Generative Adversarial Networks for generating novel policies and rewards in Reinforcement Learning

Pablo de Jesús Campos Viana

THE UNIVERSITY *of* EDINBURGH

Master of Science

Operational Research with Data Science

School of Mathematics

2018

Abstract

In the Reinforcement Learning (RL) framework, two main elements are given by the *policies*, which define some agent's way of behaving, and the *rewards*, which define the goals in a RL problem. Besides, Generative Adversarial Networks (GANs) [Goodfellow et al., 2014] are a class of generative models that have been regarded as the most interesting idea in the last years in the Machine Learning (ML) field, that are capable of produce realistic data in a wide range of domains. Nevertheless, the use of GANs in the context of RL remains a little explored research area. In this project we explore the idea of applying GANs in the context of RL in order to generate policies and rewards and also the idea of accelerating new learning by using the learned generative models. Using a simple RL environment, we show that it is completely possible to train GANs in order to be able to produce realistic policies and rewards that can be used independently to eventually accelerate new learning.

Acknowledgements

First of all, I would like to thank my best friend and partner Andry for all the priceless support and love throughout this exciting and difficult year. I thank my parents and my brother for their invaluable support and encouragement everyday.

I would like to thank my supervisor, Dr. Timothy Hospedales, for the opportunity to undertake this challenging and passionate project and for his excellent and inspiring supervision.

Finally, I thank Dr. Julian Hall, Dr. Mahdi Doostmohammadi and Dr. Sergio García Quiles for their guidance and support from the first to the last day of my postgraduate studies at the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pablo de Jesús Campos Viana*)

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In the Reinforcement Learning (RL) framework, two main elements are given by the *policies*, which define some agent's way of behaving, and the *rewards*, which define the goals in a RL problem. Besides, Generative Adversarial Networks (GANs) [Goodfellow et al., 2014] are a class of generative models that have been regarded as the most interesting idea in the last years in the Machine Learning (ML) field [1], that are capable of produce realistic data in a wide range of domains. In this project we explore the idea of applying GANs in the context of RL in order to generate policies and rewards and also the idea of accelerating new learning by using the learned generative models. We can highlight motivations in each case.

- **Policies**: Suppose that a simulated robot arm has learned models (i.e. trajectories) for how to touch different objects in the room, or how to throw objects at different targets; or suppose that a group of trained intelligent agents have slightly different strategies for playing games such as tic-tac-toe. The idea is to apply a GAN to model this space of models and sample novel policies. Aside from qualitatively observing the new behaviours that the robot or the intelligent agent can perform, one of the most exciting applications is to use it as a prior to accelerate future model learning. This is significant because the sample-inefficiency of policy learning is one of the biggest barriers in Deep Reinforcement Learning today.

- **Rewards**: In the RL framework, agents tasks correspond to reward functions. Each new task defined by the user requires the agent to learn to optimize it,

[1] `https://tinyurl.com/y9ozf4vu`

which costs time and samples (i.e. environmental interactions, causing wear and tear on the robot). Besides, in the Inverse Reinforcement Learning (IRL) framework, agents try to learn the reward function. Learning a generative model on past learned reward functions would allow the agent to propose its own reward functions, and learn to optimize them, in advance of being requested to solve it by the user. Therefore it is natural to ask if whether building a Deep Reinforcement Learning agent which can make up its own goals, and learn to solve them can eventually learn faster and better than a conventional agent that waits for the user to impose new tasks (i.e. reward functions) on it.

1.2 Contributions

In this work we evaluate the potential of an approach to policy and reward function generation based on adversarial training via GANs.

In summary, in this work we make the following contributions:

1. We identify and study recent approaches related to the general idea of connecting generative modeling via GANs with RL (or IRL) approaches.

2. We propose, implement and evaluate a proof-of-concept method for both policies and reward functions generation based on adversarial training.

3. We explore the idea of accelerating new learning by using the learned generative models for both policies and reward functions.

1.3 Dissertation Outline

Further chapters are organized in the following way. In Chapter 2 we give a brief overview regarding both Reinforcement Learning and Deep Learning frameworks, as well as the Deep Reinforcement Learning framework where both worlds are combined, and we introduce the main concepts used and applied in this work regarding both frameworks. In Chapter 3 we outline the overall approach and introduce methods that we evaluate in this work. In Chapter 4 we introduce the evaluation setup, including training settings and our approach to evaluation. Finally, in Chapter 5 we summarize our findings, and discuss potential directions for further work. Furthermore, in Appendix A we provide some details of implementation and the hardware we used for experimentation.

Chapter 2

Background

In this chapter, we give a literature review about the concepts and methods that hold a strong connection with the topic of this dissertation. The review begins with Reinforcement Learning giving particular importance to both the reward functions and policies, then moves to Deep Learning, giving special importance to Generative Adversarial Networks. Further we give a brief review of Deep Reinforcement Learning, the framework where both worlds are combined. Finally, we mention some works related to the main ideas of this dissertation.

2.1 Reinforcement Learning

2.1.1 Brief Overview

This section is primarly based on the introductory chapters of [Sutton et al., 1999]. Reinforcement Learning (RL) can be regarded as an area of Machine Learning (ML) where an environment is reactive to the decision of a learner agent, and it is about learning from interaction how to behave in order to achieve a goal. Beyond the agent and the environment, the main elements of a RL system are given by:

- **Policy**: Defines the agent's way of behaving. This is, a mapping from perceived states of the environment to actions at a given time and it is alone sufficient to determine behavior.
- **Reward function**: Defines the goal in a RL problem. On each time step, the environment sends to the agent a numeric signal called the *reward* and this implies that it defines what are the good and bad events for the agent. Furthermore, the

reward signal is the main mean for altering the policy; the policy can be changed to select actions that encourage rewards with higher values.

- **Value function**: Specifies what is good in the long run. It can be described as the total amount of reward an agent can expect to accumulate over the future when starting from that state. While rewards determine the immediate desirability of environmental states, values indicate their *long-term* desirability.

- **Model of the environment**: Mimics the behavior of the environment and allows to make inferences about how the environment will behave. Models are used for *planning*, this is, a way of deciding a course of action by considering possible future situations before they are actually experienced. Models for solving RL problem that use models of the environment and planning are called *model-based*, while *model-free* methods are those that explicitly use trial-and-error to learn. In this dissertation we are exclusively focused on the latter ones.

The mathematical framework used to formally describe RL problems is Markov Decision Processes.

2.1.2 Markov Decision Processes

Markov Decision Processes (MDPs) constitute a mathematical formulation of sequential decision making, where actions influence inmediate rewards as well as subsequent situations, given by future states and future rewards. They are meant to be a framework of the problem of learning from interaction to achieve a given goal. In this context, the decision maker is called the *agent* and the object it interacts with is called the *environment*. These two elements interact continually in such a way where the agent selects actions and the environment responds to these actions and presents new situations to the agent, including the rewards which represent numerical values that the agent seeks to maximize over time. An example of this interaction can be seen in the figure 2.1.

More formally, RL can be described as a MDP which consists of:

- A set of states $\mathcal{S}$, plus a distribution of starting states $p(\mathbf{s}_0)$.

- A set of actions $\mathcal{A}$.

- Transition dynamics $\mathcal{T}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ that map a state-action pair at time t onto a distribution of states at time $t+1$.

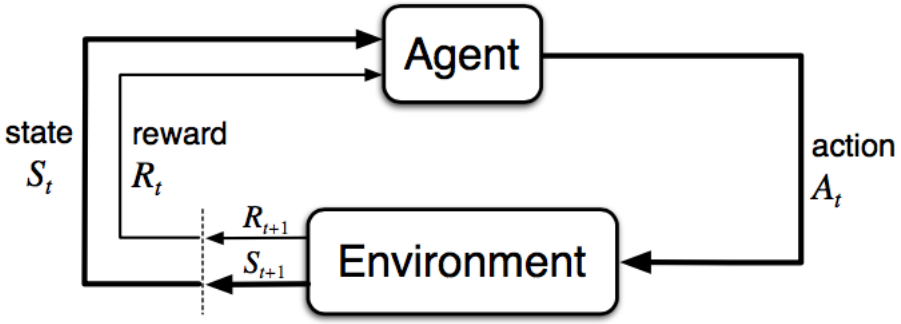


Figure 2.1: Interaction between the agent and the environment in a Markov Decision Process. Given that the agent is in state S_t and has received a reward R_t at time t, the agent executes action A_t and the environment responds by placing the agent on the new state S_{t+1} and giving a reward R_{t+1} at time $t+1$.

- An (instantaneaous) reward function $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$.
- A discount factor $\gamma \in [0, 1]$, where lower values place more emphasis on immediate rewards and greater values on long term rewards.

Furthermore, the policy π is defined as a mapping from states to a probability distribution over actions: $\pi : \mathcal{S} \rightarrow p(\mathcal{A} = \mathbf{a}|\mathcal{S})$; in particular, a policy can be deterministic: $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The MDP is said to be *episodic* if the state is reset after each episode of length T, and the sequence of states, actions and rewards in an episode defines a *trajectory* of the policy. This way, every trajectory of a policy accumulate rewards from the environment resulting in the return defined as $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$. The goal of RL therefore is to find an optimal policy π^* such that gives the maximum expected return from all states, i.e.

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \quad \mathbb{E}[R|\pi] \tag{2.1}$$

When the MDP is *non-episodic* (i.e. $T = \infty$), a discount factor $\gamma < 1$ is required in order to prevent an infinite sum of rewards. A crucial concept in the framework of MDPs and therefore in RL problems modeled as MDPs is the *Markov Property*, which refers to the fact that the future is conditionally independent of the past given the present state. It should be noted that this assumption is held by the majority of RL algorithms, though it requires the states to be *fully observable*, this is, that the state of a system is known at all times.

2.1.3 Algorithms

In the following we present the classic RL algorithms and the underlying ideas to them. They can be broadly classified into two main approaches: those based on *value functions* and those based on *policy search*. This section is mainly based on [Arulkumaran et al., 2017] and [Li, 2017].

2.1.3.1 Value functions

Value functions methods rely on the idea of estimating the expected return (the *value*) of being in a given state. The *state-value function* $V^\pi(\mathbf{s})$ is therefore defined as the expected return when starting in state $\mathbf{s}$ and following the policy π:

$$V^\pi(\mathbf{s}) = \mathbb{E}[R|\mathbf{s}, \pi] \tag{2.2}$$

The optimal policy, denoted by π^*, has a corresponding optimal state-value function $V^*(\mathbf{s})$ and therefore this one can be defined as:

$$V^*(\mathbf{s}) = \max_\pi V^\pi(\mathbf{s}) \quad \forall \mathbf{s} \in \mathcal{S} \tag{2.3}$$

Given that the transition dynamics are unknown $\mathcal{T}$, we instead use the *state-action-value* or *quality* $Q^\pi(\mathbf{s}, \mathbf{a})$, which is similar to V^π with the difference that the initial state $\mathbf{a}$ is provided and the policy π is followed from the succeeding state onwards, i.e.

