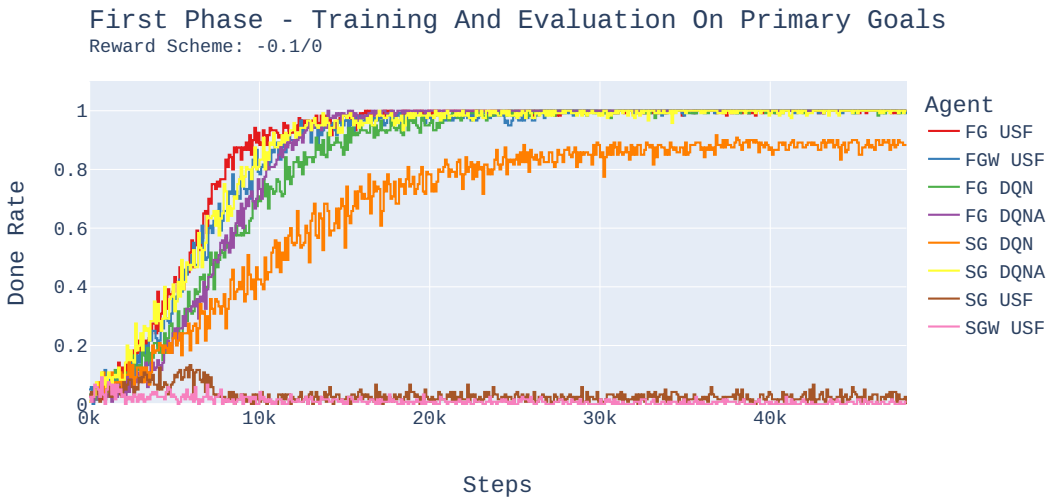


Figure 3.1: First phase of training for all the agents on the primary goals. All agents except for the SG USF and SGW USF are able to learn to navigate to the 12 goals after $48,000$ steps although the SG DQN is not able to reach a done rate of 100%.

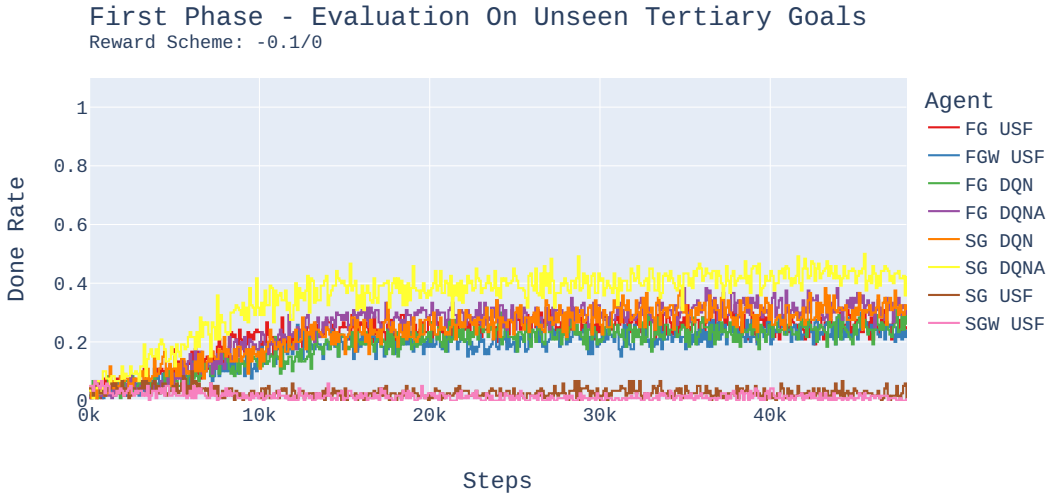


Figure 3.2: Testing the zero-shot ability to solve unseen goals by evaluating the done rate of the tertiary goals during the first phase of training. All agents except for SG USF and SGW USF are able to generalize to a certain extent on the unseen goals.

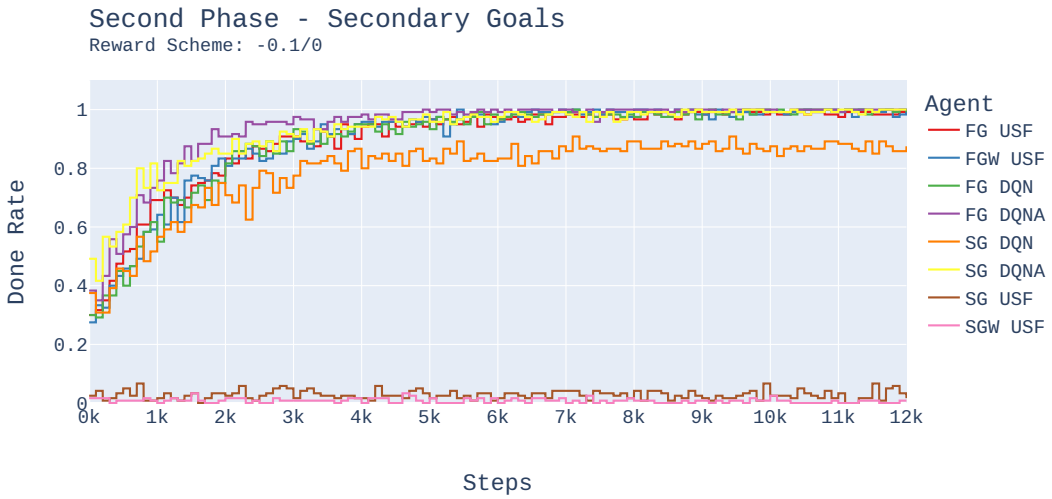


Figure 3.3: Second phase of training on the secondary goals for all the agents. The SG USF and SGW USF agents do not converge which is normal since they were not able to learn in the first phase. The rest of agents show an improved ability to learn to solve the goals compared to the first phase, as evidenced by the higher initial done rate and the lower amount of steps needed to reach 100% done rate.

### 3.1.1 Improving Convergence of SG USF and SGW USF

In order to make SG USF and SGW USF agents converge, we tried the following techniques:

- Use different learning rates.
- Use increasing batch sizes.
- Use increasing learning iterations that the neural network uses for each batch.
- Use increasing values of $W_{\psi}$ to give more significance to the loss term $L_{\psi}$. Our intuition was that this term could increase the information available to the network to learn the features and the SFs.
- Use Prioritized Experience Replay (PER)[17]. Our idea was that the network was not able to properly learn the features and the SFs because it was encountering transitions that were not useful. By using PER, we hoped to guide the network to learn better by providing more impactful transitions.

All of these techniques resulted to be unsuccessful. The results are shown in Appendix B.

One possible reason for the discrepancy between our SG USF and the one in the original work is that not all of the parameters were fully specified in the work by Ma et al.[13]. Therefore, it could be the case that the learning procedure is unstable without the right configuration. In fact, it is well known that in RL, the deadly triad[5] of function approximation, bootstrapping, and off-policy learning results in instability.

Since our attempts to replicate the results of Ma et al.[13] were partly successful, we decided to use GPI in the second phase with the USF agents that were able to learn properly, namely the FG USF and the FGW USF. In this way, we could obtain

some insight into the utility of our hypothesis that mixing GPI with USF could yield improved results.

## 3.2 Using GPI

Fig. 3.4 shows the results for using GPI during the evaluation of the agent. The lower performance indicates that the agents are being misled by the GPI procedure. It is important to remember that in this case, the agents see the same transitions as the normal case without GPI in Fig. 3.3. This means that if we were to avoid using the GPI procedure to evaluate the agents, we would have the same performance as the normal case. Thus, when using GPI in the evaluation, the agents are being misled to the point of rarely using the correct policy that can solve the goal.

Fig. 3.5 shows the results for using the GPI procedure during action selection. This means that the agents will see completely different transitions compared to the normal case in Fig. 3.3. The lower performance shows that the quality of transitions seems to be worse and the agents are unable to learn to reach the new goals. Given our previous conclusions about GPI, this is not surprising since we can assume that the GPI procedure is making the agents take actions that hinder the learning process. The results for using GPI during both action selection and evaluation are similar to the case of using GPI during evaluation in Fig. 3.4 so they are shown in Appendix C.

We were surprised with these results, since we expected that the performance should at least stay the same, if not improve. This led us to investigate deeper into the policies that were being selected with the GPI procedure.

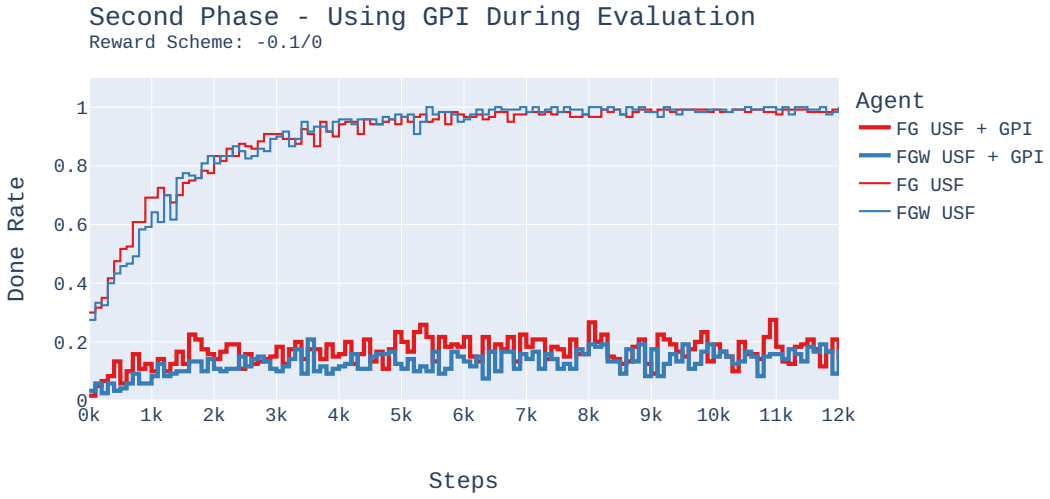


Figure 3.4: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during evaluation while the thinner lines correspond to the agents that do not use it. Both agents that use GPI achieve a done rate below 20%. This shows that the GPI procedure is misleading the agents in choosing the wrong policies during evaluation.

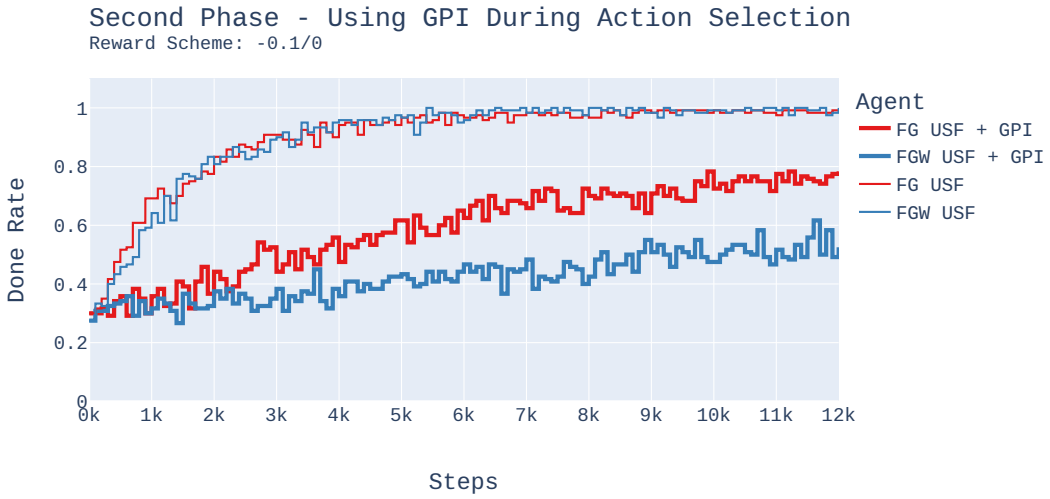


Figure 3.5: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during training to select actions while the thinner lines correspond to the agents that do not use it. The agents that use GPI are unable to reach the same performance as the agents that do not use it, which shows that the GPI procedure leads the agents to select unhelpful actions for learning.

# Chapter 4

# Changing The Reward Scheme

## 4.1 A Wrong Encoding

To understand the reason behind the results in section 3.2, we studied the trajectories of the agents. In our case, a trajectory was composed by the current goal the agent wanted to reach in that episode, the actions it took, and the policy the agent selected when using the GPI procedure at each time step along the along the path in the grid world.

We kept track of all the trajectories of the episodes where the agents were not able to reach the goal and exceeded the limit of steps. In this way, we were able to analyze what actions the agent was taking and which policies it was selecting when trying to reach a specific goal that would cause it to go the wrong way.

We learned that very often, the agents were selecting the policy that would take them to the nearest goal to their current position, disregarding everything else. We realized that this fact was fundamentally tied to the reward scheme chosen by Ma et al.[13] which encoded the idea of reaching *any* goal as soon as possible rather than reaching a specific goal.

To illustrate why this is so, consider the following example. Suppose that in the

first phase, the agent has learned to navigate to the goals `P1` and `P2` shown in Fig. 4.1. Suppose that in the second phase, the agent is currently between both of these goals, and has to learn to navigate to a secondary goal `S1`, which is immediately adjacent to `P1`.

Since the agent already knows how to navigate to `P1`, and `S1` is in the path to `P1`, we expect that the agent will use policy `P1`, and thus go left. However, the agent actually picks policy `P2` and goes right.

To see why this is so, we must analyze the SFs of the policies in this example. Since, in this case, our features are the cells, the SFs represent the expected frequency of each cell in the future with the given policy. The SFs for both policies and the goal weights for the current task look like Eq. 4.1.

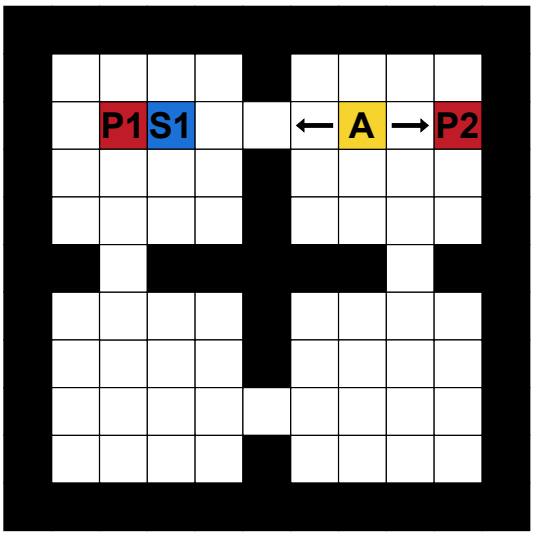


Figure 4.1: A special configuration. The agent has learned to navigate to the primary goals `P1` and `P2` during the first phase. In the second phase, it has to learn to navigate to the secondary goal `S1`. We expect the agent to use the policy `P1` to go to `S1`. However, the agent uses policy `P2` and goes right. This can be explained by analyzing the SFs of the problem, as shown in Eqs. 4.1-4.2.

$$\boldsymbol{\psi}_{P1}(A,\leftarrow) = \begin{bmatrix} 0 \\ \vdots \\ \gamma^4 \\ \gamma^3 \\ \vdots \\ 1.0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \boldsymbol{\psi}_{P2}(A,\rightarrow) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 1.0 \\ \gamma \\ 0 \\ \vdots \\ 0 \end{bmatrix} \mathbf{w}_{S1} = \begin{bmatrix} -0.1 \\ \vdots \\ -0.1 \\ 0 \\ \vdots \\ -0.1 \\ \vdots \\ -0.1 \\ -0.1 \\ -0.1 \\ -0.1 \end{bmatrix} \tag{4.1}$$

This means that the dot products, or, equivalently, the Q-function for both policies in the current task, will be as follows:

$$\boldsymbol{\psi}_{P1}(A,\leftarrow)^T\mathbf{w}_{S1} = -0.1(1+\gamma+\gamma^2+\gamma^4) < -0.1(1+\gamma) = \boldsymbol{\psi}_{P2}(A,\rightarrow)^T\mathbf{w}_{S1} \tag{4.2}$$

Since the agent uses the GPI procedure, it will maximize the Q-function over actions and policies. Since the highest value is given by the second product, it will use the second policy and go right. Therefore, with the current reward scheme, the agent will always choose the policy that learned to go to the nearest goal. This is the reason for the poor performance we saw in section 3.2.

## 4.2 New Reward Scheme

To fix the issue mentioned in the previous section, we changed the reward scheme to the following: 0 per time step and 20 at the goal position.

The idea behind this choice was not to penalize the agent for moving - which

was the main issue with the previous scheme - and give a positive reward upon discovering the goal position.

We also changed the discount factor $\gamma$ from 0.99 to 0.90. The intuition was that the if the learning was not converging for some USF agents, it could be beneficial to lower the discounting and give less importance to future rewards. However, we also ran our experiments with $\gamma = 0.99$ which shows worse results than with $\gamma = 0.90$. This tells us that the problem is very sensitive to the choice of $\gamma$. The results for $\gamma = 0.99$ can be seen in Appendix D.

Fig. 4.2 shows that with the new reward scheme, both the SG USF and SGW USF agents converge and are able to learn to reach the primary goals during the first phase of training. However, they do not show the impressive performance results that were shown in Ma et al.[13]. This is one of the first indicators of a common theme in this work: the unstable nature of this method. As we will see later, training Successor Features in this particular environment resulted to be difficult and prone to instability, where changing a small aspect would radically change the results.


Figure 4.2: First phase of training for all the agents on the primary goals with the new reward scheme. In contrast to Fig. 3.1, the SG USF and SGW USF agents are able to learn to solve the tasks and achieve a high done rate.

