THE UNIVERSITY OF EDINBURGH

INFORMATICS HONORS THESIS

Transfer Learning with Generative Adversarial Networks in Reinforcement Learning Tasks

---

Author:
Giovanni
ALCANTARA

Supervisor:
Dr. Timothy
HOSPEDALES

A thesis submitted in fulfilment of the requirements for the degree of Bachelor's of Engineering

in the

School of Informatics
College of Science & Engineering

April 5, 2018

THE UNIVERSITY OF EDINBURGH

Abstract

Faculty Name
College of Science & Engineering

Bachelor's of Engineering

Transfer Learning with Generative Adversarial Networks in Reinforcement Learning Tasks

by Giovanni
ALCANTARA

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

Chapter 1

Introduction

1.1 Motivation

Reinforcement learning is the problem faced by an agent that must learn behaviour through trial-and-error interactions with an environment.

Reinforcement learning as a field has had major successes in the past few years (Tesauro, 1995; Singh et al., 2002; Kohl and Stone, 2004; Ng et al., 2006), particularly as techniques utilising deep neural networks (DNN) have started permeating the research community. Techniques like Q-network (Mnih et al., 2015), trust region policy optimisation (Schulman et al., 2015), and asynchronous advantage actor-critic (A3C) (Mnih et al., 2016) helped stem an area of research of recent significant importance: deep reinforcement learning (DRL) (Arulkumaran et al., 2017).

Traditional reinforcement learning approaches generally lacked scalability, limiting these techniques to fairly low-dimensional problems. These limitations are in terms of memory complexity and computational complexity (Kaelbling, Littman, and Moore, 1996).

As such, using these usually becomes intractable when modelling real-world systems, due to the many variables and unknowns that are present in such systems (Strehl et al., 2006).

As DRL methods rectified some of these issues, new ones started to emerge, particularly limitations inherent to using deep neural networks. Notably, the need to have access to large datasets for training, particularly if in the context of applications that require image processing, such as autonomous vehicle control (Krizhevsky, Sutskever, and Hinton, 2012), has proven to be a critical limitation.

In real-world reinforcement learning applications, environment observations often rely heavily on computer vision and image processing (Berns, Dillmann, and Zachmann, 1992), which often provide an incomplete picture of the state that the agent is in. In such types of scenario, formally known as partially observable Markov decision processes (POMDP) (Monahan, 1982), not only do we have fragmentary observations, but it is also sometimes prohibitive to build large datasets that DRL requires to train the agent.

In deep learning, one of the ways to circumvent this constraint is *transfer learning*, the ability to leverage models trained in a particular domain on

different applications. Transfer learning has proven pivotal in achieving successes in a wide variety of applications, without the need to train expensive models from scratch (Pan and Yang, 2010).

There has been much work in improving transferability of reinforcement learning models, most notably multitask learning (Caruana, 1998) and curriculum learning (Bengio et al., 2009), though few methods have tried to bootstrap learning with the use of DNNs.

In our work we provide a general unsupervised framework that lets us speed up reinforcement learning on unseen tasks in related domains. We do this by training deep learning models over a distribution of optimal policies for different configurations of a task in a certain domain. More specifically, given a distribution of trained policies in variations of an environment, we train two models: a generative model that is able to generate policies for different configurations of a task in a domain, and a discriminative model that is able to tell whether a policy is a good one within this domain. We show how using these models while doing reinforcement learning can speed up learning on new unseen configurations.

The generative model and the discriminative model are trained using deep neural networks in an adversarial architecture also know as a Generative Adversarial Network (GAN), introduced in Goodfellow et al., 2014's seminal work. While this idea was popularised with applications in image synthesis, most notably Deep Convolutional Generative Adversarial Networks (Radford, Metz, and Chintala, 2015), there have been successes using GANs within reinforcement learning.

Specifically, work on generative adversarial imitation learning (Ho and Ermon, 2016) has shown remarkable speedups in the task of imitating behaviour given expert policies.

1.2 Goal

Given all these motivations, we can now define the **final goal of our work**:

> *Transferring knowledge* obtained with reinforcement learning techniques on certain tasks to similar, unseen tasks, by leveraging the power of *Generative Adversarial Networks* (GANs). In this way, we hope to *achieve shorter training times* and *better rewards* when training models for similar *unseen tasks*.

1.3 Data pipeline

In this section we provide a bird's eye view to the whole project by showing a visual schematic of the data pipeline (Figure 1.1)

We will be referencing this pipeline schematic throughout the whole report. In fact, as we dive deeper into technical details of each component of the data pipeline, it is critical to take them in the context of the whole pipeline as to justify certain design decisions we make for our experiments.

At this point, it is important to highlight how we achieve the goal we set in the previous subsection (subsection 1.2) with this data pipeline.

The input to our system is a set of similar tasks that would be traditionally solved with reinforcement learning techniques. What makes these tasks similar is the fact that the goal is the same, but the agent operates in different map configurations. We provide more details on the environment in Chapter 3. We split these tasks into a test set and a training set. We train all of these maps configurations with Q-learning, a reinforcement learning technique to obtain optimal policies.

Now, what we want to achieve, as per our goal, is:

1. train a GAN system to capture the distribution of the policies in the training set,
2. use the trained GAN to develop a new algorithm that is able to transfer that knowledge to speed up training of tasks in the test set of map configurations,
3. verify and benchmark that the test policies obtained with the new method are better than the ones trained with traditional Q-learning.

1.4 Structure of the report

Our report is structured in a what that strongly follows each component of the data pipeline outlined in the previous subsection.

- Chapter 2 introduces important background information on reinforcement learning, Markov Decision Processes, Q-learning and Generative Adversarial Networks.
- Chapter 3 details the environment being used in our experiments.
- Chapter 4 reports on the process of training policies on all the different tasks with traditional reinforcement learning (Q-learning).
- Chapter 5 outlines the process of training the GAN given the trained policies in the training set.
- Chapter 6 describes the new algorithm we develop to speed up learning on unseen tasks.
- Chapter

1.5 Main contributions

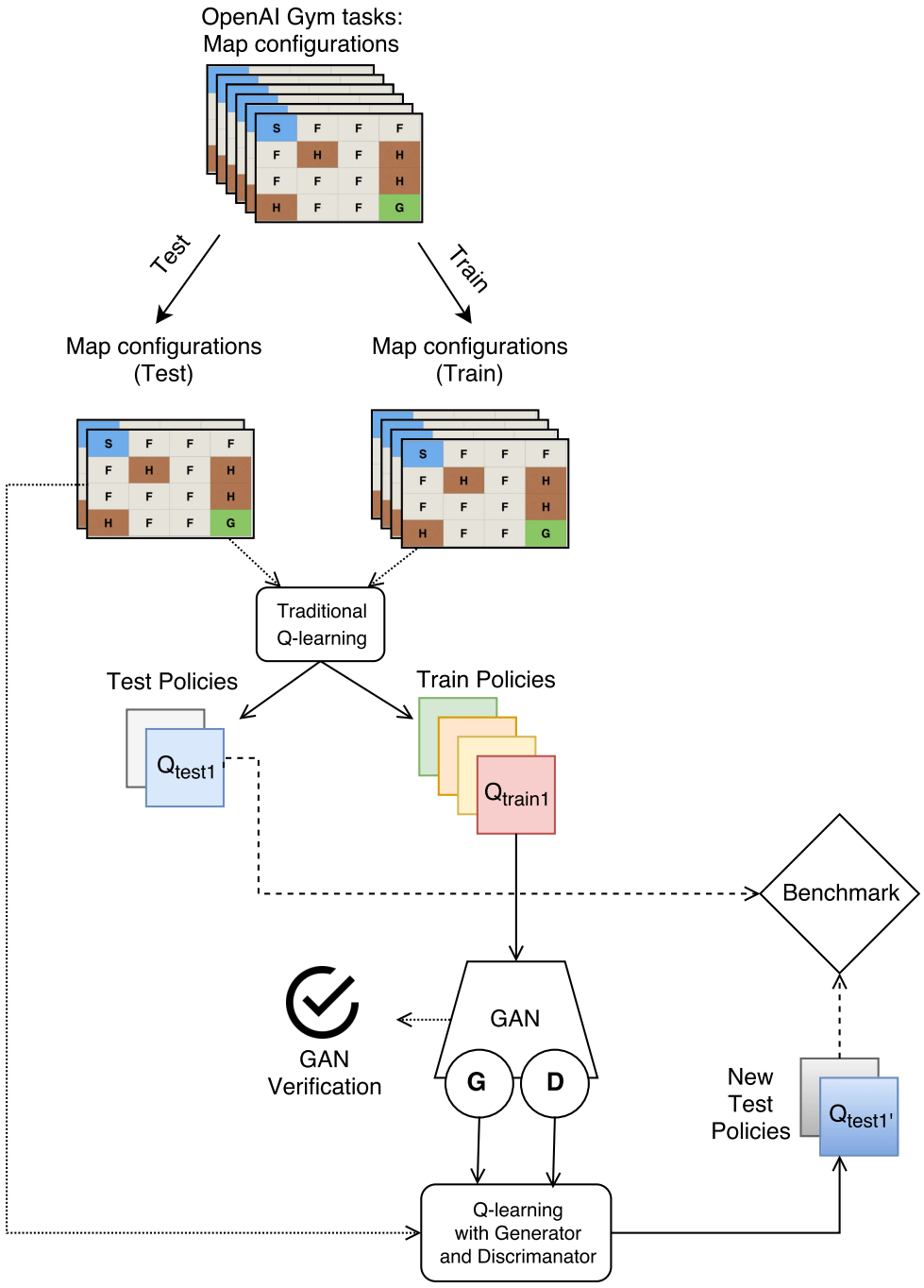


FIGURE 1.1: The Data Pipeline

Chapter 2

Background

2.1 Reinforcement Learning

2.1.1 Markov Decision Processes

Environments in traditional reinforcement learning application are usually modelled as Markov Decision Processes or MDPs.

These can be formulated as systems with the following components:

- Finite set of states $S = \{s_0, \ldots, s_n\}$ and actions $A = \{a_0, \ldots, a_m\}$.
- A distribution of probabilities $P_a(s, s')$ for transitions from state s to s' for each possible action a.
- A reward function $R : S \mapsto \mathbb{R}$ for being at a particular state.
- The goal in reinforcement learning is to maximise the final reward that an agent achieves in the environment.

As the name suggests, MDPs obey the *Markov property*, whereby the probability of the system being in a certain future state exclusively depends upon the present state, and not upon an arbitrarily-long sequence of past states.

In figure 2.1 we show a sample schematic of a Markov Decision Process, and how states, actions, and rewards could be connected between each other.

2.1.2 Q-learning

Now that we contextualised reinforcement learning environments as Markov Decision Processes, we can introduce the final objective of reinforcement

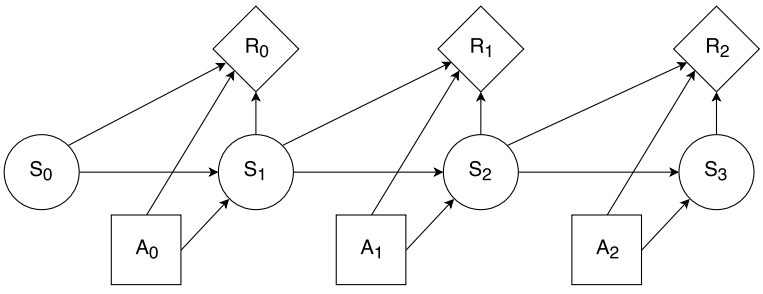


FIGURE 2.1: Sample schematic of an MDP.

learning tasks: finding a function $\Pi : S \mapsto A$ called **policy**, that maps the appropriate action $a \in A$ given the current state $s \in S$, as to maximise our agent's final reward.