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}[R|\mathbf{s}, \mathbf{a}, \pi] \tag{2.4}$$

Therefore, the best policy given $Q^\pi(\mathbf{s}, \mathbf{a})$ can be found by choosing the action $\mathbf{a}$ in a greedy fashion at every state: $\text{argmax}_\mathbf{a} Q^\pi(\mathbf{s}, \mathbf{a})$. We must also note that $V^\pi(\mathbf{s})$ can be defined by maximizing the quality function: $V^\pi(\mathbf{s}) = \max_\mathbf{a} Q^\pi(\mathbf{s}, \mathbf{a})$.

Among value functions based methods, the simplest family of methods are those based on **Dynamic Programming**. By exploiting the Markov property and defining the quality function as a Bellman equation, it takes the following recursive form:

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}[r_{t+1} + \gamma Q^\pi(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}))] \tag{2.5}$$

This implies that Q^π can be improved by means of *bootstrapping*, this is, using the current values of our estimate of Q^π to improve our estimate. This is the main idea underlying the Q-learning algorithm and the state-action-reward-state-action (SARSA) algorithm, where the update rule takes the following form:

$$Q^{\pi}(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q^{\pi}(\mathbf{s}_t, \mathbf{a}_t) + \alpha\delta \tag{2.6}$$

where α is some learning rate and $\delta = Y - Q^{\pi}(\mathbf{s}_t, \mathbf{a}_t)$ represents the temporal difference (TD) error and Y refers to a target value, just as in a standard regression problem.

On the one hand, SARSA uses transitions generated by the behavioural policy (the one derived from Q^{π}) and its target is given by $Y = r_t + \gamma Q^{\pi}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$ (i.e. it is an *on-policy* algorithm). On the other hand, Q-learning uses transitions that were not necessarily generated by the derived policy, and its target is given by $Y = r_t + \gamma \max_{\mathbf{a}} Q^{\pi}(\mathbf{s}_{t+1}, \mathbf{a})$ which directly approximates Q^* (i.e. it is an *off-policy* algorithm).

In order to find Q^* from an initial and perhaps arbitrary Q^{π}, *Generalized Policy Iteration* (GPI) is performed, which consists on two stages: *policy evaluation* and *policy iteration*. The former improves the estimate of the value function and the latter is improved by choosing actions greedily according to the updated value function. GPI therefore consists in alternating both stages such that progress can be done more rapidly, instead of performing them separately.

Another popular value function based method consists on learning an *advantage function* $A^{\pi}(\mathbf{s}, \mathbf{a})$, which rather than produce state-action values, it represents relative states-action values in the sense that it learns that one action has better consequences than others. This way, A^{π} represents a relative advantage of actions through the relationship: $A^{\pi} = Q^{\pi} - V^{\pi}$.

The value functions methods presented so far can work by using a *lookup table*, in the sense that every state $\mathbf{s}$ has an entry $V(\mathbf{s})$, or every state-action pair $(\mathbf{s}, \mathbf{a})$ has an entry $Q(\mathbf{s}, \mathbf{a})$. However, when dealing with large MDPs there could be two main problems: large memory requirements and slow learning of each state or state-action pair individually. In this case, the solution is to estimate the value function using *function approximation*, this is, $\hat{V}(\mathbf{s}, w) \approx V_{\pi}(\mathbf{s})$ or $\hat{Q}(\mathbf{s}, \mathbf{a}, w) \approx Q_{\pi}(\mathbf{s}, \mathbf{a})$, where the function approximator is usually required to be differentiable in order to perform gradient based optimization methods. Typical examples of approximators include linear combination of features and more generally, neural networks. One of the main advantages of this approach is the ability to generalise from seen states to unseen states.

2.1.3.2 Sampling Strategies

Rather than bootstrapping value functions using dynamic programming methods, Monte Carlo methods compute the expected return in expression 2.2 from a given state by averaging the return from multiple trajectories of a policy and can only be used in episodic MDPs given that a trajectory has to terminate in order to compute the return. By combining the bootstrapping strategy with the sampling strategy of Monte Carlo methods, is possible to get the best of both approaches in the Temporal Difference Learning framework. The most prominent example is the TD(λ) where the λ parameter is used to interpolate between Monte Carlo evaluation and bootstrapping, resulting in a wide spectrum of RL methods which depends on the amount of sampling performed.

2.1.3.3 Policy Search

Policy search methods directly search for an optimal policy π^* and do not need to maintain a value function model. Usually, a parametrized policy π_θ is used and its parameters are updated in order to maximize the expected return $\mathbb{E}[R|\theta]$. When searching for the policy directly, it is common to output parameters for a probability distribution and the result use to be a stochastic policy from which actions can be sampled. Under this framework we can also distinguish between gradient-free and gradient based methods, though the latter ones are more frequently used in large-scale settings.

Gradient based methods in this context, also known as *policy gradient* methods, provide a strong learning signal by improving a parameterized policy. Similar to value function approximation, typical examples of parametrized policies include the use of linear combination of features that are turned into probabilities via the softmax function, and more generally, neural networks that use a softmax activation as output layer.

If we let $J(\theta)$ be any policy objective function (in particular, $J(\theta) = \mathbb{E}_{\pi_\theta}[R]$), policy gradient algorithms search for a local maximum in $J(\theta)$ by ascending the gradients of the policy with respect to parameters θ, and the update rule is given by $\Delta\theta = \eta\nabla_\theta J(\theta)$, where $\nabla_\theta J(\theta)$ is called the *policy gradient* and η is some learning rate parameter.

In order to compute the expected return in expression 2.1, an average of plausible trajectories induced by the current policy parametrization needs to be performed. In the model-free setting, a Monte Carlo estimate of the expected return is usually computed. Nevertheless, in the Monte Carlo estimate the gradients cannot pass through these samples of a stochastic function so an estimator of the gradient is used instead an it is known as the REINFORCE rule [Williams, 1992]. Using this estimator is similar to optimize

the log-likelihood in supervised learning; performing gradient based optimization on this estimator have the effect of increasing the log probability of the sampled action, weighted by the return. The REINFORCE rule computes the gradient of an expectation over a function f of a random variable X with respect to parameters θ in the following form:

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \mathbb{E}_X[f(X;\theta)\nabla_\theta \log p(X)] \tag{2.7}$$

Given that in our context this formula would depend on the empirical return of a single trajectory, the computed gradients suffer from high variance. However, it is possible to employ variance reduction methods by introducing unbiased and less noisy estimates. Usually this is achieved by subtracting a baseline, which implies that the updates are weighted by an advantage rather than the pure return.

2.1.3.4 Actor-Critic Methods

Actor-critic (AC) methods constitute a family of algorithms where both the value function and an explicit representation of the policy are learnt. In this sense, these algorithms lie at the intersection of value functions based methods and policy search based methods. In this framework, the *actor* (policy) learns by using feedback from the *critic* (value function). AC methods can be seen as well as a subset of policy gradient methods where a learned value function is used as baseline. A diagram illustrating the general idea of these methods is shown in figure 2.2.

The critic is used to estimate the state-action-value function, i.e. $Q_w(\mathbf{s},\mathbf{a}) \approx Q^{\pi_\theta}(\mathbf{s},\mathbf{a})$, and therefore these kind of algorithms mantain two sets of parameters:

- **Critic**: Updates state-action-value function parameters w.
- **Actor**: Updates policy parameters θ, in direction suggested by the critic.

In particular, both the actor and the critic can be modeled as deep neural networks.

2.1.4 Inverse Reinforcement Learning

In most RL problems there is no natural source for the reward signal and therefore, the reward function has to be carefully designed to accurately represent the task. Usually, researchers and engineers manually tweak the rewards of the RL agent until desired behavior is observed. Another approach is to observe a human expert performing the

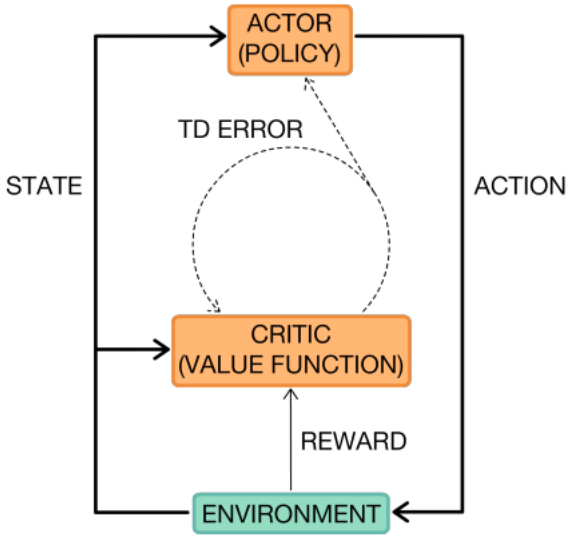


Figure 2.2: Actor-Critic framework. The *actor* (policy) receives a state from the environment and chooses an action to take. Also, the *critic* (value function) receives the state and reward resulting from the previous interaction, using the TD error computed from this information to update itself and the actor.

task in order to then automatically extract the respective rewards from these observations. Nevertheless, both of these approaches can be computationally expensive and time consuming.

The goal of inverse reinforcement learning (IRL), also known as inverse optimal control, is to infer the reward function or cost function underlying demonstrated behavior. It is typically assumed that the demonstrations come from an expert who is behaving near-optimally under some unknown reward function. The main motivation for the development of IRL is that the entire field of RL is founded on the presupposition that the reward function, rather than the policy, is the most robust and transferable definition of a task.

The first work to tackle the problem of IRL was [Ng and Russell, 2000], where a linear programming approach is followed. After that, there has been a rise of more modern approaches; a brief summary of the most popular of them is as follows:

- Apprenticeship Learning via IRL [Abbeel and Ng, 2004]
- Bayesian IRL [Ramachandran and Amir, 2007]
- Maximum Entropy IRL [Ziebart et al., 2008]
- Maximum Causal Entropy IRL [Ziebart et al., 2010]

- Maximum Entropy Deep IRL [Wulfmeier et al., 2015]

2.2 Deep Learning

2.2.1 Brief Overview

Deep Learning (DL) refers to a wide class of ML techniques and architectures with the characteristic of using several layers of non-linear information processing that are hierarchical. The essence of DL is to compute hierarchical features or representations of the observational data, where the higher-level features or factors are defined from lower-level ones. Depending on how the architectures and techniques are intended for use, a broad categorization of work done in this area is as follows:

- **Deep networks for unsupervised learning**: Intended to capture high-order correlation of the observed data for pattern analysis or synthesis purposes when no information about target class labels is available.
- **Deep networks for supervised learning**: Intended to directly provide discriminative power for pattern classification purposes. Target label data are always available in direct or indirect forms for such supervised learning.
- **Hybrid deep networks**: Designed in such a way that the goal is discrimination which is assisted, often in a significant way, with the outcomes of generative deep networks; or viceversa, this is, the goal is generation which is assisted with the outcomes of discrminative deep networks. The kind of deep networks used in this project, namely Generative Adversarial Networks, belong to this category.

The quintessential example of a DL model is the feedforward deep network, also known as multilayer perceptron (MLP). A MLP is a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions. Each application of a different mathematical function can be thought as providing a new representation of the input. An illustration of this idea is shown in figure 2.3.

2.2.2 Some Breakthroughs

In this section we highlight some of the breakthroughs in the DL research that were used along this project.

