Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Generalized Gradient Q Iteration

Relatore: Prof. Marcello RESTELLI
Correlatore: Ph.D. Matteo PIROTTA
Correlatore: Ing. Carlo D'ERAMO

Tesi di laurea di:
Germano GABBIANELLI Matr. 837834

Anno Accademico 2016–2017

Dedicata a mia nonna Paola e ai miei genitori Mercedes e Marcello...

Ringraziamenti

Abstract

Our work belongs to the Machine Learning field, and more specifically in the research area of Reinforcement Learning. Reinforcement Learning is the activity performed by an *agent* that tries to maximize a *reward signal*, while interacting with the *environment*, to achieve a *goal*. The objective of this thesis is to propose a new Reinforcement Learning method, called *Generalized Gradient Q Iteration*, that belongs to the class of approximate action-value iteration and that can be seen as generalizing some aspects of algorithms of the same class, such as Fitted Q Iteration. The main innovation of our method is that it proposes to learn directly the Bellman Optimality Operator therefore allowing us to move faster towards the optimum, in the space of action-value functions. We compare our method to Fitted Q Iteration and show how it obtains comparable or better performances under some settings of its hyper-parameters. This leads us to be optimistic about the research direction pursued in this work and motivates us to further study and improve our method.

Sommario

Il nostro lavoro si colloca nell'area di ricerca del Reinforcement Learning, che è una sottocategoria del Machine Learning (ML) e a sua volta il Machine Learning fa parte del campo di ricerca dell'Intelligenza Artificiale (IA). Nel tempo sono state date svariate definizioni di Intelligenza Artificiale. Una delle più famose categorizzazioni di queste definizioni è dovuta a [Russell and Norvig, 2016] e le partiziona lungo due dimensioni. La prima dimensione contrappone l'atto di ragionare o *pensare intelligentemente* all'atto di *agire intelligentemente*. La seconda dimensione, invece, suggerisce due modi opposti di definire il successo in IA: in termini di somiglianza ad un comportamento umano oppure di somiglianza ad una razionalità ideale. Storicamente tutte queste definizioni, che implicano obiettivi e approcci differenti, sono state ampiamente seguite e hanno dato origine ai numerosi sottocampi di cui l'Intelligenza Artificiale è composta oggi, come per esempio il natural language processing, il machine learning, l'automated reasoning o la computer vision. Le applicazioni dell'Intelligenza Artificiale sono molteplici e vanno dalla sanità, alla finanza ai videogiochi fino alle macchine a guida autonoma.

Più nello specifico il Machine Learning si occupa di costruire sistemi che posso apprendere autonomamente dall'esperienza, senza l'intervento di un operatore umano. Il Reinforcement Learning è una branca del Machine Learning e il suo scopo principale è creare agenti che siano capaci di imparare autonomamente, mentre interagiscono con l'ambiente, a raggiungere un obiettivo predefinito. A questo scopo agli agenti è consentito compiere delle azioni che abbiano un effetto sull'ambiente e dopo ogni azione di ricevere un reward numerico, che specifica quanto positiva o negativa fosse una determinata azione nella circostanza in cui è stata compiuta.

Un agente di Reinforcement Learning interagisce con l'ambiente, che è modellato secondo il formalismo matematico dei Markov Decision Process [Bellman, 1957], compiendo delle azioni e ricevendo un segnale numerico di reward. L'obiettivo dell'agente è di massimizzare il reward totale ricevuto nel tempo. Per questo motivo gli agenti usano questo segnale di reward come feedback riguardo le azioni compiute in ogni circostanza, ovvero in ogni stato dell'ambiente, rinforzando quelle che hanno portato ad un reward positivo e penalizzando le altre. Questo processo porta l'agente a imparare una *policy*, ovvero una relazione tra i possibili stati dell'ambiente e le azioni

più appropriate per quella circostanza. Al fine di trovare la miglior policy, chiamata formalmente policy ottima, molti aggenti fanno uso di value function o action-value function. Queste funzioni rappresentano la stima corrente, fatta dall'agente, riguardo quanto positivo o negativo sia trovarsi in un dato stato (value function) e quanto positivo o negativo sia compiere una determinata azione a partire da un determinato stato (action-value function).

In alcuni casi la policy ottima può essere determinata in maniera esatta adoperando alcune tecniche conosciute come *Dynamic Programming*. Tra queste, una classe importante di metodi è chiamata *Policy Iteration*, che comprende come caso speciale il metodo chiamato action-value iteration. Questo metodo identifica una sequenza di action-value function, partendo da una funzione casuale e migliorandola ad ogni passo dell'algoritmo, tramite la risoluzione di un qualche tipo di *equazione di Bellman* [Bellman, 2013]. Le equazioni di Bellman definiscono le condizioni di ottimalità per le value function e sono usate, in qualche variante, da ogni algoritmo considerato in questa tesi. Uno studio approfondito di questi metodi e molti altri è dato da [Sutton and Barto, 1998].

Sfortunatamente la maggior parte dei problemi reali sono troppo complessi per essere risolti in maniera esatta, perché, per esempio, potrebbero avere un numero molto grande o addirittura illimitato di stati. Per questo motivo, molto spesso non è possibile trovare la action-value function ottima, ma bisogna accontentarsi di una buona approssimazione, che possiamo ottenere usando tecniche di *Approximate Dynamic Programming* [Bertsekas, 2007]. Uno di questi metodi approssimati, che è centrale al nostro lavoro, è chiamato Fitted Q Iteration [Ernst et al., 2005] e può essere visto come una versione più generale e potente di action-value iteration. Anche questo metodo infatti costruisce una sequenza di action-value function (che sono anche chiamate, in letteratura, Q function).

Il nosto lavoro fa parte di questa classe di metodi e definisce un framework, chiamato Generalized Gradient Q Iteration, che può essere visto come una generalizzazione di alcuni aspetti di Fitted Q Iteration. Come ci suggerisce il nome, il nostro metodo è pensato per funzionare con algoritmi basati su metodi a gradiente, come Adam [Kingma and Ba, 2014]. Questi metodi offrono il vantaggio di permettere di rappresentare le action-value function in maniera parameterica e di richiedere molto poco sforzo nella configurazione degli hyper-parametri. Il primo modo in cui il nostro approccio generalizza Fitted Q Iteration è che permette di controllare la granularità con la quale, ad ogni step dell'algoritmo, viene appresa la action-value function corrente. Questo fatto ci permette di costruire degli algoritmi greedy che si spostano alla prossima action-value function della sequenza, senza prima aver imparato completamente la funzione precedente. L'altro e più importante aspetto sotto il quale generalizziamo consiste nel fatto di proporre di apprendere una parametrizzazione dell'operatore di Bellman ottimo, invece di apprendere direttamente le action-value

function. L'idea fondamentale è che approssimare l'operatore di Bellman potrebbe risultare più semplice e potrebbe permettere quindi di muoverci più velocemente nello spazio delle action-value function. Entrambe queste idee sono volte ad inidirizzare la principale limitazione di metodi come Fitted Q Iteration, ovvero il fatto di avere bisogno di risolvere, ad ogni iterazione, un problema di ottimizzazione completo. Il nostro metodo cerca di porre rimedio a questo difetto e quindi di fornire tempi di convergenza inferiori, ottendendo comunque la stessa soluzione.

Abbiamo eseguito degli esperimenti, in due problemi diversi e di difficoltà crescente, dove compariamo il nostro framework a Fitted Q Iteration eseguito anch'esso con metodi a gradiente. Per entrambi gli algoritmi abbiamo usato architetture lineari, sia per le action-value function, che, nel nostro caso per la parametrizzazione degli operatori di Bellman. Quello che abbiamo potuto osservare è che in entrambi i problemi, il nostro metodo mostra performance compatibili, se non superiori a quelle di Fitted Q Iteration, usando specifiche configurazioni degli hyper-parametri. Questi risultati ci fanno ben sperare, confermando che le idee delinate in questa tesi siano promettenti e meritino di essere approfondite ulteriormente con lavori futuri.

Alcuni dei possibili sviluppi futuri del nostro lavoro includono il mettere alla prova il nostro metodo in problemi più difficili e verificare il comportamento nel dettaglio di tutti i parametri, oltre che l'ideazione e la progettazione di ulteriori architetture di approssimazione per l'operatore di Bellman, includendo per esempio architetture basate sulle reti neurali artificiali, condiserando i loro recenti successi.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overivew

Our work belongs to the research area of Reinforcement Learning, which is part of the Machine Learning (ML) field. In turn Machine Learning belongs to the broad research category of Artificial Intelligence (AI). Many definitions have been given about AI, but one of the most famous categorization of these definitions, is due to [Russell and Norvig, 2016] and divides them along two dimensions. On one dimension we have thinking and reasoning intelligently opposed to acting intelligently. On the other dimension there are two opposite ways to define success for AI: in terms of similarity to human behavior on one side and to ideal rationality on the other side. Historically all of these definitions, which imply different goals and different approaches, have been followed, giving rise to the many fields of which AI is composed today, such as natural language processing, machine learning, automated reasoning and computer vision. The applications of AI are also numerous, ranging from healthcare, to finance, video games and self-driving cars.

More specifically Machine Learning deals with the task of building systems that can learn autonomously from experience, without human intervention. Reinforcement Learning is a branch of Machine Learning and its main goal is to build agents that are able to learn autonomously, while interacting with the environment, to reach a predefined objective. In order to do so, they are allowed to perform some actions, which have effect on the environment and after each action they are given a reward, signifying how good or bad that action was in that circumstance.

The goal of this thesis is to propose a framework, which characterizes a class of algorithms, to efficiently solve Reinforcement Learning problems. We tested our framework against the most successful algorithm in the literature that belongs to the same class, obtaining good results in two different problems. This suggests that our ideas are promising and should be continued with further development and studies.

1.2 Goal and Motivation

A Reinforcement Learning agent interacts with the environment, which is modeled through the mathematical formalism of Markov Decision Processes [Bellman, 1957], by performing some actions and receiving a numerical reward signal. The goal of the agent is to maximize the total amount of reward received during time. Therefore the agent uses this reward signal as a feedback for the actions it took in each circumstance, or state of the environment, to *reinforce* good actions and penalize others. This process leads the agent to learn a *policy*, that is a mapping between states of the environment and appropriate actions. In order to find the best policy, more formally called the *optimal policy*, many agents rely on *value functions* or *action-value functions*. These functions represent respectively the current estimate of the agent regarding how good or bad it is to be in a determinate state (value function) and how good or bad it is to take a determinate action in a given state (action-value function).

In some cases the optimal policy can be determined exactly, by using some techniques known as *Dynamic Programming*. One important class of those methods is *Policy Iteration*, which includes as a special case the action-value iteration method. This method identifies a sequence of action-value functions, starting with a random function and improving it at each step, by solving some form of *Bellman Equation* [Bellman, 2013]. Bellman Equations define optimality conditions for value functions and are used in some form by every algorithm considered in this thesis. A thorough study of these methods, and many others, is given in [Sutton and Barto, 1998].

Unfortunately most problems are too complex to be solved exactly because, for example, they may have an infinite number of states. Therefore the best we can do is to find a good, but often not optimal, solution using approximate value functions and approximate techniques [Bertsekas, 2007]. One approximate method that is central for our work is called *Fitted Q Iteration* [Ernst et al., 2005] and can be seen as a more general and powerful version of action-value iteration. Also this method indeed works by computing a sequence of action-value functions (which are also referred to as Q functions).

Our work pertains to this class of methods and defines a framework called *Generalized Gradient Q Iteration*, which can be seen as generalizing Fitted Q Iteration under some aspects. As the name suggests, our method is made to work with gradient-based algorithms, such as Adam [Kingma and Ba, 2014]. These methods offer the advantage that let us model value functions in a parametric way and require very little hyper-parameter tuning. The first way in which our method generalizes Fitted Q Iteration is that it allows to control the granularity with which an action-value function is learned on each step of the algorithm. This allows us to build greedy

algorithms that move to the next function of the sequence, before having learned perfectly the previous one. The other and more important generalization is that we propose to learn a parameterization of the Bellman Operator, instead of directly learning action-value functions. The core idea is that trying to approximate the Bellman Operator could be simpler and could allow us to move faster in the space of action-value functions. Both of these ideas are meant to address the principal limitation of Fitted Q Iteration, that is the requirement to solve a complete supervised learning problem on each iteration. Our method tries to remedy this deficit, therefore providing faster convergence times, while still finding the same solution.

We tested our method against Fitted Q Iteration used in conjunction with a linear action-value architecture and gradient-based methods for solving the supervised problem at each iteration. Our method shows promising results in the experiments that we ran, confirming that the ideas of this thesis deserve to be examined in more detail in future works.

1.3 Outline

The thesis is structured in the following way. Chapter 2 is dedicated to the state of the art and gives the necessary background to understand the rest of our work. It starts by introducing the Reinforcement Learning (Section 2.1), Markov Decision Processes (Section 2.2), Bellman Equations (2.2.2) and then moves to explain some of the basic Dynamic Programming algorithms (Section 2.3). After this we briefly discuss about Approximate Solutions methods (Section 2.4) and review some of the theory and techniques needed to work with real problems with large or unbounded state spaces. Lastly we introduce the algorithm of Fitted Q Iteration in Subsection 2.4.3.

Chapter 3 contains the main part of thesis and explains the theoretical foundations of our work. It starts (Section 3.1) by laying down the necessary notation by showing how, starting from Q Iteration, we can generalize several concepts step-by-step. Then it introduces the framework of Generalized Gradient Q Iteration (Section 3.2) breaking it into three procedures: the core procedure, the cost function and the weight update procedure. The next section (Section 3.3) examines two possible approximation architectures for the Bellman Operator, showing pros and cons of both. The last section of the chapter (Section 3.4) reviews the most important hyper-parameter of our framework that can be tuned to produce different algorithms and also explains how under some configuration our framework can be reduced to Fitted Q Iteration.

Chapter 4 describes some of the experiments that we ran to test the performance of our framework. We compare various configurations of the hyper-parameters, showing how in each case we manage to improve upon the performances of Fitted Q

Iteration.

Finally Chapter 5 draws our conclusions and highlights some possible future developments.

Chapter 2

State of the Art

In this chapter we will broadly explore the research field of Reinforcement Learning, starting with a general introduction (Section 2.1); a rigorous definition of the Reinforcement Learning problem using Markov Decision Processes (Section 2) and an overview of the main techniques to solve it efficiently (Sections 3-5).