We also observe that in the first phase, the best performing agents are the feature-based USF agents. However, the SG DQNA agent, which is more general and does not require the knowledge of features, performs comparably to both the FG USF and FGW USF agents. It is worth mentioning that the FGW USF architecture uses approximately 20% less network weights than SG DQNA. Therefore, this may be an indication of the benefits of using Successor Features.

Additionally, Figs. 4.2-4.4 re-confirm the fact that overall, the SG DQNA agent is the best performing agent both in terms of ability to learn the new tasks efficiently and to generalize to unseen goals. This is evidenced by the fact that it is consistently among the top best performing agents in all the cases. Furthermore, it is a testament to the robustness of this algorithm since it is stable across different reward schemes.

In Fig. 4.4 we also notice that the SGW USF agent is not able to learn the new tasks in the second phase, regardless of the fact that it was able to learn the primary tasks in the first phase.

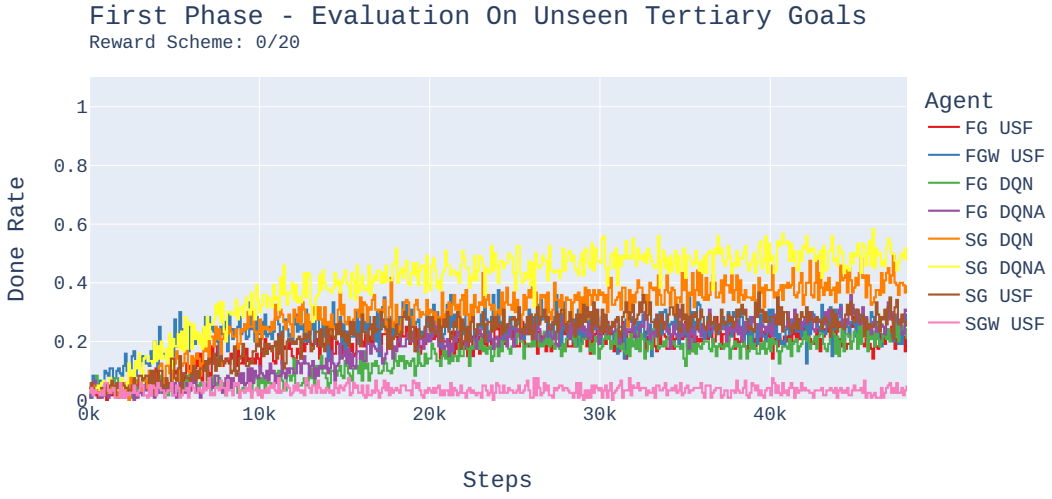


Figure 4.3: Testing the zero-shot ability to solve unseen goals by evaluating the done rate of the tertiary goals during the first phase of training using the new reward scheme. The majority of agents can generalize to some extent on the unseen goals while the SGW USF agent is not able to so so at all.

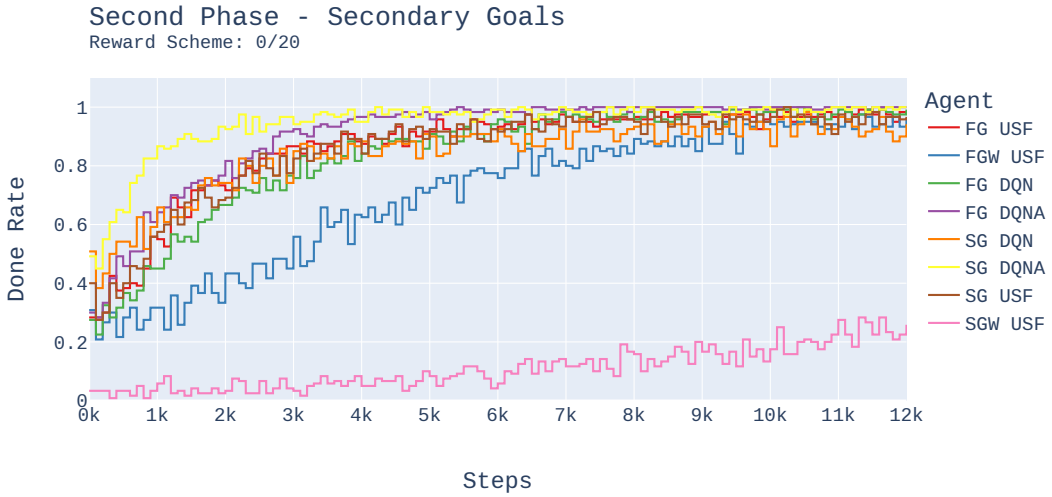


Figure 4.4: Second phase of training on the secondary set of goals with the new reward scheme. All the agents are able to learn to solve the new tasks except for the SGW USF agent.

## 4.3 Using GPI with New Reward Scheme

Figs. 4.5-4.6 show that the situation has improved compared to using GPI with the original reward scheme in section 3.2.

In Fig. 4.5 we can see that when using GPI for evaluating the agents, the FGW USF agent seems to be able to slowly improve. This means that, as the agent learns to solve the new goals, the GPI procedure is able to pick the correct policy more often. Therefore, the agent is not being completely misled anymore.

In the second case when using GPI for action selection, Fig. 4.6 shows that all USF agents slowly increase their performance over the time steps. This is further confirmation that the GPI procedure is not hindering the learning process of the agents, as was happening previously. The results for using GPI during both action selection and evaluation are similar to Fig. 4.5 so they are shown in Appendix C.

However, there is still a problem with the GPI procedure since the performance is lower than the case where the agents do not use GPI. We expect that the GPI pro-

cedure should improve, or at least maintain, the agents' performance when learning the new tasks.

To understand what is happening, we need to visualize in more detail the Successor Features that the agents are learning.

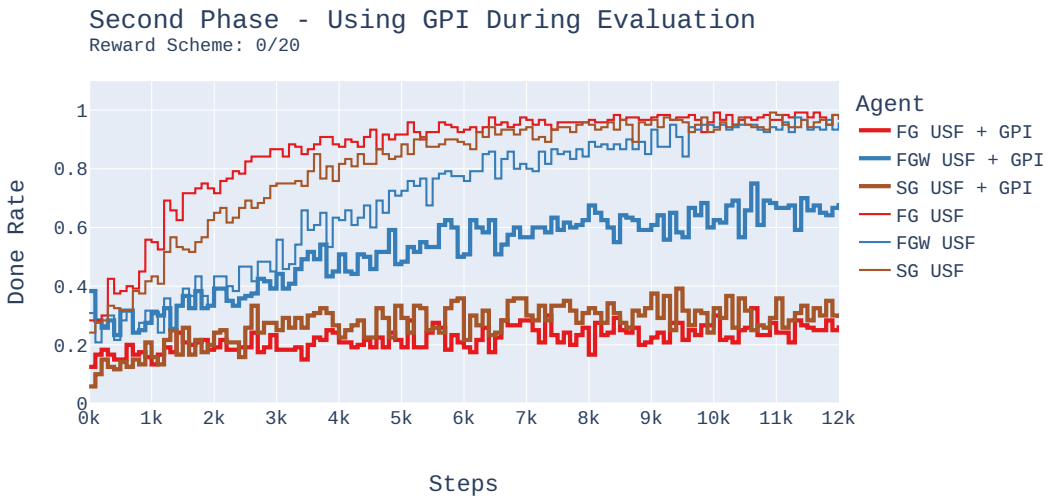


Figure 4.5: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during evaluation while the thinner lines correspond to the agents that do not use it. The FGW USF agent seems to slowly improve which means that as the agents learn to solve the new goals, the GPI procedure picks the new policies more often. However, there is still a problem with using GPI since all the agents have a low performance compared to the baseline.

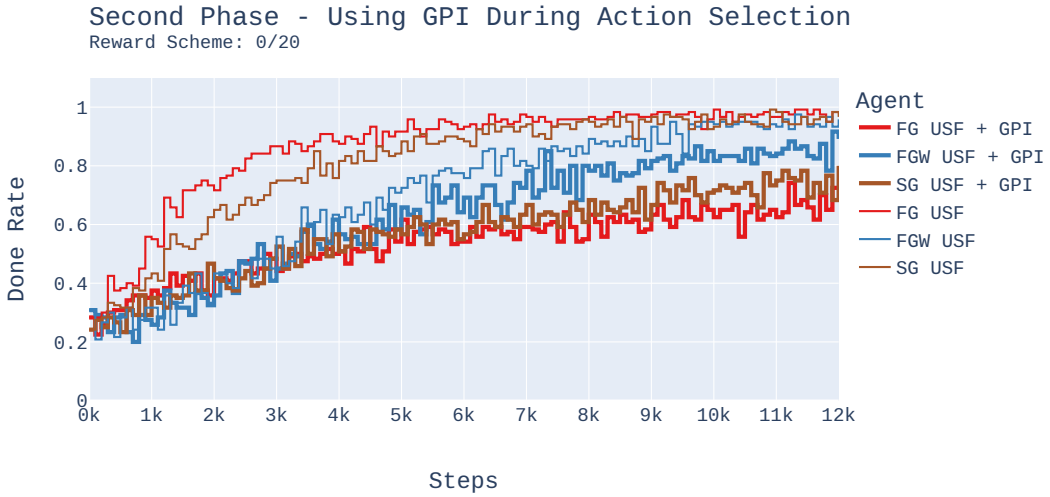


Figure 4.6: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during training to select the actions while the thinner lines correspond to the agents that do not use it. All the agents slowly increase their performance as the training proceeds which means that GPI is not hindering the learning process.

## 4.4 Visualizing The Successor Features

By analyzing the SFs that the agents learned, we realized that the main issue is that the agents are not correctly learning the SFs. Instead, they are learning to give a lot of importance to the goal states that they have learned to navigate to.

In Fig. 4.7, we analyze the SFs that a FGW USF agent has learned after the first phase. We show a particular experiment run with the following configuration: the current goal the agent wishes to navigate to is shown in purple at position $(1, 1)$; the agent is shown in yellow at position $(1, 7)$; the value of the SFs are shown in shades of blue of different intensity based on the value. We only show the SFs for the action of going left.

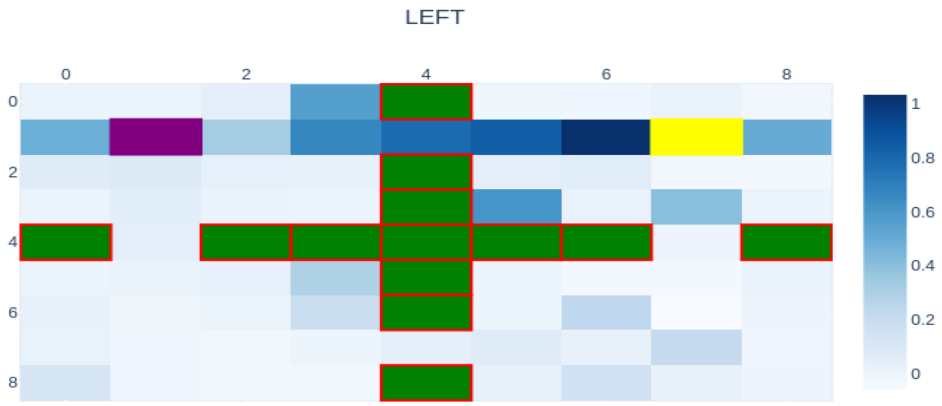


Figure 4.7: The heat plot of the SFs for a particular experiment run. The goal that the agent has to reach is shown in purple and the agent's position is shown in yellow. The SFs shown are for the action of going left. We observe that some cells close to the agent and in the lower quadrants are unusually active, considering the optimal policy. This indicates that the agent has not learned the correct SFs.

As we can see, if the agent chooses to go left, a path to the goal is activated according to the expectation of encountering those cells - also features in this case - in the future. Since going left means the agent will land in $(1, 6)$, that cell has the highest activation, as it should. Then, we see that the most active path to the goal is a straight horizontal path from the agent, as we expect.

However, we also notice that some unusual cells are very active. To the right of the agent, cell $(1, 8)$ is very active considering that the agent will go in the opposite direction. Also, cell $(3, 5)$ and $(3, 7)$ are very active considering that the agent should go in a straight path. If we look at the lower right and left quadrants, we see six cells that are more active than their neighbors.

All of these cells correspond to primary goals that the agent learned to reach in the first phase of this particular run. This is a clear example that the agent has not correctly learned the SFs for the problem. It gives a lot of importance to goal positions, regardless of the situation. So, the value of some cells are always

overestimated.

This is not problematic if we only care about reaching a single goal using a single policy because it is irrelevant if some intermediate cells are overestimated as long as the goal is reached. However, when we want to re-use past policies through GPI to help navigate to new unseen goals, the agent uses the overestimated cells (which represent new goal positions) to decide what policy to use. This then creates a problem because it causes the GPI procedure to pick the wrong policies, negatively impacting the performance.

The reason for this issue is fundamentally linked to the loss we are using given by Eq. 2.1. The dominating term is $L_Q$ which is proportional to the dot product between the Successor Features $\boldsymbol{\psi}(s, a, g)$ and the goal weights $\mathbf{w_g}$. In this case the goal weights are a $9 \times 9$ vector filled with zeros, except for the cell which corresponds to the goal position which contains the value 20. So, the dot product filters the value for all the cells except for the goal cell. On the other hand, the $L_\psi$ term, which is a vectorial loss that helps the agent to learn the SFs, has a weight of 0.01. In other words, throughout the learning process, most of the importance is placed on the learning the value of the goal cell rather than the SF vector as a whole.

To attempt to fix this, we decided to understand what would happen if we tried to correctly learn the SFs. Our idea was that if we could learn the SFs properly, then the GPI procedure was guaranteed to work when used in the second phase of training.

# Chapter 5

# Learning Successor Features

In order to correctly learn the SFs, we use only the term $L_\psi$ as the loss. This automatically excludes all the agents that need to learn the weights since $L_\psi$ does not enforce any conditions on $\mathbf{w_g}$.

We decided to focus our attention on the FGW USF agent since it only has to learn the SFs. In this way, we can focus solely on the task of learning the SFs without having interfering effects from learning the features $\boldsymbol{\phi}$.

Fig. 5.1 shows that the agent is not able to learn the SFs. It is only able to reach a done rate of $\approx 45\%$. In Fig. 5.2 we ran the same experiments on a grid world environment without rooms in order to reduce the difficulty of the problem. We can see that in this case the agent is able to reach $\approx 85\%$ done rate. This indicates that in the case of the grid world with rooms, the problem is too complex for the agent to learn the SFs with the current architecture. This conclusion applies also to the case of the simplified grid world without rooms, although to a smaller degree.

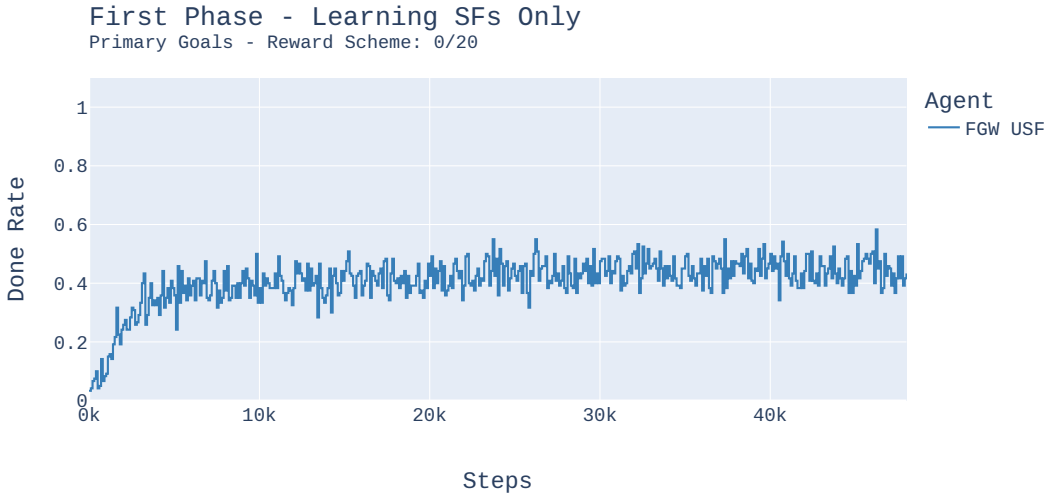


Figure 5.1: First phase of training for the FGW USF agent on the primary goals. The agent is trained in the grid world with rooms using only $L_\psi$ as the loss. It is able to reach a done rate of 45%, which means it is unable to learn to navigate to all the goals.

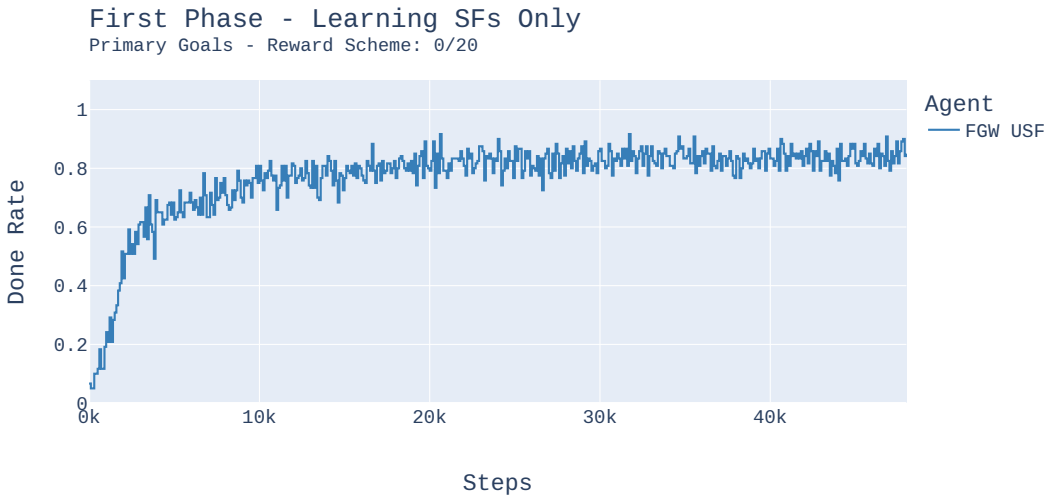


Figure 5.2: First phase of training for the FGW USF agent on the primary goals. The agent is trained in the simplified grid world without rooms using only $L_\psi$ as the loss. It is able to reach a done rate of 85%.