In particular, we will now present Q-learning (Watkins and Dayan, 1992).

Q-learning lets us learn the *quality*, or expected utility, of each state-action combination. That is, for each state, we estimate all the expected rewards we obtain by taking each possible action at that particular state. This is represented as a state action table. We can use a Q-table to define a policy by always picking the action with the highest expected return.

More formally, we estimate a function $Q : S \times A \to \mathbb{R}$. We can model Q as a mapping table (initialised with some uniform values), whose value we update at each time step of our simulations.

Here's how we update our Q-table at each time step t:

$$Q(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \times \left[\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right] \tag{2.1}$$

where:

- $\alpha \in [0, 1]$ is the *learning rate*, a coefficient that regulates how much the newly learned values will contribute in the update
- $\gamma \in [0, 1]$ is the *discount factor*, a coefficient that controls the weight of future rewards. Values closer to 0 will make our agent "short-sighted", considering only the immediate rewards.

What is our optimal policy when we do Q-learning then? After training, it is simply that function $\pi : S \to A$ that, for each state, returns the action with maximum expected utility in our Q-table.

Algorithm 1 reports the pseudocode for Q-learning.

From this procedure we need to highlight two important concepts.

The first one is the concept of episode, which is defined as a sequence of actions in which an agent (the algorithm) interacts with its environment until it reaches a certain terminal. In Q-learning we need to specify a certain number of episodes that we will run the algorithm for. The more episodes we choose, that more optimal the policy is likely to be, since the has more interactions with its environment.

The second concept is that of ϵ-greedy, which we introduce in the next subsection on the tradeoff in Reinforcement Learning between exploration and exploitation.

Algorithm 1 Q-learning

Require:
S is a set of states
A is a set of actions
γ the discount reward factor
α is the learning rate
n is number of episodes to run Q-learning
ϵ, probability to take random action, rather than follow policy
procedure Q-LEARNING
Initialize $Q(s, a)$ will all 0 utility values.
  for each episode e_i with $i = 0...n$ **do**
    Initialize s
      for each step of episode **do**
        Choose a_t from s_t using policy derived from Q with ϵ-Greedy
        Take action a_t, observe reward r and s_{t+1}
        Update Q-table using equation 6.1
      end for
  end for
end procedure

2.1.3 Exploration/Exploitation tradeoff

An important theme in reinforcement learning, and that we also focus our attention on in our project is the idea of the tradeoff between Exploration and Exploitation.

In reinforcement learning, and generally in decision making, we have a fundamental choice: *exploitation*, whereby you make the best decision given current information, or *exploration*, where we prefer gather more information.

The reason why this is a trade-off is because the best long-term strategy may involve short-term sacrifices, so we wish to gather enough information to make the best overall decisions.

For example, consider the case where we need to choose a restaurant for dinner. We can either go to our favourite restaurant (exploitation), or we could choose to try a new restaurant (exploration) with the hope that it is eventually better than our favourite, but also risking that it is worse.

In reinforcement learning this usually can be represented as an ϵ-greedy approach. In this approach, we choose to either *exploit* our current knowledge and choose the action a_t^* that we think is optimal given what we currently know, or we choose to *explore* by taking a random action.

$$a_t = \begin{cases} a_t^* & \text{with probability 1-}\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

It is critical to account for this tradeoff when developing algorithms in reinforcement learning, and we will show how we can still keep allow for both exploration and exploitation with the techniques that we develop.

2.1.4 Deep Reinforcement Learning

DQN

2.2 Generative Adversarial Networks

Generative Adversarial Networks, or GANs, are a deep neural network architecture composed of two neural networks, set against each other (thus "adversarial").

GANs were introduced in a paper by Goodfellow et al., 2014. Yann LeCun, Facebook's AI Research director referred GANs as being "the most interesting idea in the last 10 years in ML".

What have made GANs particularly successful in past work is their ability to model any distribution of data, and provide, after training, both a generative model and a discriminative model based on the initial distribution of the data. Let us define what we mean by these terms.

A *generative* model is one that is able to produce new data points that fit the distribution of the training data. For example, a Gaussian Mixture Model is a model that, after training, could generate new data based on the original distribution. More formally, training a generative model can be seen as maximum likelihood estimation problem:

$$\mathscr{L} = \prod_{i=1}^{m} p_{model}(x_i; \theta)$$

More specifically, we want to maximise the likelihood that a sample x that the generative model produces belongs to the distribution of our initial data.

A *discriminative* model is one that can discern (or "discriminate") the difference between two (or more) classes/labels. For example, we could train a Convolutional Neural Network that is able to tell us whether an input image is a face (1) or not (0).

Depending on the task at hand, we may be more interested in the generative model (for example in the case of image synthesis), or in the discriminative model, or in both.

Next, we will look at the typical architecture of Generative Adversarial Networks.

2.2.1 Architecture of GANs

Since GANs were introduced in 2014, many variants of GANs spawned from the research community. Here we will focus on the architecture of the original, vanilla GAN.

All variations of GANs will nevertheless contain two main components, which are both modelled as neural networks. These two models are trained simultaneously. One is the *generator* model, that takes an input z and produces a sample x from an implicit probability distribution. The other is the *discriminator*, a classifier that, given a sample, tries to identify it as originating from the generator model or from the original distribution.

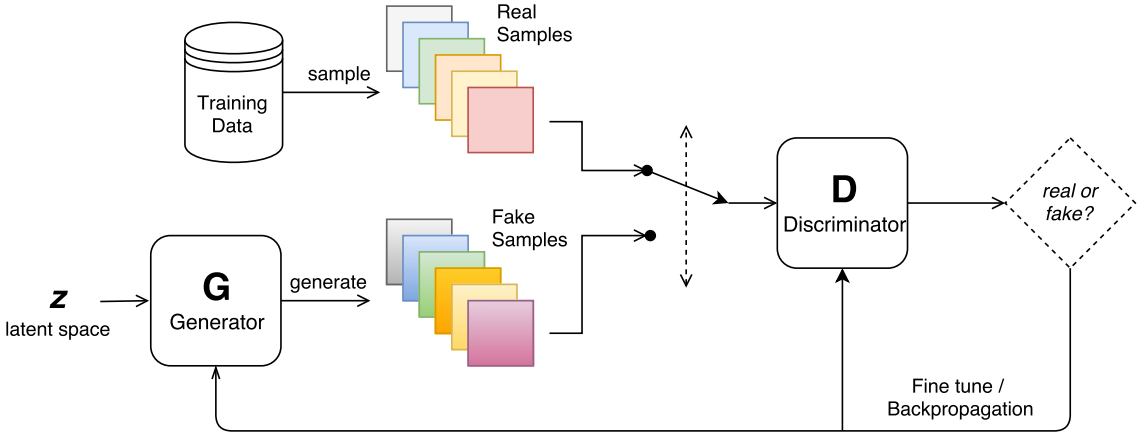


FIGURE 2.2: Architecture of a Generative Adversarial Network

An intuition to this set up is the following: the generator G can be imagined as a counterfeiter trying to produce fake banknotes or paper money; the discriminator D is the police, trying to distinguish real banknotes from the fake ones generated by the counterfeiter. As the counterfeiter becomes better and better at producing banknotes, the police will also try to improve its counterfeiting techniques.

More formally, G is a differentiable function whose parameters are trained to minimise correct assignments of D. D is also a differentiable function, which has been trained to maximise correct labels to the real and the fake samples.

In figure 2.2 summarises this whole procedure by showing the architecture of GANs.

At each iteration, the mini-batch inputs to the discriminator D are taken from both the real data sampled from the training data, and the samples generated by the generator, whose inputs is random latent variable z. Given D's prediction, we then apply a gradient-based optimisation method to update both D's and G's parameter.

To do so, we follow the loss function that defined based on the goals of each network. This cost function implicitly models a zero-sum game, which in game theory are guaranteed to achieve an equilibrium, by the minimax theorem (Du and Pardalos, 2013).

The cost function for the whole system would look like the following minimax game:

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z\sim p_g(z)}[log(1-D(G(z)))]$$

2.2.2 Successes

DCGAN

2.3 Related work

2.3.1 Reinforcement learning with unsupervised auxiliary tasks

There have few successes in transferring deep reinforcement learning across domains. Mostly notably, Jaderberg et al., 2016 introduced a technique to identify in an unsupervised way, multiple pseudo-reward functions based on all training signals that the agent collected as observations. While doing deep reinforcement learning, therefore, Jaderberg et al. would not only try to directly maximise the agent's cumulative reward, but also all the identified extrinsic rewards.

There is a potential to re-use these identified extrinsic rewards in other domains, but this is an indirect way to tackle the problem of transferring behaviour.

In this work, they would first identify what auxiliary rewards the observations can give, to then reuse them on different task. Also, we have little to no control to guide the unsupervised exploration of auxiliary rewards functions towards a related task that we have the power to define.

In fairness, Jaderberg et al. introduced auxiliary rewards as a way to speed up reinforcement learning on a single task, rather than aiming to transfer these to related tasks.

In our work we do not indirectly try to capture pseudo-rewards given a task, but we try to capture transferrable knowledge directly from the training policies.

2.3.2 Generative Adversarial Imitation Learning

An analogous way to do transfer learning in the context of reinforcement learning imitation learning, an important subfield of research in the reinforcement learning community. Imitation learning, as the name suggests, mostly deals with training agents that imitate an expert behaviour.

It does so by taking signals coming in from experts, usually encoded human behaviour. Having such signals can significantly decrease training time, as well as provide new insight onto different optimal behaviour that could achieve better rewards. Most notably, researchers at OpenAI used signals from professional Go players to build imitation learning techniques (Silver et al., 2016). This was only used as a priming step to learning the game of Go, and was used in combination with Deep Q-networks.

Generally, the imitating agent is trained on the same environment as the expert agent. Thus, imitation learning, while still being a form of knowledge transfer, only relates to techniques that do not cross task domains.

Despite this crucial difference with what the goal of our project, a particular technique that has been recently introduced for imitation learning is still very much related to some of the components of our pipeline.

Algorithm 2 Generative adversarial imitation learning

1: **Input:** Expert trajectories $\tau_E \sim \pi_E$, initial policy and discriminator parameters θ_0, w_0
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: Sample trajectories $\tau_i \sim \pi_{\theta_i}$
4: Update the discriminator parameters from w_i to w_{i+1} with the gradient
5: Take a policy step from θ_i to θ_{i+1}, using the TRPO/PPO rule.
6: **end for**

Generative Adversarial Imitation Learning (Ho and Ermon, 2016) or GAIL bases much of its architecture on GANs.

Ho and Ermon from OpenAI introduced it as a better-performing alternative to imitation learning techniques. In particular, they motivated their work on the computational cost of a typical approach to imitation learning, which does not directly train a policy that is imitating an expert. In fact, one way to do imitation learning was to use inverse reinforcement learning on the expert policy to obtain its reward function. Based on this reward function, reinforcement learning techniques could be used to train imitating agent. Given this pipeline, we can notice that it is an indirect way of conducting imitation learning, which is prone to produce a model that does not accurately represent the distribution of trajectories of the agent.

With GAIL there is no need to extract a reward function, but they let the Generative Adversarial Network capture the distribution of observations and action pairs. After training the GAN, the Generator will have been trained to produce appropriate actions, given observations. The real data that the GAN is trained with are trajectories coming in from the expert policy. Algorithm 2 also shows a simplified algorithm for GAIL.