2.1 What is Reinforcement Learning

Definition 2.1. Reinforcement Learning (RL) is the activity performed by an *agent* that tries to maximize a *reward signal*, while interacting with the *environment*, to achieve a *goal*.

Definition 2.2. A Reinforcement Learning Problem is the definition of the task that the agent is asked to perform. This includes the definition of the agent, the reward signal, the environment, and the goal.

Example 2.1. As an example, consider the game of Tetris: the agent is whoever is playing the game, either human or an algorithm; the goal is to clear all the levels; the reward signal to be maximized is the game's score, and the environment is the playing field.

In a Reinforcement Learning Problem the agent interacts with the environment by performing some *actions*, observing what effect they have both on the environment and reward signal, and using this knowledge to improve its behavior.

For example, in the Tetris Game, the possible action[1] is either moving the falling piece sideways or rotating it by $\pm 90°$. Therefore an agent that does not have any previous knowledge about the game will probably start by performing some random actions and, after a while, observing that when the fallen pieces are aligned to form a horizontal line without gaps, the score increases. A RL agent will consequently

[1] To be precise waiting while the piece falls down is also a possible action

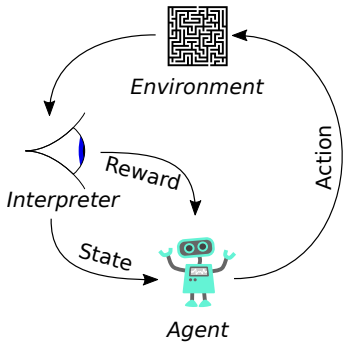

Figure 2.1: Reinforcement Learning

improve its behavior by trying to repeat the sequence of actions that led to an increase in the reward[2].

2.2 Markov Decision Processes

Markov Decision Processes (MDPs) [Bertsekas, 2005] are a mathematical formalization of the Reinforcement Learning problem.

Definition 2.3 (Markov Decision Process)**.** A Markov Decision Process is a discrete time stochastic process, represented by a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$. At each time step $t \in \mathbb{N}^0$ the agent observes the environment, which is at state $S_t \in \mathcal{S}$, and decides to perform an action $A_t \in \mathcal{A}$. The process then transitions to state S_{t+1} giving a reward $R_{t+1} \in \mathcal{R}$ to the agent, according to probability $p(S_{t+1}, R_{t+1}|S_t, A_t)$.

- S is the set of environment states observable by the agent.
- $\mathcal{A}$ is the set of actions that the agent can select in each state[3].
- $\mathcal{R} \subset \mathbb{R}$ is the set of rewards that can be given to the agent.
- $p : \mathcal{S} \times R \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a four argument function giving the probability of transitioning from state s to state s' performing action a and obtaining reward r:
$$p(s', r|s, a) \doteq \mathbb{P}\left(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\right).$$
- $\gamma \in \mathbb{R}$ is the discount factor. The smaller the discount factor the smaller the importance of future rewards compared to immediate rewards.

Unless otherwise specified we consider the sets of states $\mathcal{S}$ and actions $\mathcal{A}$ to be finite and ordered. This enables us to reference any state with the notation s_i where $0 \leq i < |\mathcal{S}|$ and every action with a_i where $0 \leq i < |\mathcal{A}|$.

[2]The increased score is effectively a positive *reinforcement* of the desired behaviour of the agent.
[3]We do not consider here the case in which the set of actions depends on the current state: $\mathcal{A}(\mathcal{S})$

Definition 2.4 (Markov Property)**.** Markov Decision Process are memory less, meaning that the current state and action are all that is needed to determine the probability of the next state and reward. In other words, the distribution probability of S_{t+1}, R_{t+1} does not depend on the *trajectory* $S_0, A_0, R_1, S_1 \dots S_t, A_t$ but only on the current state S_t and action A_t. This important property is formally expressed by the following lemma:

$$\mathbb{P}\left(S_{t+1} = s_{t+1}, R_{t+1} = r_{t+1} \mid S_0 = s_0, A_0 = a_0, \dots, S_t = s_t, A_t = a_t\right) = \\ \mathbb{P}\left(S_{t+1} = s_{t+1}, R_{t+1} = r_{t+1} \mid S_t = s_t, A_t = a_t\right).$$

Definition 2.5 (Episodic and continuing tasks)**.** Some problems, called *episodic tasks*, have a finite number of time steps, while others called *continuing tasks* have a potentially infinite number of time steps.

As an example episodic task, we can consider a card game, like Poker, in which every game is an episode and has a finite number of turns. On the other hand, a control problem, like the pole-balancing task, in which the agent has to maintain the system balanced as long as possible, is a typical example of a continuing problem.

In order to make uniform the mathematical notation for episodic and continuing task, we introduce the concept of absorbing state:

Definition 2.6 (Absorbing state)**.** Let $s \in \mathcal{S}$ be a state. Then s is *absorbing* if and only if

$$\mathbb{P}\left(S_{t+1} = s \mid S_t = s\right) = 1.$$

We can then consider every episodic task as a continuing task by adding an absorbing state ab to the set of states $\mathcal{S}$, with reward 0:

$$\mathbb{P}\left(S_{t+1} = ab, R_{t+1} = 0 \mid S_t = ab\right) = 1.$$

Definition 2.7 (Policy)**.** Any agent interacting with a Markov Decision Process follows a policy, that specifies its behavior in each state. Formally a policy $\pi(a|s)$ is a function $\mathcal{A} \times \mathcal{S} \to [0, 1]$, giving the probability of selecting action a from state s:

$$\pi(a|s) \doteq \mathbb{P}\left(A_t = a \mid S_t = s\right).$$

Definition 2.8 (Deterministic Policy)**.** A policy π is deterministic if for each state s there exists only one action a with probability greater than 0 of being selected.

$$\exists! a \text{ such that } \mathbb{P}\left(A_t = a \mid S_t = s\right) > 0.$$

Since the probability of selecting any other action is 0, the probability of selecting a is 1. Therefore we can simply the notation for deterministic policies by considering them a mapping $\mathcal{S} \to \mathcal{A}$ from states to actions:

$$\pi(s) = a \iff \pi(s|a) = 1.$$

Definition 2.9 (Goal)**.** We define the goal of the agent to be finding the policy π that leads to the maximization of the following function of the rewards sequence, called *discounted return* and denoted G_t:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

The discount factor γ is used to give more importance to rewards obtained sooner in time, as opposed to the ones obtained further in the future.

Note that G_t can be expressed recursively as:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} R_{t+1+k} && \text{let } k = j+1 \\ &= R_{t+1} + \gamma \sum_{j=0}^{\infty} \gamma^j R_{t+1+j+1} = R_{t+1} + \gamma G_{t+1} && . \end{aligned}$$

2.2.1 Value functions

In order to find the policy π that maximizes the discounted return, we need a way to rank policies. This is accomplished with *value functions*. Intuitively the higher the return while following a certain policy, the higher is its value.

Definition 2.10 (State-value Function)**.** Let π be a policy. Then $v_\pi : \mathcal{S} \to \mathbb{R}$ is the state-value function for policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right].$$

The state-value function maps each state $s \in \mathcal{S}$ to the expected return obtained from s while following policy π.

Definition 2.11 (Action-value Function)**.** Let π be a policy. Then $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the action-value function for policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right].$$

The action-value function maps each state-action pair $(s, a) \in \mathcal{S} \times \mathcal{A}$ to the expected return obtained from state s, selecting action a and then following policy π.

Theorem 2.1 (Matrix notation)**.** *If the sets of states $\mathcal{S}$ and actions $\mathcal{A}$ are finite, then any state-value function v_π and action-value function q_π can be represented respectively as a vector $\boldsymbol{v}_\pi \in \mathbb{R}^{|\mathcal{S}|}$ and as a vector $\boldsymbol{q}_\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$.*

Proof. If $\mathcal{S}$ is finite, then we can define an arbitrary ordering of its elements. Let $s_i \in \mathcal{S}$ denote the i-th state, where $0 \leq i < |\mathcal{S}|$, then we can define $\boldsymbol{v}_\pi$ as:

$$\boldsymbol{v}_\pi \doteq \left(v_\pi\left(s_0\right), \ldots, v_\pi\left(s_{|\mathcal{S}|}\right)\right)^T.$$

The proof for $\boldsymbol{q}_\pi$ is similar. □

2.2.2 Bellman Equations

Value functions can be expressed as recursive equations, called Bellman Equations, highlighting the relationship between the value of current and successive states.

Lemma 2.1. First we show how the State-value and Action-value function are mutually recursive:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) q_\pi(s, a). \end{aligned} \tag{2.1}$$

Theorem 2.2 (Bellman Equation for q_π)**.**

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \left\{r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right\} \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi(s')\right] \qquad (2.2) \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q_\pi(s', a')\right]. \qquad (2.3) \end{aligned}$$

Theorem 2.3 (Bellman Equation for v_π)**.**

$$\begin{aligned} v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a \mid s) q_\pi\left(s, a\right) \qquad (2.4) \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi(s')\right]. \qquad (2.5) \end{aligned}$$

Like value functions also the Bellman equations for v_π and q_π can be represented

in a more compact form using Matrix notation. In order to do so, let us first define a matrix form for the rewards and the transition probabilities.

Let $\boldsymbol{P}_\pi = (p_{ij}) \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{A}|}$ be the *state transition matrix*, where each element p_{ij} denotes the probability of transitioning from state s_i to state s_j under policy π. We can define $\boldsymbol{P}_\pi$ in terms of the transition probability function $p(s', r|s, a)$ as follows:

$$p_{ij} = \mathbb{P}\left(S_{t+1} = s_j \mid S_t = s_i\right) = \sum_{a\in\mathcal{A}} \pi(a \mid s_i) \sum_{r\in\mathcal{R}} p\left(s_j, r \mid s_i, a\right). \tag{2.6}$$

Let $\boldsymbol{r}_\pi = (r_i) \in \mathbb{R}^{|\mathcal{S}|}$ be the vector of expected rewards obtained from every state, and defined as:

$$r_i = \mathbb{E}\left[R_{t+1} \mid S_t = s_i\right] = \sum_{a\in\mathcal{A}} \pi(a \mid s) \sum_{r\in\mathcal{R}} r \sum_{s_j\in\mathcal{S}} p\left(s_j, r \mid s_i, a\right). \tag{2.7}$$

Then the Bellman equation for v_π (2.5) can be written in matrix form as:

$$\boldsymbol{v}_\pi = \boldsymbol{r}_\pi + \gamma \boldsymbol{P}_\pi \boldsymbol{v}_\pi. \tag{2.8}$$

A similar procedure can be done for q_π by letting $P_\pi \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|\times|\mathcal{S}||\mathcal{A}|}$ be the *state-action transition matrix* describing the probability of going from (s, a) to (s', a') under policy π and $\boldsymbol{r} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ to be the vector of expected rewards obtained from (s, a). Then the Bellman equation for q_π (2.3) can be written as:

$$\boldsymbol{q}_\pi = \boldsymbol{r} + \gamma \boldsymbol{P}_\pi \boldsymbol{q}_\pi. \tag{2.9}$$

Definition 2.12 (Bellman Operators)**.** The Bellman Equation for q_π (2.3) can be interpreted as defining a function operator T_π mapping action-value functions, in the following way:

$$\begin{aligned} T_\pi : \mathbb{R}^{\mathcal{S}\times\mathcal{A}} &\rightarrow \mathbb{R}^{\mathcal{S}\times\mathcal{A}} \\ \left(T_\pi q\right)(s, a) &\doteq \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma \sum_{a'\in\mathcal{A}} \pi(a' \mid s') q(s', a')\right]. \end{aligned} \tag{2.10}$$

The same thing can be done for v_π, by using eq. (2.5):

$$\begin{aligned} T_\pi : \mathbb{R}^{\mathcal{S}} &\rightarrow \mathbb{R}^{\mathcal{S}} \\ \left(T_\pi v\right)(s) &\doteq \sum_{a\in\mathcal{A}} \pi(a \mid s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma v(s')\right]. \end{aligned} \tag{2.11}$$

Bellman Operators can be formulated for the matrix version of the Bellman equations as well:

$$\begin{aligned} \boldsymbol{T}_\pi \boldsymbol{q} &= \boldsymbol{r} + \gamma \boldsymbol{P}_\pi \boldsymbol{q} \\ \boldsymbol{T}_\pi \boldsymbol{v} &= \boldsymbol{r}_\pi + \gamma \boldsymbol{P}_\pi \boldsymbol{v}. \end{aligned} \tag{2.12}$$

Note that although we used the same symbol T_π for the Bellman Operator of both state-value and action-value functions, it will be clear from the context which of the two is implied.

We are now ready to describe in more detail the concept of *optimal policy*, that is a policy that represents the best that can be done in a given MDP.

Definition 2.13 (Optimal policy)**.** A policy is said to be optimal if its expected return v_* is greater or equal to that of any other policy:

$$v_*(s) \doteq \max_\pi v_\pi(s).$$

We denote an optimal policy with π_* and its state-value and action-value functions respectively with v_* and q_*.

Since the Bellman Equations for v_π and q_π, that we defined above, are valid for the value functions of *any* policy, they must apply also to the state-value function v_* and action-value function q_* of the optimal policy π_*.

This enables us to find two recursive equations, called respectively the Bellman optimality equation for v_* and q_*, similar to the ones defined for v_π and q_π.