## 5.1 Improving Convergence of SF Learning

To understand the situation further, we decided to study the behavior of the agent as the complexity of the problem increased. We made the agent learn in simplified grid world environments of increasing size while keeping the ratio of goals to cells constant. This ensures that the complexity increases, but the ratio of information to learn remains approximately constant. We used the values shown in Table 5.1.

Table 5.1: The dimensions and number of goals used for learning in increasingly larger simple grid world environments without rooms. The ratio of goals to cells is approximately constant.

| Rows × Columns | Number of Goals | $\frac{N.Goals}{N.Cells}$ |
|---|---|---|
| $3 \times 3$ | 1 | 0.111 |
| $4 \times 4$ | 2 | 0.125 |
| $5 \times 5$ | 4 | 0.16 |
| $6 \times 6$ | 5 | 0.138 |
| $7 \times 7$ | 7 | 0.142 |
| $8 \times 8$ | 10 | 0.156 |
| $9 \times 9$ | 12 | 0.148 |
| $10 \times 10$ | 15 | 0.150 |

The results in Fig. 5.3 show that for an $8 \times 8$ grid world and onwards, the agent progressively gets worse when trying to learn the SFs. This effect is not related to the reward scheme, since $L_\psi$ does not depend on it. This issue is probably related to the training method, the architecture, or the choice of hyper-parameters.

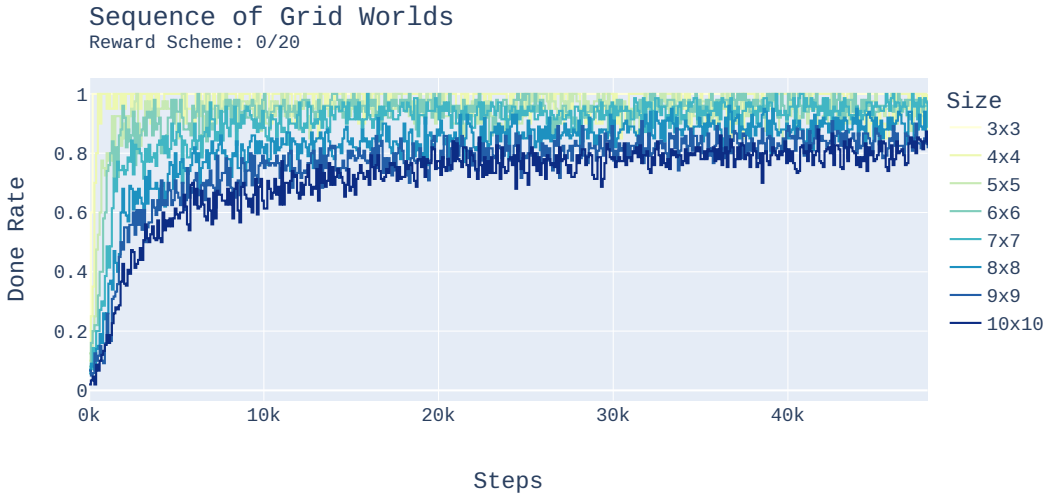


Figure 5.3: First phase of training for the FGW USF agent on the primary goals on a sequence of increasingly larger simple grid worlds using only $L_\psi$ as loss. For a size of $8 \times 8$ and onwards, the agent is unable to learn to navigate to all the primary goals.

To help the agent learn the SFs, we decided to try various approaches, namely:

- Using features based on the radial basis function rather than a one-hot encoding.
- Using a pre-trained agent as a target network and as an optimal selector of actions.
- Changing the network architecture using past related work as inspiration.
- Using the full loss in Eq. 2.1 with $W_\psi = 1.0$ and different $W_Q$.

The first two techniques resulted to be unsuccessful as they did not improve the performance of the agent nor did they provide any particular insights. The details are discussed in Appendix E.

We discuss the last two techniques in more detail.

## 5.2 Changing Network Architecture

One possibility is that the network architecture described in Ma et al.[13] is not able to properly represent the SFs for the problem we are facing. Unfortunately, there is no prescription to fix this problem as neural networks are still an active area of research and they are often treated as black boxes.

To attempt to fix this, we decided to take inspiration from two similar works: one by Borsa et al.[3] and the other by Kulkarni et al.[10].

In the first work, the authors had a similar goal to ours. Contrary to our case, they dealt with images, so their architecture contained convolutional layers and an LSTM, thus we could not take inspiration from this.

However, one interesting aspect was the way they trained their agent. The key insight is that at any one step, the USF can be trained for any policy identified by a goal $g_p$. This is because the loss $L_\psi$ depends only on $\boldsymbol{\phi}$ and $\boldsymbol{\psi}(s, a, g_p)$. Therefore, a transition of the form $(s_t, a_t, \phi_{t+1}, s_{t+1})$ would allow us to compute the term $L_\psi$ for any policy we want to update.

In the case of Borsa et al.[3], at each step, they sample a set of sub-goals close to the goal the agent is currently trying to reach and update those policies as well. In other words, at any given step, they also learn the SFs for the policies that take the agent to goals close to the current one.

In our case, this would correspond to sampling another subset of goals inside the room of the goal we currently want to reach. However, if we do this, we lose the logical separation between primary and secondary goals which helps us to analyze the capability of transferring knowledge of the method.

However, we take inspiration from this approach by updating all of the primary policies at each iteration of training. In other words, at each step of training, we sample a mini-batch of transitions and we update all of the policies identified by the

12 primary goals we are training on. So the mini-batch size is multiplied by 12. We refer to this technique as *data augmentation*. The results are shown in Fig. 5.4.

The performance has improved compared to the original case although it seems as though the network is stuck at a local minima from which it is not able to escape. Attempts to improve upon this involved changing the optimizer, the learning rate, and the batch size. None of these attempts yielded considerable improvements. The results are shown in Appendix F.

In Kulkarni et al.[10], in terms of architecture, they used a separate branch for each action to represent $\boldsymbol{\psi}(s, a)$. Therefore, we replicated this approach by changing our network structure in a similar way. We also made our network deeper, taking inspiration from some aspects of the architecture used in Borsa et al.[3]. The details of the new architectures are discussed in Appendix A.2.

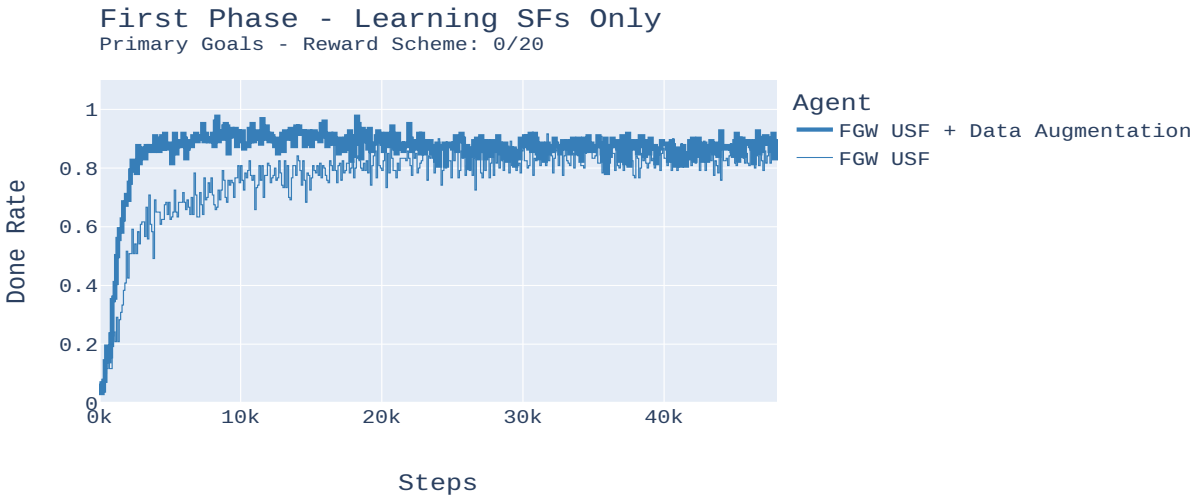


Figure 5.4: First phase of training for the FGW USF agents on the primary goals. The thicker line corresponds to an agent that uses data augmentation during training, while the thinner line is the baseline agent. The agents are trained in the grid world without rooms using only $L_\psi$ as the loss. We can see that data augmentation increases the performance of the agent during the first $20k$ steps.

In Fig. 5.5 we can see that using a deeper network improves the performance for the first $30k$ steps. On the other hand, using a separate branch for each action seems to decrease the performance. This could be due to an over-parametrization of the network or perhaps there needs to be more tuning. We were hesitant to change the network architecture since this could lead to further complications. Therefore, we only incorporated the use of data augmentation.

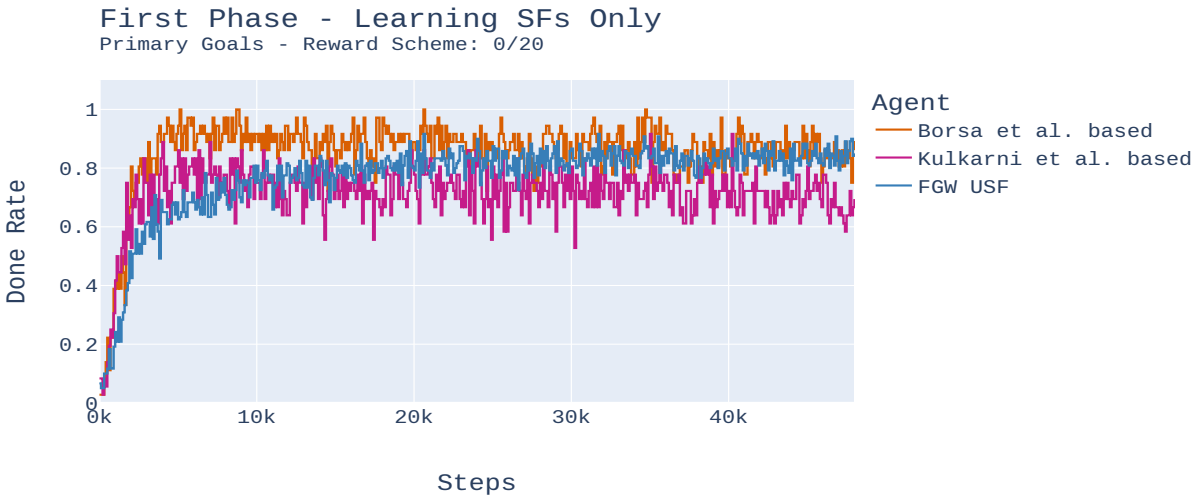


Figure 5.5: First phase of training for the FGW USF agents on the primary goals using a deeper architecture based on Borsa et al. [3] and a more parametrized one based on Kulkarni et al. [10]. The baseline agent is shown in blue. We can see that using a deeper architecture boosts the performance for the first $30k$ steps while using a more parametrized network seems to lower it.

## 5.3 Changing the Weights of the Loss

Another idea we used to improve learning $\boldsymbol{\psi}$ was to change the weights of the loss in Eq. 2.1. We reasoned that with $W_Q = 1.0$ and $W_\psi = 0.01$, we were giving all the importance to $L_Q$ and very little to $L_\psi$. This resulted in poorly learnt $\boldsymbol{\psi}$ functions, but very good goal reaching capabilities.

Therefore, we hypothesized that the opposite effect could help learning. In other

words, we give the most importance to $L_\psi$ to properly learn $\boldsymbol{\psi}$ while giving a small importance to $L_Q$ to help the agents reach the goal positions.

In practice, we used the loss in Eq. 5.1 combined with data augmentation.

$$L = L_\psi + W_Q \cdot L_Q, \quad \forall \ W_Q \in [0.01, 0.001, 0.0001] \tag{5.1}$$

The results are shown in Fig. 5.6. The agents are able to solve the tasks with a high done rate using this new loss. Even with a value of $W_Q = 0.0001$, the agent is able to perform better than the previous cases with $W_Q = 0$, although it fluctuates around 95% done rate. One possibility for this behavior, as we mentioned earlier, is that the term $L_Q$ focuses specifically on updating the goal cell. Therefore, it could be the case that with this loss we are always updating an important cell. This, in turn is able to help the agents learn.

In summary, out of all our attempts to make the agents learn the correct SFs, this was the only one that had a real success.

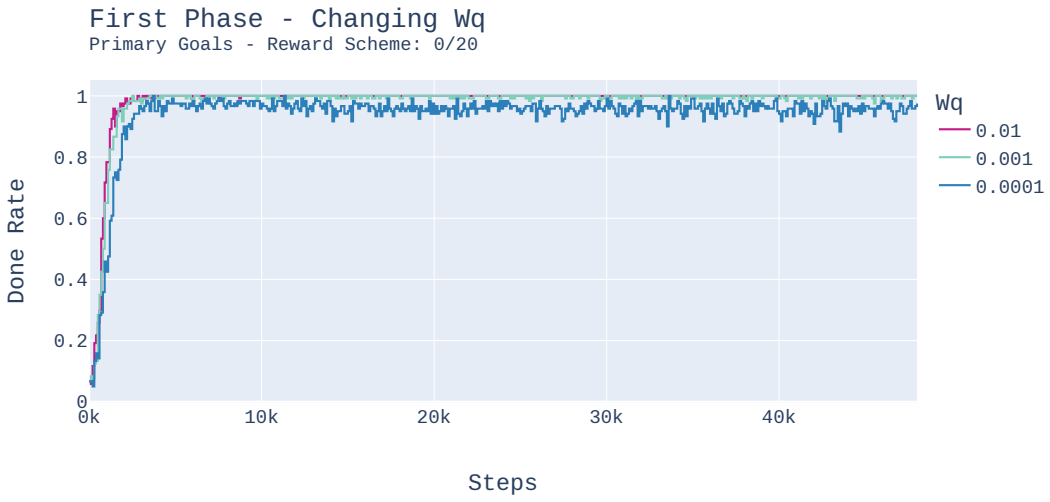


Figure 5.6: First phase of training for the FGW USF agents on the primary goals while the weight $W_Q$ is changing. This has a positive effect on the performance of the agents, as they are all able to quickly reach a done rate of 100%.

Fig. 5.7 shows the performance of the agents on the unseen tertiary goals during the first phase. We can see that the agents are able to generalize to a certain extent with a done rate of $50\% - 60\%$. In Fig. 5.8 we show the done rate of the secondary goals during the second phase of training. We can see that the agents with $W_Q = [0.01, 0.001]$ are able to reach a done rate of 100%, while the agent with $W_Q = 0.0001$ fluctuates around 95%, similarly to the first phase with the primary goals. We conclude that setting $W_Q = 0.0001$ makes the effect of $L_Q$ too small to help the agent learn to navigate to the goals. We also notice that all the agents have an initial performance that is below 50% done rate.

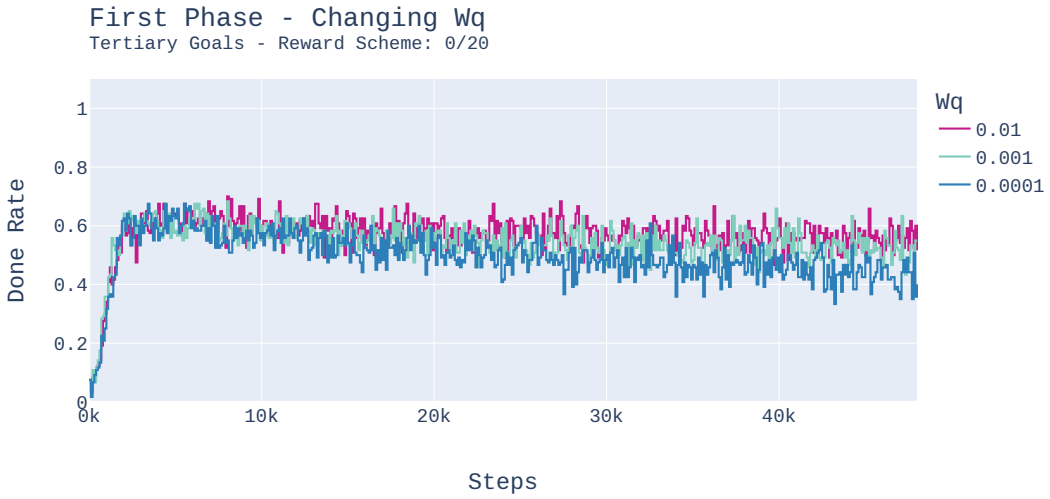


Figure 5.7: Testing the zero-shot ability to solve unseen goals by evaluating the done rate of the FGW USF agent on the tertiary goals during the first phase of training as the weight $W_Q$ is changing. All three agents can generalize to a certain extent, although the best case seems to be $W_Q = 0.01$.