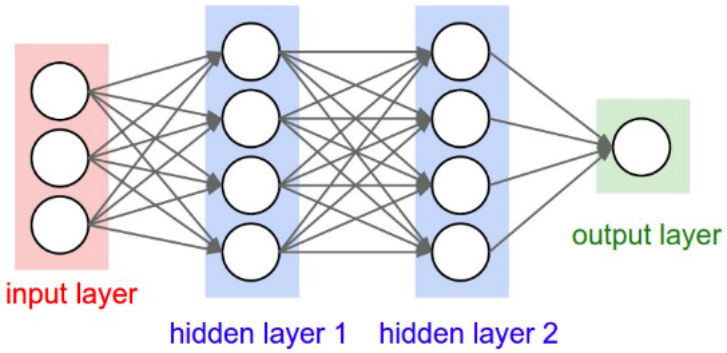


Figure 2.3: Example of a multilayer perceptron with input layer consisting of three inputs, two hidden layers of 4 neurons each and one output layer.

2.2.2.1 Activation Functions

The *sigmoid* activation function [Han and Moraga, 1995] is one of the earliest activation functions used in neural networks. It is defined as $f(t) = \frac{1}{1+e^{-t}}$, $t \in \mathbb{R}$. The main reason to use the sigmoid activation is because its range is the interval $(0, 1)$ and therefore, it is well suited to problems where the aim is to predict the probability as an output. Its generalization is the *softmax* activation function and its output can be used to represent a probability distribution over an arbitrary number of different possible outcomes.

On the other hand, the *hyperbolic tangent* (tanh) [Ramachandran et al., 2017] function has been also a common choice to use in neural networks. It is defined as $f(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$, $t \in \mathbb{R}$ and its range is the interval $(-1, 1)$, allowing negative inputs to be mapped strongly negative and positive inputs to be mapped strongly positive.

Most recently, the *rectified linear unit* (ReLU) [Nair and Hinton, 2010] has been one of the most widely used activations. It is defined as $f(t) = \max(0, t)$, $t \in \mathbb{R}$ on the outputs of a layer. It has been demonstrated to enable better training of deeper networks compared to the widely previous used activation functions, namely the sigmoid and hyperbolic tangent functions. Furthermore, many variants of this activation has been proposed. Among these, the *leaky* ReLU (LReLU) [Maas et al., 2013] is the most known and attempts to fix an issue of original ReLU activation, in which neurons can sometimes be pushed into states in which they become inactive for essentially all inputs and therefore no gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state. Hence the LReLU activation allows a small, positive gradient when the unit is not active, and they are defined as:

$$f(t) = \begin{cases} t & \text{if } t > 0 \\ ax & \text{otherwise} \end{cases} \tag{2.8}$$

where $t \in \mathbb{R}$ and a is usually set to $a = .01$.

2.2.2.2 Learning Rules

Gradient descent (GD) is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. It is a method to minimize an objective function $J(\theta)$ parameterized by a model's parameters θ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters. The learning rate η determines the size of the steps we take to reach a local minimum.

Some of the most important variants used in neural networks training and used along this project are the following.

- **Minibatch Gradient Descent** [Bottou, 2010]:

  While the standard GD computes the gradient of the cost function with respect to the parameters for the entire training dataset with the update rule $\theta_t = \theta_{t-1} - \eta\nabla_\theta J(\theta)$ and stochastic gradient descent (SGD) computes the gradient of the cost for each randomly selected training example $s^{(i)}$ with the update rule $\theta_t = \theta_{t-1} - \eta\nabla_\theta J(\theta; s^{(i)})$, the minibatch SGD is a variant that splits the training dataset into small batches called *minibatches* and performs an update for every minibatch of randomly selected n training examples of the form:

  $$\theta_t = \theta_{t-1} - \eta\nabla_\theta J(\theta; s^{(i:i+n)})$$

  taking the best of both worlds and reducing the variance of the parameter updates, which can lead to more stable convergence. Minibatch GD is typically the algorithm of choice when training a neural network and the term SGD is usually employed also when minibatches are used.

- **Momentum** [Qian, 1999]: This a method that helps accelerate GD in the relevant direction and dampens oscillations. It does this by adding a fraction γ_m of the update vector of the past time step to the current update vector. The learning rule takes the following form:

$$v_t = \gamma_m v_{t-1} + \eta \nabla_\theta J(\theta_t)$$
$$\theta_t = \theta_{t-1} - v_t$$

where γ_m is usually set to $\gamma_m = 0.9$. The momentum term v_t increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we can get faster convergence.

- **Nesterov Momentum** [Nesterov, 1983]:

  Nesterov momentum (or Nesterov accelerated gradient) is a slightly different version of the momentum update that enjoys stronger theoretical convergence guarantees and in practice it usually works slightly better than standard momentum. The learning rule takes the following form:

$$v_t = \gamma_m v_{t-1} + \eta \nabla_\theta J(\theta_{t-1} + \gamma_m v_{t-1})$$
$$\theta_t = \theta_{t-1} - v_t$$

  where it can be noted that the key difference between standard momentum and Nesterov momentum is that the former compute the gradient before applying the momentum term, while the latter computes the gradient after doing so.

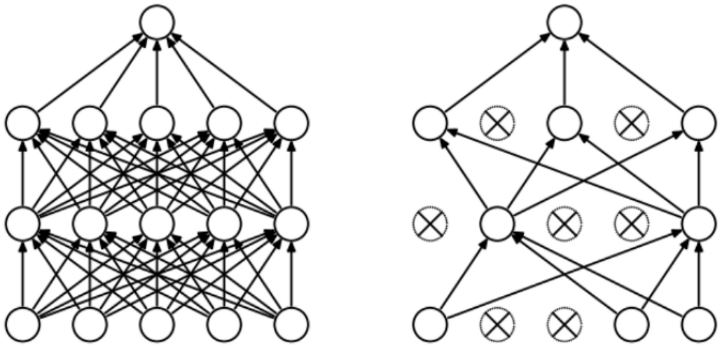
- **Adaptive Moment Estimation** [Kingma and Ba, 2015]: Adaptive moment estimation (Adam) is a method that is part of a family of methods that adapt the learning rate to the parameters. It stores an exponentially decaying average of past squared gradients v_t, but this method also keeps an exponentially decaying average of past gradients m_t, similar to Momentum. The learning rule takes the following form:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

where m_t and v_t are estimates of the first moment and the second moment of the gradients, respectively, and $\hat{m}_t$ and $\hat{v}_t$ are computed as a bias-correction mechanism. Usually $\beta_1 = 0.9$, $\beta_2 = 0.999$ and ϵ is set to a small number to avoid division by zero.

2.2.2.3 Dropout

Dropout [Srivastava et al., 2014] is a regularization technique[Hastie et al., 2001] for neural networks. It has been demonstrated to produce better generalization performance in many settings. The general idea is to drop a given number of neurons randomly at each iteration. This is, at each training stage, individual neurons are either dropped out of the network with probability $1 - p$ or kept with probability p, so that a reduced network is left; incoming and outgoing edges (i.e. parameters) to a dropped-out neurons are also removed. This process is illustrated in figure 2.4.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.4: Examples of neural networks with and without dropout.

2.2.2.4 Batch Normalization

Batch normalization [Ioffe and Szegedy, 2015] is a technique that normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch

standard deviation.

In [Ioffe and Szegedy, 2015] it was suggested that a change in the distribution of activations because of parameter updates slows learning. This phenomenon is called *internal covariate shift*. To alleviate this phenomenon, they proposed a techinique called batch normalization. During GD training, each activation of the minibatch is centered to zero-mean and unit variance. The mean and variance are measured over the whole minibatch, independently for each activation. A learned offset β and multiplicative factor γ are then applied. This process is called *batch normalization* and is illustrated in algorithm 1.

Algorithm 1 Batch Normalizing step, applied to activation x over a minibatch.

Input: Values of x over a minibatch: $\mathcal{B} = \{x_1, \ldots, x_m\}$;

Parameters to be learned: γ, β.

Output: $y_i = \left\{\mathrm{BN}_{\gamma,\beta}(x_i)\right\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$

$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$

$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$

$\hat{y}_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$

By making normalization part of the model architecture, we can be able to use higher learning rates and take less care to the initialization parameters. It is widely accepted that batch normalization additionally acts as a regularizer, reducing the need of other forms of regularization.

At test time, fixed values of the mean μ and the variance σ are needed. In order to do so, during training, this layer keeps a running estimate of its computed mean and variance by using an exponential moving average with a momentum factor m in the following way:

$$\mu_t = m\mu_{t-1} + (1-m)\mu_{\mathcal{B}}$$
$$\sigma^2_t = m\sigma^2_{t-1} + (1-m)\sigma_{\mathcal{B}}$$

2.2.2.5 Initialization Strategies

Neural networks need careful initialization of their weights and biases to work properly. Some of the most popular initialization strategies are the LeCun uniform initialization [LeCun et al., 1998], where initial weights w_i are sampled according to:

$$w_i \sim \mathcal{U}\left\{-\sqrt{\frac{3}{n_{in}}}, \sqrt{\frac{3}{n_{in}}}\right\} \tag{2.9}$$

and the Glorot uniform initialization [Glorot and Bengio, 2010], where initial weights w_i are sampled according to:

$$w_i \sim \mathcal{U}\left\{-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right\} \tag{2.10}$$

where in both cases, $\mathcal{U}(-a,a)$ is the uniform distribution in the interval $(-a,a)$, n_{in} is the size of the previous layer and n_{out} is the size of the next layer.

2.2.3 Generative Adversarial Networks

Generative models are a branch of unsupervised learning that, as their name suggests, are used to generate new data based on some joint probability distribution that the model learns from a training set. This type of models have many applications, such as image denoising, reconstruction of images (*inpainting*), improving the resolution of an image and even pretraining neural networks in cases where obtaining new data is expensive.

In the past few years, Generative Adversarial Networks (GANs), proposed by [Goodfellow et al., 2014], have become a promising model for this task. The objective of this architecture is to create a generative model through an adversarial process based on the concept of a two person zero-sum game first proposed by [Wei Li, 2013].

This is accomplished by simultaneously training two networks. The task of the first network, called the *generative* network, is to generate new samples with the highest possible resemblance to the structure of the original data distribution, and the task of the second network, called the *discriminative* network, is to estimate the probabilities of the data coming from the training data rather then being the output from the *generative* model. These two networks are trained iteratively, learning from each other and improving their performance with each iteration until they converge to an equilibrium.

In their standard formulation by [Goodfellow et al., 2014], GANs are generative models that learn a mapping from random noise vector z to output some kind of samples y, i.e. $G : z \to y$. The generator G is trained to produce outputs that cannot be distinguished from real samples by an adversarially trained discriminator D, which is trained to do well at detecting the generator's fake samples. More formally and following a game-theoretical approach, the objective of a GAN can be expressed as

$$\mathcal{L}_{\text{GAN}}(G, D) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D(y)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))] \tag{2.11}$$

where p_{data} is the true data distribution, p_z is the random noise distribution, and where G tries to minimise this objective agains D that tries to maximise it, therefore $G^* = \arg\min_G \max_D \mathcal{L}_{\text{GAN}}(G, D)$. While this approach is very general, both G and D are commonly defined by deep neural networks, which allows GANs to model complex distributions.

A solution to the objective in expression 2.11 is called a *Nash Equilibrium*. Nevertheless, in practice, solving analytically the objective to find a Nash Equilibrium is intractable so the objective function is trained by means of gradient based methods. It must be noted that in practice, rather than training G to minimize $\log(1 - D(G(z))$, it is preferred to maximize $\log D(G(z))$. This modified objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning.