Theorem 2.4. *The value $v_*(s)$ of each state $s \in \mathcal{S}$, while following an optimal policy π_*, must be equal to the maximum action-value $q_*(s,a)$ obtainable from that state:*

$$v_*(s) = \max_{a \in \mathcal{A}} q_* (s,a) .$$

Proof. First we note how v_* and q_* must obey the mutually recursive relation defined in Lemma 2.1:

$$v_*(s) = \sum_{a \in \mathcal{A}} \pi_*(a \mid s) q_* (s,a) . \qquad \text{(by eq. 2.1)}$$

Then we prove that v_* must be less than or equal to $\max_a q_* (s,a)$:

$$
\begin{aligned}
\forall a \in \mathcal{A} \left(q_*(s,a) \leq \max_{\bar{a} \in \mathcal{A}} q_*(s,\bar{a}) \right) &\implies \\
\sum_{a \in \mathcal{A}} \pi_*(a \mid s) q_*(s,a) \leq \sum_{a \in \mathcal{A}} \pi_*(a \mid s) \max_{\bar{a} \in \mathcal{A}} q_*(s,\bar{a}) &\implies \\
\sum_{a \in \mathcal{A}} \pi_*(a \mid s) q_*(s,a) \leq \max_{\bar{a} \in \mathcal{A}} q_*(s,\bar{a}) &\implies \\
v_*(s) \leq \max_{a \in \mathcal{A}} q_*(s,a).
\end{aligned}
$$

Finally we show by contradiction that $\max_a q_*(s,a)$ cannot be greater than v_*. Suppose that there exist a state $\sigma \in \mathcal{S}$ where $\max_a q_*(\sigma,a) > v_*(\sigma)$. Then we can construct a policy π', that behaves identically to π_* except in σ, where we set:

$$
\pi'(\sigma) = \operatorname*{argmax}_{a \in \mathcal{A}} q_*(\sigma,a).
$$

Then we have:

$$
v_{\pi'}(\sigma) = \max_{a \in \mathcal{A}} q_*(\sigma,a) > v_*(\sigma).
$$

But this contradicts the fact that π_* is an optimal policy, and therefore proves our theorem. □

Proposition 2.1 (Bellman optimality equation for v_*)**.**

$$
\begin{aligned}
v_*(s) &= \max_a q_*(s,a) && \text{(by Theorem 2.4)} \\
&= \max_a \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_*(s') \right]. && \text{(by eq. 2.2)}
\end{aligned}
$$

Proposition 2.2 (Bellman optimality equation for q_*)**.**

$$
\begin{aligned}
q_*(s) &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_*(s') \right] && \text{(by eq. 2.2)} \\
&= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]. && \text{(by Theorem 2.4)}
\end{aligned}
$$

2.3 Dynamic Programming

In this section we introduce a series of methods, known in the literature as Dynamic Programming, that *exactly* solve the Reinforcement Learning problem of finding the optimal policy π_*, assuming a *perfect knowledge* of the underlying Markov Decision Process.

These methods are called *exact methods* because they store value functions in a tabular form, without approximations. More specifically, to exactly represent a state-value function v_π or an action-value function q_π, we need to store respectively

a value for each state $s \in \mathcal{S}$ and for each state-action pair $s, a \in \mathcal{S} \times \mathcal{A}$. Therefore the memory required is proportional to $\mathcal{S}$ in the former case and to $\mathcal{S} \times \mathcal{A}$ in the latter.

These methods are called *Model-based methods* because they require a model of the environment.

Definition 2.14 (Models)**.** A model is something that simulates the behavior of the real environment and allows to obtain simulated experience while interacting with it. Models can be of two types: *distribution models* and *sample models*.

Distribution models are a complete model of the environment, in the form of a Markov Decision Process, therefore including the set of states $\mathcal{S}$, actions $\mathcal{A}$, rewards $\mathcal{R}$ and transition probabilities p.

Sample models, which are usually easier to obtain, only allow the agent to *sample* values according to the transition probabilities, but do not give direct access to the underlying distributions. That is, given an initial state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, an agent can ask for a simulated next state $s' \in \mathcal{S}$ and reward $r \in \mathcal{R}$, but cannot directly know the probability $p(s', r \mid s, a)$.

All the algorithms presented in this section assume to have a distribution model.

Before exploring Policy Iteration, which is the central algorithm of this section, we will look more closely at two other algorithms: Policy Evaluation and Policy Improvement. These will form the basis on which to build the Policy Iteration algorithm.

2.3.1 Policy Evaluation

The objective of policy evaluation is to compute the state-value function v_π associated with a policy π. This can be done by finding the value that satisfies the Bellman Equation for v_π (2.5) for each state $s \in \mathcal{S}$:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi(s')\right].$$

Assuming the knowledge of the environment transition probabilities (i.e. to have a distribution model) then the above condition is equivalent to solving the linear system of $|\mathcal{S}|$ equations expressed by the matrix form of the Bellman equation for v_π (2.8).

The solution to the above system of equations can also be computed in a more convenient way, using an iterative method, which is shown in Algorithm 2.1.

The algorithm starts with a random[4] estimate $\boldsymbol{v}_0$ of the state-value function v_π, and at each step k it produces a new estimate $\boldsymbol{v}_{k+1}$ by applying the Bellman operator (2.12) to the previous estimate $\boldsymbol{v}_k$:

[4]The only requirement is that absorbing states are initialized to 0

Algorithm 2.1 Policy Evaluation

`input:` π policy for which to compute v_π
`param:` θ threshold > 0, to determine convergence

$\boldsymbol{v}_0 \leftarrow 0$
$k \leftarrow 0$
$\delta \leftarrow +\infty$
`while` δ `>` θ`:`
  $\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{T}_\pi \boldsymbol{v}_k = \boldsymbol{r}_\pi + \gamma \boldsymbol{P}_\pi \boldsymbol{v}_k$
  $\delta \leftarrow \|\boldsymbol{v}_{k+1} - \boldsymbol{v}_k\|_\infty$
  $k \leftarrow k + 1$

$$\boldsymbol{v}_{k+1} \leftarrow \boldsymbol{r}_\pi + \gamma \boldsymbol{P}_\pi \boldsymbol{v}_k.$$

The sequence of state-value functions $\{\boldsymbol{v}_0, \boldsymbol{v}_1, \dots\}$ is guaranteed to converge to $\boldsymbol{v}_\pi$ as k approaches infinity, because the Bellman operator (2.12) represents a contraction mapping with $\boldsymbol{v}_\pi$ as the unique fixed point.

Since convergence happens only in the limit, we need a more practical stopping criterion. For this reason Algorithm 2.1 has a parameter $\theta \in \mathbb{R}$ and keeps iterating until $\|\boldsymbol{v}_{k+1} - \boldsymbol{v}_k\|_\infty \leq \theta$.

2.3.2 Policy Improvement

Policy evaluation allows us to estimate the state-value function v_π of a policy π. We will now show how we can use this information to improve the current policy π, thanks to the *Policy Improvement Theorem*.

Theorem 2.5 (Policy Improvement Theorem)**.** *Let π and $\bar{\pi}$ be policies such that $\forall s \in \mathcal{S}$:*

$$\mathbb{E}_{\bar{\pi}} \left[v_\pi(S_t) \leq q_\pi(S_t, A_t) \mid S_t = s \right]. \tag{2.13}$$

Then policy $\bar{\pi}$ is as good as, or better than, policy π. Meaning that $v_\pi(s) \leq v_{\bar{\pi}}(s)$ for all states $s \in \mathcal{S}$.

Algorithm 2.2 Policy Improvement

`input:` $\boldsymbol{\pi} \in \mathcal{A}^{|\mathcal{S}|}$deterministic policy as vector
$\boldsymbol{v} \in \mathbb{R}^{|\mathcal{S}|}$ vector representing the state-value function
// Compute vector $\boldsymbol{q} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ *with components* $q(s,a)$ *as:*
$q(s,a) \leftarrow \sum_{j,r} p(s', r \mid s, a)[r + \gamma v(s)]$
$\pi'(s) \leftarrow \texttt{argmax}_a q(s,a)$ *// Compute new policy* π' *for each* $s \in \mathcal{S}$
`is-stable` $\leftarrow \pi = \pi'$
`return is-stable`

Proof.

$$
\begin{aligned}
\mathbb{E}_{\bar{\pi}}\left[v_\pi(S_t) \leq q_\pi(S_t, A_t) \mid S_t = s\right] &\implies \\
\mathbb{E}_{\bar{\pi}}\left[\sum_{k=0}^{\infty} \gamma^k v_\pi(S_{t+k}) \leq \sum_{k=0}^{\infty} \gamma^k q_\pi(S_t, A_t) \mid S_t = s\right] &\implies \\
\mathbb{E}_{\bar{\pi}}\left[\sum_{k=0}^{\infty} \gamma^k v_\pi(S_{t+k}) \leq \sum_{k=0}^{\infty} \gamma^k \left(R_{t+k+1} + \gamma v_\pi(S_{t+k+1})\right) \mid S_t = s\right] &\implies \\
\mathbb{E}_{\bar{\pi}}\left[\sum_{k=0}^{\infty} \gamma^k v_\pi(S_{t+k}) \leq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} + \sum_{k=0}^{\infty} \gamma^{k+1} v_\pi(S_{t+k+1}) \mid S_t = s\right] &\implies \\
\mathbb{E}_{\bar{\pi}}\left[\sum_{k=0}^{\infty} \gamma^k v_\pi(S_{t+k}) \leq G_t + \sum_{j=1}^{\infty} \gamma^j v_\pi(S_{t+j}) \mid S_t = s\right] &\implies \\
\mathbb{E}_{\bar{\pi}}\left[v_\pi(S_t) \leq G_t \mid S_t = s\right] &\implies \\
v_\pi(s) \leq \mathbb{E}_{\bar{\pi}}\left[G_t \mid S_t = s\right] &\implies \\
v_\pi(s) \leq v_{\bar{\pi}}(s).
\end{aligned}
$$

□

Proposition 2.3. *Note that the precondition* (2.13) *of the Policy Improvement Theorem, in case policy* π *is deterministic, becomes:*

$$v_\pi(s) \leq q_\pi\left(s, \bar{\pi}(s)\right).$$

Definition 2.15 (Greedy Policy)**.** Let π be a policy and q_π its action-value function. Then we define the *greedy policy* with respect to q_π as:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\text{argmax}} q_\pi(s,a).$$

Note how the greedy policy π' respects by construction the precondition (Proposition 2.3) of the Policy Improvement Theorem:

$$q_\pi\left(s, \pi'(s)\right) = \max_{a \in \mathcal{A}} q_\pi\left(s, a\right) \geq v_\pi(s),$$

and therefore it is guaranteed to be as good as, or better than π. That is $v_{\pi'}(s) \geq v_\pi(s)$ for all states $s \in \mathcal{S}$.

Theorem 2.6 (Stability implies optimality)**.** *Let π be a policy and π' the greedy policy with respect to v_π. Then*

$$\pi' = \pi \implies \pi' \text{ is optimal.}$$

Proof.

$$\begin{aligned} \pi(s) = \pi'(s) \qquad & \implies \\ \pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} q_\pi\left(s, a\right) & \implies \\ v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a). & \end{aligned}$$

□

We can then create an algorithm, called Policy Improvement Algorithm (2.2), that given a policy π and its state-value function v_π, computes a new improved policy π' that is greedy with respect to q_π. Theorem 2.6 shows how the greedy policy π' computed by the Policy Improvement algorithm is always strictly better than π, unless π is already optimal. The algorithm returns `True` in case the policy is *stable*, meaning that $\pi' = \pi$, and `False` otherwise.

2.3.3 Policy Iteration

The Policy Iteration Algorithm, shown in Algorithm 2.3, works by taking as input an initial policy π and repeatedly invoking *policy evaluation* and *policy improvement*, until the policy π converges to the optimal policy π_*.

Algorithm 2.3 policy-iteration

```
input: π initial policy
for s ∈ S:
  v_π(s) = 0
policy-stable = False
while not policy-stable:
  policy-evaluation(π, v_π)
  policy-stable = policy-improvement(π, v_π)
```

The initial policy π supplied to the algorithm, can be any policy at all. Usually it is taken to be as the random policy[5].

[5]The random policy is the policy that at each state assigns equal probability to all possible

2.4 Approximate Solution Methods

In this section we show approximated methods for solving the Reinforcement Learning problem. As opposed to the exact ones of the previous sections, we assume that these methods cannot exactly represent value functions and therefore we call them *approximated methods*. The reason for this is that many interesting Reinforcement Learning problems have an infinite or very large state space $\mathcal{S}$, that is either very hard to fit into memory or too computationally expensive to deal with. Therefore we assume that approximated value functions are parameterized by a weight vector $\boldsymbol{w} \in \mathbb{R}^d$, and, following Sutton's notation, we denote them as:

$$\tilde{v}(s, \boldsymbol{w}) \approx v_\pi(s).$$

The purpose of parameterizing our function with a weight vector $\boldsymbol{w}$, is that the number of weights is usually much smaller than the number of states ($d \ll |\mathcal{S}|$). The most common example is that of linear function approximation:

$$\tilde{v}(s, \boldsymbol{w}) \doteq \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w} = \sum \phi_i(s) \cdot w_i,$$

where $\boldsymbol{\phi}(s) \in \mathbb{R}^d$, called feature vector for state s, is a vector of the same dimension of $\boldsymbol{w}$.

In this section we focus on the task of approximating the true value function v_π of a policy π. For this purpose we interact with the environment to collect a series of *samples,* which mimic the desired input-output behavior of the value-function. We denote those samples $S_t \mapsto U_t$, where S_t is the state to be updated (input) and U_t is the desired target value (output). We can then use potentially any supervised learning method to produce an approximation of v_π.

Since we cannot represent all state-value pairs exactly, we need a metric to indicate how well we are approximating our target. This is accomplished by defining a cost function $C(\boldsymbol{w})$ as the prediction objective that we aim to minimize. We select our prediction objective function to be the *Mean Squared Value Error* between the true value function v_π and our approximation $\tilde{v}$:

$$C(\boldsymbol{w}) \doteq \frac{1}{2} \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \tilde{v}(s, \boldsymbol{w}) \right]^2. \tag{2.14}$$

2.4.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a supervised learning method that updates the weight vector $\boldsymbol{w}$ at each time step t, according to the gradient of the cost function C with respect to $\boldsymbol{w}$:

---

actions

name	target U_t	bootstrap	exact
True Value	$v_\pi(S_t)$	×	✓
Monte Carlo	G_t	×	×
DP	$\sum_{a,s',r} \pi(a \mid S_t)\, p(s', r \mid S_t, a)\, [r + \gamma \tilde{v}(s', \boldsymbol{w}_t)]$	✓	✓
TD(0)	$S_t \to R_{t+1} + \gamma \tilde{v}(S_{t+1}, \boldsymbol{w}_t)$	✓	×

Table 2.1: Update targets

$$\boldsymbol{w}_{t+1} \doteq \boldsymbol{w}_t - \alpha \nabla C, \tag{2.15}$$

where the gradient ∇C is the column vector of partial derivatives of C, defined as follows:

$$\nabla C(\boldsymbol{w}) \doteq \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_d} \right)^T. \tag{2.16}$$

The gradient ∇C is scaled by a small positive factor α in equation (2.15), which must be decreased over time, to ensure convergence to a local minimum of C.

This method is an instantiation of the more general Gradient Descent algorithm and is called stochastic because the gradient is computed on a single, stochastically selected sample.