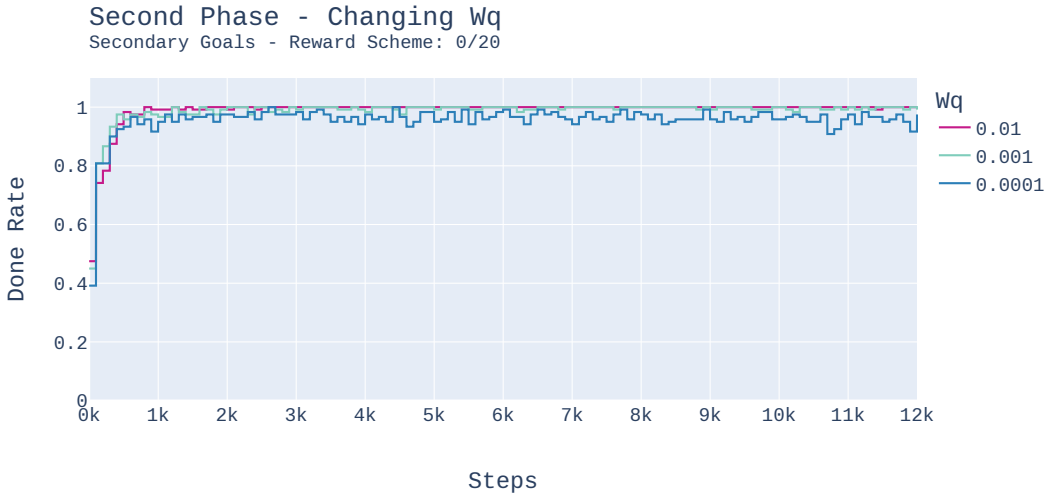


Figure 5.8: Second phase of training on the secondary goals for FGW USF agents as the weight $W_Q$ is changing. All the agents are able to learn to solve the new tasks with a high done rate. However, the agent with $W_Q = 0.0001$ is only able to reach 95%, which means the effect of $L_Q$ is too small.

## 5.4 Using GPI

In the previous section we obtained an agent that is able to correctly learn the SFs for the simplified grid world environment. We used two techniques: data augmentation and we included the loss term $L_Q$ with a small weight $W_Q$. We saw that with $W_Q \in [0.01, 0.001]$, the agents are able to correctly solve all the tasks, reaching a done rate of 100%

Figs. 5.9-5.10 show how these agents perform on the secondary goals using the GPI procedure during evaluation and action selection, respectively. In Fig. 5.9, we observe that using GPI during evaluation allows the agents to have a higher initial performance compared to not using it. In fact, when using GPI during evaluation, the done rate for all the agents is above 60% while without GPI, the best agent starts at 50% done rate. This is an improvement of 10% if we compare the worst agent with GPI to the best agent without it. If we compare the same agents then

the improvement is 20%. This is the first successful instance of our purpose of combining GPI with USFs.

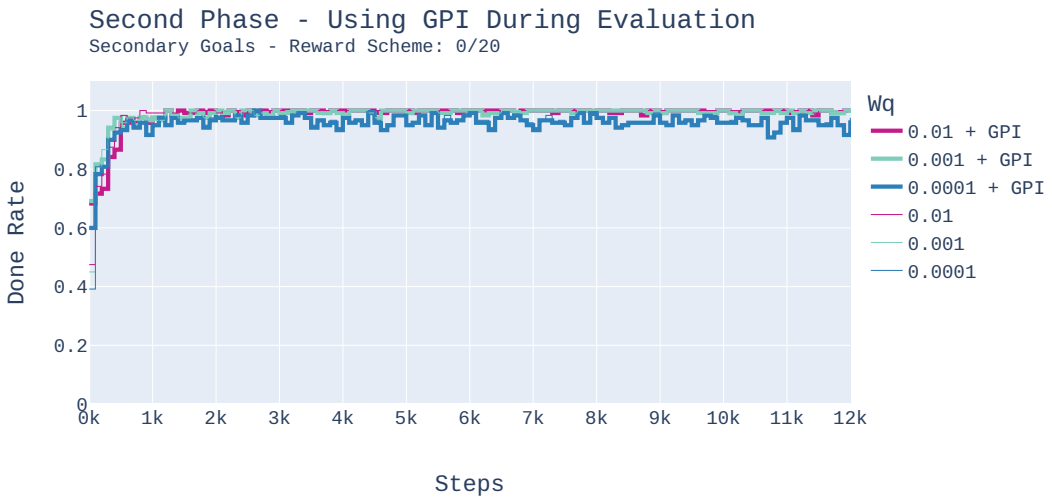


Figure 5.9: Second phase of training on the secondary goals with different values of $W_Q$. The thicker lines show the FGW USF agents that use GPI during evaluation while the thinner lines correspond to the agents that do not use it. All the agents that use GPI have a higher initial done rate than the ones that do not. This shows that using GPI is beneficial to jump-start the performance on new tasks.

In Fig. 5.10 we analyze the use of GPI to select actions during training. As we mentioned previously, in this case the transitions that the agent will encounter will be different from the case without GPI. Therefore, the performance of the agents will depend on the quality of the GPI procedure in selecting the actions. Fig. 5.10 shows that that there seems to be no significant improvement.

This is not necessarily evidence that the GPI procedure is not beneficial for action selection; rather, it is evidence that in this particular case, the task is simple and the neural network is able to generalize well. Therefore, to understand the possible benefits of using GPI during action selection, we would need to analyze a more complex problem where the neural network is not able to generalize well and the agent can select better actions to find useful transitions sooner.

The results for using GPI during both action selection and evaluation are similar

to Fig. 5.9 so they are shown in Appendix C.

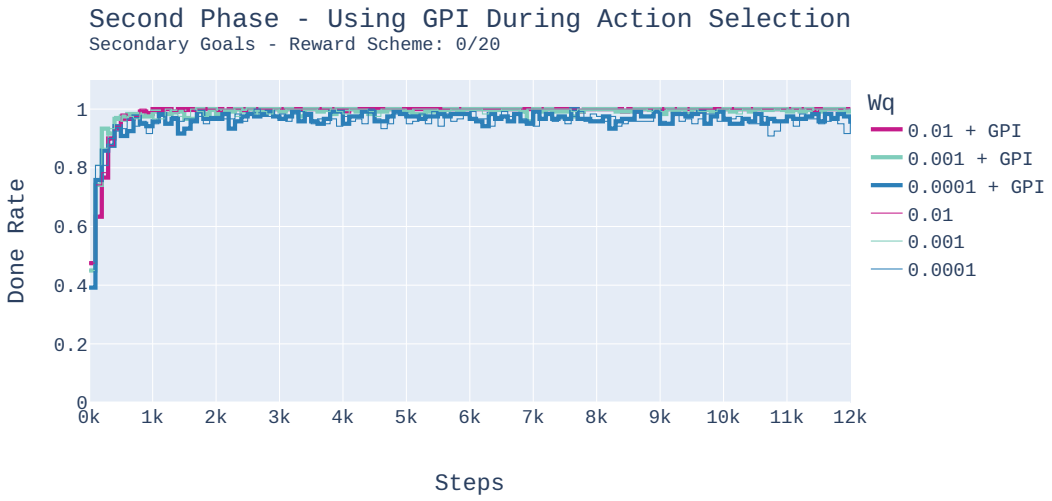


Figure 5.10: Second phase of training on the secondary goals with different values of $W_Q$. The thicker lines show the FGW USF agents that use GPI during training to select actions while the thinner lines correspond to the agents that do not use it. There seems to be no substantial benefit to using GPI in this case.

# Chapter 6

# Extending to Grid World with Rooms

We extend the results to the original grid world environment with rooms. We only analyze the FGW USF agent that we have studied in depth. The expectation is that the environment will be more complex so the GPI procedure can yield improvements both by selecting actions that yield useful transitions sooner and by allowing the agent to obtain a higher initial performance when it is evaluated on the new tasks. However, this expectation heavily relies on the assumption that the agent is able to correctly learn the SFs for this problem. Otherwise, as we have seen, the agent can be misguided and perform worse.

In Fig. 6.1 we show how the agents perfom in this environment when using the loss described in Eq. 5.1 with changing $W_Q$, with and without data augmentation.

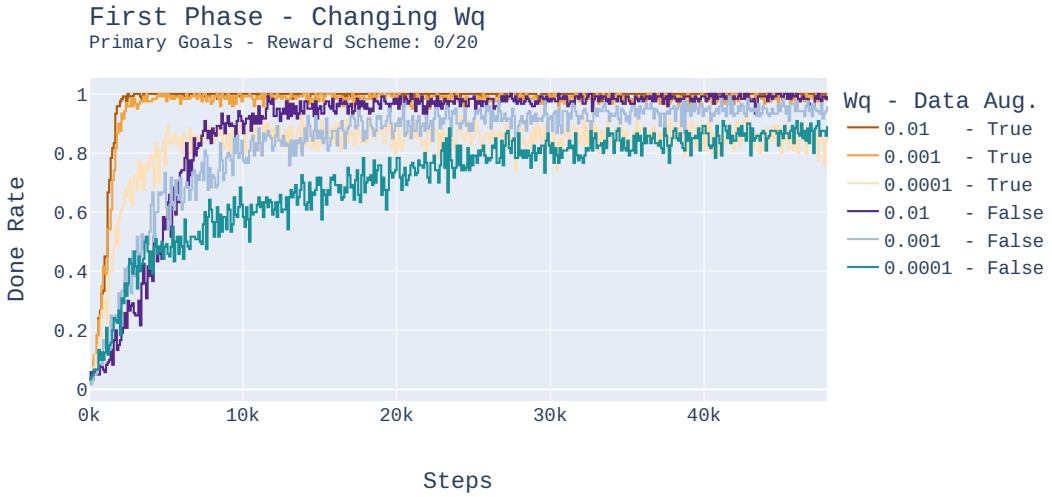


Figure 6.1: First phase of training in the full grid world environment with rooms for the FGW USF agents on the primary goals while the weight $W_Q$ is changing, with and without data augmentation. We can see that using data augmentation with $W_Q \in [0.01, 0.001]$ allows the agents to achieve the best performance. However, also $W_Q = 0.01$ without data augmentation is able to learn to solve all the tasks with 100% done rate.

As we can see, with $W_Q = [0.01, 0.001]$ and data augmentation the agents are able to learn to solve the tasks with a done rate of 100%. Without data augmentation, only the agent with $W_Q = 0.01$ is able to reach a done rate of 100%, while the agent with $W_Q = 0.001$ only arrives to 95%. On the other hand, both with and without data augmentation, setting $W_Q = 0.0001$ does not create an effect strong enough to be able to learn the tasks.

It is important also to note that data augmentation has a crucial role in the performance of the agents. In fact, the agents that use this technique during training are able to almost immediately reach a perfect performance.

In the Fig. 6.2, we overlay the results for the best agents with those obtained in Fig. 4.2 for the original agents that used $W_Q = 1.0$ and $W_\psi = 0.01$ and no data augmentation. To differentiate between the different FGW USF agents, we will call the ones with $W_Q = [0.01, 0.001]$ *improved* FGW USF.

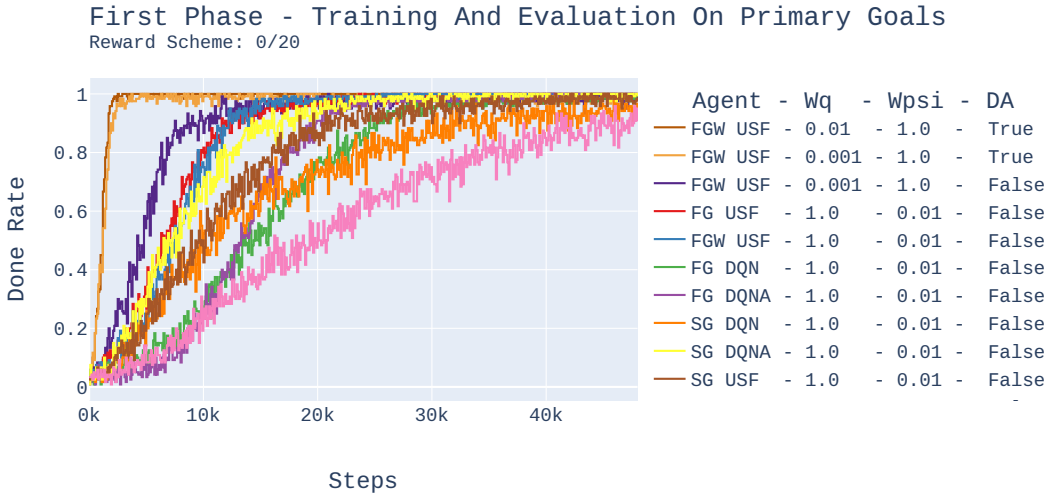


Figure 6.2: Comparing the improved FGW USF agents with the original ones in Fig. 4.2. The improved agents are able to learn to solve the tasks at a significantly faster rate than all the original agents.

We can see that the improved FGW USF agents outperform all the other agents by a considerable margin, in particular, the agents that use data augmentation learn to perfectly solve the tasks within the first $2,000$ steps. Furthermore, the improved FGW USF without data augmentation is able to reach a better performance quicker than any of the original agents. This is probably due to the fact that $L_\psi$ is more informative than $L_Q$. This indicates that there seems to be an advantage in using a framework based on Successor Features.

Now, we also analyze the performance of the improved FGW USF agents on the unseen tertiary goals during the first phase of training. We overlay the results with those of the original agents in Fig. 4.3.

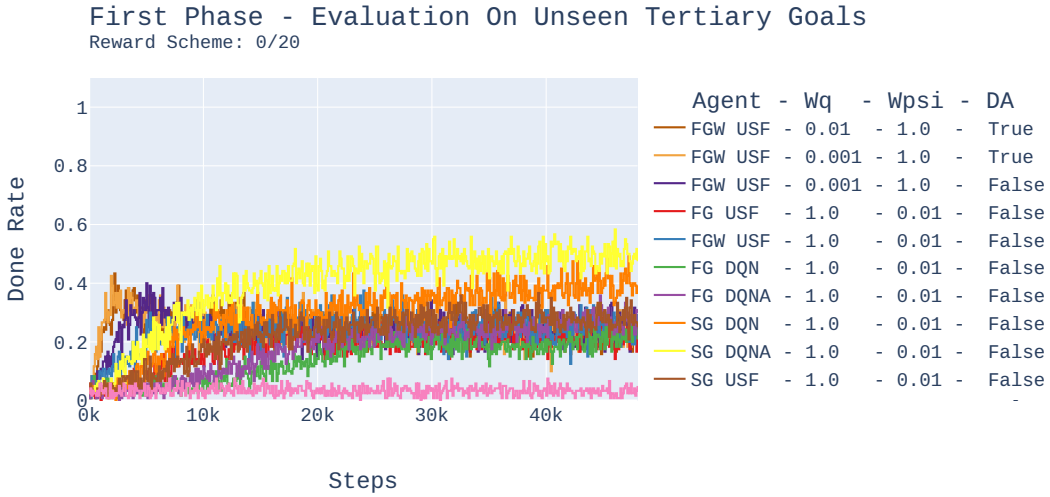


Figure 6.3: Comparing the zero-shot ability to solve unseen goals of the improved FGW USF agents and the original ones in Fig. 4.2. The improved agents are initially able to generalize better than the original ones, but their final performance is lower than the SG DQN and SG DQNA agents.

As we can see, the improved FGW USF agents initially perform better than the other original agents on the unseen tertiary goals. However, after $10k$ steps, they start to perform worse and are outperformed by the original SG DQNA and SG DQN agents. One possibility for this behavior is that while $L_\psi$ is more informative, the task of generalizing to unseen goals is more difficult since we are learning an entire vector representation rather than a single value. As a consequence, this could cause the agent to overfit on the current tasks and have worse zero-shot performance on the unseen goals.

In Fig. 6.4, we show the results for the second phase of training.

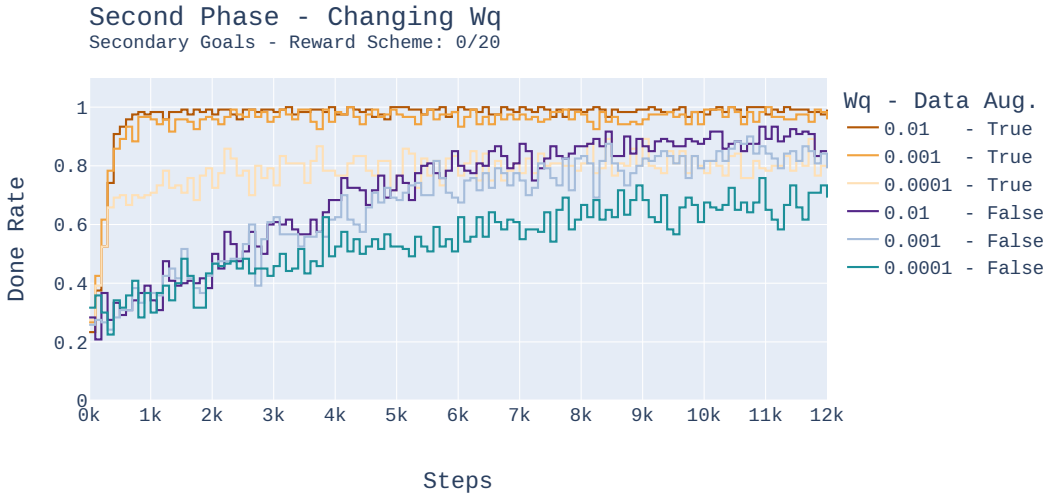


Figure 6.4: Second phase of training on the secondary goals for FGW USF agents as the weight $W_Q$ is changing with and without the use of data augmentation. Only the agents with data augmentation and $W_Q \in [0.01, 0.001]$ are able to fully solve the new tasks.