However, one of the major drawbacks of GANs is their instability during training. In practice this implies that extensive architecture design and hyperparameter tuning is required to successfully train a GAN.

2.3 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) refers to the combination of the DL and RL approaches. Nowadays, DL is enabling RL to scale to problems that were previously intractable. The range of applications goes from allowing a machine to learn to play video games directly from raw pixels, to allow a robot to learn control policies directly from camera inputs in the real world.

2.3.1 Major Breakthroughs

2.3.1.1 The Deep Q-Network

The well-known function approximation properties of neural networks led to the use of DL to regress functions for use in RL agents. A major breakthrough in DRL was the development of the Deep Q-network (DQN) by DeepMind to learn to play Atari video games. The key idea was to use deep neural networks to represent the Q-network (a function approximator of the state-action-value function). An illustration of this idea is shown in figure 2.5. Previous attempts to combine RL with neural networks had largely

failed due to unstable learning. To address these instabilities, the DQN algorithm stores all of the agent's experiences and then randomly samples and replays these experiences to provide diverse and decorrelated training data; this is known as *experience replay*.

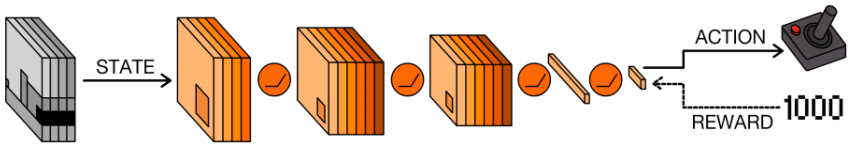


Figure 2.5: The Deep Q-network. It takes the states (a stack of frames from the video game) and processes it with several hidden layers and ReLU activations in between each layer. At the final layer, the network outputs a discrete action. Given the current state and chosen action, the game returns a new score and the difference between the new score and the previous one is used as reward.

2.3.1.2 Experience Replay

The experience replay technique [Lin, 1992] consists in storing the agent's experiences at each time-step, $e_t = (\mathbf{s}_t, \mathbf{a}_t, \mathbf{r}_t, \mathbf{s}_{t+1})$ in a dataset $D_t = \{e_1, \ldots, e_t\}$. During learning, we apply Q-learning updates on samples (or minibatches) of experience $(\mathbf{s}, \mathbf{a}, \mathbf{r}, \mathbf{s})$, drawn uniformly at random from the pool of stored samples. After performing experience replay, the agent selects and executes an action according to an ϵ greedy policy.

This approach is powerful because of three main reasons:

- **Data efficiency**: Each step of experience is potentially used in many weight updates.
- **Variance reduction**: learning from consecutive samples is inefficient, due to the strong correlations between the samples. By randomizing the samples, these correlations are broken and therefore reduces the variance of the updates.
- **Smoothing out learning**: By using experience replay the behavior distribution is averaged over many of its previous states, therefore avoiding oscillations or divergence in the parameters.

It should be noted that when using experience replay, it is necessary to learn off-policy and that is another motivation of using Q-learning jointly with this technique. Whilst in its original formulation this technique uses uniform sampling, it has been shown that prioritizing samples based on TD errors can be more effective for learning [Schaul et al., 2016].

2.3.2 Algorithms

Following the categorization of RL algorithms described in section 2.1.3, we briefly highlight some of the most important algorithms developed in the DRL research in recent years.

- **Value functions**: Double Q-learning [van Hasselt, 2010] is a method that was proposed to overcome the fact that the single estimator used in the Q-learning update rule overestimates the expected return due to the use of the maximum action value as an approximation of the maximum expected action value, by providing a better estimate that uses a double estimator. Dueling DQN [Wang et al., 2016] is an approach to modify the DQN architecture and decompose the Q-function into meaningful functions; in particular, it constructs Q^π by adding together separate layers that compute the state-value function V^π and the advantage function A^π. Besides, dueling DQN combined with prioritised experience replay [Schaul et al., 2016] constitutes one of the techniques with best performance in discrete action settings.

- **Policy search**: *Stochastic Value Gradients* (SVG) [Heess et al., 2015] is a family of algorithms that exploit use the REINFORCE rule [Williams, 1992] or the reparametrization trick [Rezende et al., 2014] in order to treat neural networks as stochastic computation graphs that can be optimized. *Guided Policy Search* (GPS) [Levine and Koltun, 2013] is a way to search directly for a policy represented by a neural network with many parameters, by taking sequences of actions from another system and by using supervised learning in combination with importance sampling [Levine and Abbeel, 2014]. *Trust region policy optimization* (TRPO) [Schulman et al., 2015] is a method that optimizes an importance sampled advantage estimate, constrained using a quadratic approximation of the KL divergence. *Proximal policy optimization* (PPO) [Schulman et al., 2017] follows a similar idea but performs unconstrained optimization, requiring only first-order gradients.

- **Actor-critic methods**: These methods can benefit from research in both value function methods and policy gradient methods. *Deterministic policy gradients* (DPGs)[Silver et al., 2014] is a family of algorithms that extend the standard policy gradient theorems for stochastic policies [Williams, 1992] to deterministic policies. Deep DPGs (DDPGs) represent an extension to DPGS which

utilize deep neural networks to operate on high dimensional visual state spaces [Lillicrap et al., 2016]. Interpolated policy gradients (IPGs) [Gu et al., 2017] unifies on-policy and off-policy methods in AC algorithms and represent one of the most recent state-of-the-art continuous DRL algorithms. Another promising approach consists of exploiting parallel computations; methods for training deep neural networks by means of asynchronous gradient updates have been developed for use on both single machines [Mnih et al., 2016] and distributed systems [Dean et al., 2012]. Following this paradigm, *asynchronous advantage actor-critic* (A3C) [Mnih et al., 2016] algorithm is one of the most popular DRL techniques in recent years; it combines advantage updates with the AC formulation, and relies on asynchronously updated policy and value function networks trained in parallel over several processing threads. When A3C is reformulated such that it uses only one agent [Wang et al., 2017a], or when segments from trajectories of multiple agents can be processed together in a batch by performing GPU computations [Schulman et al., 2017], the algorithm is called *advantage actor-critic* (A2C).

2.4 Related Work

In this section we highlight some publications that are somehow related with the general idea of connecting generative modeling via GANs with RL (or IRL) approaches. To our knowledge and at the moment of writing this dissertation, there has not been prior works that explore the idea of applying GANs on policies or reward functions in the way that we have done it.

In [Pfau and Vinyals, 2016] a connection between GANs and AC methods is established, following a multilevel optimization perspective. It is highlighted that in both kind of methods, the information flow is a simple feedforward pass from one model which either takes an action (AC) or generates a sample (GANs) to a second model which evaluates the output of the first model; the second model is the only one which has information in the environment, in the form of reward signals (AC) or real samples from the distribution in question (GANs). In summary, the first model must learn based on error signals from the second model alone. Besides, practical considerations are highlighted, specially those related to stability issues. Last, it is shown that GANs can be viewed as AC methods in an environment where the actor cannot affect the reward signal.

In [Finn et al., 2016a] it is stated that while the idea of learning cost functions is relatively new to the field of generative modeling, learning costs has long been studied in the IRL domain, typically for imitation learning from demonstrations. It is shown that certain IRL methods are in fact mathematically equivalent to GANs. Interestingly, a particular equivalence between a special kind of maximum entropy IRL [Ziebart et al., 2008] and a GAN in which the density of the generator can be evaluated and is provided as an additional input to the discriminator. Furthermore, it is also shown that this equivalence is more general and it is discussed the interpretation of GANs as an algorithm for training energy-based models.

In [Fu et al., 2017] it is proposed an algorithm called *Adversarial Inverse Reinforcement Learning* (AIRL) based on an adversarial reward learning formulation, motivated by the fact that previous IRL approaches have been proven to be very difficult to apply to high-dimensional problems with unknown dynamics. It is shown that the proposed algorithm is able to recover reward functions that are robust to changes in dynamics, therefore enabling to learn policies under significant variation in the environment seen during the training stage. In a similar way, in [Henderson et al., 2017], it is used the *options* framework [Sutton et al., 1999], in which a generalization of primitive actions to include temporally extended courses of action are provided, and it is proposed an extension of such framework to simultaneously recover reward options in addition to policy options by using GANs and using only observed expert states. It is shown that this approach works well in both simple and complex continuous control tasks.

More recently and more related to the ideas presented in this dissertation, in [Held et al., 2017] it is proposed a method called *Goal GAN* that allows a RL agent to automatically discover the range of tasks that it is capable of performing in its environment, by using a generator network to propose tasks for the agent to try to accomplish, where each task is specified as reaching a certain parametrized subset of the state-space. Furthermore, the generator produces tasks that are always at the appropriate level of difficulty for the agent. It is also shown that an agent can automatically learn to perform a wide set of tasks without requiring any prior knowledge of its environment.

Chapter 3

Models

3.1 Introduction

Our approach can be summarized in the following stages:

1. Given a particular kind of RL environment, train an RL agent on multiple tasks in order to collect data regarding optimal policies and estimated reward functions.
2. Train GANs models on both policies and reward functions.
3. Use the learned generative models, independently, to accelerate new learning.

3.2 The Environment

The RL environment chosen for this project is the Open AI Gym's [Brockman et al., 2016] *FrozenLake* (FL) environment [1]. The simplest version of this environment is a 4×4 grid which consists of four possible areas, namely: Safe (S), Frozen (F), Hole (H) and Goal (G). The set of actions that the agent can take is the set of directions: $\{Left, Up, Right, Down\}$. The agent moves starting from the Safe area around the grid until it reaches the Goal or a Hole. If the agent falls into a hole, the process is over and it is rewarded a value of zero; if it reaches the goal it is rewarded a value of one. In the RL setting, the agent must continue playing until it learns from every mistake and reaches the goal eventually. Therefore, in order to reach the goal, the agent must learn to walk over the Frozen areas and avoid the Holes. An illustration of this environment is shown in figure 3.1.

[1] `https://gym.openai.com/envs/FrozenLake-v0/`

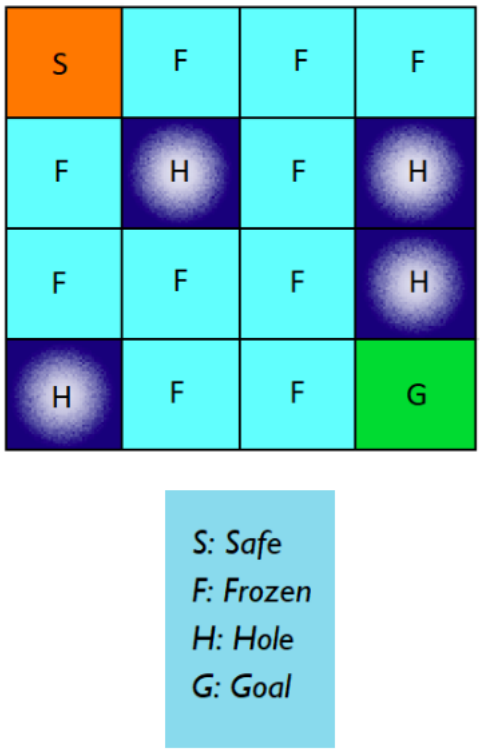


Figure 3.1: 4×4 FrozenLake environment.