Theorem 2.7. *The cost $C(\boldsymbol{w}_t)$ decreases with each update of $\boldsymbol{w}$:*

$$\Delta C \doteq C(\boldsymbol{w}_{t+1}) - C(\boldsymbol{w}_t) \leq 0 \qquad \forall t.$$

Proof.

$$\begin{aligned} \Delta C &\approx \nabla C \cdot \Delta \boldsymbol{w} \\ &= \nabla C\, (-\alpha \nabla C) \\ &= -\alpha \left\| \nabla C \right\|^2 \leq 0. \end{aligned}$$

□

The Stochastic Gradient Descent update rule (2.15) can be expressed in a more explicit form by expanding the definition of cost function (2.14) and computing the gradient:

$$\begin{aligned} \boldsymbol{w}_{t+1} &\doteq \boldsymbol{w}_t - \alpha \nabla C \\ &\approx \boldsymbol{w}_t - \alpha \nabla \left[v_\pi \left(S_t \right) - \hat{v} \left(S_t, \boldsymbol{w}_t \right) \right]^2 \\ &= \boldsymbol{w}_t + \alpha \left[v_\pi \left(S_t \right) - \hat{v} \left(S_t, \boldsymbol{w}_t \right) \right] \nabla \hat{v} \left(S_t, \boldsymbol{w}_t \right). \end{aligned} \tag{2.17}$$

Note that the gradient of v_π is 0 because we assume to have the true value-function, which does not depend on the weight vector $\boldsymbol{w}$. In the general case, though, the target U_t of our training samples $S_t \rightarrow U_t$ won't be the true value $v_\pi \left(S_t \right)$ of state S_t, but an approximation of it. However SGD is still guaranteed to converge to a local minimum of C, in the case in which U_t is an unbiased estimate of v_π for all time steps t:

$$\mathbb{E} \left[U_t \mid S_t \right] = v_\pi (S_t).$$

The Monte Carlo update target $U_t \doteq G_t$ is an example of unbiased estimate. In this case the update rule 2.17 becomes:

$$\boldsymbol{w}_{t+1} \doteq \boldsymbol{w}_t + \alpha \left[U_t - \hat{v} \left(S_t, \boldsymbol{w}_t \right) \right] \nabla \hat{v} \left(S_t, \boldsymbol{w}_t \right). \tag{2.18}$$

2.4.2 Semi Gradient Methods: PVI, LSPE and TD(0)

If the target U_t is a bootstrapping estimate of $v_\pi \left(S_t \right)$, such as the TD(0) target or the DP target (both shown in Table 2.1 on page 35), then it is biased and the update rule (2.18) is not guaranteed to converge. The reason is that, in this case, the target U_t depends on the weight vector $\boldsymbol{w}_t$, but we are not taking this fact into account when computing the gradient of C. Therefore bootstrapping methods that use eq. (2.18) as their update rule are not proper instances of Stochastic Gradient Descent, but are instead called *Semi Gradient methods*.

These methods, despite not converging to a local minimum as robustly as Gradient Descent methods, offer nonetheless some advantages, including faster learning and the ability to update the weight vector $\boldsymbol{w}_t$ immediately without waiting for the end of the episode. One notable instance of such methods that is guaranteed to converge in the linear case is Semi-Gradient TD(0), which uses the update rule (2.18) with the TD(0) target (shown in Table 2.1 on page 35).

Two other methods belonging to the same class and that converge to the same solution as Semi Gradient TD(0) are Projected Value Iteration (PVI) and Least Squares Policy Evaluation (LSPE). The difference between them is that the former is model-based and uses the DP update target, while the latter is model-free and uses the TD(0) update target. Both of those methods differ from Semi Gradient

TD(0) in that they perform the gradient computation using more than one sample, while stochastic methods discussed so far use only one.

2.4.3 Fitted Q Iteration

Fitted Q Iteration [Ernst et al., 2005] is a batch mode Reinforcement Learning algorithm, that approximates the optimal action-value function q_*, by solving a sequence of regression problems.

In batch mode learning, the agent does not interact directly with the environment, but receives a dataset $\mathcal{D}$ of four-tuples $(S_t, A_t, R_{t+1}, S_{t+1})$ collected by an other agent, which must be used to learn the best possible approximation of the optimal policy.

Let us define the following sequence of action-value functions, by repeatedly applying the optimal Bellman operator:

$$\begin{aligned} \boldsymbol{q}_0 &\doteq 0 \\ \boldsymbol{q}_k &\doteq \boldsymbol{T}\boldsymbol{q}_{k-1}. \end{aligned}$$

This is the sequence computed by the action-value iteration algorithm and therefore is guaranteed to converge to the unique fixed point of $\boldsymbol{T}$, that is the optimal action-value function $\boldsymbol{q}_*$. The Fitted Q Iteration algorithm (2.4) computes a similar sequence that is an approximation of the above. It starts by initializing the current action-value function $\tilde{q}_0$ to 0 and then at each iteration it builds a training set using the dataset $\mathcal{D}$ and the current action-value function $\tilde{q}_k$, and uses it to compute the next action-value function $\tilde{q}_{k+1}$ by applying a supervised learning method.

The Fitted Q Iteration algorithm supports the use of any supervised learning method to fit the next action-value function $\tilde{q}_{k+1}$ at each step, even non-parametric ones like kernel based methods. To emphasize that a weight vector $\boldsymbol{w}$ is not required in all cases, we denote the action-value functions as $\tilde{q}\,(s, a)$ instead of $\tilde{q}(s, a, \boldsymbol{w})$.

Algorithm 2.4 Fitted Q Iteration

`input:` $\mathcal{D}$ dataset of four-tuples

```
while not stopping-condition():
  // Build dataset consisting of input i = (i_i) and
  // output o = (o_i) vectors of size |D|
  for (s, a, r, s') ∈ D:
    i_i ← (s, a)
    o_i ← r + γ max_a q̃_k (s', a)
  k ← k + 1
  q̃_k ←fit(i, o)
```

If we assume our problem to be deterministic and to known the exact action-value function q_k, then the output vector $\boldsymbol{o}$ built by the algorithm at iteration k, is indeed equal to the true value of q_{k+1} for each input element $(s, a) \in \boldsymbol{i}$. This training set can then be used by the `fit` procedure to generalize to unseen (s, a) samples and produce an approximation $\tilde{q}_{k+1}$ of q_{k+1}. In a stochastic context, the values of the output vector $\boldsymbol{o}$ are not identical anymore to the corresponding true action-values q_k. However their expected value is equal to the true action-value function q_k and therefore the `fit` procedure can provide anyway an approximation $\tilde{q}_{k+1}$ of q_{k+1}.

The convergence of the sequence of action-value functions identified by Fitted Q Iteration has been proven to converge under the assumption of using kernel-based methods (see [Ernst et al., 2005]). Unfortunately those methods are non-parametric and therefore cannot be used for our work. Nonetheless Fitted Q Iteration has been shown to provide good results even when using Artificial Neural Networks as the approximation architecture for the action-value function [Riedmiller, 2005].

Chapter 3

Theoretical Analysis

In this chapter we introduce the original part of our work: the development of the Generalized Gradient Q Iteration (GGQI) framework. GGQI is a generic method that designates a class of different algorithms, therefore we refer to it as a framework.

Gradient-based GGQI pertains to the category of approximate dynamic programming (or approximate solutions methods). More specifically it requires a *parametric* approximating architecture that can be fitted with *gradient*-based methods. This is a stricter requirement compared to the approximate methods introduced in Chapter 1. Most of those methods can use any supervised learning algorithm because they are able to obtain input/output samples $x \mapsto y$, which define the behavior of the function to approximate. In our case we assume this is not possible. Therefore gradient-based optimization algorithms, like SGD and Adam, are at the core of our method, since they allow us to approximate a target function simply by minimizing a cost function C, without providing explicit input/output samples.

Generalizing Q Iteration The method that we propose can be seen as a generalization[1] of various methods belonging to the class of action-value iteration, more commonly referred to as Q iteration. These methods, among which are tabular action-value iteration and Fitted Q Iteration, try to compute a good approximation of the optimal q function by generating a sequence of action-value functions $\{\tilde{q}_0, \tilde{q}_1, \ldots, \tilde{q}_K\}$ through repeated application of some Bellman operator.

Learning the Bellman Operator The core idea of our approach, and at the same time the main difference with other methods, is to build an approximation of the projected optimal Bellman operator, that we can use to guide our search for the best action-value function.

[1] Meaning that it can be reduced to those algorithms, in particular circumstances

3.1 Generalizing Q Iteration

We now go through the process of introducing the necessary concepts and notation needed to properly describe our method.

Q Iteration If we assume to know the environment's transition probabilities, we are able to compute the Bellman Optimality Operator, for any action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$:

$$\begin{aligned} T : \mathbb{R}^{\mathcal{S}\times\mathcal{A}} &\to \mathbb{R}^{\mathcal{S}\times\mathcal{A}} \\ (Tq)(s,a) &\doteq \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_a q(s', a')\right]. \end{aligned}$$

Then the easiest thing to do in order to obtain the optimal action-value function q_*, is to use the Q iteration algorithm, which by repeatedly applying the Bellman operator identifies a sequence of action-value functions $\{q_0, q_1, \ldots, q_K\}$ that can be defined as:

$$q_{k+1} = Tq_k. \tag{3.1}$$

Approximation Even with the strong assumption of the knowledge of the Bellman Operator, most of the interesting Reinforcement Learning problems have state and action spaces too big to be represented exactly in a tabular way. Therefore we need to resort to approximate solution methods and choose an approximation architecture for our action-value functions.

Let $\tilde{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$ denote a function representing our chosen approximation architecture. For example, in the linear case we might have:

$$\tilde{Q}(s, a, \boldsymbol{w}) \doteq \boldsymbol{\phi}(s, a)^T \boldsymbol{w}.$$

Then $\mathcal{Q}$ is the set of all action-value functions representable by our approximation architecture:

$$\mathcal{Q} \doteq \left\{\tilde{q}_{\boldsymbol{w}} \in \mathcal{S} \times \mathcal{A} \to \mathbb{R} \mid \tilde{q}_{\boldsymbol{w}}(s, a) = \tilde{Q}(s, a, \boldsymbol{w})\right\} \subset \mathbb{R}^{\mathcal{S}\times\mathcal{A}}.$$

Unfortunately using an approximation architecture restricts the set of action-value functions that we can represent and therefore prevents us from applying the Q Iteration algorithm, defined by eq (3.1). This is because the Bellman operator acts on the broader set of all possible q functions $\mathbb{R}^{\mathcal{S}\times\mathcal{A}}$, and thus applying it to a function in $\mathcal{Q}$ can result in a function that is outside of it and therefore not representable by our architecture.

Projection We need a way to "bring back" (or project) the result of Tq_w into the set $\mathcal{Q}$ of representable action-value functions. Let us define $\tilde{\boldsymbol{q}}_{\boldsymbol{w}} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ as the vector with components $\tilde{q}_{\boldsymbol{w}}(s,a)$ for each state-action pair (s,a) and $\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}}$ as the vector resulting from the application of T to each element of $\tilde{\boldsymbol{q}}_{\boldsymbol{w}}$. Then we would like to find the vector $\tilde{\boldsymbol{q}}_{\boldsymbol{x}}$ that minimizes the distance, according to some chosen euclidean norm, with $\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}}$:

$$\left\|\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}} - \tilde{\boldsymbol{q}}_{\boldsymbol{x}}\right\|^2.$$

The operation of finding the vector $\boldsymbol{v}$ in a subspace $\mathcal{V} \subseteq \mathbb{R}^n$, that minimizes the norm with an other vector $\boldsymbol{u}$ in the *parent* space $\mathbb{R}^n$, is often called *projection* and denoted as:

$$\boldsymbol{v} = \Pi\boldsymbol{u} \doteq \operatorname*{argmin}_{\boldsymbol{v}\in\mathcal{V}} \left\|\boldsymbol{u} - \boldsymbol{v}\right\|^2.$$

Therefore $\tilde{\boldsymbol{q}}_{\boldsymbol{x}}$ can be written as:

$$\tilde{\boldsymbol{q}}_{\boldsymbol{x}} = \Pi\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}} = \operatorname*{argmin}_{\tilde{\boldsymbol{q}}_{\boldsymbol{x}}\in\mathcal{Q}} \left\|\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}} - \tilde{\boldsymbol{q}}_{\boldsymbol{x}}\right\|^2. \tag{3.2}$$

Equation (3.2) defines the vector form $\Pi\boldsymbol{T}$ of a mapping on $\mathcal{Q}$ between approximate action-value functions, called Projected Bellman Operator and denoted ΠT.

Note that the minimization in eq (3.2) is performed on the set of representable action-value functions $\mathcal{Q}$, but since action-value functions are exactly characterized by their weight vector $\boldsymbol{w} \in \mathbb{R}^d$, we can rewrite the above equation as a minimization on $\mathbb{R}^d$, which is more convenient for our purposes:

$$\begin{aligned} \Pi T : \mathbb{R}^d &\rightarrow \mathbb{R}^d \\ (\Pi T)(\boldsymbol{w}) &\doteq \operatorname*{argmin}_{\boldsymbol{x}\in\mathbb{R}^d} \left\|\boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}} - \tilde{\boldsymbol{q}}_{\boldsymbol{x}}\right\|^2. \end{aligned} \tag{3.3}$$

Then the usual definition of the Projected Bellman Operator as a mapping between action-value functions, can be done using (3.3), simply as:

$$\begin{aligned} \Pi T : \mathcal{Q} &\rightarrow \mathcal{Q} \\ (\Pi T\tilde{q}_{\boldsymbol{w}})(s,a) &\doteq \tilde{Q}(s,a,(\Pi T\boldsymbol{w})). \end{aligned} \tag{3.4}$$

Assuming the knowledge of the Bellman Operator T, we can compute $\Pi T\tilde{q}_{\boldsymbol{w}}$ using any supervised learning method. For example we can use a gradient method like Stochastic Gradient Descent by defining a cost function

$$C(\boldsymbol{w}) \doteq \left\| \boldsymbol{T}\tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{q}}_{\boldsymbol{w}} \right\|^2, \tag{3.5}$$

and minimizing it with respect to $\boldsymbol{w}$. Even though in most cases this will not lead us to find the action-value function $\tilde{q}_{\boldsymbol{x}}$ representing the global minimum of eq. (3.2), but only a local minimum, this is the best we can do.