The most important aspect we notice is that without the data augmentation technique, none of the agents are able to learn to solve the new tasks with a 100% done rate. This behavior could be connected to the difficulty of learning the $\psi$ function with $L_\psi$. However, with data augmentation, the agents with $W_Q = [0.01, 0.001]$ are able to learn the environment perfectly.

We overlay the results for the aforementioned agents with the original agents in Fig. 4.4 to perform a comparison.

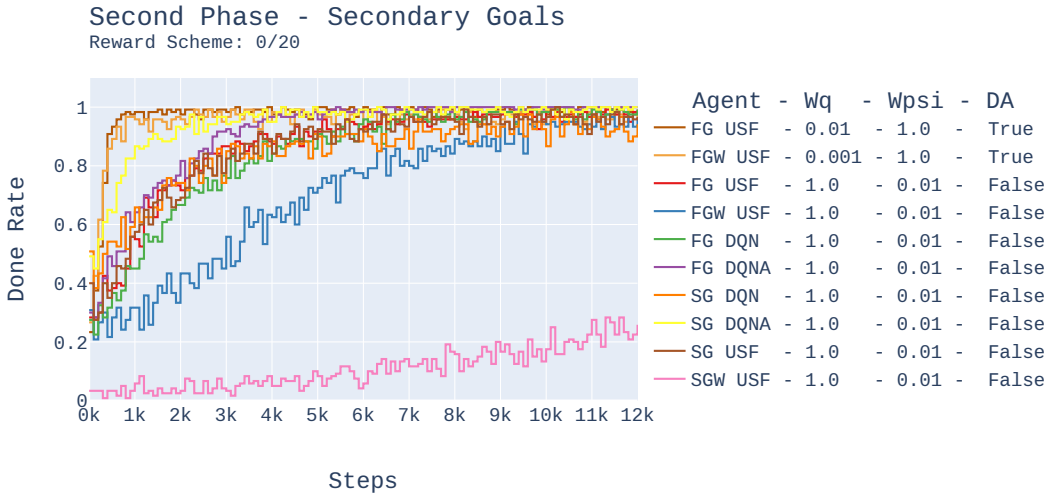


Figure 6.5: Comparing the improved FGW USF agents with the original ones in the second phase. The improved agents are able to solve the new tasks quicker than the SG DQNA, which originally performed the best.

We notice that the improved FGW USF agents perform better on the secondary goals compared to the original agents. They are able to solve the new tasks after $1,000$ steps. This highlights the effectiveness of our method to enable transfer learning and also to improve over the goal-conditioned DQN architectures. However, it is important to remember that the improved agents are using data augmentation, therefore, they are actually training on a larger batch size. In our defense, as we saw previously, increasing the batch size for the original agents had no considerable effect, which makes the comparison more fair. However, to make a fully fair comparison, we should use the same training techniques for all agents.

## 6.1 Using GPI

Finally, we show what happens when we use the GPI procedure with the improved FGW USF agents. Since we properly learned the SFs, we expect to have a similar improvement to what we saw for the simple grid world environment. The results

are shown in Figs. 6.6-6.7.

In Fig. 6.6, when using the GPI procedure for evaluating the agents, we can see that we obtain a performance boost in the beginning. The agents start with a done rate of $\approx 50\%$ while without GPI this figure sits below 30%. Unfortunately, this is the only benefit since the subsequent performance is lower and the agents with GPI are not able to solve all the tasks with a done rate of 100%. This is surprising considering that we correctly learned the SFs previously. One possibility for this behavior is that since we are using a single neural network to represent all the policies, as soon as the agent starts training on the new tasks, the SFs for the previous tasks change. As we saw earlier, if the SFs are not correctly learned, the agent can be misguided.

In Fig. 6.7, when using GPI to select actions, there seems to be no significant benefit. This may be due to a similar reason to what was discussed for the case of using GPI for evaluation. If the agent learns imperfect SFs, it may be misguided and no real improvements can be seen in this case. However, it is still able to reach a done rate of 100%.

The results for using GPI during both action selection and evaluation are similar to Fig. 6.6 so they are shown in Appendix C.

Overall, these results were surprising since we expected that correctly learning the SFs of the problem would be enough to guarantee seeing an improvement by using the GPI procedure. However, we can see that learning the SFs in this case is difficult and seems to be unstable.

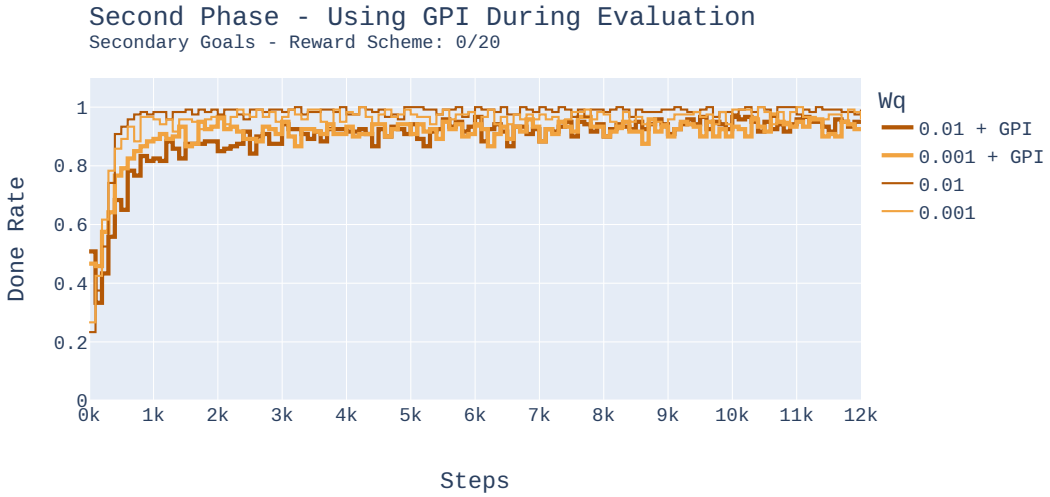


Figure 6.6: Second phase of training for the improved FGW USF agents on the secondary goals. The thicker lines show the agents that use GPI during evaluation while the thinner lines correspond to the agents that do not use it. Initially there is a boost in performance. However, in the long run, the agents using GPI are not able to solve the tasks with 100% done rate.

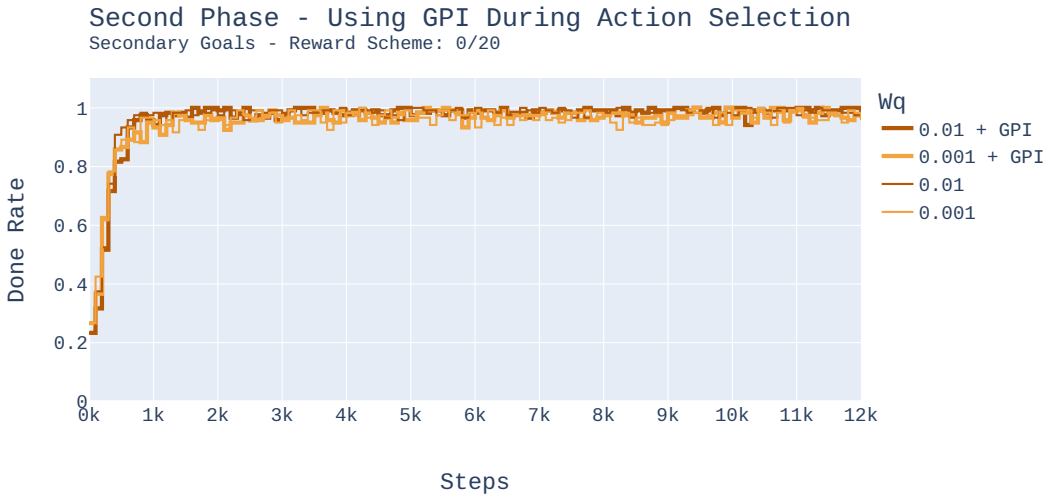


Figure 6.7: Second phase of training for the improved FGW USF agents on the secondary goals. The thicker lines show the agents that use GPI during training to select actions while the thinner lines correspond to the agents that do not use it. The performance is the same regardless of whether the agents use GPI or not.

# Chapter 7

# Conclusions & Further Work

In this work we wanted to improve upon the framework of Ma et al.[13] by combining their proposed USF architecture with the GPI procedure to further enhance the ability of an agent to learn to reach new goals in a grid world environment with rooms. The intention was to allow the agent access to the policies it learned during the first phase of training to reach the new goals quicker during the second phase of training. The agent was allowed to use the GPI procedure both during action selection and when it was evaluated on the new set of goals it had to solve.

We encountered some difficulties in replicating some of the results of the original work of Ma et al.[13]. We used various techniques to make the SG USF agents converge, such as hyper-parameter tuning and Prioritized Experience Replay[17]. This issue highlights the unstable nature of many RL frameworks and emphasizes the need for making results easy to reproduce.

In the initial attempt at extending the USF framework with GPI using the same experimental setup as Ma et al.[13], we realized that their reward scheme was incorrect for our objectives, as it encoded a preference for proximity, rather than specificity. With this reward scheme, the GPI procedure would choose a policy that drove the agent to one of the nearest old goals rather than the current one it needed

to reach. Consequently, we changed the reward scheme and obtained very different behaviors for the agents that previously were not converging.

During our second attempt at using the GPI procedure with the new reward scheme we discovered that the agent was not correctly learning the Successor Features and was being misguided when using GPI. We showed that this was fundamentally tied to the loss function that was being used, which gave more importance to the Q-function rather than the Successor Features themselves. This caused the agent to learn information mainly about the goal states rather than the entire SFs as a whole.

This realization ultimately led us to focus on correctly learning the Successor Features in a simplified grid world using only the FGW USF agent. This task proved to be difficult since the agent was not converging to the optimal solution. We tried various techniques such as using radial basis features, using pre-trained agents, changing the network architecture, using data augmentation, and changing the loss weights.

The last strategy, coupled with the data augmentation technique, led to being able to successfully learn the SFs in the simplified grid world. We showed that in this case, the FGW USF agent is able to benefit from the GPI procedure and obtain a performance boost when learning the new set of goals. This shows that our application has some promise in enhancing the transferability of past skills to solve new tasks.

When extending these results to the full grid world environment with rooms, we found that our improved FGW USF agents that did not use GPI were able to learn the primary and secondary goals at a faster rate than the original USFs and the goal-conditioned DQN baselines. One drawback is that our agent requires knowledge of both features and weights, which can be restrictive in many scenarios. However, it shows promise in using the Successor Features framework to enable

transfer learning. Finally, the use of GPI with our improved FGW USF agents did not show any improvements in this more complex environment. In fact, it lowered the performance to the point that the agents using GPI were not able to perfectly solve the new set of tasks. We theorize that this is possibly due to the difficulty of learning the SFs and the utilization of a single neural network to represent all the policies.

There are two key conclusions from this work. First, we empirically illustrated that SFs can be advantageous to increase the transferability of past skills, however, they are not always easy to learn. This is a drawback to this method. Second, and most important, we observed how unstable some RL algorithms and applications can be. This highlights the importance of reproducibility in all research work.

Finally, we discuss some possible future steps. Due to time constraints, we were unable to widen our scope to all of the agents, as we mainly focused on the FGW USF agent in the end. It would be interesting to understand if our results could be extended to the other agents as well. Additionally, to fix the issue of GPI, it could be interesting to try to separate the two networks: one for the previous policies, which is never updated during the second phase, and one for the new tasks it has to learn. In this way, the knowledge of the past policies is not affected when training on the new tasks. However, this comes at the expense of doubling the memory requirements, so it may have limited practical interest.

# Acknowledgements

I would like to extended my deepest thanks to the entire RobotLearn team, without whom this endeavor would not have been possible. They made me feel welcome in their research team from the first day I arrived. In particular, I would like to thank Chris Reinke whose invaluable supervision allowed me to dive deeper in the complicated, yet beautiful field of Reinforcement Learning and kept me motivated in the face of the difficulties we faced.

I would also like to thank Gaetan Lepage, who patiently answered many of my questions, debugged code with me, and welcomed me to the world of Vim - blazingly fast!

I would also like to thank my supervisor Antonio Celani for doing a wonderful job of teaching the class of Reinforcement Learning which started my passion for this field. I would also like to thank him for his guide and insights in directing the present work.

I would like to thank my girlfriend, who always provided love and support as she listened to all my complaints, fears, and joyous moments throughout my degree.

Finally, I would like to thank my mother, father, and brother, who have always supported my endeavors and without whom none of this would have been remotely possible from the start.

# Bibliography


[1] André Barreto et al. *Successor Features for Transfer in Reinforcement Learning*. Apr. 12, 2018. arXiv: 1606.05312[cs]. URL: http://arxiv.org/abs/1606.05312.

[2] André Barreto et al. *Transfer in Deep Reinforcement Learning Using Successor Features and Generalised Policy Improvement*. Jan. 30, 2019. arXiv: 1901.10964[cs]. URL: http://arxiv.org/abs/1901.10964.

[3] Diana Borsa et al. *Universal Successor Features Approximators*. Dec. 18, 2018. arXiv: 1812.07626[cs,stat]. URL: http://arxiv.org/abs/1812.07626.

[4] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610.7930 (Oct. 2022), pp. 47–53. ISSN: 1476-4687. DOI: 10.1038/s41586-022-05172-4. URL: https://doi.org/10.1038/s41586-022-05172-4.

[5] Hado van Hasselt et al. *Deep Reinforcement Learning and the Deadly Triad*. Dec. 6, 2018. arXiv: 1812.02648[cs]. URL: http://arxiv.org/abs/1812.02648.

[6] INRIA. *RobotLearn*. URL: https://team.inria.fr/robotlearn/.

[7] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI:

10.1038/s41586-021-03819-2. URL: https://doi.org/10.1038/s41586-021-03819-2.

[8] B. Ravi Kiran et al. *Deep Reinforcement Learning for Autonomous Driving: A Survey*. Jan. 23, 2021. arXiv: 2002.00444[cs]. URL: http://arxiv.org/abs/2002.00444.

[9] Jens Kober, J. Bagnell, and Jan Peters. "Reinforcement Learning in Robotics: A Survey". In: *The International Journal of Robotics Research* 32 (Sept. 2013), pp. 1238–1274. DOI: 10.1177/0278364913495721.

[10] Tejas D. Kulkarni et al. *Deep Successor Reinforcement Learning*. June 8, 2016. arXiv: 1606.02396[cs,stat]. URL: http://arxiv.org/abs/1606.02396.

[11] Nathan Lambert. *Illustrating Reinforcement Learning from Human Feedback (RLHF)*. URL: https://huggingface.co/blog/rlhf.

[12] Long-Ji Lin. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 293–321. ISSN: 0885-6125. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699.

[13] Chen Ma et al. *Universal Successor Features for Transfer Reinforcement Learning*. Jan. 4, 2020. arXiv: 2001.04025[cs,stat]. URL: http://arxiv.org/abs/2001.04025.

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602.

[15] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[16] Chris Reinke and Xavier Alameda-Pineda. *Successor Feature Representations*. Dec. 16, 2022. arXiv: 2110.15701[cs]. URL: http://arxiv.org/abs/2110.15701.

[17] Tom Schaul et al. *Prioritized Experience Replay*. Feb. 25, 2016. arXiv: 1511.05952[cs]. URL: http://arxiv.org/abs/1511.05952.

[18] Tom Schaul et al. "Universal Value Function Approximators". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1312–1320. URL: https://proceedings.mlr.press/v37/schaul15.html.

[19] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Dec. 5, 2017. arXiv: 1712.01815[cs]. URL: http://arxiv.org/abs/1712.01815.

[20] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[21] SPRING. *Socially Pertinent Robots in Gerontological Healthcare - SPRING*. URL: https://spring-h2020.eu/.

[22] Richard Sutton et al. "Horde : A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction Categories and Subject Descriptors". In: vol. 2. Jan. 2011.

[23] Christopher Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[24] Zhuangdi Zhu et al. *Transfer Learning in Deep Reinforcement Learning: A Survey*. May 16, 2022. arXiv: 2009.07888[cs,stat]. URL: http://arxiv.org/abs/2009.07888.

# Appendix A

# Additional Architecture Details

This appendix is devoted to discussing the architectures used in more detail.

## A.1 Base Architectures

The base architecture for all the models is shown in Fig. A.1. $\theta_\psi^{(1,2,3)}$ and $\theta_w$ are all fully connected feedforward networks. $\theta_\psi^{(1)}$, $\theta_\psi^{(2)}$, and $\theta_\psi^{(3)}$ contain one hidden layer of size 81, 64, and 256, respectively. The output of $\theta_\psi^{(2)}$ was chosen to have the same dimensionality as $\phi(s)$ which is 81. $\theta_w$ consists of two hidden layers of size 64. In all cases, we use ReLU activations.

As mentioned in section 2.1.4, all the USF agents can be derived from Fig. A.1 by eliminating a sub-branch from Fig. A.1. For example, the architecture for the FG USF is the same as Fig. A.1 without $\theta_\psi^{(1)}$ and it takes as input $\phi(s)$ directly.