In this project, we used the 8×8 version of the FL environment since it represents a more complex problem than the simpler 4×4 version and at the same time, it allows for faster experimentation of the ideas presented in our work, compared to other Open AI Gym's environments that are more computationally expensive. This grid has $8 \times 8 = 64$ possible blocks where the agent will be at a given time. At the current state, the agent will have four possibilities of deciding the next state. From the current state, the agent can move in any direction, which gives a total of $64 \times 4 = 256$ possibilities. Following notation introduced in section 2.1.2, we have that $|\mathcal{S}| = 64$ and $|\mathcal{A}| = 4$.

3.3 Data Collection

In order to generate new and different tasks (or problems) for the agent to solve, we have modified the original (static) Open AI Gym's version of the 8×8 FL environment to make it dynamic, allowing to create environments with pseudorandom location of Start (located at some one of the four possible edges), Goal (located at the opposite direction of the Start) and a fixed number of Holes (7 Holes, located anywhere in the map). This way, we ensure that we create tasks that are at the same level of difficulty for the agent.

In each created task, we made the following:

- Perform (tabular) Q-learning to find optimal state-action-value function Q^* and optimal (deterministic) greedy policy π^* with respect to Q^*.
- Estimate the reward function $\mathcal{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ via multiple linear regression.

We collected data for a total of 800 tasks, where there were 200 tasks per each possible direction of the task (Start in top edge and Goal in bottom edge, Start in bottom edge and Goal in top edge, Start in left edge and Goal in right edge, Start in right edge and Goal in left edge). Once the data was collected, we were able to train generative models to try to sample new reward functions and new policies.

The policies we collected are deterministic, mapping directly states to actions. With the purpose of visualizing the policies, we perform the following mapping of actions to symbols: Left $\mapsto$ " $<$ ", Up $\mapsto$ " $\wedge$ ", Right $\mapsto$ " $>$ ", Down $\mapsto$ " $\vee$ ".

Figures 3.2, 3.3 and 3.4 show the target map, the optimal value function $V^*(\mathbf{s}) = \max_{\mathbf{a}} Q^*(\mathbf{s}, \mathbf{a})$ and the greedy policy π^* with respect to Q^* for some generated task in the training set, respectively.

```
FFFFFFSF
FFFFFFHF
FHFFFFFF
FFFFFHFF
FFHFFFFF
FHFFFFFF
FFFHFFFF
FFFFFGFF
```

Figure 3.2: Example of target map for a generated task in the training set.

3.3.1 Training settings

We performed standard (tabular) Q-learning with the following parameters: number of episodes n_episodes = 2000, learning rate $\alpha = 0.81$, discount factor $\gamma = 0.96$, maximum number of steps (actions) taken per episode max_steps = 100. In order to perform enough exploration, we followed a decaying $\epsilon-$greedy algorithm with update rule $\epsilon = \epsilon - (1/\text{n\_episodes})$ with initial $\epsilon = 0.9$. As initial Q table, we use a zero initialized table. For estimating the reward function for each task, we collect data from the

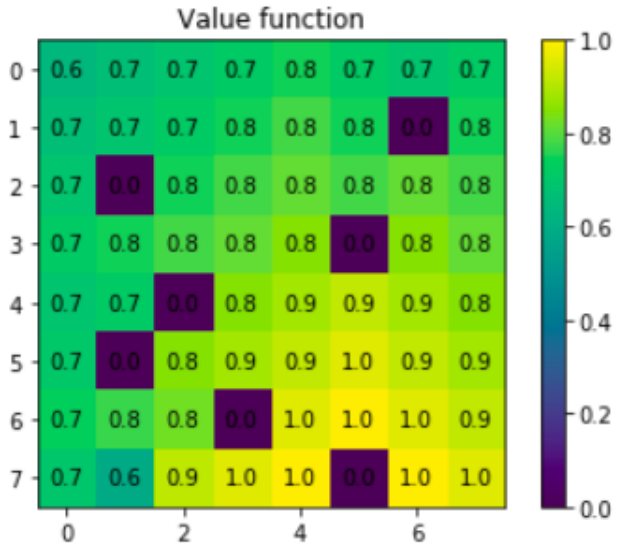


Figure 3.3: Illustration of optimal value function $V^*(\mathbf{s})$ found for the target map shown in figure 3.2. The value function show how good is a state for the RL agent to be in. In this case, it is clear which states correspond to the holes and which states are approaching the goal.

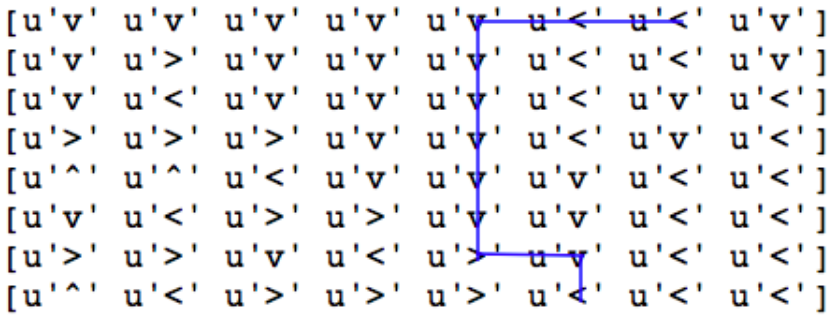


Figure 3.4: Illustration of optimal policy π^* found for the target map shown in figure 3.2. The RL agent has learned how to reach the goal from any position in the map; in particular, the path from the Start state to the Goal state is shown with a line.

environmental interactions in the form of tuples $(\mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r})$ and after the RL agent has performed all the episodes, we regress $\mathbf{r}$ on $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ using a *one-hot encoding* representation for each element in the tuple, and then merging each representation into one single vector; this is, we perform a mapping of the form $(\mathbf{s}, \mathbf{a}, \mathbf{s}') \mapsto \{0,1\}^{|\mathcal{S}|+|\mathcal{A}|+|\mathcal{S}|}$.

3.4 Generative Models

3.4.1 Reward Functions

Let us denote by **FC(n)** a dense (fully connected) layer of dimension n, by **LReLU** a Leaky Rectified Linear Unit activation [Maas et al., 2013], by **BatchNorm** a Batch Normalization layer [Ioffe and Szegedy, 2015] and by **Dropout** a Dropout layer [Srivastava et al., 2014]. As in its standard formulation, we used the binary cross-entropy as loss function for the GAN and the architectures used for the generator network and the discriminator network are described as follows.

Generator

Input: $z \in \mathbb{R}^{16}$, $z \sim \mathcal{N}(\vec{0}, \mathbb{1})$
`FC(256), LReLU, BatchNorm, Dropout`
`FC(512), LReLU, BatchNorm, Dropout`
`FC(1024), LReLU, BatchNorm, Dropout`
FC($\lvert\mathcal{S}\rvert + \lvert\mathcal{A}\rvert + \lvert\mathcal{S}\rvert$), `Sigmoid`

Discriminator

Input: $G(z) \in \mathbb{R}^{\lvert\mathcal{S}\rvert+\lvert\mathcal{A}\rvert+\lvert\mathcal{S}\rvert}$
`FC(512), LReLU`
`FC(256), LReLU`
FC(1), `Sigmoid`

Hyperparameters

Maximum number of epochs 5000 with early stopping[2] with default parameters, batch size of 30 observations, dimension of latent space $\frac{|\mathcal{S}|}{4} = 16$ (therefore $z \in \mathbb{R}^{16}$), leakage parameter of LReLU layer $a = 0.2$, momentum parameter of BatchNorm layer $m = 0.8$, and dropout rate of Dropout layer of $p = 0.8$. Adam optimization algorithm was used with learning rate $\eta = 0.0001$ and parameters $\beta_1 = 0.5$, and $\beta_2 = 0.999$, for both the generator and the discriminator. All the aforementioned hyperparameters

[2] https://keras.io/callbacks

were chosen by performing grid search and cross-validation[Hastie et al., 2001]. All networks parameters were initialized using Glorot uniform initialization.

3.4.2 Policies

As in its standard formulation, we used the binary cross-entropy as loss function for the GAN and following notation described in section 3.4.1, the architectures used for the generator network and the discriminator network are described as follows.

Generator network

Input: $z \in \mathbb{R}^{16}$, $z \sim \mathcal{N}(\vec{0}, \mathbb{1})$
`FC(256), LReLU, BatchNorm, Dropout`
`FC(512), LReLU, BatchNorm, Dropout`
`FC(1024), LReLU, BatchNorm, Dropout`
FC($\lvert\mathcal{S}\rvert \times \lvert\mathcal{A}\rvert$), `ReLU`
Reshape($\lvert\mathcal{S}\rvert, \lvert\mathcal{A}\rvert$), `Softmax`, `Argmax`, `One-hot`

Discriminator network

Input: $G(z) \in \mathbb{R}^{\lvert\mathcal{S}\rvert \times \lvert\mathcal{A}\rvert}$
Reshape($\lvert\mathcal{S}\rvert \times \lvert\mathcal{A}\rvert$)
`FC(256), LReLU`
FC(1), `Sigmoid`

Hyperparameters

All hyperparameters were set as described in section 3.4.1, except for the maximum number of epochs which was 10000 in this case, with early stopping with default parameters.

3.5 Accelerating New Learning

3.5.1 Reward Functions

We consider the setting where the RL agent is free to play in an environment, but what is costly is the human supervision of its reward for its RL problem. In this case, one would want the agent to learn to work with the environment using some intrinsic motivation, and then be more efficient at learning when an extrinsic task is given to it by the user. A similar setting has been addressed before in [Finn et al., 2016b], where the reward function can only be evaluated in a set of *labeled* MDPs, and the agent must generalize its behavior to the wide range of states it might encounter in a set of *unlabeled* MDPs, by using experience from both settings. In the case of our RL environment, we separate the role of holes (domain) and goals (task), and we consider the following phases of training:

- **Phase 1 training** (unrewarded, target map): The user doesn't give a task (goal). Let the RL agent make up new rewards via the trained GAN generator and learn to solve them with a RL algorithm. In the process, it should learn something about the target map (for example, where are located all the holes in the target map).

- **Phase 2 training** (rewarded, target map): Using the same target map, the user now specifies a task (goal) and the RL agent should learn to solve it. Given the previous phase, the RL agent must know something about the current environment and it should be able to solve the target task more quickly.

3.5.2 Policies

Given that we have trained a GAN on a set of policies obtained from a set of source maps, then the next step is to accelerate training on a set of new target maps. In this sense, the general idea is to have the subspace of reasonable policies learned by the GAN help accelerate the search by constraining it away from the larger space of all policies. One way to do this is to initialize the search of the target problem by sampling the generator rather than taking a random initial condition. In order to do so, we needed an algorithm where explicit representation of the policies are learned and we have implemented a simple version of the state-of-the-art Advantage Actor-Critic (A2C) algorithm with one single RL agent.

We represent the states using *one-hot encoding*, we used mean squared error (MSE) as loss function and following notation described in section 3.4.1, the architectures used for the actor network and the critic network are described as follows.