Projected Q Iteration With those definitions, we can come back at the Q iteration algorithm, defined by (3.1) and just replace T by ΠT, to obtain the following sequence of action-value functions:

$$q_{\boldsymbol{w}_{k+1}} \doteq \Pi T q_{\boldsymbol{w}_k}. \tag{3.6}$$

We refer to this, not surprisingly, as *Projected Q Iteration* and we can alternatively define it as the corresponding sequence on weight vectors $\boldsymbol{w} \in \mathbb{R}^d$:

$$\boldsymbol{w}_{k+1} \doteq \Pi T \boldsymbol{w}_k. \tag{3.7}$$

Although this may seem like to solve our problem of finding the representable action-value function q_{w_*} that is closer to the optimal action-value function q_*, this is not often the case. The repeated application of T, and therefore the Q Iteration algorithm, is guaranteed to converge to the optimal action-value function q_* because the Optimal Bellman Operator T is a *contraction mapping* with q_* as its unique *fixed-point*. Unfortunately the same cannot be said for ΠT, therefore the sequence defined by eq 3.7 is not guaranteed to converge, except for some special cases.

Definition 3.1 (Contraction Mapping)**.** Let (X, d) be a metric space. Then a map $T : X \to X$ is called a *contraction mapping* on X if there exists $\lambda \in [0, 1)$ such that

$$d(Tx, Ty) \leq \lambda d(x, y)$$

for all $x, y \in X$.

Theorem 3.1 (Banach fixed-point theorem)**.** *Let (X, d) be a non-empty complete metric space with a contraction mapping $T : X \to X$. Then T admits a unique fixed-point x_* in (i.e. $Tx_* = x_*$). Furthermore, x_* can be found as follows: start with an arbitrary element $x_0 \in X$ and define a sequence $\{x_n\}$ by $x_n = Tx_{n-1}$, then $x_n \to x_*$. [Banach, 1922]*

However even though convergence is not guaranteed, there are many cases in which algorithms based on some form of Projected Q Iteration obtain good results in practice. As an example see [Riedmiller, 2005] that describes Fitted Q Iteration adapted to work with Artificial Neural Networks.

Simulation A further generalization is to give up the hypothesis of knowledge of the environment's transition probabilities and therefore the ability to compute the Optimal Bellman Operator. Without this knowledge we have to resort to simulation to approximate the value of T. We accomplish this task by defining the *Empirical Bellman Operator*.

Definition 3.2 (Empirical Bellman Operator)**.** Let $\hat{T}$ be a mapping taking as input an action-value function $\tilde{q} \in \mathcal{Q}$ and producing as output a function $\hat{T}\tilde{q} : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \to \mathbb{R}$. Then $\hat{T}$ is called *empirical Bellman operator* and is defined as:

$$\begin{aligned} \hat{T} : \mathcal{Q} &\to \mathbb{R}^{\mathcal{S}\times\mathcal{A}\times\mathcal{R}\times\mathcal{S}} \\ \left(\hat{T}\tilde{q}\right)(s,a,r,s') &\doteq r + \gamma \max_{a\in\mathcal{A}} \tilde{q}\left(s',a\right). \end{aligned}$$

To justify the name of the empirical Bellman operator, we note that its expected value, given any four-tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ collected starting in state $s \in \mathcal{S}$ and taking action $a \in \mathcal{A}$, is equal to the value of the optimal Bellman operator for that state-action pair:

$$\mathbb{E}\left[\left(\hat{T}\tilde{q}\right)(S_t, A_t, R_{t+1}, S_{t+1}) \mid S_t = s, A_t = a\right] = \left(T\tilde{q}\right)(s,a).$$

Let $\tilde{q} \in \mathcal{Q}$ be an action-value function and let $\mathcal{D}$ denote a dataset of four-tuples $(S_t, A_t, R_{t+1}, S_{t+1})$ collected by interacting with the environment, possibly by an other agent:

$$\mathcal{D} \doteq \left\{\left(s_i, a_i, r_i, s_i'\right) \mid 0 \le i < |\mathcal{D}|\right\}.$$

Then $\tilde{\boldsymbol{q}} = (\tilde{q}_i) \in \mathbb{R}^{|\mathcal{D}|}$ is the vector obtained by computing $\tilde{q}(s,a)$ for each tuple in $\mathcal{D}$:

$$\tilde{q}_i \doteq \tilde{q}\left(s_i, a_i\right).$$

Moreover we can define the matrix form of the empirical Bellman operator as the vector $\hat{\boldsymbol{T}}\tilde{\boldsymbol{q}} \in \mathbb{R}^{|\mathcal{D}|}$, with components:

$$\left(\hat{T}\tilde{q}\right)_i \doteq \left(\hat{T}\tilde{q}\right)\left(s_i, a_i, r_i, s_i'\right).$$

The Empirical Bellman Operator is used by many simulation based methods, such as TD(0) or Fitted Q Iteration. More specifically the sequence of action-value functions $\{\tilde{q}_0, \tilde{q}_1, \ldots, \tilde{q}_K\}$ computed by Fitted Q Iteration can be formulated, using the notation we developed so far, as follows:

$$\tilde{\boldsymbol{q}}_{k+1} = \Pi\hat{\boldsymbol{T}}\tilde{\boldsymbol{q}}_k, \tag{3.8}$$

where $\Pi\hat{\boldsymbol{T}}$ can be defined in the same way that we did for T, both as a mapping on $\mathcal{Q}$ and as a mapping on weight vectors $\boldsymbol{w}$, by just replacing T with $\hat{T}$ respectively in equations (3.4) and (3.3).

3.2 Generalized Gradient Q Iteration

In the previous section we summarized the differences between the most important methods belonging to the class of Q Iteration, by showing how you can obtain more powerful algorithms that generalize simpler ones by incrementally dropping most of their assumptions. In this section we break down our algorithm in three procedures and analyze them separately to make more clear and simple our exposition.

Q Iteration assumes a perfect knowledge of the environment (i.e. a distribution model) and to be able to represent any action-value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Projected Q Iteration drops the latter assumption and uses an approximation architecture $\tilde{Q}(s, a, \boldsymbol{w})$ to represent action-value functions at the cost of not being able to ensure convergence in most common cases. Finally simulation based methods like Fitted Q Iteration drop also the former assumption, by not requiring the knowledge of the environment's transition probabilities and instead using the Empirical Bellman Operator (def. 3.2) to replace the Optimal T.

The core idea of our approach moves a step further in this quest for generalization. Note how all the methods considered so far learn a sequence of action-value function by applying some sort of Bellman Operator to the previous function of the sequence. Each step requires learning a new function and therefore performing an expensive minimization problem. What we propose is to learn instead the behavior of the Bellman Operator using a parameterized approximation architecture $\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})$ with weight vector $\boldsymbol{\omega} \in \mathbb{R}^m$ and then exploiting it to compute the next action-value function of the sequence.

Therefore our method works by computing at the same time two sequences. One sequence of action-value functions $\{\tilde{q}_0, \tilde{q}_1, \ldots, \tilde{q}_k\}$ and one sequence of approximate Bellman Operators $\left\{\tilde{T}_0, \tilde{T}_1, \ldots, \tilde{T}_k\right\}$. To be more precise, the sequences computed by the algorithm are of weight vectors $\boldsymbol{w} \in \mathbb{R}^d$ and $\boldsymbol{\omega} \in \mathbb{R}^m$, characterizing respectively the action-value functions and the Bellman operators.

Definition 3.3 (Approximate Bellman Operator)**.** Let $\tilde{T} : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d$ denote a function representing our chosen approximation architecture for the Bellman operator. Then $\mathcal{T}$ is the set of all approximate Bellman operators representable by our architecture:

$$\mathcal{T} \doteq \left\{\tilde{T}_{\boldsymbol{\omega}} \in \mathbb{R}^d \to \mathbb{R}^d \mid \tilde{T}_{\boldsymbol{\omega}}(\boldsymbol{w}) = \tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})\right\}.$$

Therefore each element $\tilde{T}_{\boldsymbol{\omega}}$ of $\mathcal{T}$ represents an *approximate Bellman operator*, characterized by the weight vector $\boldsymbol{\omega} \in \mathbb{R}^m$. Each $\tilde{T}_{\boldsymbol{\omega}} \in \mathcal{T}$ is a function that takes as input a weight vector $\boldsymbol{w}_k$ and computes the next element $\boldsymbol{w}_{k+1}$ of the sequence.

We defined approximate Bellman operators to be mappings on weight vectors $\boldsymbol{w}$, but the definition can be easily extended to action-value functions, in the same way we did for the Projected Bellman Operator in eq (3.4):

$$\begin{aligned} \tilde{T} : \mathcal{Q} &\rightarrow \mathcal{Q} \\ \left(\tilde{T}\tilde{q}_{\boldsymbol{w}}\right)(s,a) &\doteq \tilde{Q}\left(s, a, \left(\tilde{T}\boldsymbol{w}\right)\right). \end{aligned} \tag{3.9}$$

3.2.1 Core procedure

The core procedure of our method is shown in Algorithm 3.1 and works as follows: at each iteration k, the weight vector $\boldsymbol{\omega}_k \in \mathbb{R}^m$ representing the approximate Bellman operator $\tilde{T}_{\boldsymbol{\omega}_k}$ is learned and it is then used to compute the next weight vector $\boldsymbol{w}_k \in \mathbb{R}^d$ representing the approximate action-value function $\tilde{q}_{\boldsymbol{w}_k}$, by invoking the `update-w` procedure.

Algorithm 3.1 Generalized Gradient Q Iteration

`input:` $\mathcal{D}$ dataset of four-tuples
`param:` N_e number of training epochs to perform each iteration
N_p stop training after N_p epochs that the cost does not decrease

// Initialize $\boldsymbol{w}_0$ *such that* $\forall s, a : \tilde{Q}(s, a, \boldsymbol{w}_0) = 0$
// Initialize $\boldsymbol{\omega}_0$ *in any way*
$k \leftarrow 0$
`while not stopping-condition():`
  $e \leftarrow 0$
  $p \leftarrow 0$
  $\boldsymbol{\omega}_k \leftarrow 0$
  $k \leftarrow k + 1$
  `while` $e < N_e$ `and` $p < N_p$`:`
    // find the weights $\boldsymbol{\omega}$ *that minimize the cost function* C
    $\boldsymbol{\omega}_e \leftarrow$ `adam-train-step(C)`
    `if` $C(\boldsymbol{\omega}_e) \leq C(\boldsymbol{\omega}_k)$`:`
      $p \leftarrow 0$
      $\boldsymbol{\omega}_k \leftarrow \boldsymbol{\omega}_e$
    `else:`
      $p \leftarrow p + 1$
    $e \leftarrow e + 1$
  $\boldsymbol{w}_k \leftarrow$`update-w(`$\boldsymbol{\omega}_{k-1}$`,` $\boldsymbol{w}_{k-1}$`)`

In order to learn a good approximation of the Bellman operator we use Adam

[Kingma and Ba, 2014], a gradient-based optimization algorithm, that works very well in our case and has the advantage of requiring very little hyper-parameter tuning. This is represented in algorithm 3.1 by the `adam-train-step` procedure which performs a one-step minimization of the cost function C over the dataset $\mathcal{D}$. The details of the cost functions are given in the next subsection.

Moreover the procedure shown in Algorithm 3.1 can be tuned by specifying two hyper-parameters: N_e and N_p, which define how the weight vector $\boldsymbol{\omega}_k$ is learned. At each iteration k, many Adam optimization steps (or epochs) are performed, by invoking the `adam-train-step` procedure. The training is continued until the number of epochs is greater or equal to N_e or until N_p epochs are performed without obtaining a decrease in the cost function, and therefore without being able to improve the current weight vector $\boldsymbol{\omega}_e$.

The hyper-parameter N_e, representing the maximum number of training epochs to perform, can be set to any positive integer, or to ∞ to denote that the training should be halted only when the cost C does not decrease for N_p consecutive epochs. Note that while also the patience hyper-parameter N_p can be set to ∞, to denote the intention to always interrupt the training after N_e epochs, care should be taken to ensure that at least one of the hyper-parameter is finite, otherwise the procedure does not terminate.

3.2.2 Cost function

Like other gradient-based methods, Adam requires a cost function C to minimize. In order to achieve our objective we propose to minimize the sum of the Bellman errors at N_s steps, by considering N_s-times applications of the approximated Bellman operator.

Definition 3.4 (Cost Function)**.** Let $C : \mathbb{R}^m \to \mathbb{R}_{\geq 0}$ be the cost function of weight vectors $\boldsymbol{\omega} \in \mathbb{R}^m$, representing the prediction objective that we aim to minimize:

$$\begin{aligned} C : \mathbb{R}^m &\to \mathbb{R}_{\geq 0} \\ C(\omega) &\doteq \sum_{n=0}^{N} \left\| \hat{\boldsymbol{T}} \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n+1)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} \right\|^2, \end{aligned}$$

where $\tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n)}$ denotes the matrix form of the n-step approximated Bellman operator.

Let $\tilde{T}_{\boldsymbol{\omega}}^{(n)} : \mathbb{R}^d \to \mathbb{R}^d$ be a mapping between weight vectors $\boldsymbol{w}$, called n-step approximated Bellman operator, and defined as:

$$\begin{aligned}\tilde{T}_{\boldsymbol{\omega}}^{(0)}(\boldsymbol{w}) &\doteq \boldsymbol{w} \\ \tilde{T}_{\boldsymbol{\omega}}^{(n)}(\boldsymbol{w}) &\doteq \tilde{T}_{\boldsymbol{\omega}}^{(n-1)}\left(\tilde{T}_{\boldsymbol{\omega}}\boldsymbol{w}\right).\end{aligned}$$

Note that this is simply the n-times application of the approximated Bellman operator $\tilde{T}_{\boldsymbol{\omega}}$ that we defined above. Indeed in the case in which $n = 1$, we have

$$\tilde{T}_{\boldsymbol{\omega}}^{(1)}\boldsymbol{w} = \tilde{T}_{\boldsymbol{\omega}}\boldsymbol{w}.$$

The case $n = 0$ corresponds to 0 applications and therefore returns the weight vector $\boldsymbol{w}$ unchanged.

The definition of the n-step approximated Bellman operator acting on action-value functions $\tilde{T}_{\boldsymbol{\omega}}^{(n)} : \mathcal{Q} \to \mathcal{Q}$ is identical to the one we provided for $\tilde{T}_{\boldsymbol{\omega}}$ in equation (3.9).