In figure 2.3 we present a schematic of each component involved in training a Generative Adversarial Network for Imitation Learning. We notice right away that the GAIL architecture closely resembles the Vanilla GAN, as we showed in figure 2.2. After training, the Generator will represent the imitating policy, which will hopefully perform as well as the expert policy. Again, there is no need to extract a reward function in this whole pipeline.

In this work, Ho and Ermon have shown groundbreaking results in imitation learning using GANs, as they achieved much better performances with traditional imitation learning techniques like Behavioural Cloning or Dataset Aggregation.

In our work, we aim to achieve as great results in transferring knowledge across domains.

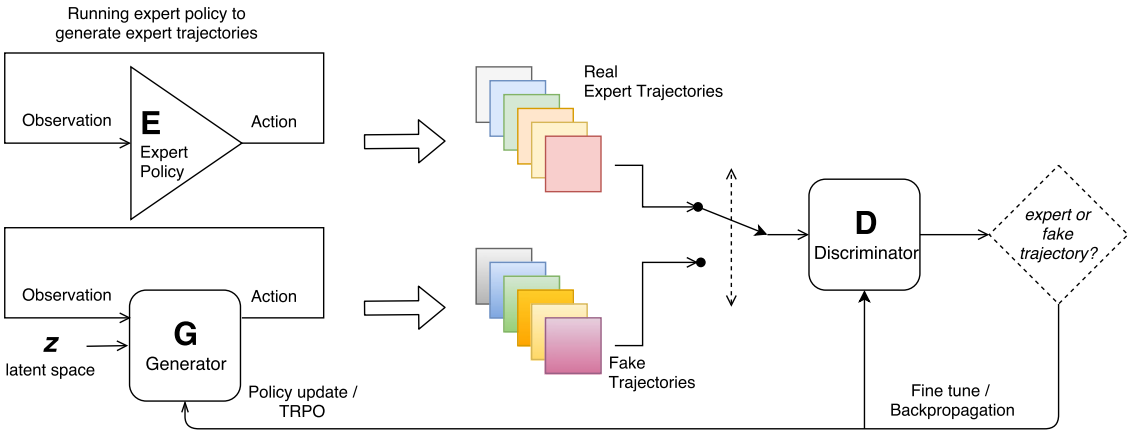


FIGURE 2.3: Generative Adversarial Imitation Learning (GAIL) architecture

Chapter 3

Environment

In this chapter we report the process that went into choosing the environments that we will be using in the project's simulations, and from which we will be basing our simulations.

There are many choices of environments and tasks that are publicly available. Some of these we will be introducing in this chapter.

What makes this step non-trivial and deserving of its own chapter is that different reinforcement learning techniques are more suitable to different categories of tasks. Similarly, different machine learning and deep learning techniques are more or less efficient when applied to different tasks.

In a research effort that is heavily dependent on building reinforcement learning and deep learning models, the choice of environment is a critical one.

Furthermore, we also need to achieve this without losing focus on the main motivations for the whole project (Chapter 1): *optimising reinforcement learning algorithms by adding transferability of pre-trained models on unseen maps or configurations of a task*.

This last point implies that a substantial part of the computational work in the project will be about training hundreds of thousands of reinforcement learning models to build a dataset over a distribution of different maps (we present this in Chapter 4). This is an important point: in the many experiments we ran, this turned out to be the biggest bottleneck, and required plans to distribute computations across multiple machines to make the computational time feasible in the timespan allocated to the project.

With these points in mind and given the experimental nature of the work, we conclude that a bottom-up approach in complexity is preferable. Given successful results with "easier" tasks, we can scale up in complexity and hopefully formalise and generalise our approach to more tasks (Chapter 7).

Easier tasks will enable us to explore different reinforcement learning approaches that we introduced in Subsection 2.1 with the guarantee that they will give satisfiable results. We can use these results as a foundation of the further steps (specifically Generative Adversarial Networks training in Chapter 5).

Before introducing candidate environments, let us define what is meant by an "easy" reinforcement learning task. What we are looking for is ideally a task with a discrete and relatively small set of observable states and actions. Why does this condition make the task easier?

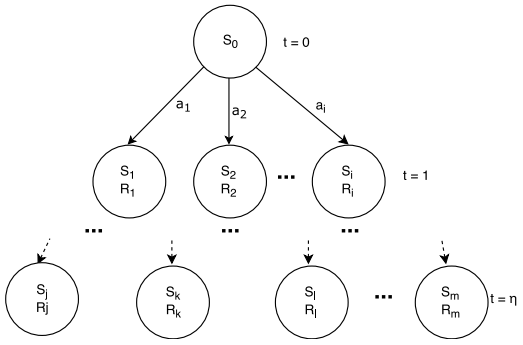


FIGURE 3.1: Sample state-action tree

Imagine building a decision tree (such as the one shown in figure 3.1 with all possible states-action transitions, until we either: 1) reach a goal state, or 2) reach an arbitrarily maximum iteration time step $t = \eta$ or depth of the tree (to prevent infinite iterations). Also assume we were building this tree in a bruteforce manner (worst-case scenario of reinforcement learning resolutions), such that we need to build all possible paths or trajectory that the agent will need to take. Therefore, we would need to visit each node of the tree, until we arrive at the leaves. At the leaves, we can find the achieved reward for the agent given the path it has taken to get there.

The breath and depth of the state-action tree will increase as we increase the possible set of states we would need to traverse, adding up to the space and time complexity of our solution, which is exponential in $\#S$ and $\#A$ for both space and time ($\#S$ indicates the cardinality of a set S).

With that said, there is a solid line between our ideal environment and a trivial one that could be solved without the aid of reinforcement learning techniques.

Let's explore some candidate environments next.

3.1 OpenAI Gym

OpenAI Gym (Brockman et al., 2016) provides a toolkit that aids in the process of building reinforcement learning systems and evaluating algorithms to solve such tasks.

OpenAI Gym provides an environment interface `Env`. The interface abstracts the following operations on an environment:

- `step(action)` – Simulates one time step by executing the provided action. This returns `observation` (the new current state), the `reward` for taking such action, a flag `done` indicating whether the system has reached a final state, and `info` providing additional information dependent on the environment.
- `reset()` – Resets the environment, i.e. the initial state is restored
- `render()` – Renders the environment in human-readable format.

Now, we could either build implementations for such interface (if we were to implement our own environment's logic), or use the provided implementations for several environments in the OpenAI Gym library, which includes board games, algorithm-based or physics-based environments, Atari games, etc.

3.1.1 Motivation

In this project we will mostly deal with environments provided in the OpenAI Gym. There are several reasons for such decision:

- It abstracts the need to implement the logic of a separate environment. Implementing our own environment adds a point of failure to our whole (quite experimental) work, as well as imposing a bigger time constraint to the one we already have;
- OpenAI Gym has become a standard academic tool for reinforcement learning researchers, therefore many papers and articles build on top of this framework;
- Environment implementations in the OpenAI Gym library are constantly expanding and being revised by an active community, also thanks to the support of the overarching organisation (OpenAI);
- The core implementation of the `gym` module (*openai/gym on Gihub*) allows for straightforward extensions on existing environments, which, as we will see in subsection 3.3, will be critical in this project;
- While sadly not applicable anymore, OpenAI Gym used to provide and support an online platform for developers to compare the performance of different reinforcement learning algorithms for each task. This was in a form of a leaderboard, measuring the performances, as well as providing the implementation and data of winning techniques. Such data could have been used directly as input to our Generative Adversarial Networks.

3.2 Baseline: `FrozenLake-v0`

One of our baseline environments is `FrozenLake-v0` (*OpenAI Gym*), one of the algorithmic environments provided in the OpenAI Gym. In this section we describe the task, motivation for choosing it as one of our baseline models, and some shortcomings that we faced in using this environment in our project's pipeline.

3.2.1 Description of the task

In `FrozenLake-v0` we control an agent in a grid world, more precisely the grid shown in figure 3.1. The objective is to go from a starting tile (`S`) to a

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	**G**

TABLE 3.1: `FrozenLake-v0`'s default 4x4 configuration

goal state (`G`), by moving in four possible directions from a tile: `up`, `down`, `left` and `right`.

What differentiates this from a trivial path-finding or planning problem? A couple of things:

1. there are both walkable and non-walkable tiles in the grid (these are respectively frozen tiles `F` and holes `H`), and

2. tiles are "slippery", as in the agent could "slip" while navigating the grid world, meaning that the movement direction of the agent is uncertain and only partially depends on the direction we tell the agent to follow, i.e. the direction is non-deterministic.

The reward in `FrozenLake-v0` is 1 for reaching the goal state, and 0 for being in any other state. The system stops executing when the agent falls into a hole, and the environment needs to be reset.

3.2.2 Motivation and shortcomings

`FrozenLake-v0` is generally classified as a straightforward reinforcement learning task. An optimal solution could be found by creating a model of the environment, that is merely recording where frozen tiles are while exploring the grid world. This is a *model-based* approach to reinforcement learning, and it is perhaps less interesting than learning by exploration, without being biased by the particular configurations of the map.

Model-free algorithms, like Q-learning, need no accurate representation specific to the environment, and they are therefore more transferable to different configurations of the map or even to different tasks, which is the final objective of our project.

In fact, let's take the case of Q-learning applied to `FrozenLake-v0`: we do not need to have an explicit knowledge of the dynamics of each different tiles. We do not build a policy that explicitly favours movements towards frozen tiles or towards the goal. In fact, the agent does not have a model of what a frozen tile is, nor a model of the goal tile or a hole–it just learns by exploration that it is rewarded when it gets to the goal state, and that is what it implicitly aims for.

This is the sort of behaviour that we can transfer to different tasks–again our ultimate objective.

In its current OpenAI Gym implementation, `FrozenLake-v0` is a static map with the fixed configuration that is shown in figure 3.1. There is currently no

way to generate random configurations of the map, so next up, we will be extending this implementation to account for that.

3.3 Extended baseline: Randomised Frozen Lake

We would like to to extend OpenAI Gym's implementation of `FrozenLake-v0` so that it can generate random configurations of the map.

Before we move on, let us contextualise this in the bigger picture as to not lose focus of what we are trying to achieve. Why do we need different configurations of the map, again? We want to train reinforcement learning models on different configurations so that we can have a distribution of policies over different maps which we can use as input to our GAN. After training our GAN, we will have a Generator network spawning new policies for unseen configurations without having to find it through (computationally expensive) reinforcement learning algorithms!

Listing 3.1 shows how we implemented the algorithm to generate random maps. The critical line is line 4, which uses `numpy`'s `random.choice()` method that samples a matrix of a given size, given probabilities for each element (`'F'` and `'H'` tiles in our case).

So, we can pass in the desired size of the map. By default, it will generate 4x4 grids like the one in `FrozenLake-v0`, but we could generate maps of arbitrary size, which will result in higher task complexity. We explore these harder extensions in Chapter 7.

We can also pass it the probability that a tile will be a frozen one through the parameter `p`. The presence of fewer frozen tiles, and therefore of more holes, makes the goal harder to achieve for the agent.

```
1 def generate(size=4, p=0.8):
2     valid_map = False
3     while not valid_map:
4         config = np.random.choice(['F','H'], (size, size), p=[p, 1-
      p])
5         config[0][0] = 'S'    # set top left to be the starting tile
6         config[-1][-1] = 'G' # set bottom right to be goal tile
7         valid_map = is_valid(config)
8         p *= 1.05             # increase probability of frozen tile
9     return ["".join(x) for x in config]
```

LISTING 3.1: Algorithm to generate random configurations. Utility function `is_valid()` of line 7 is shown in listing 3.2

Notice how the `generate()` function in listing 3.1 only returns valid maps, that is, maps that have at least one frozen path from start to goal. Surely, we could train models on environment configurations that are not solvable. Q-learning, for example, would just return a `Q` matrix with all Q-values equals to 0, since the agent will never get to the goal tile and get its reward. If it is unclear why, refer to the Q-learning subsection 2.1.2.