The Actor

Input: $s \in \{0,1\}^{\lvert\mathcal{S}\rvert}$
`FC(4), ReLU`
`FC(128), ReLU`
`FC(`$\lvert\mathcal{A}\rvert$`), Linear`

The Critic

Input: $s \in \{0,1\}^{\lvert\mathcal{S}\rvert}$
`FC(4), ReLU`
`FC(1), Linear`

Chapter 4

Experiments and Analysis

4.1 Generative Models

4.1.1 Evaluation Metrics

As noted by [Borji, 2018], despite large strides in terms of theoretical progress, evaluating and comparing GANs remains a difficult task. In [Borji, 2018] it is presented a review and discussion about quantitative and qualitative measures for evaluating generative models with a particular emphasis on GANs based models. While several measures have been introduced, there is no consensus as to which measure best captures strengths and limitations of the generated models. Furthermore, many proposed measures remain to be domain specific. Among qualitative measures, visual examination of samples is one of the common and most intuitive ways to evaluate GANs. In the case of this project, given the novelty of our approach and taking advantage of the small size of the tasks, we consider visual inspection to be sufficient to evaluate the quality of generated samples, both on reward functions and policies. Since we have collected data and we have estimated reward functions and and obtained optimal policies for the generated tasks, it is straightforward to visually compare the generated samples with the samples from the true data distributions in each case.

4.1.2 Results: Reward Functions

In order to visualize the samples generated by the GAN and evaluate their quality, we compute $\hat{\mathbf{r}}(\mathbf{s}) = \max_{\mathbf{a},\mathbf{s}'} \hat{\mathcal{R}}(\mathbf{s},\mathbf{a},\mathbf{s}')$ for each state $\mathbf{s}$ and plot these values in a heatmap. Figures 4.1, 4.2, 4.3, and 4.4 show good samples generated by the generative model. The GAN has learned that highest values (i.e. proposed goals) must be located at the

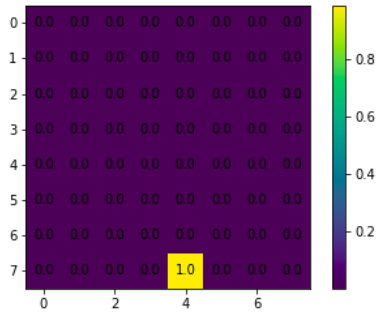


Figure 4.1: Illustration of a sampled reward function with goal located in bottom edge.

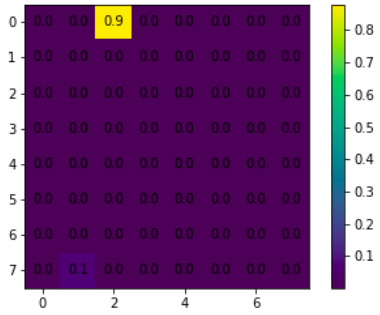


Figure 4.2: Illustration of a sampled reward function with goal located in top edge.

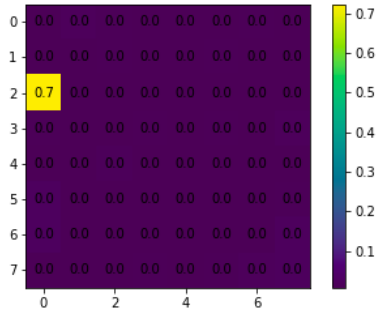


Figure 4.3: Illustration of a sampled reward function with goal located in left edge.

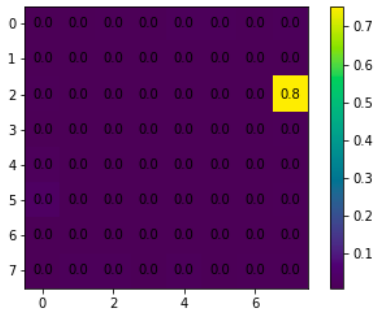


Figure 4.4: Illustration of a sampled reward function with goal located in right edge.

edges of the target map. Nevertheless, there are also bad samples that don't look like the samples from the true data distribution of reward functions in the training data. For example, figures 4.5 and 4.6 show samples where there are more than one possible goal and therefore they could not be good for a RL task in our context, although all of them are located at some of the edges of the target map; on the other hand, figures 4.7 and 4.8 show samples where there is not a clear goal since all values in the map are very similar.

4.1.3 Results: Policies

In order to visualize the policies sampled by the GAN, we sample vectors of policies and we reshape them in a matrix of dimensions consistent with the target map and making the mapping from actions to symbols, just as previously shown in figure 3.4. Figure 4.9 shows a sampled policy at the beginning of training where only random noise is observed, and 4.10 shows a sampled policy at 500 epochs of training, where it can be observed that small meaningful paths begin to appear; figures 4.11, 4.12, 4.13 and 4.14

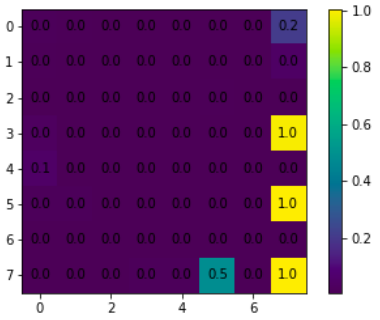


Figure 4.5: Illustration of a sampled reward function that is not consistent with the true data distribution.

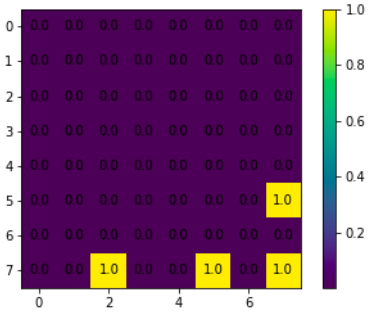


Figure 4.6: Illustration of a sampled reward function that is not consistent with the true data distribution.

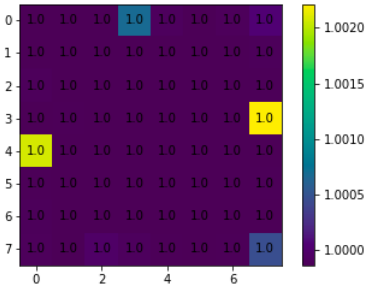


Figure 4.7: Illustration of a sampled reward function that is not consistent with the true data distribution.

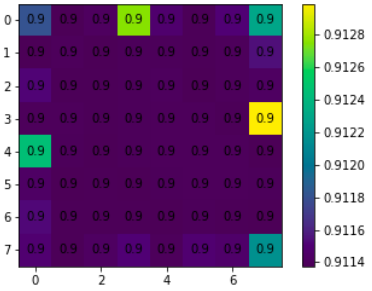


Figure 4.8: Illustration of a sampled reward function that is not consistent with the true data distribution.

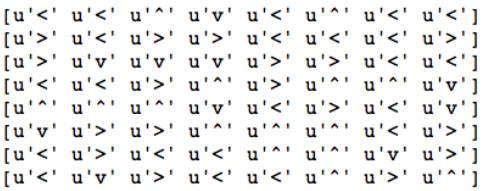


Figure 4.9: Sampled policy at 0 epochs of training.

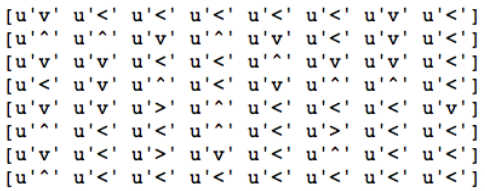


Figure 4.10: Sampled policy at 500 epochs of training.

Figure 4.11: Sampled policy at 1000 epochs of training.

Figure 4.12: Sampled policy at 2500 epochs of training.

show sampled policies at 1000, 2500, 5000 and 7500 epochs of training, where we can gradually observe policies with increasingly meaningful paths.

Finally, figures 4.16, 4.15, 4.17 and 4.18 show samples of policies for the final GAN model trained with 10000 epochs. It can be noted that at this point, the GAN is able to produce policies with meaningful paths from one edge to another opposite edge. In each figure, two possible paths from some point at the start edge to some point at the end edge are shown with lines.

4.2 Accelerating New Learning

4.2.1 Baselines and Evaluation Metrics

In the case of this stage, we consider that learning the target problems *from scratch* to be the natural choice for the baseline. On the other hand, we regard the following as the evaluation metrics:

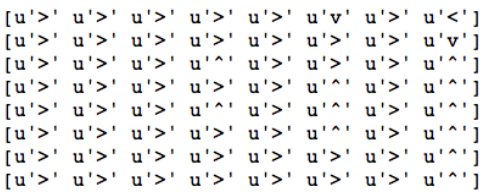


Figure 4.13: Sampled policy at 5000 epochs of training.

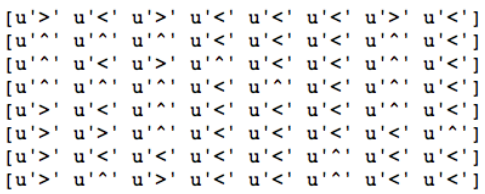


Figure 4.14: Sampled policy at 7500 epochs of training.

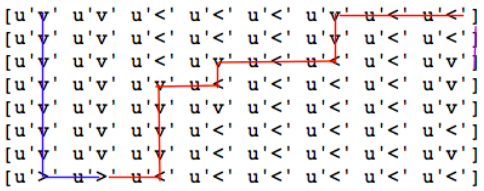


Figure 4.15: Illustration of a sampled policy with paths from top edge to bottom edge.

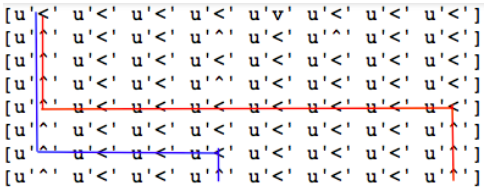


Figure 4.16: Illustration of a sampled policy with paths from bottom edge to top edge.

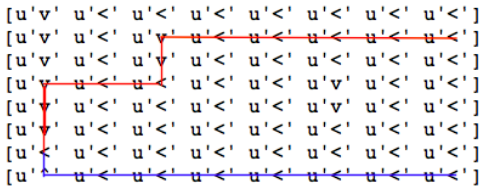


Figure 4.17: Illustration of a sampled policy with paths from right edge to left edge.

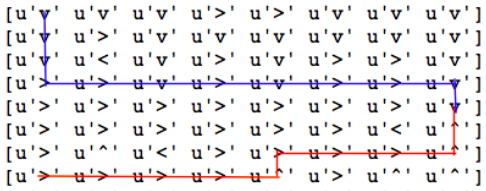


Figure 4.18: Illustration of a sampled policy with paths from left edge to right edge.

- **Rewards**: Number of total environmental interactions.
- **Policies**: Time to achieve a given solved performance level. We measure this with the number of episodes needed until the actor network has learned a policy to solve the task.

4.2.2 Results: Reward Functions

Following the setting described in section 3.5.1, we conduct the following experiment:

- **Phase 1 training**: We generate a target map in a similar way to the data collection step described in section 3.3 (specifically, this target map give us location of Frozen and Holes states). After, we sample a reward function from the trained generator network which induces a Goal state in the target map, and we set the Start state to some pseudorandomly state located on the opposite edge to the one where the Goal is located. Finally, we let the RL agent to solve the problem via Q-learning and computing the rewards via the previously sampled reward function.
- **Phase 2 training**: Using the same target map, we simulate the fact that the user now can specify a new goal by pseudorandomly changing the Goal state to another location in one of the other edges and again, we set the Start state to some

pseudorandomly state located on the opposite edge to the one where the Goal is located. Therefore, we let the RL agent to use the Q table obtained in Phase 1 as prior Q, which now must encapsulate information about the target map (desirable states and Holes states), and computing the rewards via the true reward function.