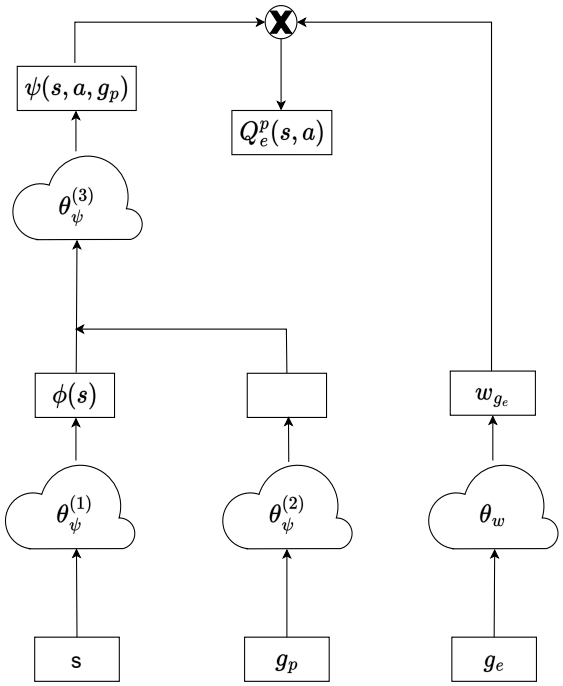


Figure A.1: The modified USF architecture in order to combine GPI with the USF framework. All USF agents use two goals as input: $g_p$ which selects the policy that learned to navigate to $g_p$ by generating the SFs $\boldsymbol{\psi}(s, a, g_p)$ and $g_e$ which represents the current goal the agent is trying to reach.

The goal-conditioned DQN architectures correspond to Fig. A.1 without $\theta_w$, as shown in Fig. A.2. $\theta^{(1)}$, $\theta^{(2)}$, and $\theta^{(3)}$ are fully connected feedforward networks that contain one hidden layer of size 81, 64, and 256, respectively. The FG DQN use $\phi(s)$ as input instead of $s$.

The augmented goal-conditioned DQN agents contain more layers at the head. In particular, $\theta^{(3)}$ contains three hidden layers of size 256, 352, and 64, respectively. The number of learnable parameters in the networks are shown in Table A.1.

Table A.1: The number of learnable parameters in the architectures. As we can see, by augmenting the DQN models, we make their representation power comparable to that of the USF.

| Model | USF | DQN | Augmented DQN |
|---|---|---|---|
| Learnable Parameters | $145,859$ | $54,531$ | $166,403$ |

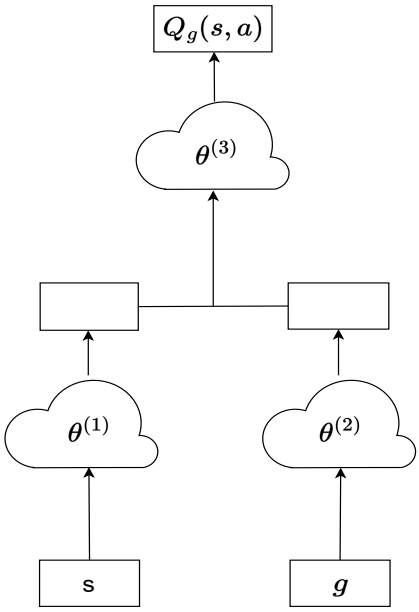


Figure A.2: The base architecture used for all goal-conditioned DQN agents.

## A.2 Extra Architectures

The architecture used based on Kulkarni et al.[10] is the same as Fig. A.1 except that we use a different branch for each of the four actions to represent the Successor Features $\psi(s, a, g)$. This means that there are 4 copies of $\theta_\psi^{(3)}$. Each of these is used in a dot product with $\mathbf{w_g}$ to calculate the Q-function for the corresponding action. Furthermore, we added a network $\theta_\psi^{(4)}$ of two hidden layers of size 64 between $\boldsymbol{\phi}(s)$ and the concatenation with the output of $\theta_\psi^{(2)}$.

The architecture used by taking inspiration from Borsa et al.[3] uses deeper versions of $\theta_\psi^{(2,3)}$. In particular, $\theta_\psi^{(2)}$ is composed of two hidden layers of size 64 and $\theta_\psi^{(3)}$ is composed of two hidden layers of size 256 and 512, respectively. This network also uses the same $\theta_\psi^{(4)}$ described above.

# Appendix B

# Parameter Search - SG and SGW USF

Below we show the results of all the attempts to make the SG USF and SGW USF agents converge.

We tried to:

- Tune the learning rate across various orders of magnitude.
- Increase the batch size.
- Increase the number of passes the neural network does on a single batch.
- Tune the value of $W_\psi$.
- Use Prioritized Experience Replay[17].

None of these attempts yielded successful results, as we will see presently. One possibility is that the problem is very unstable and since Ma et al.[13] did not fully specify the architecture and parameters they used, the results are difficult to reproduce.

## B.1 Using Different Learning Rates

Keeping all the other hyper-parameters the same, we changed the learning rate to observe any possible improvements. To speed up the measurements, we limited our experiments to the first $10,000$ steps, as this would already give a good indication if the agent was converging properly or not. The values we used for the learning rate were $5e^{-5}$, $5e^{-4}$, and $5e^{-3}$.

The results are shown in Figs. B.1-B.2. As we can see, none of the results show any improvements towards helping the agents converge.

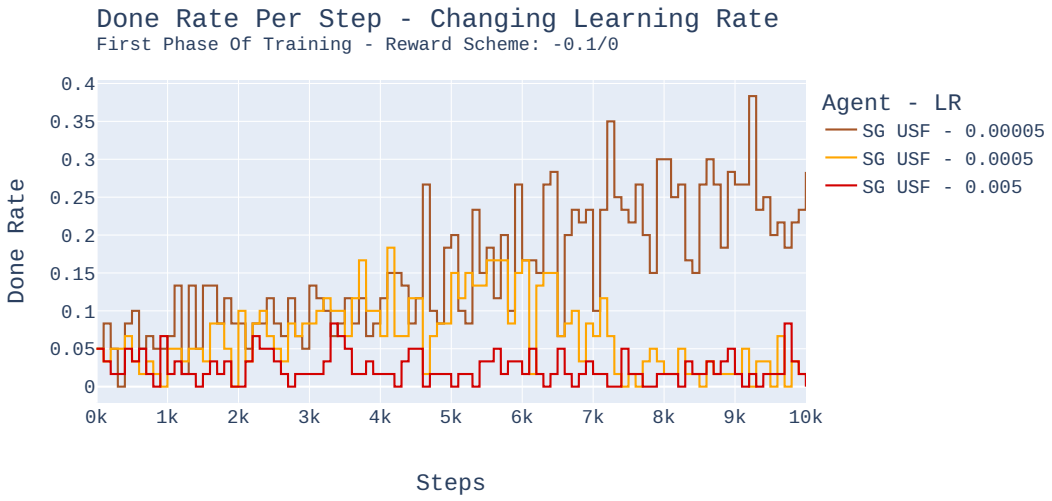


Figure B.1: Done rate per step on the primary goals for the SG USF agent as the learning rate is changed. A sight improvement is shown for learning rate $5e^{-5}$, however it still does not converge.

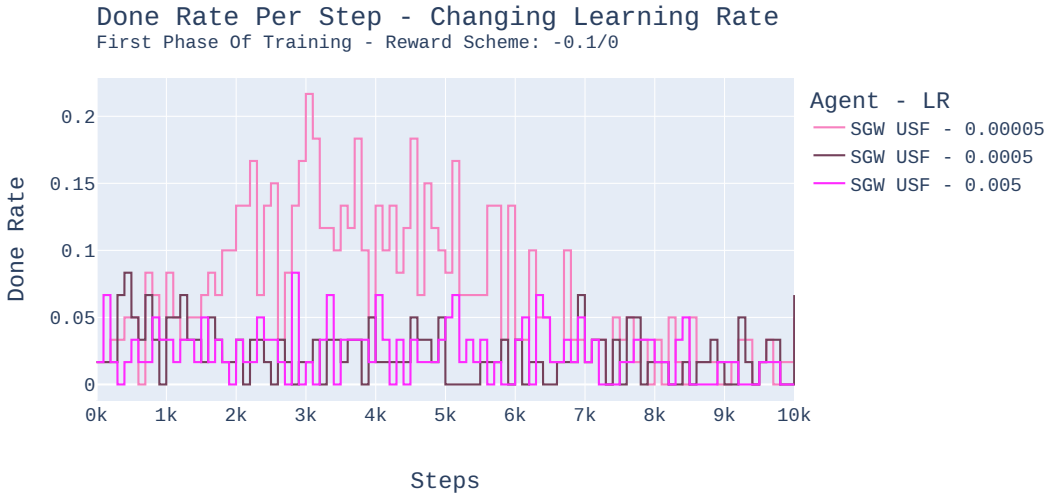


Figure B.2: Done rate per step on the primary goals for the SGW USF agent as the learning rate is changed.

## B.2 Using Different Batch Sizes

Keeping the learning rate fixed at $5e^{-4}$ as in the original paper, we increased the batch size in the hope that this would allow the agent to learn better. The results are shown in Figs. B.3-B.4. We can observe that increasing the batch size has no effect on the convergence of the agents.

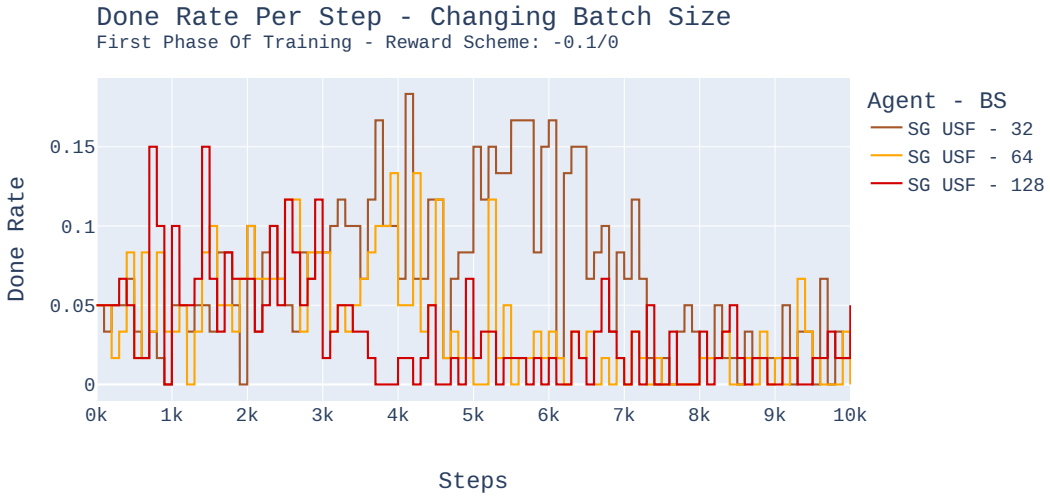


Figure B.3: Done rate per step on the primary goals for the SG USF as the batch size is increased.

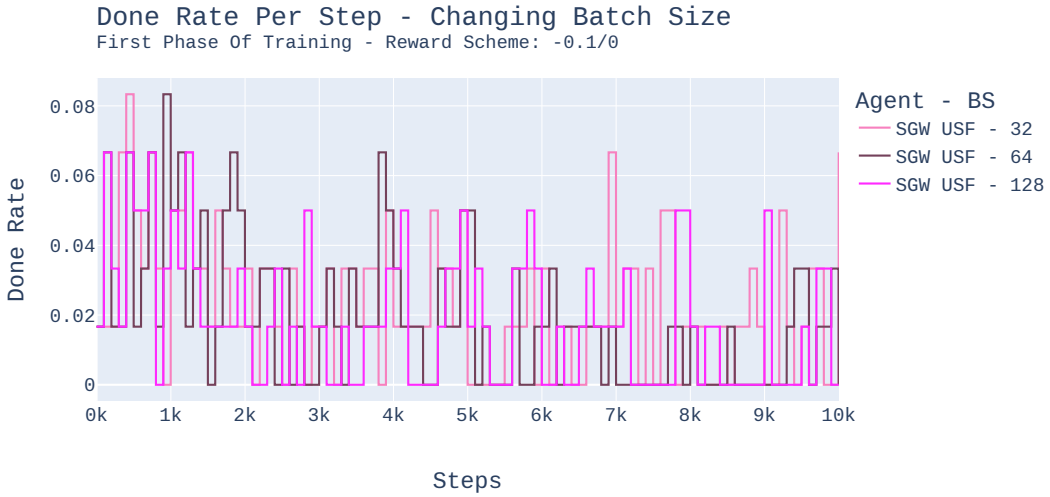


Figure B.4: Done rate per step on the primary goals for the SGW USF as the batch size is increased.

## B.3 Using Different $W_\psi$ Values

Fixing the learning rate and batch size the same as in the original paper, we tried to tune the value of the weight $W_\psi$.

Since the Ma et al.[13] did not justify their choice of $W_\psi$, we tried to increase this value to see if this could have some positive effect on the convergence of the agent. We thought that perhaps the value for this parameter was misspecified in the original work. The results are shown in Figs. B.5-B.6. None of the results yielded significant improvements.

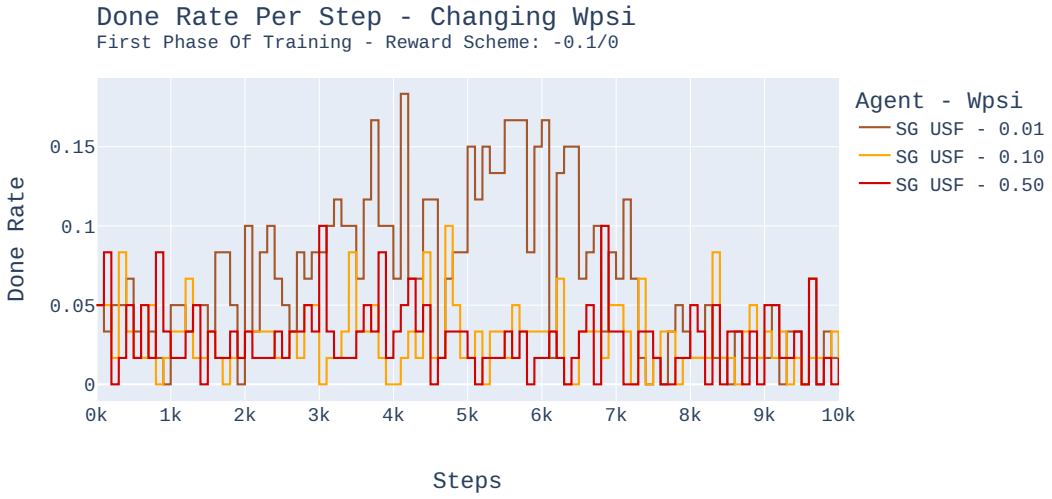


Figure B.5: Done rate per step on the primary goals for the SG USF agent as the value of $W_\psi$ is increased.

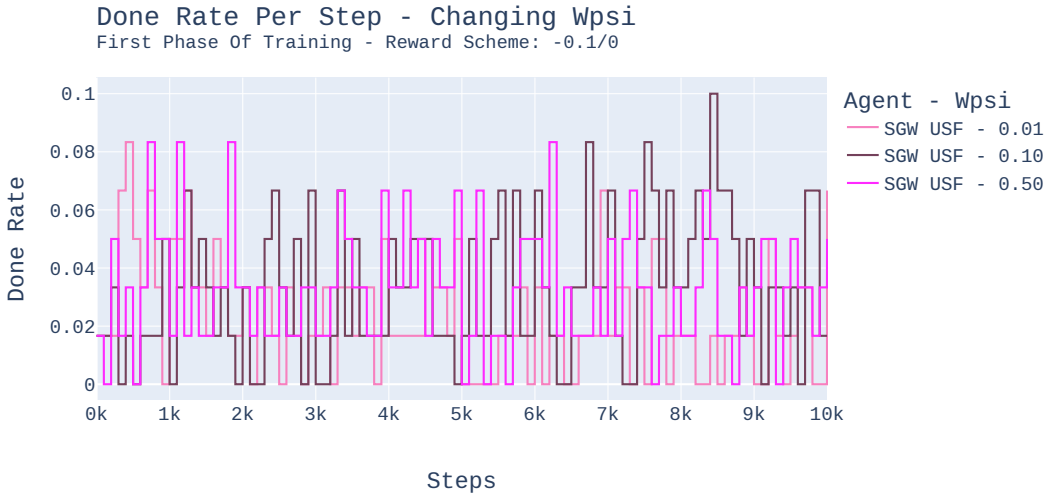


Figure B.6: Done rate per step on the primary goals for the SGW USF agent as the value of $W_\psi$ is increased.

## B.4 Using Different Iterations of Learning

Whenever we sample the batch size, we can choose to do $n$ passes which update the network using that same batch. By default, this parameter is set to 1, however we wanted to experiment whether increasing it could have some further benefits. The results are shown in Figs. B.7-B.8. None of the results yielded significant improvements.

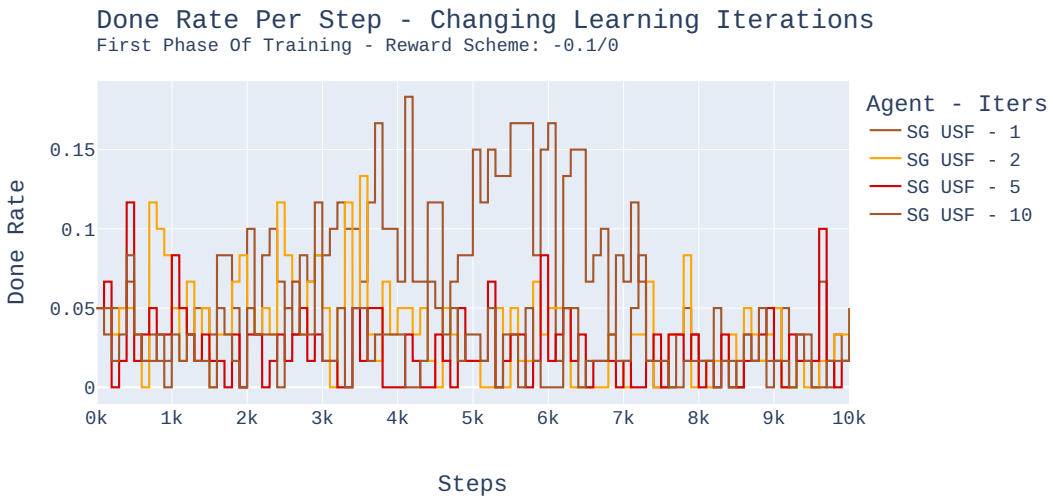


Figure B.7: Done rate per step on the primary goals for the SG USF agent as the number of passes per batch increases.