Using such constraint on map validity, we can limit the number of models we need to train by a significant amount, therefore reducing training time.

For the 4x4 grid, rather than having $2^{(16-2)} = 2^{14} = 16,384$ possible configurations, we can reduce it to only 3,827 possible valid map configurations.

To check whether a map is solvable we use depth-first search from the start tile to the goal. If there is such path, then it is a valid map. Listing 3.2 shows the algorithm:

```
def is_valid(arr, r=0, c=0):
    if arr[r][c] == 'G':
        return True

    tmp = arr[r][c]
    arr[r][c] = "#" # temporary mark with '#' to remember visited
    tiles

    if r+1 < size and arr[r+1][c] not in '#H': # go down
        if is_valid(arr, r+1, c) == True:
            arr[r][c] = tmp
            return True
    if c+1 < size and arr[r][c+1] not in '#H': # go right
        if is_valid(arr, r, c+1) == True:
            arr[r][c] = tmp
            return True
    if r-1 >= 0 and arr[r-1][c] not in '#H': # go up
        if is_valid(arr, r-1, c) == True:
            arr[r][c] = tmp
            return True
    if c-1 >= 0 and arr[r][c-1] not in '#H': # go left
        if is_valid(arr,r, c-1) == True:
            arr[r][c] = tmp
            return True

    arr[r][c] = tmp
    return False
```

LISTING 3.2: Depth-first search to check if a frozen lake map is valid

So the `generate()` function returns a valid random frozen lake map represented as a list of strings. An example of an output is `["SFHH", "HFHH", "HFHH", "HFFG"]`. Each string in the list encodes a row configuration of the frozen lake.

What is left to do now, is to extend `FrozenLake-v0` so that we could pass in any map configuration, therefore exposing our randomly generated maps to the `gym` environment interface described earlier in section 3.1. Listing 3.3 shows how to register a new environment by extending a pre-existing one.

```
random_map = generate(size=4, p=0.8)
register(
    id='RandomisedFrozenLake',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'is_slippery': True, 'desc': random_map},
    max_episode_steps=100,
)
```

LISTING 3.3: Code to extend `FrozenLake-v0` with random maps.

Now, to start doing simulations on our new environment we can initialise it just like any other OpenAI Gym environment:

```
env = gym.make('RandomisedFrozenLake')
```

We created a pull request to the OpenAI Gym's Github repository integrating this feature supporting random maps for `FrozenLake-v0`, so that developers and researchers could make use of these functionalities (*Add ability to generate random frozen lake maps by gvsi · Pull Request #835*).

3.3.1 Adjusting the reward system

The vanilla OpenAI Gym implementation of `FrozenLake-v0` has sparse rewards, meaning that the agent is rewarded a value of 1 if it reaches the goal, and a value of 0 if it falls into a hole.

The problem with this system of rewards is the fact that it does not take into account the number of timesteps that it takes the agent to solve the task.

Consider the three paths shown in figure 3.2. All three paths would yield a final reward of 1 in the OpenAI Gym's implementation. But intuitively, in a real-world scenario, we'd like to have a path-finding robot that is able to navigate the environment with as few actions as possible.

More generally, reinforcement learning environments have rewards that are inversely proportional to the number of steps that it takes the agent to solve (Bellemare et al., 2013; Brockman et al., 2016; Zamora et al., 2016)

We wish to leverage this alternative reward system to train agents that can therefore reach the goal state as quickly as possible.

In terms of extending our `RandommiseFrozenLake` implementation, we can just update the final reward as

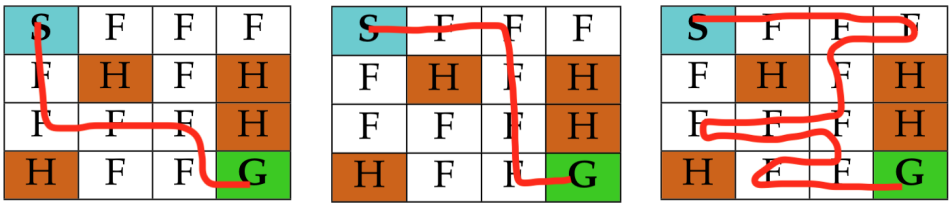
$$r = \begin{cases} 0 & \text{if agent falls into a hole} \\ 6/t & \text{if agent reaches goal state} \end{cases}$$

where t is the number of timesteps needed to reach the goal state.

If the agent reaches the goal state in the minimum number of timesteps (6), the reward would be 1, otherwise we add a penalty for any additional timestep required.

The rewards of each path in figure 3.2 will therefore be, from left to right, $r = 1$, $r = 1$ and $r = 6/14 = 0.43$.

In section 4.4 and 4.5, we will show how this reward system (favouring paths that are shorter) will help us define the type transferable knowledge that we wish to transfer across tasks.



(a) Optimal path to goal tile

(b) Another optimal path to goal tile

(c) Non-optimal path to goal tile

FIGURE 3.2: Different paths to the goal tile

Chapter 4

Dataset creation

In Chapter 3 we finalised our OpenAI environment choice and have a systematic way to generate different configurations of our environments.

The next step is to build up our dataset by solving as many of these configurations as possible with the use of reinforcement learning algorithms.

We justified in previous sections (specifically section 2.1.2 and section 3.2.2) how using *model-free* reinforcement learning algorithms will put us in the right direction to achieve the project's goals highlighted in section 1.1: improve the transferability of pre-trained models to different configurations, environments, and tasks.

One choice that is left to make before we move on to train models using our randomised environments is what reinforcement learning algorithm we should use.

4.1 Q-learning on `RandomisedFrozenLake`

We have presented Q-learning in section 2.1.2 as a method that generates a policy by building up a table Q of values corresponding to the expected utility of taking each action at each observable state.

This table Q is represented as a 2D-matrix of size (`env.observation_space.n`, `env.action_space.n`), which in the case of `RandomisedFrozenLake` is just a (16x4) matrix.

This is good news. Having a policy that is encoded as a 2-dimensional data structure makes it an obvious input to our Generative Adversarial Network later on in the next chapter. In fact, as we presented in section 2.2.2, GANs have proven successful in image synthesis applications, where inputs were images, that is 2D matrices.

The choice of Q-learning as our reinforcement learning technique therefore becomes preferable. Other model-free algorithms like policy search may not have a clear 2D representation in the way the trained parameter set θ is encoded.

4.2 Experiment set up

Before we proceed, we need to be able to record some details about the process of training our data. If our ultimate objective is to benchmark performance of traditional reinforcement learning agains our proposed approach,

we need to record running time of our training, and see if that improves at the end.

Let's encapsulate training of a configuration in an `Experiment` class, whose code is shown in listing 4.1. An `Experiment` is initialised with an OpenAI Gym environment, which in our case is an instance of a `RandomisedFrozenLake`, and an integer value `num_episodes`, indicating the number of independent simulations the Q-learning algorithm will be doing. By calling `Experiment`'s `run()` instance method, we actually start training the model, with Q being updated at each iteration.

The instance variable `score` could be used as an evaluation criteria of the effectiveness of our training. It records the average reward the agent achieves for each episode during training.

`Experiment` also has a utility method called `dumps()` that serialises all this data and allows us to save it on disk.

```
1  class Experiment(object):
2      def __init__(self, env, num_episodes=10000):
3          self.env = env
4          self.Q = np.zeros([self.env.observation_space.n, self.env.
       action_space.n])
5          self.num_episodes = num_episodes
6          self.score = None
7          self.valid_score = None
8          self.start = None
9          self.end = None
10
11     def run(self):
12         self.start = datetime.now()
13         #
       ------------------------------------------------------------------
14         # Run Q-learning algorithm, saving the rewards of each
       episode
15         #
       ------------------------------------------------------------------
16         self.end = datetime.now()
17         self.score = sum(rewards)/self.num_episodes
18
19     def dumps(self):
20         return dumps({'Q': self.Q, 'start': self.start, 'end': self
       .end, 'score': self.score, 'num_episodes': self.num_episodes})
```

LISTING 4.1: `Experiment` wrapper class to train one instance of `RandomisedFrozenLake`

It's critical to be able to evaluate the quality of the Q-table after training. To do so we just use the optimal policy (that is the policy that picks the action that has the maximum expected utility according to the Q-table) and run if for a certain number of episodes. A validation score could be then defined by the average reward achieved at each episode. Listing 4.2 shows the code to achieve that.

```
def validate(self):
  rewards = 0
    for i in tqdm(range(num_episodes)):
        while j < 200: # Limit to 200 time steps
            j+=1
            # Choose an action by pick best action from Q-table
            a = np.argmax(Q[s,:])

            # Get new state and reward from environment
            s1,r,d,_ = env.step(a)
            rewards += r
            s = s1
            if d == True:
                break
     self.valid_score = rewards / num_episodes
```

LISTING 4.2: Code to validate a trained Q-table

4.3 Distributing Q-learning with MapReduce

Now that we formalised our experiment setup, we can run Q-learning on each of the map configurations of our `RandomisedFrozenLake`. For the 4x4 grid there are 3,827 possible valid map configurations. It takes an average of 15 seconds to train each Q-learning table on a 2.5 GHz Intel Core i7 processor. On a single machine, it would take around 16 CPU hours to run the whole set of experiments.

To make this step of our pipeline faster and scalable to more data, we decide to set up a distributed processing on a cluster with multiple machines. More precisely, we set up a MapReduce framework (Dean and Ghemawat, 2004) implementation running on an Hadoop cluster (Shvachko et al., 2010) of 25 machines.

A MapReduce program is composed of a Map procedure that takes in some (large) input and performs a particular operation whose output is then fed into another Reduce procedure, which outputs the final result. The power of MapReduce is that the framework orchestrates the processing of these procedures by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

Figure 4.1 shows the distributed architecture to train multiple Q-learning instances. The input to the system is a list of strings, each representing the map configuration of a `RandomisedFrozenLake`, e.g. `"SFHHHFHHHFHHHFFG"`. These inputs are fed into Mapper programs running on different machines. The mapper's task is to initialise the experiment and run Q-learning on the environment. It outputs a key-value pair, where the key is the string that uniquely identifies a map configuration, and the output is the `Experiment` object that encapsulates the already trained Q-table. We have yet to assign a validation score to this table we trained, and that's the job of the reducer.

Each experiment's result is then written to an output file by the reducer. Each line will be again a key-value pair, with the map configuration string

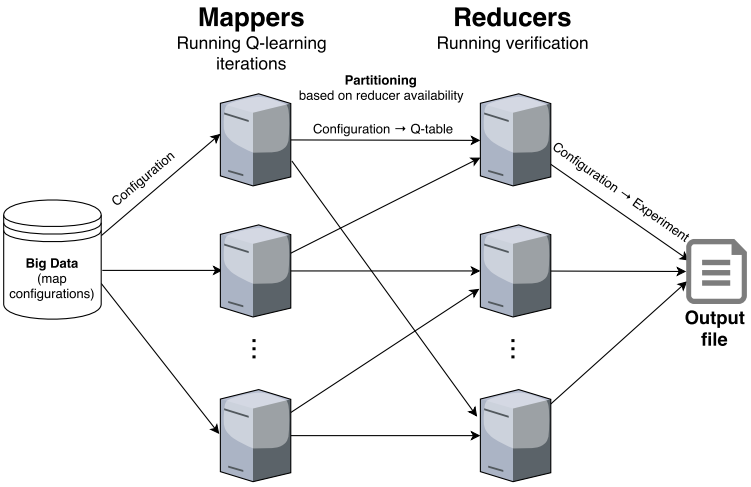


FIGURE 4.1: Schematic of distributed Q-learning with validation on MapReduce

as key and the `Experiment` result dumped in string format. In the following sections, we can just parse and load these results to conduct our further analysis.