In order to conduct our experiments and compare learning with prior Q against learning from scratch, we performed standard (tabular) Q-learning with the following parameters: number of episodes n_episodes = 2000, learning rate $\alpha = 0.81$, discount factor $\gamma = 0.96$, maximum number of steps (actions) taken per episode max_steps = 100. We followed a decaying $\epsilon-$greedy algorithm with update rule $\epsilon = \epsilon - (1/\text{n\_episodes})$ with initial $\epsilon = 0.9$. In the case of learning the problem from scratch, we use a zero initialized Q table. We repeated the experiment a number of 100 times.

Figure 4.19 shows an example of a target map with Goal located according to a sampled reward function, while figure 4.20 shows the same target map with modified location of Start and Goal states. Figures 4.21 and 4.22 illustrate the value function learned by the RL agent by using the true reward function and the sampled reward function for the target map shown in 4.19, showing that similar value functions can be obtained if the RL agent uses the sampled reward function instead of the true reward function. Finally, figures 4.23 and 4.24 illustrate the value function learned by using the value function shown in 4.22 as prior Q for the target map shown in 4.20, and the value function if the RL agent performs Q-learning from scratch, respectively.

On the one hand, we compute the percentage of won episodes and on the other hand, the total number of environmental interactions divided by the number of won episodes and we compare these distributions when using a prior Q table and from scratch (zero initialized table). The tables 4.1 and 4.2 show the results of this comparison. The results show that in average, learning to solve the problem with an extrinsic goal (i.e. with modified location of Goal) by using a prior Q is able to achieve more won episodes thus having a better performance, compared with learning to solve the problem from scratch. Also, the results show that using a prior Q would lead to require a slightly lower number of average environmental interactions relative to the number of won episodes, compared with learning to solve the problem from scratch.

4.2.3 Results: Policies

Following the setting described in section 3.5.2, we conduct the following experiment:

We generate a target map in a similar way to the data collection step described in

FFFFFFFG
SFFFFFFF
FFFFFFFF
FFHFFFHF
FHFHFFFF
FFFFFFFF
FFFFHFFF
HFFFFHFF

Figure 4.19: Example of a target map with Goal located according to a sampled reward function.

FFFFFFFF
FFFFFFFS
FFFFFFFF
FFHFFFHF
FHFHFFFF
FFFFFFFF
FFFFHFFF
GFFFFHFF

Figure 4.20: Example of the same target map with modified location of Start and Goal.

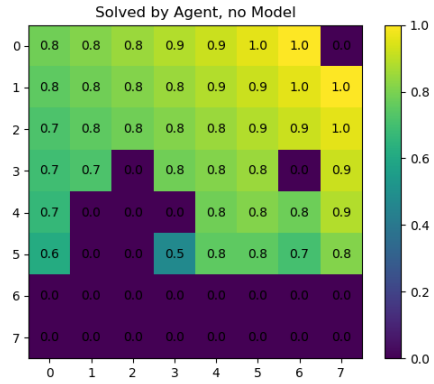


Figure 4.21: Value function learned by the agent by using the true reward function for the target map shown in 4.19.

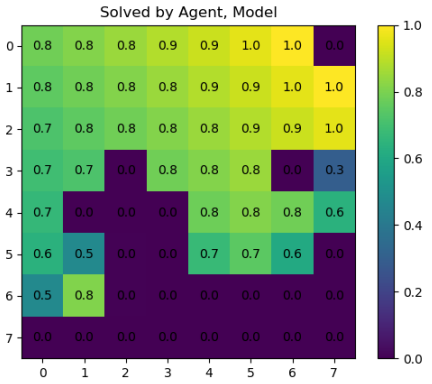


Figure 4.22: Value function learned by the agent by using the sampled reward function for the target map shown in 4.19.

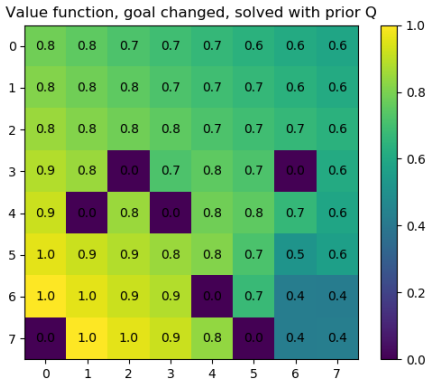


Figure 4.23: Value function learned by using the value function shown in 4.22 as prior Q for the target map shown in 4.20.

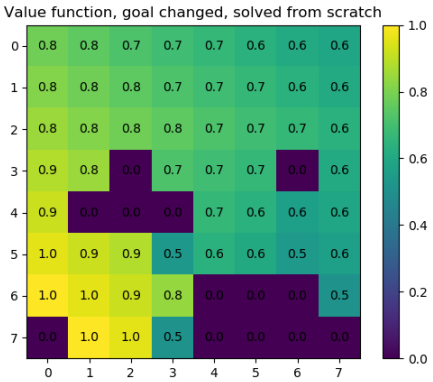
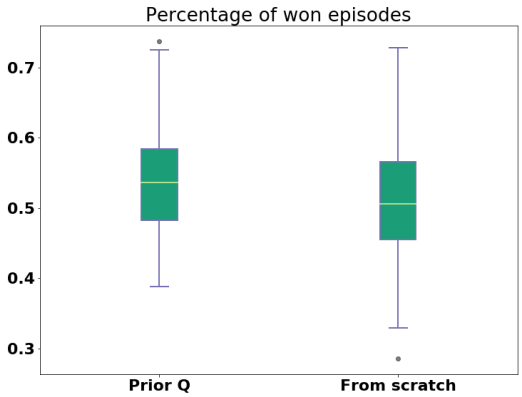
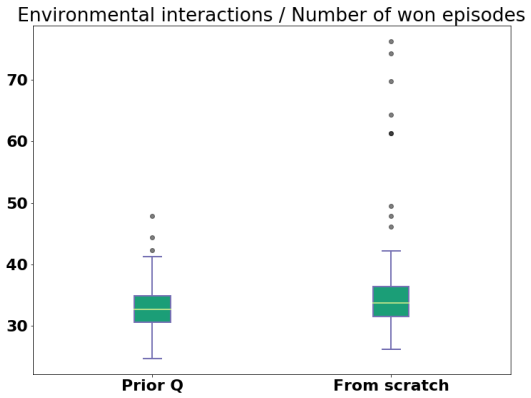


Figure 4.24: Value function learned from scratch for the target map shown in 4.20.



	From scratch	Prior Q
observations	100	100
mean	0.51	0.54
std	0.09	0.08
min	0.29	0.39
25th percentile	0.46	0.48
50th percentile	0.51	0.54
75th percentile	0.57	0.58
max	0.73	0.74

Table 4.1: Summary statistics of percentage of won episodes.



	From scratch	Prior Q
observations	100	100
mean	36.06	33.07
std	9.13	4.07
min	26.18	24.67
25th percentile	31.48	30.59
50th percentile	33.66	32.67
75th percentile	36.32	34.85
max	76.31	47.88

Table 4.2: Summary statistics of number of environmental interactions divided by the number of won episodes.

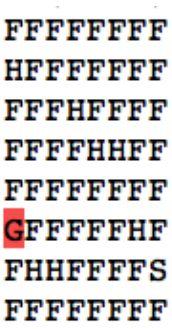


Figure 4.25: Example of target map generated for the set of experiments described in section 4.2.3.

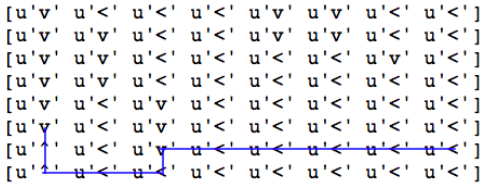


Figure 4.26: Sampled policy from the generator network and followed by the RL agent in the earlier episodes of learning.

section 3.3. After, we sample set of policies from the trained generator network and for each sampled policy, we let the RL agent to try to solve the problem following the sampled policy for a certain number of episodes instead of taking the actions dictated by the actor network, which follows a pseudorandom initialization. We repeated the experiment a number of 100 times and each time we sampled a set of 10 policies.

Figure 4.25 shows an example of target map generated for this set of experiments, while figure 4.26 shows the sampled policy from the generator network and followed by the RL agent in the earlier episodes of learning. On the other hand, optimal policies achieved by the actor network when the RL agent performs standard AC2 algorithm and AC2 following the sampled policy from the generator network are shown in figures 4.27 and 4.28, respectively.

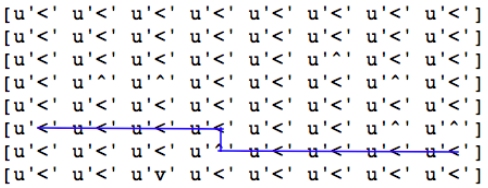


Figure 4.27: Optimal policy found for the target map in figure 4.25 when the RL agent performs standard A2C algorithm.

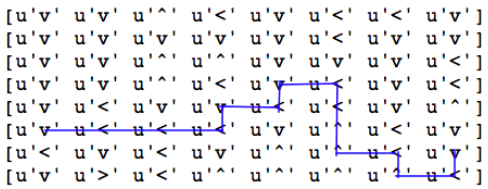
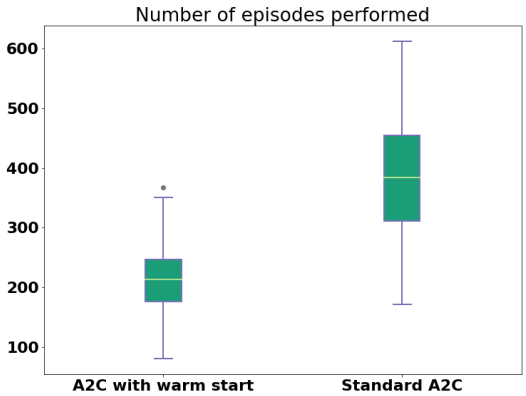


Figure 4.28: Optimal policy found for the target map in figure 4.25 when the RL agent performs A2C algorithm following the sampled policy shown in 4.26 in the earlier episodes of learning.

In order to conduct our experiments and compare learning by following a policy sampled from the trained generator network (we refer to this setting as A2C with warm start) against learning following the standard A2C algorithm, we used the following parameters: maximum number of episodes n_episodes = 1000, number of episodes following the sampled policy warm_start = 200, learning rate $\alpha = 0.81$, discount factor $\gamma = 0.975$, maximum number of steps (actions) taken per episode max_steps = 40, batch size of 30 observations and length of experience replay buffer of 40 observations. We followed a decaying $\epsilon-$greedy algorithm with update rule $\epsilon = \epsilon - (1/\text{n\_episodes})$ with initial $\epsilon = 0.5$, with respect to the actor network. For the optimization, SGD algorithm with Nesterov momentum was used with learning rate $\eta = 0.1$ and momentum parameter $\gamma_m = 0.9$, for both the actor network and the critic network. All networks parameters were initialized using LeCun uniform initialization.