Therefore we can rewrite the n-step cost function in (def. 3.4) more explicitly as:

$$\begin{aligned}C(\boldsymbol{\omega}) &\doteq \sum_{n=1}^{N_s}\left\|\hat{\boldsymbol{T}}\tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n-1)}\tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n)}\tilde{\boldsymbol{q}}_{\boldsymbol{w}_k}\right\|^2 \\ &= \sum_{n=1}^{N_s}\sum_{i}^{|\mathcal{D}|}\left(\left(\hat{T}\tilde{T}_{\boldsymbol{\omega}}^{(n-1)}\tilde{q}_{\boldsymbol{w}_k}\right)(s_i, a_i, r_i, s_i') - \left(\tilde{T}_{\boldsymbol{\omega}}^{(n)}\tilde{q}_{\boldsymbol{w}_k}\right)(s_i, a_i)\right)^2 \\ &= \sum_{n=1}^{N_s}\sum_{i}^{|\mathcal{D}|}\left(r_i + \gamma\max_{a\in\mathcal{A}}\tilde{Q}\left(s_i', a, \tilde{T}_{\boldsymbol{\omega}}^{(n-1)}\boldsymbol{w}_k\right) - \tilde{Q}\left(s_i, a_i, \left(\tilde{T}_{\boldsymbol{\omega}}^{(n)}\boldsymbol{w}_k\right)\right)\right)^2 \quad (3.10)\end{aligned}$$

The cost function is implemented as a procedure that can be supplied as argument to Adam, and its code is shown in algorithm 3.2.

3.2.3 Weight Update

The only part of our method that still needs to be described is the weight update procedure. The core procedure (Algorithm 3.1) calls the weight update procedure, `update-w`, on each iteration k, after the current Bellman Operator $\hat{T}_{\boldsymbol{\omega}_k}$ has been learned, in order to compute the new action-value function's weights.

The `update-w` procedure, shown in Algorithm 3.3, is simply the result of applying N times the current approximate Bellman Operator, defined by the weight vector $\boldsymbol{\omega}$, to the current estimate of the action-value function defined by the weight vector $\boldsymbol{w}$.

The procedure has a single hyper-parameter N_u defining the number of updates to perform.

Algorithm 3.2 Cost Function

`input:` $\boldsymbol{\omega}$ vector of weights characterizing a Bellman Operator $\tilde{T}_{\boldsymbol{\omega}}$
`param:` N_s number of steps of $\tilde{T}_{\boldsymbol{\omega}}$ to learn

$\boldsymbol{w} \leftarrow \boldsymbol{w}_k$
`cost` $\leftarrow 0$
`for` $i \leftarrow 0 \dots N$`:`
  // compute the cost for each element of $\mathcal{D}$
  `for` $(s, a, r, s') \in \mathcal{D}$`:`
    // compute the target t *as* $\hat{T}q_{\boldsymbol{w}}$
    $t \leftarrow r + \gamma \texttt{mm}_a \tilde{Q}(s', a, \boldsymbol{w})$
    // compute the prediction p *as* $\tilde{T}q_{\boldsymbol{w}}$
    $\boldsymbol{w} \leftarrow \tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})$
    $p \leftarrow \tilde{Q}(s, a, \boldsymbol{w})$
    `cost` $\leftarrow$`cost` $+ (t - p)^2$
`return cost`

Algorithm 3.3 Weight Update

`input:` $\boldsymbol{\omega}$ vector of weights characterizing a Bellman Operator $\tilde{T}_{\boldsymbol{\omega}}$
  $\boldsymbol{w}$ vector of weights to be updated, characterizing an action-value function $\tilde{q}_{\boldsymbol{w}}$
`param:` N_u number of times to apply $\tilde{T}_{\boldsymbol{\omega}}$

`for` $i \leftarrow 0 \dots N_u$`:`
  $\boldsymbol{w} \leftarrow \tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})$

3.3 Approximation Architectures for the Bellman Operator

In the previous section we detailed our algorithm and we said that it uses a parameterized architecture $\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})$ to approximate the Bellman optimality operator, but we did not give any example. In this section we are going to examine two possible approximate architectures for the Bellman Operator.

Note that the architecture has to be parametric, therefore we cannot consider all the non-parametric architectures, like for example Extremely Randomized Trees [Geurts et al., 2006], that obtain very good results when used to approximate action-value function in the Fitted Q Iteration algorithm.

Artificial Neural Networks Artificial Neural Networks are one of the most widely used non-linear architectures and have obtained extra-ordinary results also when used in the field of Reinforcement Learning. For example, see the results ob-

tained by the DeepMind team with the DQN algorithm [Mnih et al., 2015]. Artificial Neural Networks have the advantage of being able to approximate very well almost any function [Csáji, 2001], but are usually a lot slower to train compared to simpler, maybe linear, architectures.

In our case Artificial Neural Network have another disadvantage: they cannot model explicitly a contraction mapping and therefore we are not guaranteed that they have any fixed point.

Fixed Point Architecture A more interesting approach is to try to build an approximate architecture that explicitly models a contraction mapping. An example of such an architecture that we considered in our experiments is the following:

$$\tilde{T}(\boldsymbol{w}, \boldsymbol{A}, \boldsymbol{b}) = \boldsymbol{w} + \alpha \left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{w}\right), \tag{3.11}$$

where is $\alpha \in \mathbb{R}$ scalar, such that $0 < \alpha \leq 1$, $\boldsymbol{A}$ is a real $d \times d$ matrix and $\boldsymbol{b} \in \mathbb{R}^d$ is a column vector. In this case the parameters of our architecture are denoted with the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ instead of the vector $\boldsymbol{\omega}$.

The fixed point of the system is thus obtained as:

$$\begin{aligned}
\boldsymbol{w}_* &= \tilde{T}(\boldsymbol{w}_*, \boldsymbol{\omega}) && \Longrightarrow \\
\boldsymbol{w}_* &= \boldsymbol{w}_* + \alpha \left(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{w}_*\right) && \Longrightarrow \\
\boldsymbol{A}\boldsymbol{w}_* &= \boldsymbol{b} && \Longrightarrow \\
\boldsymbol{w}_* &= \boldsymbol{A}^{-1}\boldsymbol{b}.
\end{aligned}$$

Therefore the matrix $\boldsymbol{A}$ needs to be invertible.

Note that the system can be also written as:

$$\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega}) = \left(\boldsymbol{I} - \alpha \boldsymbol{A}\right) \boldsymbol{w} + \alpha \boldsymbol{b},$$

where $\boldsymbol{I}$ is the identity matrix. The stability of such a system is guaranteed when $(\boldsymbol{I} - \alpha \boldsymbol{A})$ is a matrix with eigenvalues of modulus smaller than one. One way to ensure this condition is to require the matrix $\boldsymbol{A}$ to be positive definite. This implies that its eigenvalues are positive real and therefore it is possible to find a value for α small enough to guarantee that the eigenvalues of $(\boldsymbol{I} - \alpha \boldsymbol{A})$ have modulus less than one. Therefore stability can be proved.

The big advantages of this approximate architecture are its simplicity that results in fast training times and above all the fact that we can find the fixed point of $\tilde{T}$ at any time by computing $\boldsymbol{A}^{-1}\boldsymbol{b}$. The obvious disadvantage is the fact that this architecture can represent only a small subset of the possible Bellman operators and therefore may not be a good fit in all cases.

Moreover note that the approximation architecture defined by (3.11), represents exactly the same dynamic system that we obtain while proving the convergence of most of the semi gradient methods like Projected Value Iteration or Semi Gradient TD(0), under linear function approximation.

3.4 Hyper-Parameters

In the previous sections we examined the details of our algorithm, giving details about both the structure of code and the approximation architectures that can be used with it. In this section we will review the main hyper-parameters that can be tuned and explain how changing their values, in combination with different approximation architectures leads to different algorithms. This is the reason why we defined our method as a framework rather than a single specific algorithm.

Let us start by giving an overview of the main hyper-parameters:

$\boldsymbol{N_s}$ Number of steps of the approximate Bellman operator $\tilde{T}_{\boldsymbol{\omega}}$ to use in the cost function. Setting this parameter higher than 1 causes the algorithm to try to learn more steps of the Bellman operator. Therefore its approximation will be less local, at the expense of a higher computational expense.

$\boldsymbol{N_e}$ Number of training epochs to perform while training the approximate Bellman operator. This number together with N_p defines the level of accuracy to which we want to learn the weight vector $\boldsymbol{\omega}$ on each iteration of the algorithm. If this parameter is set to ∞, the training will continue until there are no more improvements for a certain period (called patience). The hyper-parameter N_p representing the patience, is not considered in this section, because its goal is similar to that of N_e and can be fined tuned without impacting the following study.

$\boldsymbol{N_u}$ Number of updates to perform to the weight vector $\boldsymbol{w}$. On each iteration after we computed the current estimate of the Bellman operator $\tilde{T}_{\boldsymbol{\omega}}$, we use it to find the next action-value function, by applying it N_u times to the current weight vector $\boldsymbol{w}$.

Gradient Fitted Q Iteration Let us first show how our algorithm can be reduced to a gradient-based version of Fitted Q Iteration.

Definition 3.5 (Constant architecture)**.** Let our approximation architecture for the Bellman operator be the function $\tilde{T} : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d$, defined as:

$$\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega}) \doteq \boldsymbol{\omega},$$

where we must have $|\boldsymbol{\omega}| = |\boldsymbol{w}|$ or equivalently $m = d$. This function does not depend on the weight vector $\boldsymbol{w}$ and therefore is called constant.

Let us consider the case in which have $N_s = 1$. Then the cost function (def. 3.4) of our algorithm reduces to:

$$\begin{aligned} C(\boldsymbol{\omega}) &\doteq \sum_{n=1}^{N_s} \left\| \hat{\boldsymbol{T}} \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n-1)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(n)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} \right\|^2 \\ &= \left\| \hat{\boldsymbol{T}} \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(0)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}}^{(1)} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} \right\|^2 \\ &= \left\| \hat{\boldsymbol{T}} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{T}}_{\boldsymbol{\omega}} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} \right\|^2 \\ &= \left\| \hat{\boldsymbol{T}} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k} - \tilde{\boldsymbol{q}}_{\boldsymbol{\omega}} \right\|^2 . \end{aligned} \tag{3.12}$$

If we set the hyper-parameter to N_e to ∞, we obtain an algorithm that at each iteration k, minimizes the cost (3.12) until it converges to a local minimum. This is equivalent to performing the full projection of $\hat{\boldsymbol{T}} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k}$ into the space of action-value functions $\mathcal{Q}$, in order to find the next function of the sequence:

$$\tilde{\boldsymbol{q}}_{\boldsymbol{w}_{k+1}} \doteq \Pi \hat{\boldsymbol{T}} \tilde{\boldsymbol{q}}_{\boldsymbol{w}_k}.$$

This is exactly what the Fitted Q algorithm does when using a gradient-based methods to solve the supervised problems at each iteration.

Greedy Gradient Fitted Q Iteration By setting the parameter N_e to a finite value, and in the more extreme case to 1, while maintaining $N_s = 1$ and the constant approximation architecture explained in the previous paragraph, we obtain a variant of Fitted Q Iteration, that moves to the next action-value function before completing the projection operation. The idea is that it could turn out not to be necessary to learn perfectly every action-value function of the sequence as long as we keep moving in the correct direction in the space of action-value functions $\mathcal{Q}$.

Other algorithms with non-constant architecture The most interesting case are the ones when we do not use the constant architecture that we detailed in the previous paragraph. This is because in that way we are effectively not training the Bellman Operator, but approximating directly the action-value functions, ignoring thus the main feature of our algorithm.

Like we have shown for the previous algorithms, by varying the hyper-parameter N_e from 1 to infinite we can control the detail to which the approximate Bellman Operator is trained at the current iteration, before moving to the next action-value function. The lower the parameter the more greedier the algorithm becomes.

By using a non constant architecture such as the ones described in Section 3.3, we also gain the ability to fine-tune the hyper-parameters N_s and N_u, which otherwise lose meaning since we have no Bellman Operator to consider.

Chapter 4

Experiments

In this chapter we study the performance of our framework, under various configurations of the hyper-parameters, in two problems: the Linear-quadratic-Gaussian control problem and the Car on the Hill problem and compare its results to the solutions obtained by Fitted Q Iteration.

4.1 Linear–quadratic–Gaussian Problem

As a first problem we chose a variation of the Linear-quadratic-Gaussian control problem [Athans, 1971]. The unidimensionality of the problem and the fact that it admits a closed form solution, considerably simplifies the analysis, and makes it a good candidate for a first test of the performances of our framework.

Environment In this task the agent moves on a one dimensional straight line, starting from a random position, with the goal of arriving in the center (i.e. position 0). The agent receives a negative reward each time step proportional to the square of the distance from the center and to the square of the length of the step it takes as the current action.

The state space $\mathcal{S}$ is unbounded and is equal to the real line $\mathbb{R}$, therefore it is imperative to use approximate action-value functions:

$$\mathcal{S} \doteq \mathbb{R}.$$

The action space $\mathcal{A}$, in our case, is a discrete set of 21 elements, ranging from -8 to 8 and defined as:

$$\mathcal{A} \doteq \{0.8i \mid i \in \mathbb{Z}, -8 \leq i \leq 8\}.$$

The next state S_{t+1} at each time step is computed according to the following equation:

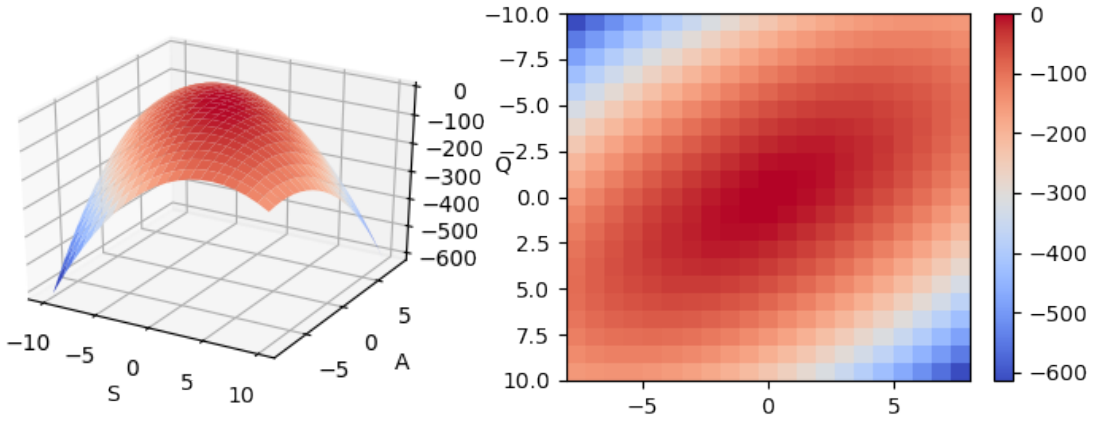

Figure 4.1: LQG Optimal action-value function

$$S_{t+1} \doteq aS_t + bA_t + \sigma,$$

where the parameters a and b have been set to 1 and the term σ represents a discrete-time Gaussian white noise process.