While this particular architecture does not make full use of the power of MapReduce (combining, partitioning and sorting), it is an optimal and convenient pipeline to distribute our computations across a cluster of machines, thereby drastically reducing the training time of the whole experiment set.

4.4 Analysis of results

In this section we report some statistics on the data we just trained. This analysis will be useful when we perform our final benchmarking.

Figure 4.2 shows the average intensity values of the Q-tables in the training subset.

We notice that the utility values in the second and third columns are, in most cases, the highest values at each row. We interpret this as our models favouring, on average, the action of going south (second column) and east (third column) over going west or north.

Indeed, according to the reward system that we described in section 3.3.1, our trained models will try and reach the goal state with as few actions as possible. This validates why on average, the best actions to make at each state will be going south or going east.

STILL NEED TO FINISH THIS ANALYSIS

4.5 Transferable knowledge

Before proceeding, we need to start a discussion on the type of knowledge that we wish or we expect to be transferred. Having transferred knowledge is what will allow us to speed up training time on unseen tasks.

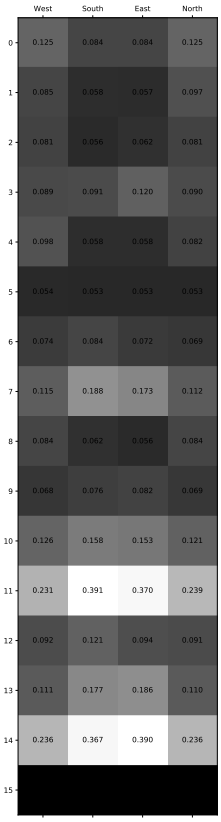


FIGURE 4.2: Average intensity values of the Q-tables in the training subset

By using Q-learning to train policies on `RandomisedFrozenLake`, we effectively train agents to go from the top-left corner to the bottom-right corner. In section 3.3.1 we adjusted the reward system so that we reward the agent more if it reaches the goal state in fewer time steps. We implicitly favour actions that go south or east. We have just seen in our results analysis how on average the agent is indeed inclined towards the south-east direction.

In this lays our expectation for transferred knowledge: we hope that this bias towards south-east directions in our distribution of training policies is captured by our Generative Adversarial Network.

If that is the case, we can therefore use this captured knowledge when we train new policies on unseen tasks in the test set of map configurations.

In fact, we expect the training process on new maps to be faster given the knowledge that in order to reach goal state, we should give priority to the south-east direction.

As an intuition, imagine there are two people in a labyrinth, looking for a treasure. The first person knows that they will likely eventually find the treasure if they keep going south-east. The other does not. We expect the first one to reach the treasure much faster. The concept of "going south-east" is the knowledge that we hope to transfer in this task domain.

4.6 Task generalisation

We explained how we hope to capture transferrable knowledge that is generalisable across similar tasks. We do so by training a GAN using the policy distribution in our training set of policies, and we show this process in the next chapter (Chapter 5).

Before we do that, we ought to talk about how this pipeline generalises on different reinforcement learning task domains. In our work we focused on the Frozen Lake environments, but the ultimate goal of our work is to develop a data pipeline that generalises to any reinforcement learning task domain.

If we hope to transfer knowledge within a domain, one key requirement is that there needs to be transferrable knowledge across these tasks. In our main experiment, we decoded this as being the action of "going south-east". However, with many and more complex domains with a larger action space and observation space, it may not be as trivial to identify such knowledge.

In our work, one of the reasons we chose Frozen Lake environments is that it is straightforward enough to verify that our agents are leveraging and benefitting from previous knowledge, since we can intuitively define and predict what this knowledge is.

But in many other applications, we do not expect to be able to even decode the transferred knowledge in words, as we do when we say "going south-east". We will leave that up to the deep neural networks and their inherent statistical foundations to model such behaviour given the distribution we pass it in an unsupervised way.

By developing our pipeline for transferring knowledge, we provide a generalisable tool that people can use on their own task domains.

Chapter 5

Adversarial Networks Training

Now that we built our dataset, we have the data we need to proceed with the adversarial training step of our pipeline.

It is once again important to place this chapter in the context of our data pipeline, as we introduced it in section 1.3 and as we showed it in figure 1.1. The dataset we just created is composed of Q-tables, which are effectively a representation of our policy. More specifically we have 3,827 such policies, corresponding to all the 3,827 different valid map configurations of our environment, `RandomisedFrozenLake`. We split the dataset in a training subset (80%) of the total number of policies, and the test subset (the remaining 20%).

In section 2.2 we introduced Generative Adversarial Networks, detailing their vanilla architecture, as they were introduced by Goodfellow et al. We now build off of this architecture to build a GAN architecture for policy generation.

In this section we present such architecture, explain the design decisions that were involved in configuring it, and describe the process of training it in an adversarial manner. We also provide a detailed breakdown of the generator and the discriminator's deep neural networks. We also make our Keras implementation of our Policy GAN publicly available.

The input to our Policy GAN will only be the training subset of the policies we trained with Q-learning. In this way, we hope to be able to model, through our GAN, a distribution on these policies such that we can capture knowledge that we can transfer to other unseen tasks. These unseen tasks are exactly the ones corresponding to the map configurations of the test subset, and we therefore do not include those test policies as our input to our GAN.

5.1 Policy GAN

In figure 5.3 we show the architecture of our Policy GAN. We already described most of the details involved when training GANs in section 2.2.

Training a GAN is an iterative process that runs for a set amount of epochs (an epoch is one full training cycle on the training set). At each iteration, the mini-batch inputs to the discriminator `D` are taken from both the real data sampled from the training data (in our case policies from the training set that we created in chapter 4), and the samples generated by the generator. Given `D`'s prediction, we then apply a gradient-based optimisation method to update both `D`'s and `G`'s parameters.

Building neural net architecture that work well in practice generally involves finding and tuning many parameters and making several design decisions that make the issue not trivial. In the following subsections, we introduce such decisions and hyperparameters choice with the aim of allowing reproducibility of our results.

As we can notice, the Generator network outputs a vector of size dimension 64, which we can then reshape to a 16x4 matrix. This is our 'fake', generated Q-table. Its input is just a the latent space vector z of size 100 whose values we can set randomly. We can interpret z as a noise vector which deterministically influences the Q-table that we generate. While we can vary z and obtain different Q-tables, these generated Q-tables should still fall within the initial distributions of Q-tables we have in our dataset, provided that the Generator was trained properly.

The Discriminator network, in turn, takes in an input a vector of size 64 (which is a reshaped 16x4 Q-table), and outputs a single scalar in $[0, 1]$ representing the predicted probability that the inputted Q-table is a generated Q-table or one sampled from the dataset.

5.1.1 Gradient descent optimisation

Learning rates and gradient descent optimisations algorithms have historically been among the trickiest hyperparameters to set, as they drastically influence the final results, perhaps more than all other hyperparameters.

Gradient descent (Lemaréchal, 2012), a seminal approach to optimisation that dates back to Cauchy a few centuries back, still proves relatively successful in many machine learning applications. Since then, a lot of work has been put in devising algorithms that have adaptive learning rates and that lead to faster convergence. In our design, we specifically explored two such adaptive learning rates algorithms: RMSProp (Hinton, Srivastava, and Swersky, 2012) and Adam (Kingma and Ba, 2014)

Both perform local optimisation with different techniques and metrics that are constructed from the history of previous interactions.

RMSProp (algorithm 3) is a modification of the AdaGrad optimiser (Duchi, Hazan, and Singer, 2011), that modifies the gradient accumulation into a moving average that is exponentially weighted.

Adam (algorithm 4) is best described as a combination of RMSProp and Stochastic Gradient Descent (SGD) with momentum (Sutskever et al., 2013). Momentum accelerates SGD by multiplying the learning rate by a parameter that increases as we go towards the right direction in the gradient update.

Like in most applications involving deep learning, there is no definite way to see which approach would yield the best results without going through the whole training process.

We therefore empirically experiment with each of these approaches to see which choice is more suitable to our data distribution and yields the best results.

"Best", in our case, is seen in the context of both the Generator network and the Discriminator network. We look for an optimisation technique that