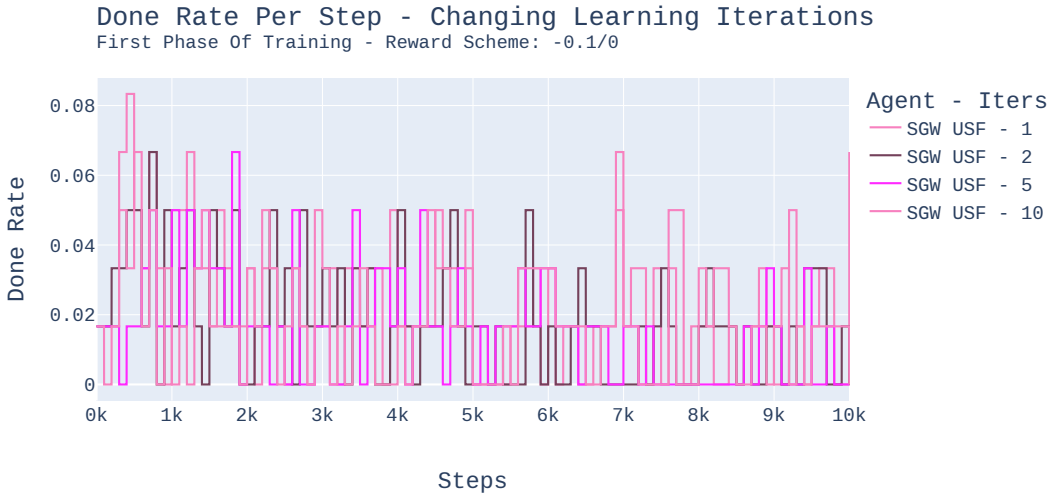


Figure B.8: Done rate per step on the primary goals for the SGW USF agent as the number of passes per batch increases.

## B.5 Using Prioritized Experience Replay

Prioritized Experience Replay[17] (PER) is a sampling method that can be used with methods that rely on memory buffers such as DQN[14]. It enables sampling random transitions according to a distribution that gives more importance to transitions that are more impactful according to some measure. In our case we used the decrease in the loss as a measure of the impact of the transition.

In particular, PER uses two hyper-parameters $\alpha$ and $\beta$. $\alpha$ dictates how much priority we should give to impactful transitions. Setting it too high will cause the distribution to be very skewed causing the agent to keep revisiting the same subset of transitions. $\beta$ dictates how much we want to correct the skewed behavior. It is recommended that we start with a low value to be able to see more impactful transitions at the beginning of the learning and then slowly anneal $\beta$ to 1 towards the end of training.

We varied $\alpha$ and $\beta$ in the range $[0.0, 0.5, 1.0] \times [0.0, 0.5, 1.0]$. The results are

shown in Figs. B.9-B.10. As we can observe, the use of PER does not seem to have a significant impact on improving the convergence of the agent.

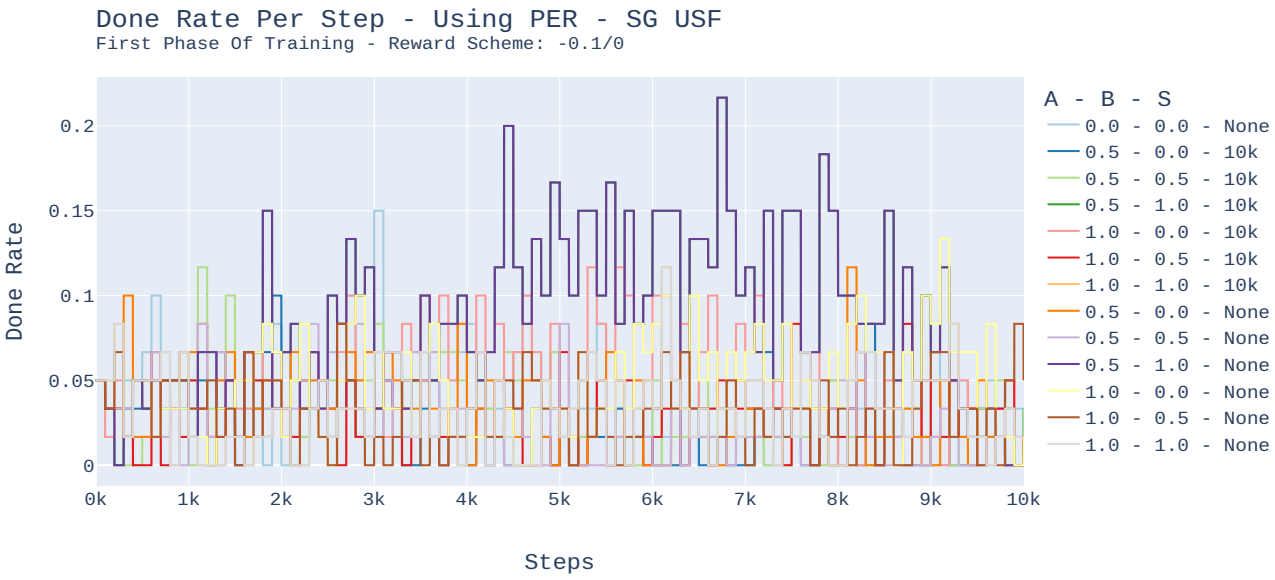


Figure B.9: Done rate per step on the primary goals for the SG USF as the parameters for PER are changed.

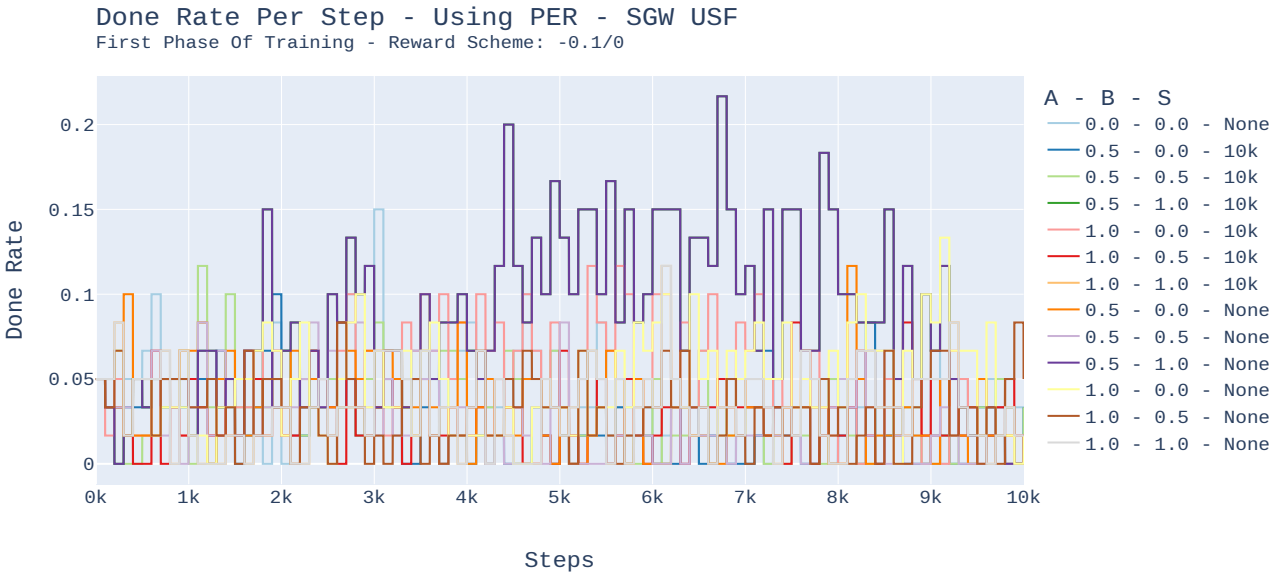


Figure B.10: Done rate per step on the primary goals for the SGW USF as the parameters for PER are changed.

# Appendix C

# Using GPI in Both Cases

We show the additional measures of using the GPI procedure both during action selection and during evaluation of the agents. In all the cases, the results are very similar to using the GPI procedure only for evaluation because this is the dominating effect in the performance of the agent. In fact, if the GPI procedure picks the wrong policies during evaluation, the agent will perform poorly, even if the actions it is using to learn are helpful.

Fig. C.1 shows the results in the case of the original reward scheme of $-0.1$ per time step and 0 at the goal position. The performance is low in this case because the reward scheme is not helpful for the GPI procedure.

Fig. C.2 shows the results in the case of the modified reward scheme of 0 per time step and 20 at the goal position. We can see an improvement with respect to Fig. C.1, however, there is still a problem with GPI. Fig. C.3 shows the results in the case of the FGW USF agent in the simplified grid world with changing $W_Q$. This is the first successful instance of using GPI, as evidenced by the higher initial done rates of the agents that use it. Finally, Fig. C.4 shows the results in the case of the improved FGW USF agents in the full grid world environment with rooms. The agents using GPI have a higher initial done rate, however, they are unable to

reach a done rate of 100%.

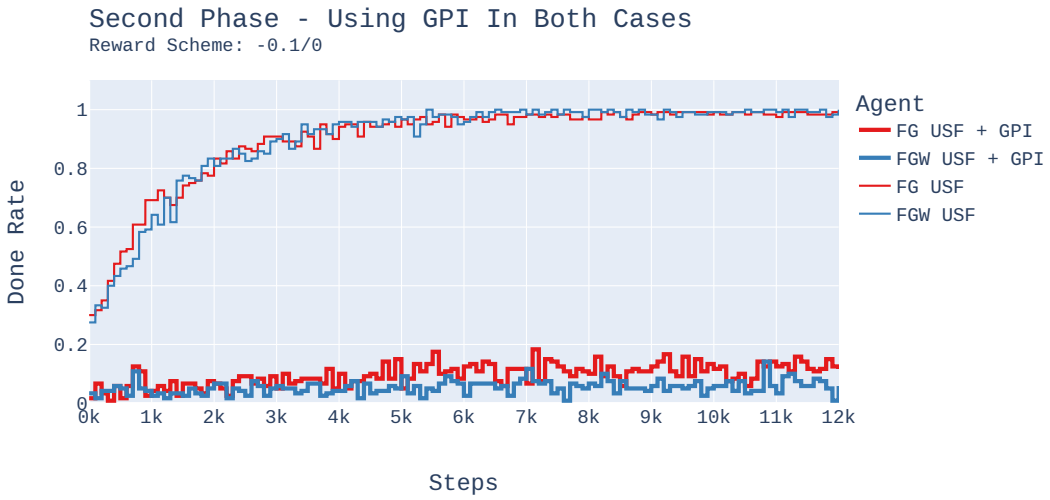


Figure C.1: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during training to select actions and during evaluation on the secondary goals. The thinner lines correspond to the agents that do not use GPI. The performance is worse than Fig. 3.4 because the agents are choosing the wrong policies during evaluation and unhelpful actions during training.

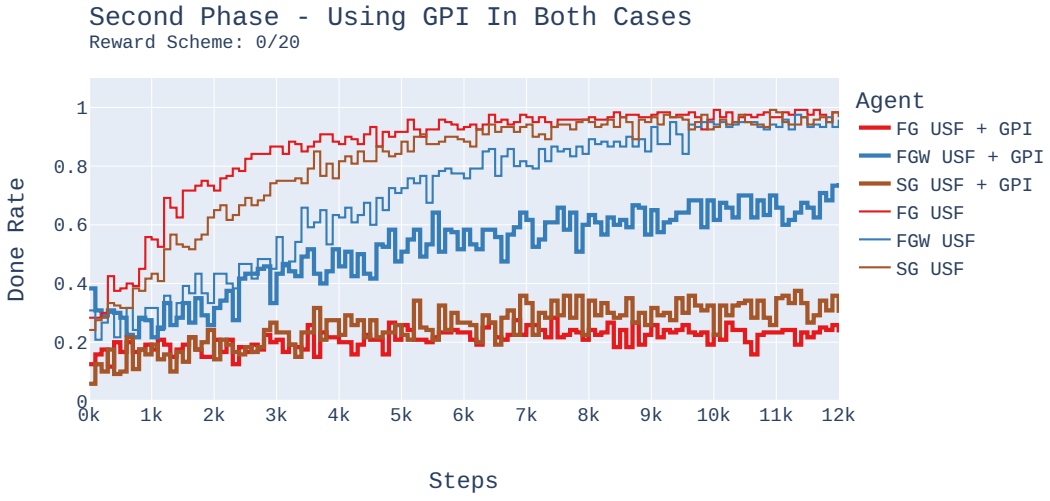


Figure C.2: Second phase of training on the secondary goals. The thicker lines show the agents that use GPI during training to select actions and during evaluation on the secondary goals. The thinner lines correspond to the agents that do not use GPI. The dominating effect is that the GPI procedure misleads the agent during evaluation.

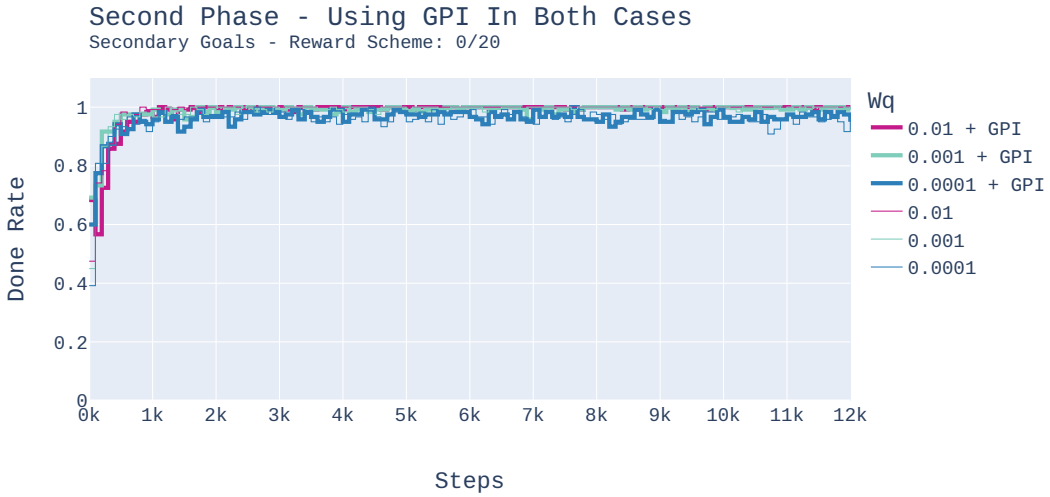


Figure C.3: Second phase of training on the secondary goals with different values of $W_Q$. The thicker lines show the FGW USF agents that use GPI both during training to select actions and during evaluation while the thinner lines correspond to the agents that do not use it. All of the agents that use GPI have a higher initial done rate than the ones that do not. This shows that using GPI during evaluation and action selection is beneficial to jump-start the performance on new tasks.

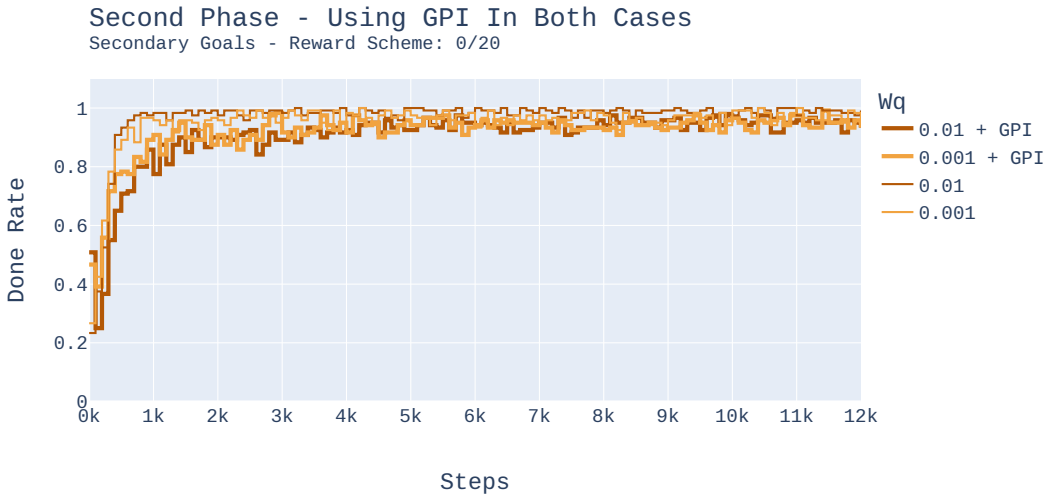


Figure C.4: Second phase of training for the improved FGW USF agents on the secondary goals. The thicker lines show the agents that use GPI during both during training to select actions and during evaluation while the thinner lines correspond to the agents that do not use it. The performance of the agents that use GPI is lower because it selects the wrong policies during evaluation, which is the dominating effect.