In each experiment, among the sampled policies, we measure the percentage of times where learning using sampled policies at the first episodes were faster than standard A2C algorithm, in terms of number of episodes taken in order to solve the task. The table 4.3 shows the summary statistics of the aforementioned random variable. Quantitatively, we observed that approximately, 25 percent of the time, starting learning following a sampled policy results in a faster learning, while qualitative, we observed that (consistent with intuition) this usually happens when the sampled policy suggests a path that can take the RL agent near the real goal in the target map (just as in the example shown in figures 4.25 - 4.26). In the example, the sampled policy suggest to go from the right edge to the left edge, just as in the goal proposed in the target map.

On the other hand, for those target maps where there was at least one sampled policy which resulted in faster learning than standard A2C, we take the best result (minimum number of episodes performed) and we contrast this distribution against the distribution



	A2C with WS	A2C
observations	97	97
mean	215.28	385.04
std	58.56	97.08
min	81.00	172.00
25th percentile	176.00	311.00
50th percentile	213.00	384.00
75th percentile	247.00	454.00
max	367.00	612.00

Table 4.4: Summary statistics of number of episodes performed until solve the task.

of number of episodes taken when standard A2C was performed. The table 4.4 shows the results of this comparison. The results show that under these considerations, in average, significantly less time (i.e. number of episodes of training) is needed to solve the task when A2C with warm start is performed, compared with learning to solve the problem with standard A2C algorithm.

Table 4.3: Summary statistics of percentage of times where learning using sampled policies was faster than standard A2C algorithm.

observations	100
mean	0.25
std	0.11
min	0.00
25%	0.17
50%	0.24
75%	0.35
max	0.49

Chapter 5

Conclusions and Future Work

In summary, the conclusions drawn from this project are the following:

- We have proposed, implemented and qualitatively evaluated a method for generating both reward functions and policies via adversarial training, based on a particular and simple RL environment but that can be generalizable to other more complex RL environments. We have found that it is completely possible to train GANs in order to be able to produce realistic policies and reward functions that can be used to eventually try to accelerate new learning. Nevertheless, consistent with the rest of GANs related work and research, GAN training in our case has required a substantial amount of extensive architecture design and hyperparameter tuning.

- We have explored and qualitatively and quantitatively evaluated the idea of accelerating new learning by using the learned generative models for both policies and reward functions. We have found that is possible to accelerate new learning under certain conditions and the lines of future research on this topic are wide open.

Furthermore, after completing this project, we highlight the following ideas as possible lines of future research:

- In this project we have explored the idea of using GANs in order to generate both reward functions and policies. Similar to GANs, Variational Autoencoders (VAEs) [Kingma and Welling, 2013] are expressive latent variable models that can be used to learn complex probability distributions from training data. Future research could be done by following our approach and using VAEs as the generative models

to produce reward functions and policies in the context of RL and contrast the results produced by the GANs.

- In this project we have used a simple RL environment in order to test our ideas. Given the results observed, a natural and immediate extension would be to keep trying out the ideas presented here on more complex environments provided by the OpenAI Gym toolkit.

- In this work we have used a simple approach to estimate and parametrize the reward function $\mathcal{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ in order to eventually sample the trained generator network of rewards to propose new tasks for the RL agent. One possible extension would be to use some IRL approach instead to recover and parametrize the reward function.

- In this work we have collected deterministic policies and have trained the GAN to output also deterministic policies. One immediate extension would be to consider stochastic policies.

- While in this project we have explored the idea of accelerating learning of new target problems by sampling the trained generator network of policies, there are other possible ways in which we can still explore this idea. Some of them are the following:

    1. **Regularization using the discriminator network**: Optimize the target problem cost $J(\theta)$ in a policy gradient (or actor-critic) algorithm, by adding a regularization term $D(\pi)$, where $D(\pi)$ stands for the trained discriminator network evaluated at the policy π. The general idea would be that the discriminator regularizes the search to prefer policies that look more realistic. Naturally, the discriminator would need not to be a well trained one since at convergence, the discriminator will not be able to the distinguish between real and fake samples.

    2. **Regularization using the generator network**: Optimize the target problem cost $J(\theta)$ in a policy gradient (or actor-critic) algorithm, by adding a regularization term $p(\pi|G)$, where $p(\pi|G)$ stands for the likelihood of the policy π under the generative model G. The general idea would be to regularize the search in order to prefer policies that have high likelihood under the generative model. While in its original formulation and

most of their variants GANs don't provide this likelihood directly, likelihood estimation would be possible following the approach presented in [Eghbal-zadeh and Widmer, 2017].

3. **Search in the latent space**: Rather than search policy parameters directly to optimize $J(\theta)$, perform the search by using the noise parameter of GAN instead, each of which correspond to some kind of policy; this is, optimize $J(G(z))$. If the GAN has modelled the manifold of policies well, this could be more efficient than searching in raw parameter space; particularly if the dimension of the policy (i.e. output of the GAN) is smaller than the dimension of the latent space (input noise of the GAN).

Appendices

Appendix A

Implementation and Hardware

All models were implemented using the TensorFlow[1] library and the Keras[2] high-level Python API, using the Conda[3] package management system.

We used the CPU version of TensorFlow running on a local machine with macOS Sierra with a 1.8 GHz Intel Core i5 processor.

Experimentation and visualization were performed using Jupyter Notebooks[4].

Important code used for this project can be found at `https://github.com/pjcv89/msc_dissertation`.

[1] `https://www.tensorflow.org/`
[2] `https://keras.io/`
[3] `https://conda.io/docs/`
[4] `http://jupyter.org/`

Bibliography

[Abbeel and Ng, 2004] Abbeel, P. and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning, proceedings of the twenty-first international conference on machine learning. In *Proceedings of the 21st international conference on Machine learning*.

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *IEEE Signal Processing Magazine*.

[Borji, 2018] Borji, A. (2018). Pros and cons of GAN evaluation measures. *CoRR*, abs/1802.03446.

[Bottou, 2010] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

[Dean et al., 2012] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Tucker, A. S. P., and Le, K. Y. Q. V. (2012). Large scale distributed deep networks. *NIPS*.

[Eghbal-zadeh and Widmer, 2017] Eghbal-zadeh, H. and Widmer, G. (2017). Likelihood estimation for generative adversarial networks. *CoRR*, abs/1707.07530.

[Finn et al., 2016a] Finn, C., Christiano, P. F., Abbeel, P., and Levine, S. (2016a). A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *CoRR*, abs/1611.03852.

[Finn et al., 2016b] Finn, C., Yu, T., Fu, J., Abbeel, P., and Levine, S. (2016b). Generalizing skills with semi-supervised reinforcement learning. *CoRR*, abs/1612.00429.

[Fu et al., 2017] Fu, J., Luo, K., and Levine, S. (2017). Learning robust rewards with adversarial inverse reinforcement learning. *CoRR*, abs/1710.11248.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *AISTATS, volume 9*.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc.

[Gu et al., 2017] Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R. E., Scholkopf, B., and Levine, S. (2017). Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. *NIPS*.

[Han and Moraga, 1995] Han, J. and Moraga, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. *Springer*, pages 195–201.

[Hastie et al., 2001] Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA.

[Heess et al., 2015] Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., and Tassa, Y. (2015). Learning continuous control policies by stochastic value gradients. *NIPS*.

[Held et al., 2017] Held, D., Geng, X., Florensa, C., and Abbeel, P. (2017). Automatic goal generation for reinforcement learning agents. *CoRR*, abs/1705.06366.

[Henderson et al., 2017] Henderson, P., Chang, W., Bacon, P., Meger, D., Pineau, J., and Precup, D. (2017). Optiongan: Learning joint reward-policy options using generative adversarial inverse reinforcement learning. *CoRR*, abs/1709.06683.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Karen Simonyan, 2014] Karen Simonyan, A. Z. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Kingma and Ba, 2015] Kingma, D. and Ba, J. (2015). Adam: a method for stochastic optimization. *International Conference on Learning Representations*.

[Kingma and Welling, 2013] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *CoRR*, abs/1312.6114.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Orr, G. B., and Muller, K.-R. (1998). Efficient backprop.

[Levine and Abbeel, 2014] Levine, S. and Abbeel, P. (2014). Learning neural network policies with guided policy search under unknown dynamics. *NIPS*.

[Levine and Koltun, 2013] Levine, S. and Koltun, V. (2013). Guided policy search. *ICLR*.

[Li, 2017] Li, Y. (2017). Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274.

[Lillicrap et al., 2016] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. *ICLR*.

[Lin, 1992] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning, 8(3-4)*, pages 293–321.

[Maas et al., 2013] Maas, A., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *ICML*.

[Mehdri Mirza, 2014] Mehdri Mirza, S. O. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *ICLR*.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted boltzmann machines. *ICML*.

[Nesterov, 1983] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). *Soviet.Math.Docl., volume 269*, pages 543–547.

[Ng and Russell, 2000] Ng, A. Y. and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *17th International Conf. on Machine Learning*, pages 663–670. Morgan Kaufmann.

[Pfau and Vinyals, 2016] Pfau, D. and Vinyals, O. (2016). Connecting generative adversarial networks and actor-critic methods. *CoRR*, abs/1610.01945.

[Philip Isola, 2017] Philip Isola, Jun-Yan Zhu, T. Z. A. A. E. (2017). Image-to-image translation with conditional adversarial networks. *arXiv preprint*.

[Qian, 1999] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks: The Official Journal of the International Neural Network Society*.

[Ramachandran and Amir, 2007] Ramachandran, D. and Amir, E. (2007). Bayesian inverse reinforcement learning. In *Proc. IJCAI*, pages 2586–2591.

[Ramachandran et al., 2017] Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Searching for activation functions. *CoRR*, abs/1710.05941.

[Recht et al., 2011] Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. *NIPS*.

[Rezende et al., 2014] Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. *ICML*.

[Schaul et al., 2016] Schaul, T., Quan, J., Antonoglou, I., and Silver., D. (2016). Prioritized experience replay. *ICLR*.

[Schulman et al., 2015] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. *ICML*.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv:1707.06347*.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. *ICML2014*.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*.

[Sutton et al., 1999] Sutton, R., Precup, D., and Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence 112*, pages 181–211.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

[van Hasselt, 2010] van Hasselt, H. (2010). Double q-learning. *NIPS*.

[Wang et al., 2017a] Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2017a). Learning to reinforcement learn. *CogSci*.

[Wang et al., 2017b] Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2017b). Learning to reinforcement learn. *CogSci*.

[Wang et al., 2016] Wang, Z., de Freitas, N., and Lanctot, M. (2016). Dueling network architectures for deep reinforcement learning. *ICLR*.

[Wei Li, 2013] Wei Li, Melvin Gauci, R. G. (2013). A convolutionary approach to learn animal behavior through controlled interaction. *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning, 8(3-4)*.

[Wulfmeier et al., 2015] Wulfmeier, M., Ondruska, P., and Posner, I. (2015). Deep inverse reinforcement learning. *CoRR*, abs/1507.04888.

[Yunus Saatchi, 2017] Yunus Saatchi, A. G. W. (2017). Bayesian gan. *NIPS*.

[Ziebart et al., 2008] Ziebart, B., Maas, A., Bagnell, J. A., and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning. *AAAI Conference on Artificial Intelligence*.

[Ziebart et al., 2010] Ziebart, B. D., Bagnell, J. A., and Dey, A. K. (2010). Modeling interaction via the principle of maximum causal entropy. *ICML*.