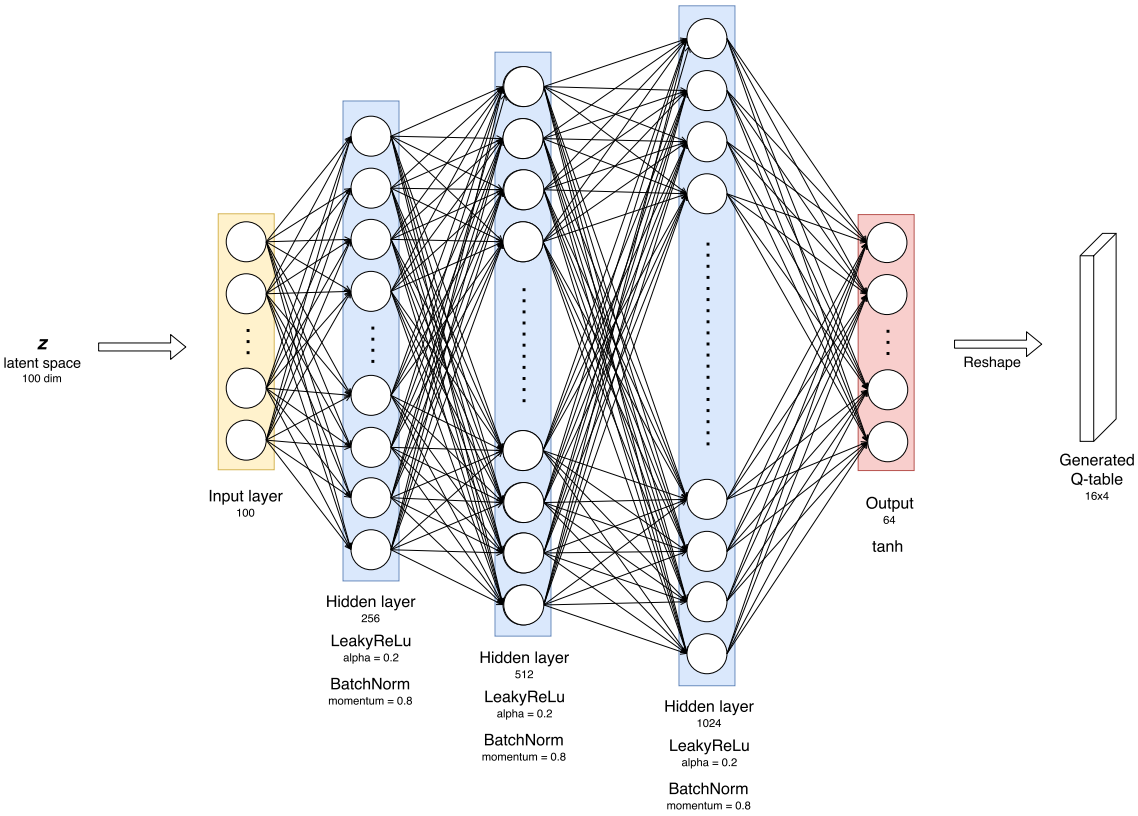


FIGURE 5.1: Architecture of the Generator network

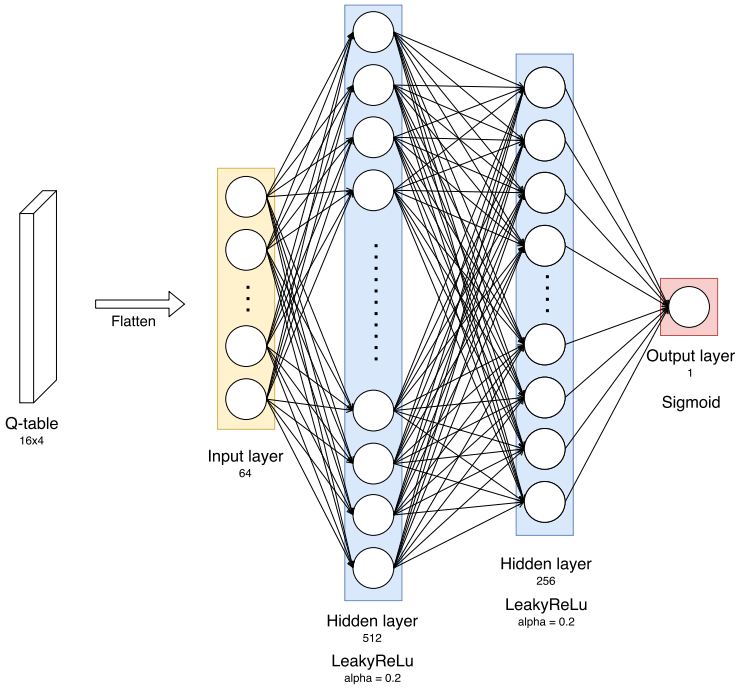


FIGURE 5.2: Architecture of the Discriminator network

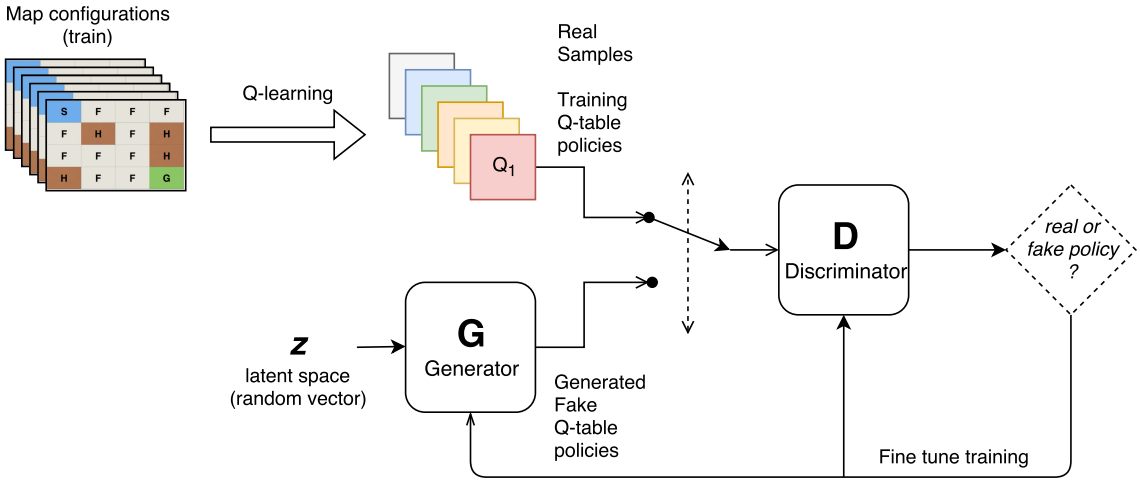


FIGURE 5.3: Our Policy GAN architecture

Algorithm 3 RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter θ
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small number
Initialize accumulation variables $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{x^{(1)}, ..., x^{(m)}\}$ with corresponding targets y^i.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i \boldsymbol{L}(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (\boldsymbol{1} - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\boldsymbol{\Delta\theta} = -\frac{\epsilon}{\sqrt{\delta + r}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + r}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

leads to stable costs for both networks. In most deep learning problems, convergence of the cost function is limited to one single neural network. In adversarial architectures like GANs, or architectures with multiple deep neural networks like Variational Autoencoders (VAE), or Actor-Critic networks, convergence of the model is not a property that we can evaluate as trivially (Kingma and Welling, 2013; Grondman et al., 2012).

We look for an asymptotic behaviour that has both networks converge to an optimum, without having one network prevail over the other. For example, we could have a Discriminator that produces perfect predictions after a few epochs of training, but that is a meaningless result if the Generator is not able to produce results that can "fool" the discriminative model.

The overall cost of the architecture, in short, must take into account both components. The cost of the whole architecture has the following closed

Algorithm 4 Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilisation (Suggested default: 10^{-8})
Require: Initial parameters θ
Initialise 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}, \mathbf{r} = \mathbf{0}$
Initialise time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{x^{(1)}, ..., x^{(m)}\}$ with corresponding targets y^i.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \boldsymbol{\nabla}_\theta \sum_i \boldsymbol{L}(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \boldsymbol{\rho_1 s} + \boldsymbol{(1 - \rho_1) g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \boldsymbol{\rho_2 r} + \boldsymbol{(1 - \rho_2) g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{\boldsymbol{1 - \rho_1^t}}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{\boldsymbol{1 - \rho_2^t}}$
Compute update: $\boldsymbol{\Delta\theta} = -\boldsymbol{\epsilon} \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{\Delta\theta}$
end while

form:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_g(z)}[log(1 - D(G(z)))]$$

With both RMSProp and Adam, we obtain a much better convergence compared to vanilla SGD. While we do not notice a noticeable difference between these two results, having adaptive learning rates clearly helps convergence for both models.

After optimising our hyperparameters with grid-search, our final optimal optimisation algorithm is the following:
Adam, step size $\epsilon = 0.0002$, exponential decay rates $\rho_1 = 0.5$, $\rho_2 = 0.999$.

5.1.2 Activation functions

Another important design decision when building neural networks pertains to the choice of activation function we use in our hidden layers.

In the context of neural network training, an activation function φ is one that we apply to the outputs of hidden layers, i.e. $\varphi(y)$ where $y = W^T x + b$.

What makes activation functions so critical is the fact that they allow us to model non-linear data distributions, enabling us to build more complex representations of our predictions.

Historically, non-linear activations functions like the logistic sigmoid function or tanh function, while successful with certain data distributions, have proven difficult to train, mostly due to their non-zero centered property and

slope of the function (Xu, Huang, and Li, 2016). Many activations functions have been introduced in machine learning literature, with some working well with many practical applications.

In this section, we provide an overview of such activation functions, all of which we evaluated in our GAN training experiments.

We trained different Policy GAN models using different combinations of activations functions for the Generator and the Discriminator's hidden layers. The activation we accounted for were the following: sigmoid function, ReLU function (Rectified Linear Unit) (Arora et al., 2016), Leaky ReLU (Xu et al., 2015), ELU or Exponential Linear Units (Clevert, Unterthiner, and Hochreiter, 2015), and SELU or Scaled Exponential Linear Units (Klambauer et al., 2017).

Following are the definitions of the considered activations functions with their respective gradients, followed, in figure 5.4, by a their visualisation on the x-y axis.

Sigmoid:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp -x} \tag{5.1}$$

ReLU:

$$\text{relu}(x) = \max(0, x) \tag{5.2}$$

Leaky ReLU:

$$\text{lrelu}(x) = \begin{cases} \alpha x & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases} \tag{5.3}$$

ELU:

$$\text{elu}(x) = \begin{cases} \alpha(\exp(x) - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases} \tag{5.4}$$

SELU:

$$\text{selu}(x) = \lambda \begin{cases} \alpha(\exp(x) - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0. \end{cases} \tag{5.5}$$

As we expected, different activations functions yield varying degree of successful results, with Leaky ReLu ($\alpha = 0.2$) producing the best overall results for both the Generator and the Discriminator model.

Using all other activations functions in our hidden layer, with the exception of the sigmoid function, yield good results. The sigmoid activation function model likely incurred incurred into the vanishing gradient problem (Pascanu, Mikolov, and Bengio, 2012), where a big amount of possible inputs are 'squashed' into a relatively small range (0, 1). Other activation functions like LeakyReLu, ELU or SELU address this problem by giving a wider range of continuous values for negative inputs.

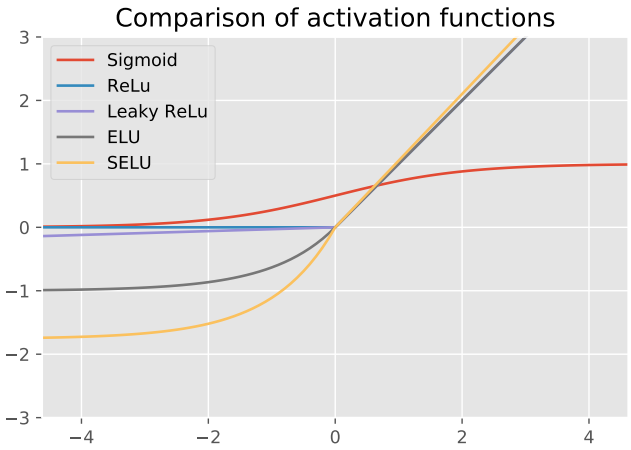


FIGURE 5.4: Plots of activation functions

While the use of the sigmoid function as activation for the hidden layers creates this issue, it is still greatly functional to model a binary classification problem. More specifically, the Discriminator network, which is effectively a binary classifier determining whether the input is 'fake' or 'real', can still make use of a sigmoid activation in its output layer to model the probability of the input being real.

5.1.3 Weight initialisation

A common easy approach to weight initialisation is to use small random values as weights, most commonly sample from a normal distribution $N \sim \mathcal{N}(0, \sigma)$. Having a 'smarter' weight initialisation, however, may lead to faster convergence when training our models.

One such approach is using sampling from the following uniform distribution:

$$W \sim U(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}) \tag{5.6}$$

where n_{in} is the number of incoming connections of each unit in the hidden layer.

Another, slightly more sophisticated approach to weight initialisation is the Glorot/Xavier initilisation (Glorot and Bengio, 2010), whereby weights are sampled from the following uniform distribution:

$$W \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right) \tag{5.7}$$

where, once again, n_{in} is the number of incoming connections of each neuron in the hidden layer. Similarly, n_{out} is the number of outgoing connections.

We experimented will all these three variations to initialise the weights in our neural network. Since there is randomness involved in these techniques,

we ran the GAN training multiple times. On average, Glorot/Xavier initialisation led to 6% fewer epochs required before both the generator and the discriminator converged.

5.1.4 Batch Normalisation

Batch Normalisation (Ioffe and Szegedy, 2015) is a technique that improves stability and performance of neural networks by normalising the inputs of each layer such that they have a mean output activation of zero and standard deviation of one. Benefits of using batch normalisation include faster training time (i.e. faster convergence to optimal model). This also allows higher learning rates and and makes weight initialisation not as critical.

We first compute the mean and variance of each hidden unit activation across the minibatch (size M):

$$\mu_i \leftarrow \frac{1}{M}\sum_{m=1}^{M} u_i^m$$
$$\sigma_i^2 \leftarrow \frac{1}{M}\sum_{m=1}^{M}(u_i^m - \mu_i)^2$$

The result of doing batch normalisation will then be:

$$u_i = w_i x$$
$$\hat{u_i} = \frac{u_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$
$$z_i = \gamma_i \hat{u_i} + \beta_i = \text{batchNorm}(u_i)$$

where γ and β are parameters updated with gradient descent that will scale and shift the normalised activations.