# Appendix D

# New Reward Scheme - Additional Measures

We show the results for the agents with the new reward scheme of 0 per time step and 20 at the goal position using $\gamma = 0.99$. The results are shown in Figs. D.1-D.2. As we can see, there are no significant improvements for the SG USF and SGW USF. Furthermore, the performance of FGW USF decreases. This is an indicator that learning is unstable since we saw that setting $\gamma = 0.90$ in Fig. 4.2 improves the performance of all the agents.

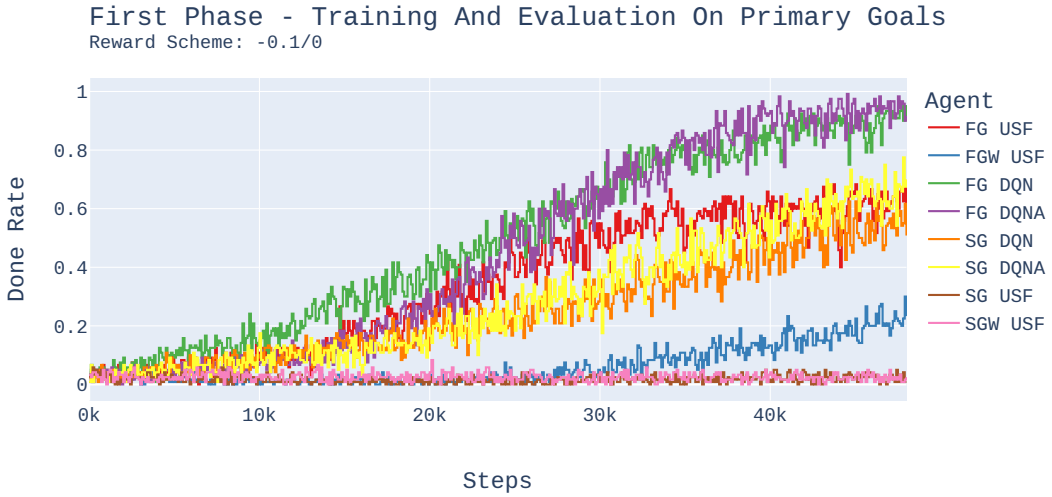


Figure D.1: First phase of training on the primary goals using the new reward scheme and $\gamma = 0.99$. The performance is lower compared to Fig. 4.2 with $\gamma = 0.90$, which is evidence that the learning is sensitive to the choice of $\gamma$.

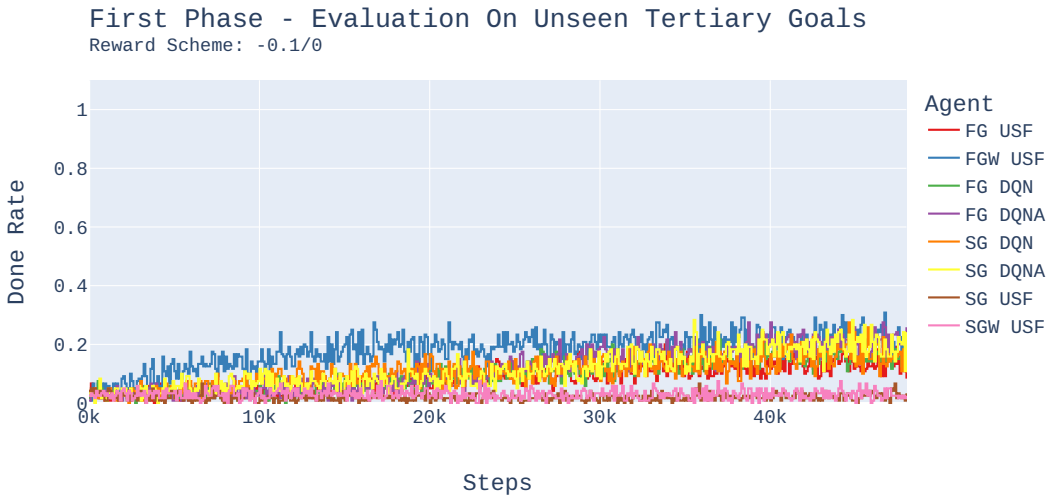


Figure D.2: Testing the zero-shot ability to solve unseen goals by evaluating the done rate of the tertiary goals during the first phase of training using the new reward scheme with $\gamma = 0.99$. The generalization capability for all the agents is lower than Fig. 4.3 with $\gamma = 0.90$.

# Appendix E

# Using RBF Features and Pre-trained Agents

## E.1 Using Gaussian Radial Basis Features

One idea to improve learning the SFs in the simplified grid world environment was to use a Gaussian radial basis function (RBF) to generate the features rather than a one-hot encoding. Our intuition was that the one-hot encoding was too discontinuous to encode a sense of proximity/neighborhood, which would be useful to the agent for learning. In fact, any two vertically adjacent cells in the grid world are 9 indexes apart in the one-hot representation.

In the RBF representation we use the squared euclidean distance of the agent's position $A_{ij} = (A_i, A_j)$ from each cell position $C_{ij} = (C_i, C_j)$ in the grid world, as shown in Eq. E.1 Then we compute the RBF features by using Eq. E.2.

$$d_{ij}^2 = (\mathrm{A}_i - \mathrm{C}_i)^2 + (\mathrm{A}_j - \mathrm{C}_j)^2 \tag{E.1}$$

$$\mathrm{RBF}_{ij} = e^{-d_{ij}^2}. \tag{E.2}$$

Fig. E.1 shows the results obtained by using the RBF features. We can see that there seems to be some form of unlearning when using this technique. This is a form of instability for agents that learn with bootstrapping techniques.

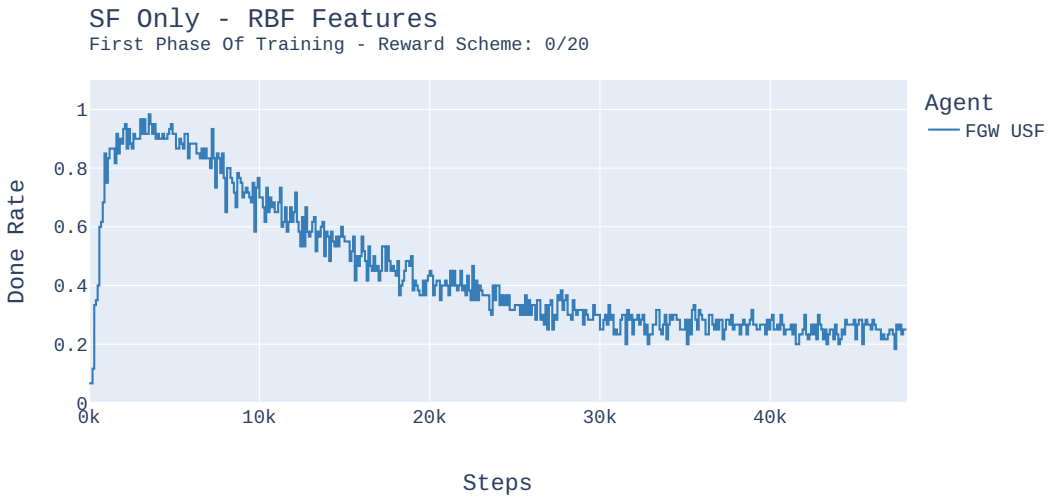


Figure E.1: First phase of training on the primary goals for the FGW USF agent in a simplified grid world using the RBF features. We see there is a form of unlearning which can happen with bootstrapping techniques.

## E.2 Using a Pre-trained Agent

Another idea to improve learning the SFs is to use a pre-trained agent in two contexts:

1. As an optimal target network to lead ourselves back to a supervised learning scenario to understand the representation power of our network.

2. As an optimal agent to select actions for the agent that is learning. This is done to increase the useful actions that the learning agent experiences.

In the first case, at each training iteration, the policy network uses the external optimal target network to obtain the targets in the loss. We never update the

target network since it is already considered optimal. This means that this becomes a supervised learning problem where the policy network is learning to imitate the optimal behavior of the target network. In this way, we hope to understand better if the architecture we are using is capable of representing an optimal behavior.

As the optimal target network, we use a FGW USF agent trained on the grid world environment using $L = L_Q + 0.01 \cdot L_\psi$, which we saw was able to learn to solve the tasks with 100% done rate. The agent that is learning - i.e. the policy network - uses the loss $L = L_\psi$.

The results are shown in Fig. E.2 which shows that the agent is not able to learn the optimal behavior. This is a possible indication that the architecture is not powerful enough to learn the solution to this problem. It is worth recalling that we are trying to learn the behavior of an agent that used a different loss, which may be a problem. However, ideally, the Successor Features should be able to represent the same optimal behavior as the target agent.

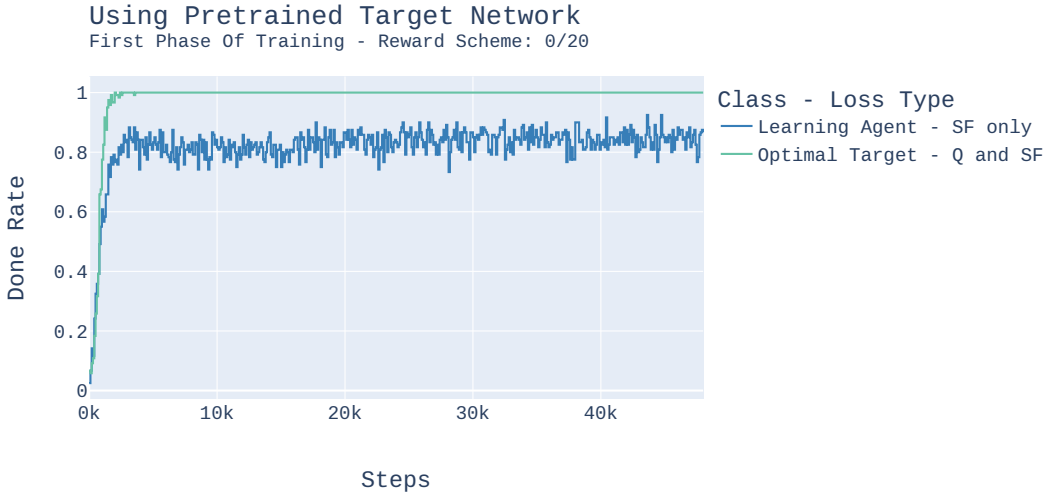


Figure E.2: Done rate per step on the primary goals for an FGW USF agent that learns to imitate the behavior of an optimal target network. The learning agent is not able to learn the optimal behavior with the loss based on $L_\psi$ only.

In the second case we decided to use a pre-trained optimal agent to choose the

actions for our learning agent. Our idea was that in this way, the quality of the transitions that the agent uses to learn are higher since they are more likely to contain useful information about the goal by finding it with less exploration. We hoped that in this way, the agent would be able to learn better. We used the same pre-trained agent as above for this scenario.

The results are shown in Fig. E.3 where we can observe that this attempt at improving the learning seems to actually lower the performance in the long run. This could be explained with the fact that with an optimal action selector, the agent continually observes the same transitions and thus learns less.

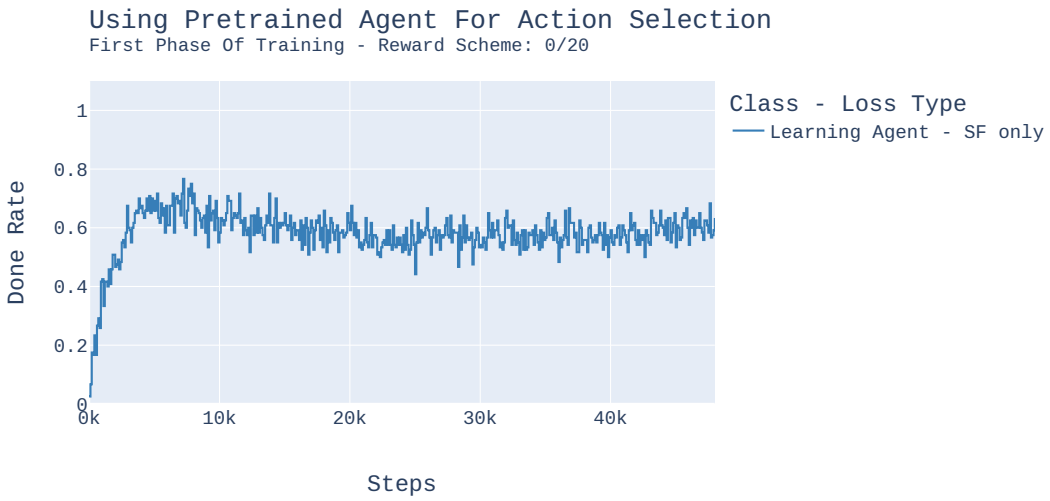


Figure E.3: Done rate per step on primary goals for an FGW USF agent using a pre-trained optimal agent to select the actions for the learning agent. This technique is unsuccessful since the agent is only able to achieve a done rate of 60%, which means it is not able to learn to reach the all the primary goals.

# Appendix F

# Parameter Search - FGW USF

As part of our experiments we also did a grid search for the best hyper-parameters to use with the data augmentation technique. The hope was that by tuning some hyper-parameters we could make the agent obtain a perfect done rate.

We first analyzed the learning rate and the optimizer we used. The learning rate was changed through 4 orders of magnitude and we used both Adam and Stochastic Gradient Descent (SGD) as optimizers, with default parameters.

In Fig. F.1, we see that SGD performs considerably worse than Adam. Therefore we keep using Adam in all experiments. Furthermore, we can see that using Adam with learning rates $5e^{-5}$, $5e^{-4}$, and $5e^{-3}$, we obtain similar results. The best performance is obtained with $5e^{-4}$ which is the learning rate we always use.

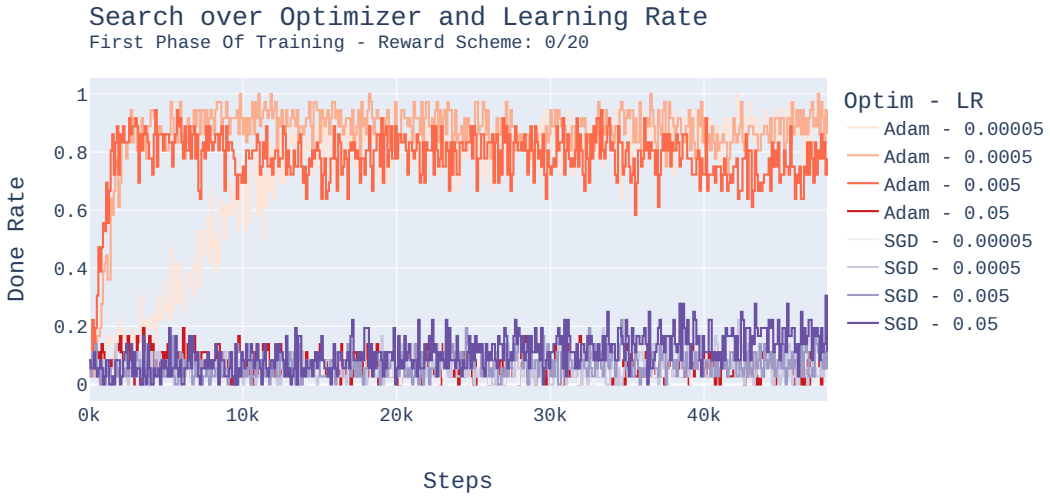


Figure F.1: Tuning of learning rate and optimizer used. We use Adam and SGD with default parameters. The best performance is obtained when using Adam and a learning rate of $5e^{-4}$.

Having determined the optimizer and learning rate to use, we now do a grid search over the batch size and the learning iterations. The batch size was changed through 32, 64, and 128. However, since we are using data augmentation for all of these experiments, the real size is multiplied by 12. The learning iterations were varied from 1, 5, and 10. The results are shown in Fig. F.2. We can observe that none of the hyper-parameters combinations cause a signficant improvement.

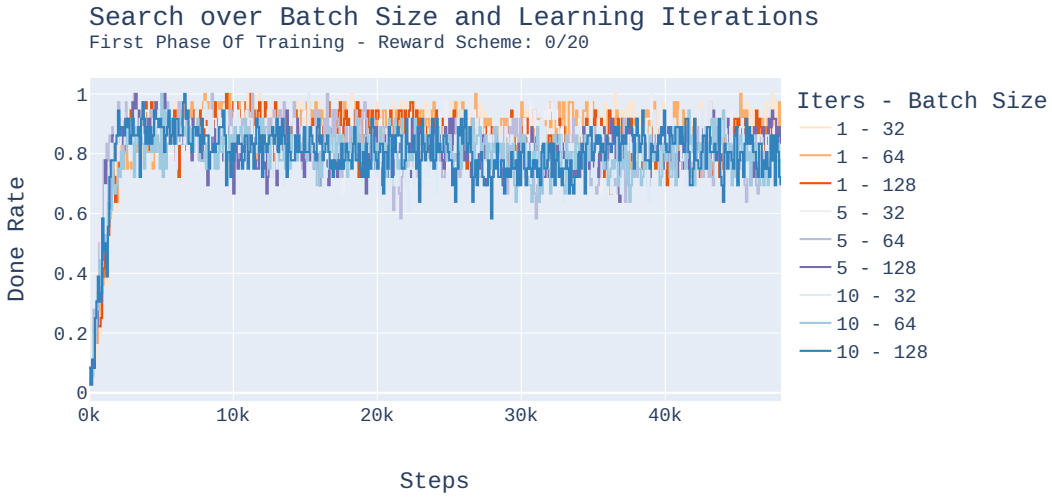


Figure F.2: Tuning of batch size and learning iterations. We used Adam with default parameters and a learning rate of $5e-4$. None of the results yield any significant differences.