The reward given to the agent is always negative and quadratic with respect to the current state and action, except in position 0, where the reward is null. The reward is specified by the following equation:

$$R_{t+1} \doteq -qS_t^2 - rA_t^2,$$

where the parameters q and r have been both set to 0.9.

The optimal policy can be computed in closed form and has the form:

$$\pi(s) \doteq -l \cdot s,$$

where

$$l \quad = \quad abx \left(b^2 x + r\right)^{-1},$$

and x is determined by solving the Riccati equation:

$$x = a^2 \left(x - \left(xb^2 + r\right)\right) + q.$$

Dataset collection The dataset has been collected by uniformly sampling the state space $\mathcal{S}$ in the interval $[-10, 10]$ and performing a step by selecting every possible action in $\mathcal{A}$, starting from each state.

The states in the dataset $\mathcal{D}$ are:

$$\overline{\mathcal{S}} \doteq \left\{ \frac{i}{15} \mid i \in \mathbb{Z}, -150 \leq i \leq 150 \right\} \subset \mathcal{S}.$$

The size of the set of states $\overline{\mathcal{S}}$ is 301, and therefore the size of the dataset $\mathcal{D}$ is given by:

$$|\mathcal{D}| = |\overline{\mathcal{S}}| \cdot |\mathcal{A}| = 301 \cdot 21 = 6321.$$

Approximation Architectures Both the approximation architecture for the action-value functions $\tilde{Q}(s, a, \boldsymbol{w})$ and the one for the Bellman operator $\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega})$ have been chosen to be linear, except for Gradient Fitted Q Iteration and its greedy variant that do not train the Bellman operator and therefore have a constant $\tilde{T}(\boldsymbol{w}, \boldsymbol{\omega}) = \boldsymbol{\omega}$, like explained in Section 3.4 on page 51.

The action-value function architecture has the form:

$$\tilde{Q}(s, a, \boldsymbol{w}) \doteq \boldsymbol{\phi}(s, a)^T \boldsymbol{w} = \sum_i \phi_i(s, a) w_i,$$

where $\boldsymbol{\phi}(s, a) = (\phi_1(s, a), \phi_2(s, a))^T \in \mathbb{R}^2$ represents the feature vector for the state-action pair (s, a) and is defined as:

$$\begin{aligned} \phi_1(s, a) &\doteq s \cdot a \\ \phi_2(s, a) &\doteq s^2 + a^2, \end{aligned}$$

while the approximate Bellman operator architecture is the Fixed Point architecture defined in 3.3.

Fitted Q Iteration with Least Squares The first step of the evaluation of our framework is to run the Fitted Q Iteration algorithm to be able to compare its solution to the ones obtained by our method. Since we are using a linear approximation architecture for the action-value function we decided to use SciKit Learn's Least Square Regressor [Pedregosa et al., 2011] to solve the supervised problem performed at each step of Fitted Q Iteration, due to its speed in this case compared to gradient-based algorithms.

The algorithm converges approximately in 6 iterations, to the solution:

$$\boldsymbol{w} \approx (-2.9076, -2.3517)^T,$$

which results in an action-value function that has a mean squared error of ≈ 6.37 with respect to the optimal action-value function, evaluated over all the state-action

it.	w_1	w_2	mse*
1	1.41794758e-17	-8.99999969e-01	16031.722623
2	-1.78116456	-1.79148525	2328.277009
3	-2.66976558	-2.23696794	86.890258
4	-2.87080458	-2.33493751	5.062161
5	-2.90443403	-2.35043844	6.094046
6	-2.90766497	-2.35178957	6.386884
7	-2.90761501	-2.35173596	6.378294

Table 4.1: LQG FQI Least Squares
Weights computed by 7 steps of FQI with Least Squares in the LQG problem.

pair in the dataset $\mathcal{D}$. Table 4.1 shows the values of the weights and the mean squared error with the optimal action-value function at each iteration.

Gradient Fitted Q Iteration Gradient Fitted Q Iteration produces results almost identical to the Least Squares version. The weight vector $\boldsymbol{w}$ has been initialized to 0, to ensure that the first target that is learned is equal to the reward. Note that the algorithm learn the weight vector $\boldsymbol{\omega}$, even if in this case it simply represents the weights of the action-value function, and at the end of each iteration $\boldsymbol{\omega}$ is copied into $\boldsymbol{w}$.

The weight $\boldsymbol{\omega}$ have been initialized in an intentionally bad way in order to see if the algorithm can recover from bad initial weights:

$$\begin{aligned} \boldsymbol{\omega}_{bad} &\doteq (5, 10) \\ \boldsymbol{\omega}_0 &\doteq \boldsymbol{\omega}_{bad}. \end{aligned}$$

See Figure 4.2 on page 59 to see the performance of this algorithm compared to the corresponding one with the Fixed Point architecture.

Gradient Fitted Q Iteration with $N_e = 1$ The greedy version of the previous algorithm, that is the one obtained by training the action-value function for just one epoch each iteration ($N_e = 1$). The algorithm diverges to $+\infty$ in the case we initialize it with $\boldsymbol{w}_0 = 0$ (and $\boldsymbol{\omega}_0 = \boldsymbol{\omega}_{bad}$).

In case we initialize the weights (both $\boldsymbol{w}_0$ and $\boldsymbol{\omega}_0$), with the same weights found after one step by Gradient Fitted Q Iteration, which are the weights learned when trying to approximate the reward:

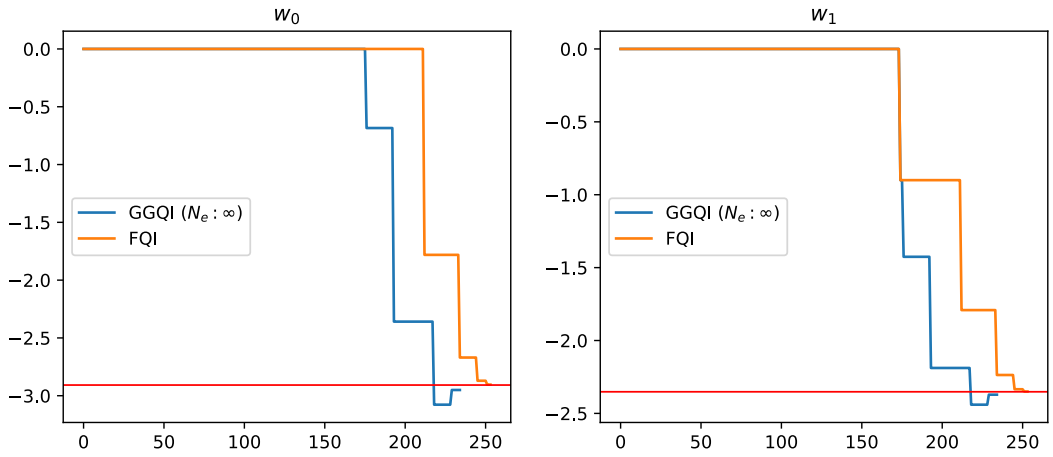

Figure 4.2: FQI vs GGQI on LQG
The two plots show respectively the changes in the two components of the weight vector $\boldsymbol{w} = (w_0, w_1)$. Each plot shows the same weight component for two algorithms: Gradient FQI and GGQI with $N_s = 1$ and $N_e = \infty$. In this experiment our method is able to learn the same solution of FQI, slightly faster. On the horizontal axis are the epochs of learning divided by 100.

$$\boldsymbol{w}_r \doteq (4.71210274e-15, -8.99998803e-01)\,,$$

the algorithm goes towards the fixed point but then oscillates around it. See Figure 4.3 on page 60 for a comparison of this method with the corresponding greedy method with Fixed Point Architecture and $N_s = 2$.

Fixed Point architecture with $N_e = \infty$ Another trial is to run our algorithm with the Fixed Point architecture described in Section 3.3 and setting $N_e = \infty$. This setting is able to converge to the fixed point in the same number of total epochs as Gradient Fitted Q Iteration.

In order to make a fair comparison with Gradient Q Iteration, the initial weights have been set in the following way:

$$\begin{aligned} \boldsymbol{w}_0 &\doteq 0 \\ \boldsymbol{A}_0 &\doteq \boldsymbol{I} \\ \boldsymbol{b}_0 &\doteq \boldsymbol{\omega}_{bad}. \end{aligned}$$

Setting $\boldsymbol{A}$ and $\boldsymbol{b}$, which represent the weights of the Fixed point architecture, in this way ensures that the initial fixed point is equal to $\boldsymbol{\omega}_{bad}$:

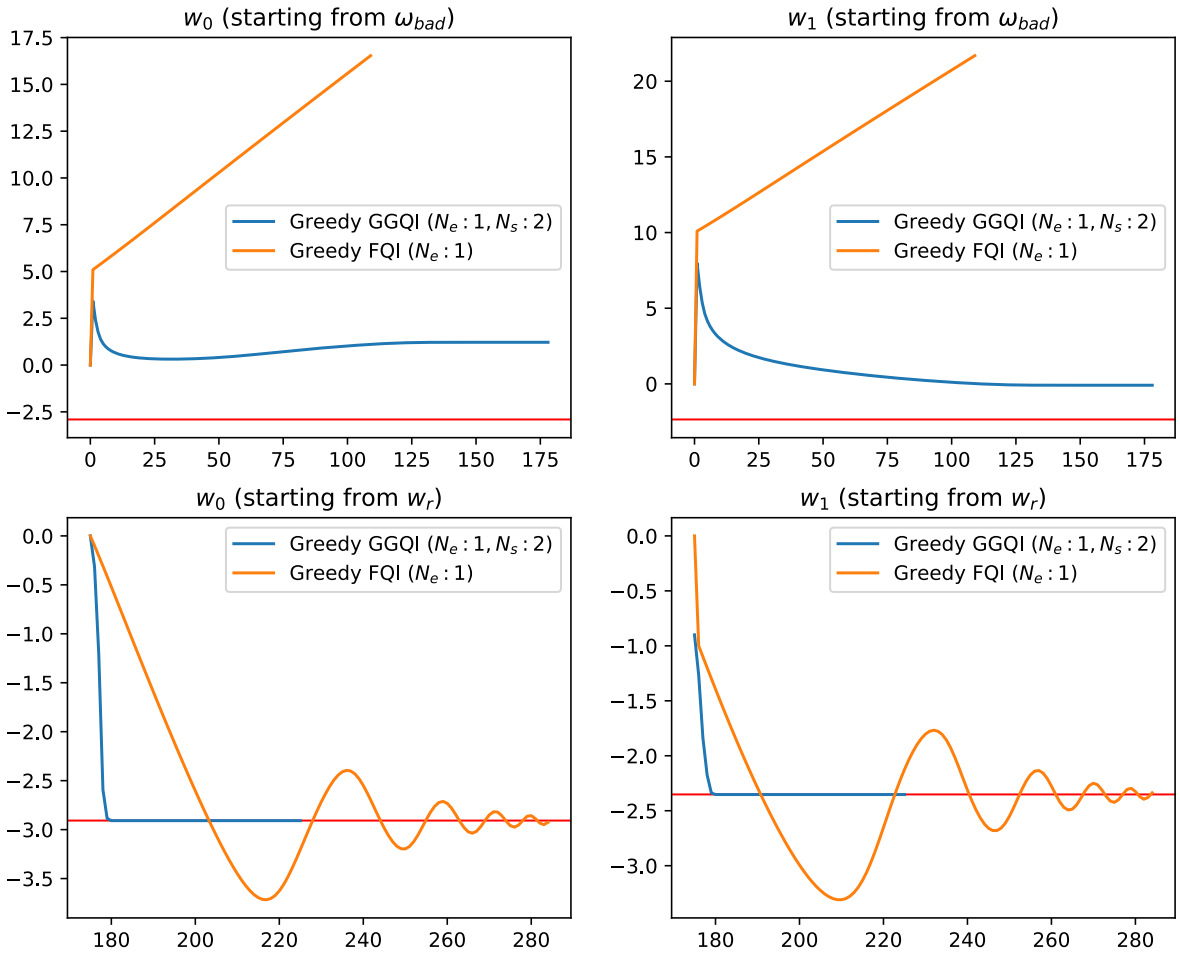

Figure 4.3: Greedy Algorithms on LQG

These plots, divided in two rows and two columns, show on each column the evolution of a component of the weight vector $\boldsymbol{w} = (w_0, w_1)$. The first rows compares the greedy version ($N_e = 1$) of Gradient FQI with the greedy ($N_e = 1$) GGQI with Fixed Point architecture (with $N_s = 2$), when their initial weights are set to ω_{bad}. In this case Greedy FQI diverges almost immediately, while the Fixed Point architecture does not diverge but does not go to the correct fixed point either.

In the second row, we initialized the algorithms from $\boldsymbol{w}_r$, and in this case we can see how Greedy FQI oscillates for very long, while the greedy Fixed Point converges almost immediately to the correct solution.

$$\boldsymbol{w}_0^* = \boldsymbol{A}_0^{-1}\boldsymbol{b}_0 = \boldsymbol{\omega}_{bad}.$$

Fixed Point architecture with $N_e = 1$ and $N_s = 2$ Running the greedy version of Fitted Q Iteration (i.e. with $N_e = 1$) goes very fast towards the fixed point, but oscillates around it. We can try to fix this by learning two steps of the approximate Bellman Operator, instead of moving directly in the action-value function space like Fitted Q Iteration.

We tried again starting the algorithm, like in the previous setting, that is with $\boldsymbol{w}_0 = 0$ and with $\boldsymbol{A}^{-1}\boldsymbol{b} = \boldsymbol{\omega}_{bad}$. In this case the algorithm does not diverge to $+\infty$, like Fitted Q Iteration with $N_e = 1$, but converges to point that is distant from the correct solution.