We trained two different models for both the Generator and the Discriminator models: one with Batch Normalisation and one without, keeping the rest of the architecture and hyperparameter settings fixed. Using Batch Normalisation in the Generator network leads to much faster convergence than the model that does not make use of it. The Discriminator network, effectively being a much less complex model than the Generator, did not benefit from batch-normalised inputs in terms of speed of model convergence and overall cost.

5.2 Final architecture and results

We report here the architectures of final optimal deep neural networks for the Generator model and the Discriminator model. Figure 5.1 and figure 5.2 show a visualisation for the two networks respectively.

Generator:

- Input: 100 dim
- Hidden Layer 1: 256 dim
  - Activation: LeakyReLu(alpha = 0.2)
  - BatchNormalisation(momentum = 0.8)
- Hidden Layer 2: 512 dim
  - Activation: LeakyReLu(alpha = 0.2)
  - BatchNormalisation(momentum = 0.8)
- Hidden Layer 3: 1024 dim
  - Activation: LeakyReLu(alpha = 0.2)
  - BatchNormalisation(momentum = 0.8)
- Output Layer: 64 dim
  - Activation: tanh

Discriminator:

- Input: 64 dim
- Hidden Layer 1: 512 dim
  - Activation: LeakyReLu(alpha = 0.2)
- Hidden Layer 2: 256 dim
  - Activation: LeakyReLu(alpha = 0.2)
- Output Layer: 1 dim
  - Activation: sigmoid

Let us look at the convergence of both networks. We plot the losses of both Generator and Discriminator in figure 5.5. The graph on the left reports the actual values for both losses. Since these tend to oscillate greatly during training (as it is typical with gradient-based optimisations), we also provide a version smoothed with IIR filtering, on the right. We notice that both losses asymptotically converge after around 1,000 epochs of training.

Initially, we observe that the Discriminator's loss (in blue) hovers just around 0. That is due to the fact that the Generator was not properly trained yet (notice the its high losses in orange). In early epochs, the Generator is likely producing unrealistic policies that the Discriminator is easily able to discern.

However, as we train the Generator a few more epochs, the Q-tables/policies that it produces tend to fit the initial distribution more, which makes its loss decrease, and the Discriminator's loss increase slightly. Both then converge after a few more hundred epochs, at around epoch 1,000.

It takes the Generator more epochs to train since it is a neural network with much higher complexity (one additional hidden layer, and more hidden units). This explains the trend of both losses that the figure reports.

5.3 Verification

It is useful to run a series of verification steps as a sanity check to the two neural networks that we just trained. While these do not give us a definite

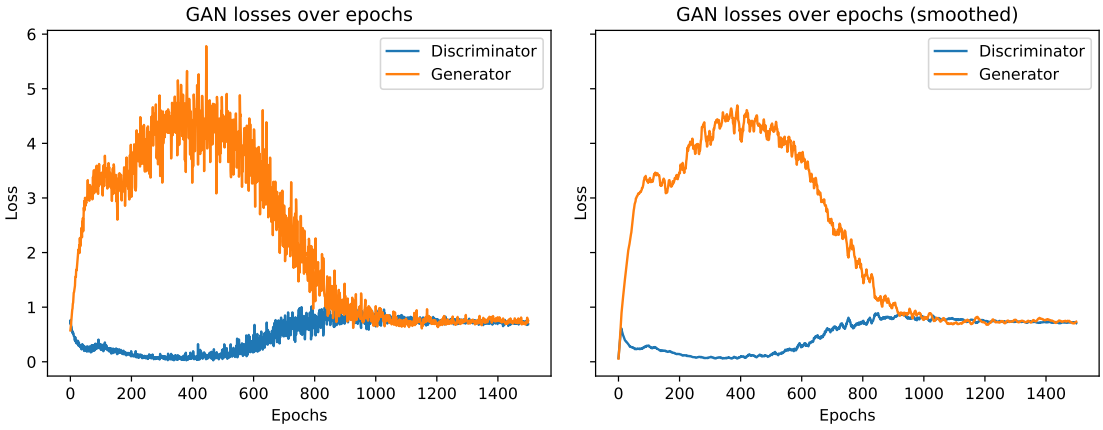


FIGURE 5.5: Generator and Discriminator losses over number of epochs

indication that the GAN was able to correctly capture and model the distribution in the training data, it is still useful to have a verification step in our pipeline.

5.3.1 Generator

Figure 4.2 back in section 4.4 showed the average intensity values of the Q-tables in the training test. As part of our verification process for the Generator, it is useful to show some samples of Q-tables that the neural network we just trained can produce.

In figure 5.6 we show sets of sampled Q-tables (16x4 matrices) from the Generator for different epochs, as we trained the GAN.

At epoch 0, we expect the Generator to produce random matrices, and that looks like the case. As we train the GAN, we produce Q-tables that will tend to fit within the initial distribution more and more.

It is still a fairly hard task to verify this visually. If we were training on the MNIST dataset of handwritten digits (0-9), we would have a clear visualisation of the digits evolving from random matrices to scribbles, to real-looking handwritten digits.

While we do not have a comparable, visual confirmation, we still notice a few things that are promising:

1. In many sampled Q-tables, the second column or third column are the ones with the highest intensity in their row. This signifies that the policies our generator is producing tend to favour the actions of "going south" and "going east".

2. Higher intensive values then accumulate in the bottom rows, since those are the closest to the goal state, and we expect the agent to have high probability of reaching the goal at that point.

3. In all Q-tables, the last row (row 16), which corresponds to the goal state, has all its values set to 0 (that is because an episode ends when

we reach the goal state, and we do not update the Q-table any further at that point).

Another way we used to verify that the Generator has been trained well for our purposes is to use sample a Q-table and check that there is a map configuration in the training set that can be solved using the policy derived from the Q-table.

Indeed, we sampled 1,000 different Q-tables, and 826 of the policies derived from these Q-tables solve at least one task from the test set. We consider the task to be solved if it achieved at least 80% of the score of the corresponding policy trained with Q-learning.

In comparison, sampling 1,000 random 16x4 matrices and deriving policies from these only yielded 23 solved tasks. These 23 tasks are the ones which have very few holes in the Frozen Lake environment, and in which it is therefore considerably easier to reach the goal state with random actions.

5.3.2 Discriminator

In this subsection we present two sanity checks on the Discriminator network.

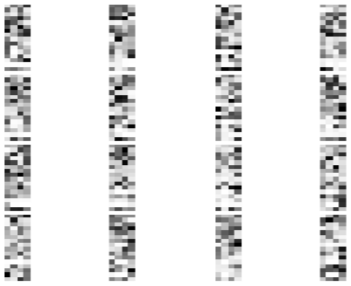
A first verification method is to run the Discriminator network with the Q-tables/policies in the test set. In other words, run a forward pass given the Q-tables in the test set as inputs.

We expect the Discriminator to label these as real Q-tables. Recall that the output of the Discriminator network is a single scalar value in $(0, 1)$ representing the probability that the input policy is real or fake. The average output predicted value for the 765 policies in the test set is 0.79, indicating that the Discriminator correctly leans towards categorising these policies as being real.

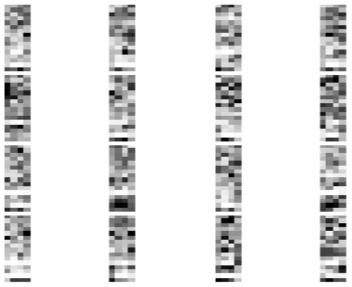
As an additional check, we also retrain the test map configurations with Q-learning, but with half as many number of episodes (5,000 instead of 10,000). With fewer simulations and iterations in Q-learning, the policies that we train will yield lower average scores. We run these 1,000 of "half-trained" policies through the Discriminator once more. We obtain an average predicted score of 0.66. This is lower than the 0.79 we obtained with the policies in the test set, and it intuitively makes sense that we now obtain a lower average predicted value, since these "half-trained" policies do not fit the distribution of "fully-trained" policies in the training data. Another set of sanity checks we conduct is to make sure that the Discriminator is not biased towards categorising any input as being real. We generate 1,000 random 16x4 matrices, and the average predicted output of the Discriminator is 0.008, which is a good sign that we have a healthy neural network.

In table 5.1 we report the scores we obtained using these three different policies.

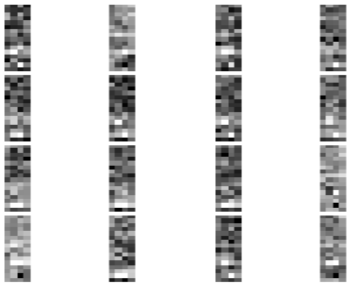
(a) **Epoch 0**



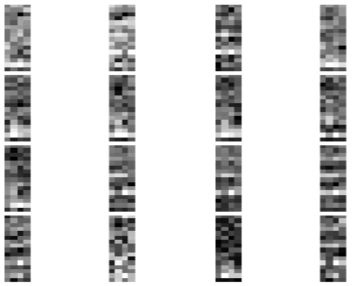
(b) **Epoch 200**



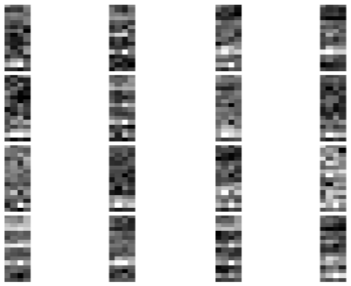
(c) **Epoch 400**



(d) **Epoch 600**

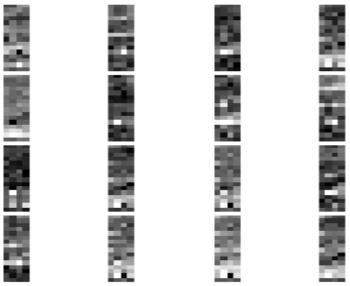


(e) **Epoch 800**



(f) **Epoch 1000**

(g) **Epoch 1200**



(h) **Epoch 1500**

FIGURE 5.6: Sampled policies produced by Generator over different epochs

Policies	Mean Discriminator Score
756 test policies	0.79
1,000 "half-trained" policies	0.66
1,000 random policies	0.008

TABLE 5.1: Mean Predicted Discriminator Scores for different policies

Chapter 6

Generative Adversarial Transfer Learning

At this point of our Data Pipeline, we have two trained deep neural networks: the Generator `G` and the Discriminator `D`. It is once again important to contextualise this work in the Data Pipeline presented in section 1.3 and as we showed it in figure 1.1.

`G` is able to generate Q-tables, which we can effectively consider our optimal policies that solve a set of similar tasks. `D` returns an indication of on the "realness" or quality of a Q-table. Both `G` and `D` hopefully capture the data distribution of the training data, which were just a subset of the Q-tables we generated in Chapter 4.

Next up in our work is to develop and propose different reinforcement learning algorithms that can leverage the knowledge that we captured through the Generative Adversarial Network. We aim to transfer such knowledge to the tasks in the test set. We hope to achieve better rewards as well as lower training times than the policies we trained with vanilla Q-learning.

The algorithms that we propose in this Chapter mostly build on the Q-learning procedure that we presented back in section 2.1.2. In the techniques that we propose, we integrate Q-learning with the Generator `G` and Discriminator `D` neural networks in different ways.

In the context of our discussion on task generalisation that we already started in section 4.6, we also need to discuss how these techniques generalise to other reinforcement learning task domains.

We empirically test all these techniques on our `RandomisedFrozenLake` environment setup, and therefore we get an indication on the successes and failures of such techniques with a bias on this environment. We report the results of each technique in the following chapter (Chapter 7 - Benchmarking).

Given this disclaimer, we nevertheless endeavour to provide a data pipeline for transfer learning using GANs that generalises as much as possible to any reinforcement learning task.

Because of this, some of the techniques we propose here have not necessarily been successful with our environment setup, but can, within reasonable expectations, be in other task domains with different transferred knowledge, environment dynamics, and data distributions of policies.

As it is common in reinforcement learning and deep learning applications, empirical exploration of different techniques is the most effective way to verify which is a best fit in the given problem.

6.1 Using the trained Discriminator

The Discriminator network `D` takes in as an input a Q-table, and outputs a single scalar in $[0, 1]$ representing the predicted probability that the inputted Q-table is a genuine one.

One way to leverage this to achieve our goal is by asking the following question:

> "How should we update the values of the Q-table that we inputted to `D`, so that if we input the updated Q-table, the score that the Discriminator returns is higher (i.e. the Q-table becomes more genuine)?"

This in fact turns out to be the same fundamental question of any gradient-based optimisation technique in machine learning. When doing vanilla gradient descent, for example, we ask ourselves a similar question: "how should we update our model's parameter set so that we minimise the loss function?", the answer to which is "by moving in the opposite direction of the loss function's gradient with respect to our parameter".

This gives us a way to find a Q-table update. The loss function in our case is how far we are from having the discriminator label the Q-table as being real, that is how far is the predicted output from a label of one: $1 - y_{predicted}$.

So what we need to calculate is the discriminator loss's gradient with respect to the Q-table, which we call $\nabla D(Q_t)$.

Recall the algorithm for Q-learning. The way we can use $\nabla D(Q_t)$ is by combining the vanilla Q-learning update step of equation 6.1 to an update with the gradient step.

Algorithm 5 Q-learning with gradient update

Require:

S is a set of states
A is a set of actions
γ the discount reward factor
α is the learning rate
n is number of episodes to run Q-learning
ϵ, probability to take random action, rather than follow policy
procedure Q-LEARNING
Initialize $Q(s, a)$ will all 0 utility values.
    for each episode e_i with $i = 0...n$ **do**
      Initialize s
        for each step of episode **do**
          Choose a_t from s_t using policy derived from Q with ϵ-Greedy
          Take action a_t, observe reward r and s_{t+1}
          Update Q-table using $\nabla D(Q_t)$
        end for
    end for
end procedure

6.1.1 Baseline approach: Q-learning

This is the basic update step of vanilla Q-learning.

$$Q(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \times \times \left[\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right] \tag{6.1}$$

6.1.2 Approach 1: Update whole Q-table towards $\nabla D(Q_t)$

In this approach we update whole Q-table in the direction of the gradient. We precede this with a vanilla Q-learning update step.

$$Q_{t+1} = Q_t - \underbrace{\beta}_{\text{discriminator learning rate}} \times \underbrace{\nabla D(Q_t)}_{\text{gradient of D at } Q_t} \tag{6.2}$$

6.1.3 Approach 2: Update state/action value pair towards $\nabla D(Q_t)$

This approach is similar to the one before, but just like Q-learning, we only update the state/action pair of the step the agent just took in the environment. This approach combines Q-learning's update approach with our gradient-based update.

$$\begin{aligned} Q(s_t, a_t) = {} & \underbrace{Q(s_t, a_t)}_{\text{old value}} + \\ & + \underbrace{\alpha}_{\text{learning rate}} \times \left[\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right] + \\ & - \underbrace{\beta}_{\text{discriminator learning rate}} \times \underbrace{\nabla D(Q)(s_t, a_t)}_{\text{gradient of D at } Q_t} \end{aligned} \tag{6.3}$$

6.1.4 Approach 3: Update state/action row towards $\nabla D(Q_t)$

Similar to the approaches above, but updates the whole row of actions for the state the agent is in.

$$
\begin{aligned}
&\forall a \in A : \\
Q(s_t, a) = &\underbrace{Q(s_t, a_t)}_{\text{old value}} + \\
&+ \underbrace{\alpha}_{\text{learning rate}} \times \Bigg[\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg] + \\
&- \underbrace{\beta}_{\text{discriminator learning rate}} \times \underbrace{\nabla D(Q)(s_t, a_t)}_{\text{gradient of D at } Q_t}
\end{aligned}
\tag{6.4}
$$

6.2 Using the trained Generator

We can categorise Q-learning and the approaches that we just presented as being "local search" in the space of possible policies that solve the task. That is, we search over possible Q-tables/policies by just applying local changes when we update the Q-table at each time step.

Another heuristic method for optimisation problems is global search, that is searching all possible solutions and choosing the one that maximises the achieved rewards.

6.2.1 Approach 1: Global Search

Global search is effectively a bruteforce method to obtain a policy that that solves a task. While this may seem like a non-optimal approach, if we already have a distribution of potential policies, global search may be the least computationally expensive approach. In fact, we can try and sample a fixed number of Q-tables/policies from the Generator network, and check the rewards that each of these policies yields.

6.2.2 Approach 2: Local/Global search hybrid

The issue with global search, is that the solutions that we may find, while yielding good rewards, are not necessarily optimised for the specific task at hand. We could be achieving better overall rewards if we did some local search using the Q-table we found through global search.

To do that we can do our global search on a fixed number of sampled policies, and choose the best Q-table. Now, to do local search, we can choose any of approaches we introduced in the previous section, but instead of initialising the Q-table with 0 values, we initialise it as the Q-table we just found with global search.

Algorithm 6 Global search and Q-learning with gradient update

Require:

p number of policies to sample for global search
S is a set of states
A is a set of actions
γ the discount reward factor
α is the learning rate
n is number of episodes to run Q-learning
ϵ, probability to take random action, rather than follow policy
procedure GLOBAL SEARCH
    for i in $i = 0...p$ **do**
        Sample Q-table from the Generator
        Evaluate the Q-table/policy
    end for
Let Q_{best} be the Q-table that yields the best policy
end procedure
procedure Q-LEARNING
Initialize Q = Q_{best}
    for each episode e_i with $i = 0...n$ **do**
        Initialize s
        for each step of episode **do**
            Choose a_t from s_t using policy derived from Q with ϵ-Greedy
            Take action a_t, observe reward r and s_{t+1}
            Update Q-table using $\nabla D(Q_t)$
        end for
    end for
end procedure

Chapter 7

Benchmarking

Chapter 8

Conclusion

Bibliography

Add ability to generate random frozen lake maps by gvsi · Pull Request #835. https://github.com/openai/gym/pull/835. (Accessed on 01/26/2018).

Arora, Raman et al. (2016). "Understanding Deep Neural Networks with Rectified Linear Units". In: *CoRR* abs/1611.01491. arXiv: 1611.01491. URL: http://arxiv.org/abs/1611.01491.

Arulkumaran, Kai et al. (2017). "A Brief Survey of Deep Reinforcement Learning". In: *CoRR* abs/1708.05866. arXiv: 1708.05866. URL: http://arxiv.org/abs/1708.05866.

Bellemare, Marc G et al. (2013). "The Arcade Learning Environment: An evaluation platform for general agents." In: *J. Artif. Intell. Res.(JAIR)* 47, pp. 253–279.

Bengio, Yoshua et al. (2009). "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 41–48.

Berns, Karsten, Rüdiger Dillmann, and U Zachmann (1992). "Reinforcement-learning for the control of an autonomous mobile robot". In: *Intelligent Robots and Systems, 1992., Proceedings of the 1992 lEEE/RSJ International Conference on*. Vol. 3. IEEE, pp. 1808–1815.

Brockman, Greg et al. (2016). *OpenAI Gym*. eprint: arXiv:1606.01540.

Caruana, Rich (1998). "Multitask learning". In: *Learning to learn*. Springer, pp. 95–133.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *CoRR* abs/1511.07289.

Dean, Jeffrey and Sanjay Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

Du, Ding-Zhu and Panos M Pardalos (2013). *Minimax and applications*. Vol. 4. Springer Science & Business Media.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

Goodfellow, Ian et al. (2014). "Generative adversarial nets". In: *Advances in neural information processing systems*, pp. 2672–2680.

Grondman, Ivo et al. (2012). "A survey of actor-critic reinforcement learning: Standard and natural policy gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6, pp. 1291–1307.

Hinton, Geoffrey, Nitish Srivastava, and Kevin Swersky (2012). *Neural networks for machine learning lecture 6a overview of mini-batch gradient descent*.

Ho, Jonathan and Stefano Ermon (2016). "Generative adversarial imitation learning". In: *Advances in Neural Information Processing Systems*, pp. 4565–4573.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

Jaderberg, Max et al. (2016). "Reinforcement learning with unsupervised auxiliary tasks". In: *arXiv preprint arXiv:1611.05397*.

Kaelbling, Leslie Pack, Michael L Littman, and Andrew W Moore (1996). "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4, pp. 237–285.

Kingma, Diederik P. and Jimmy Ba (2014). "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980. arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

Kingma, Diederik P and Max Welling (2013). "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114*.

Klambauer, Günter et al. (2017). "Self-Normalizing Neural Networks". In: *CoRR* abs/1706.02515.

Kohl, Nate and Peter Stone (2004). "Policy gradient reinforcement learning for fast quadrupedal locomotion". In: *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*. Vol. 3. IEEE, pp. 2619–2624.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Lemaréchal, Claude (2012). "Cauchy and the gradient method". In: *Doc Math Extra* 251, p. 254.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, p. 529.

Mnih, Volodymyr et al. (2016). "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*, pp. 1928–1937.

Monahan, George E (1982). "State of the art—a survey of partially observable Markov decision processes: theory, models, and algorithms". In: *Management Science* 28.1, pp. 1–16.

Ng, Andrew Y et al. (2006). "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX*. Springer, pp. 363–372.

OpenAI Gym. https://gym.openai.com/envs/FrozenLake-v0/. (Accessed on 01/25/2018).

openai/gym on Gihub. https://github.com/openai/gym/tree/master/gym. (Accessed on 01/25/2018).

Pan, Sinno Jialin and Qiang Yang (2010). "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10, pp. 1345–1359.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2012). "Understanding the exploding gradient problem". In: *CoRR* abs/1211.5063.

Radford, Alec, Luke Metz, and Soumith Chintala (2015). "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434*.

Schulman, John et al. (2015). "Trust region policy optimization". In: *International Conference on Machine Learning*, pp. 1889–1897.

Shvachko, Konstantin et al. (2010). "The hadoop distributed file system". In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, pp. 1–10.

Silver, David et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, pp. 484–489.

Singh, Satinder et al. (2002). "Optimizing dialogue management with reinforcement learning: Experiments with the NJFun system". In: *Journal of Artificial Intelligence Research* 16, pp. 105–133.

Strehl, Alexander L et al. (2006). "PAC model-free reinforcement learning". In: *Proceedings of the 23rd international conference on Machine learning*. ACM, pp. 881–888.

Sutskever, Ilya et al. (2013). "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*, pp. 1139–1147.

Tesauro, Gerald (1995). "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3, pp. 58–68.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8.3-4, pp. 279–292.

Xu, Bing, Ruitong Huang, and Mu Li (2016). "Revise Saturated Activation Functions". In: *CoRR* abs/1602.05980. arXiv: 1602.05980. URL: http://arxiv.org/abs/1602.05980.

Xu, Bing et al. (2015). "Empirical Evaluation of Rectified Activations in Convolutional Network". In: *CoRR* abs/1505.00853.

Zamora, Iker et al. (2016). "Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo". In: *arXiv preprint arXiv:1608.05742*.