If we initialize the weights, with the ones found after one step of Gradient Fitted Q Iteration:

$$\begin{aligned} \boldsymbol{w}_0 &\doteq \boldsymbol{w}_r \\ \boldsymbol{A}_0^{-1}\boldsymbol{b}_0 &\doteq \boldsymbol{w}_r. \end{aligned}$$

Then the algorithm does not diverge and goes very quickly to the correct fixed point.

4.2 Car on the Hill Problem

We consider now the Car on the Hill problem, in which an agent needs to drive a car on top of a hill (See Figure 4.4 on page 62). The state space is continuous and consists of tuples (p, s) representing the position and speed of the agent:

$$\mathcal{S} = \{(p, s) \in \mathbb{R} \mid -1 < p < 1 \wedge -3 \leq s \leq 3\}.$$

The agent start at position -0.5, at the bottom of the valley and can select only two actions

$$\mathcal{A} \doteq \{-4, 4\},$$

that influence the acceleration of the car, making it respectively smaller and bigger. The goal of the agent is to bring the car on top of the hill as fast as possible while avoiding to reach positions $p < -1$ and speeds $|v| > 3$.

Note that due to the forces acting on the car, the agent cannot simply accelerate to win, but needs first go back to gain momentum.

For the exact equations governing the system, see [Ernst et al., 2005, Appendix C.2].

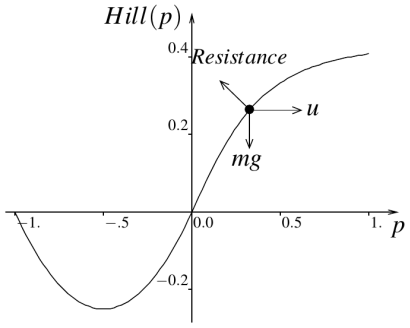

Figure 4.4: The Car on the Hill control problem
Representation of the shape of the Hill (taken from [Ernst et al., 2005])

In order to compare the performance of our algorithm graphically, we computed a baseline action-value function for this problem performing 10 steps of Fitted Q Iteration with Extremely Randomized Trees (see Figure 4.5 on page 63). We then measured the mean squared error of the other algorithms that we ran with respect to this action-value function.

Moreover for each algorithm we computed the score like suggested by [Ernst et al., 2005].

Dataset collection The dataset has been collected similarly to what has been done for the previous problem, that is by uniformly sampling the state space $\mathcal{S}$ and performing a step by selecting every possible action in $\mathcal{A}$, starting from each state.

The states in the dataset $\mathcal{D}$ are:

$$\overline{\mathcal{S}} \doteq \{(0.02 \cdot i, 0.2 \cdot j) \mid i, j \in \mathbb{Z}, -50 \leq i \leq 50 \wedge -15 \leq j \leq 15\} \subset \mathcal{S}.$$

The size of the set of states $\overline{\mathcal{S}}$ is $101 \cdot 31 = 3131$, and therefore the size of the dataset $\mathcal{D}$ is given by:

$$|\mathcal{D}| = |\overline{\mathcal{S}}| \cdot |\mathcal{A}| = 3131 \cdot 2 = 6262.$$

Approximation Architectures The approximation architecture used in this problem, for the action-value functions and the Bellman Operators are both linears, like in the previous case. The architecture used for the approximate Bellman Operator, when used, is the Fixed Point architecture (Section 3.3).

The features for the action-value functions, in this case are tiles. We used 8 tilings, each composed of 5×5 tiles, with displacement vector set to the first odd integers $(1, 3, 5, 7, \dots)$ as recommended by [Miller and Glanz, 1996].

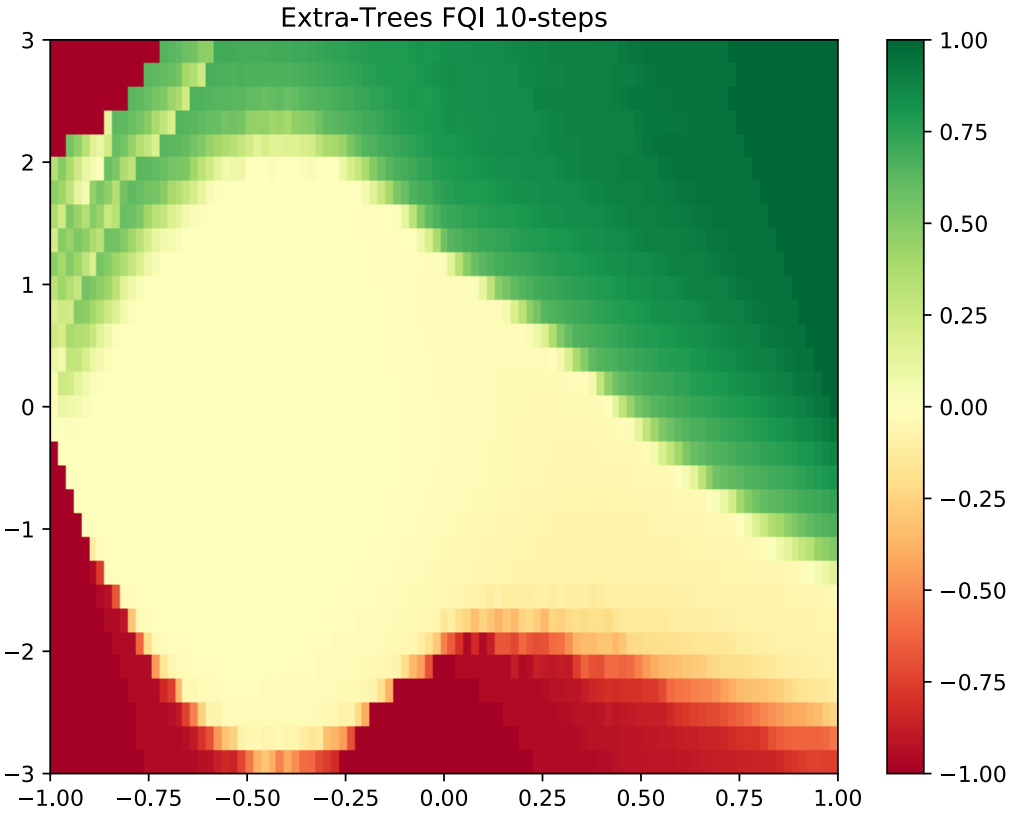

Figure 4.5: V function of Extra-Trees
This plot shows the value function computed by an Extra-Trees regressor after 10 iterations of Fitted Q Iteration. On the horizontal axis there are the positions p, ranging from -1 to 1. On the vertical axis there is the speed s, ranging from 3 to -3.

Gradient Fitted Q Iteration and Fixed Point Architecture with $N_e = \infty$ As in the same problem we executed Gradient FQI for 10 iterations. It obtained a score of 0.2274.

We then ran our algorithm with $N_e = \infty$ and 10 iterations and it obtained a slightly better score of 0.2339.

Figure 4.6 on page 65 compares the mean squared error of these two algorithms with respect to the baseline action-value function computed with Extra-Trees. Both algorithm tend to get closer and closer to that function, while our algorithm seems to get there slightly faster.

Greedy Variants $N_e = 100$ In this context we ran our greedy algorithms with a less aggressive setting of $N_e = 100$, as opposed to $N_e = 1$ that we used for the previous problems. This means that the algorithms learn for 100 epochs before moving to the next action-value function of the sequence.

We executed our algorithm with $N_s = 1$ and compared it with the performance of the greedy version of gradient FQI. They obtained both lower scores than the versions with $N_e = \infty$ and we had to initialize their weights with the ones obtained by learning the reward with one step of FQI in order to have better results. The scores are 0.1533 for FQI and 0.1574 for GGQI.

Figure 4.7 on page 66 shows the mean squared error computed while running those algorithms with respect to the baseline action-value function learned with FQI and Extra-Trees. They both converge almost bringing the error to 0, even though GGQI seems to be more stable towards the end.

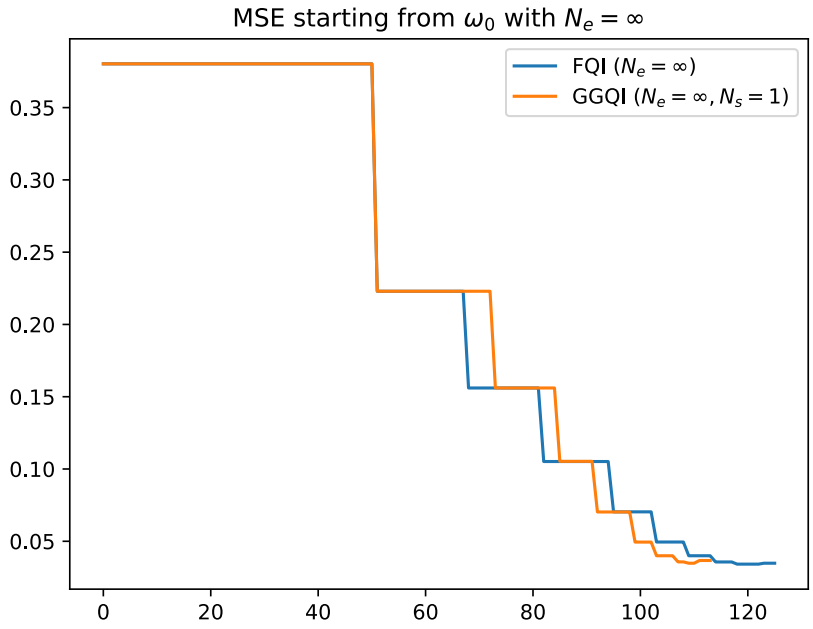

Figure 4.6: FQI vs GGQI on Car on the Hill

This plot compares the mean squared error of FQI and GGQI (with $N_e = \infty$ and $N_s = 1$) with respect to the action-value function computed with Extra-Trees and 10 steps of FQI. On the horizontal axis are the epochs of learning divided by 100.

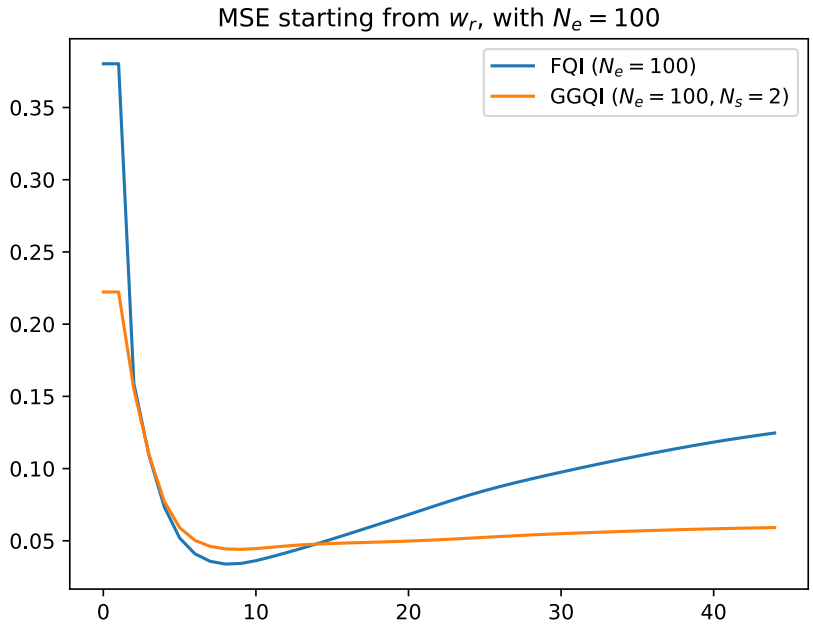

Figure 4.7: Greedy Algorithms on Car on the Hill
This plot compares the mean squared error of FQI ($N_e = 100$) and GGQI (with $N_e = 100$ and $N_s = 2$) with respect to the action-value function computed with Extra-Trees and 10 steps of FQI. On the horizontal axis are the epochs of learning divided by 100.

Chapter 5

Conclusions and Future Work

This thesis belongs to the research area of Reinforcement Learning and its goal was to propose a new framework: Generalized Gradient Q Iteration.

This framework consists of a generic method with a series of hyper-parameters that can be tuned in order to obtain different algorithms. Our method belongs to the class of Q Iteration, and indeed we showed how under some setting of the hyper-parameters it can be reduced to Fitted Q Iteration.

The first characteristics that sets apart our method from Fitted Q Iteration is the ability to fine-tune the level at which action-value functions are approximated on each step of the algorithm. This allows us to avoid performing the full projection and moving to the next action-value function before having fully learned the previous one. As long as the algorithm keeps moving in the correct direction, convergence is sped up by a significant factor.

The second and bigger characteristic of our method is that we have the ability to learn an approximation of the Bellman optimality operator. The core idea is that if we manage to approximate in a good way the Bellman operator we can then use this knowledge to move very quickly towards the optimal action-value functions. We proposed various approximation architectures for the Bellman Operator, among which one that we called Fixed Point architecture has an explicit representation of its fixed point. This has the obvious advantage of allowing us to move directly to such fixed point.

We tested our framework in various experiments and showed how in every case, under the appropriate settings, it manages to get performances that are comparable or slightly better than those of Fitted Q Iteration. This encourages us to think that with this work we moved in the right direction and that with more study we can further improve and refine our method.

One possible future development is to test our method in other more challenging tasks and to try and experiment with different approximation architectures for the Bellman Operator, in particular with Deep Neural Networks, considering their recent

spectacular successes. Another interesting future improvement would be to prove the convergence of our method, using the above mentioned Fixed Point architecture, in the linear case to the same solution found by semi-gradient methods such as semi-gradient TD(0).

Bibliography

[Athans, 1971] Athans, M. (1971). The role and use of the stochastic linear-quadratic-gaussian problem in control system design. *IEEE transactions on automatic control*, 16(6):529–552.

[Banach, 1922] Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3(1):133–181.

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, pages 679–684.

[Bellman, 2013] Bellman, R. (2013). *Dynamic programming*. Courier Corporation.

[Bertsekas, 2005] Bertsekas, D. P. (2005). *Dynamic programming and optimal control*, volume 1.

[Bertsekas, 2007] Bertsekas, D. P. (2007). *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 3rd edition.

[Csáji, 2001] Csáji, B. C. (2001). Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48.

[Ernst et al., 2005] Ernst, D., Geurts, P., and Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556.

[Geurts et al., 2006] Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, 63(1):3–42.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Miller and Glanz, 1996] Miller, W. and Glanz, F. H. (1996). The university of new hampshire implementation of the cerebellar model arithmetic computer-cmac. *Robotics Laboratory, University of New Hampshire, Durham, New Hampshire*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Riedmiller, 2005] Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer.

[Russell and Norvig, 2016] Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.