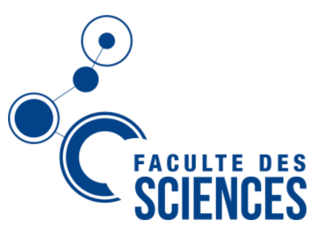


Faculté des sciences

Deep Reinforcement Learning: An introduction through video games

Author : Corentin ZONE

Supervisors : Robin VAN OIRBEEK, Andrea PENNISI

Reader : Donatien HAINAUT

Academic year 2021-2022

Master en science des données, orientation statistique

Abstract

An increasing number of technologies involving artificial intelligence (AI) are currently being developed all over the world. These technologies could lead to enormous breakthroughs in their sectors. These AI developments are largely made possible thanks to deep learning techniques that use neural networks in their algorithm.
In this master thesis, we discover the Deep Reinforcement Learning through three algorithms: *Q-Learning*, one of the more basic algorithms, *Deep Q-Network*, an evolution of the Q-Learning using neural networks, and the *Proximal Policy Optimization*, currently one of the most efficient RL algorithms.
To compare these algorithms, we use two games implemented in a Python package called Gym developed by OpenAI. The first game is basic and often used in the literature, *Mountain Car*. The second is the classical *Super Mario Bros*.
For all these algorithms, we have designed several models with various hyperparameter values and compared their performance by looking at the cumulative average reward produced by each game. For the Mario environment, the models were also compared by their ability to end the first stage.

Acknowledgement

Most importantly, I sincerely want to express my gratitude to my supervisor, Dr. Robin Van Oirbeek. His precious advice and feedbacks have helped me considerably in the realization of this work. His enthusiasm and goodness have made this master thesis very pleasant and rewarding.

I also want to acknowledge my friends who made these years of studies the most fun and enjoyable of my life.

Ultimately, I would like to heartily thank you, Julie, for the support you brought me during these uncommon years, specially during the lockdown. Your attentive listening and encouragement in everything I do are precious and inestimable. Thank you for bringing out the best in me.

Contents

Abstract i

Acknowledgement ii

Contents iii

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Overall Picture . . . 1
- 1.2 Research Questions . . . 2
- 1.3 Summary of Results . . . 2
- 1.4 Structure of the Thesis . . . 3

2 State of the Art 5
- 2.1 Reinforcement Learning . . . 5
  - 2.1.1 Introduction . . . 5
  - 2.1.2 Formal Definition . . . 6
  - 2.1.3 Solving Methods . . . 10
- 2.2 Neural Networks . . . 13
  - 2.2.1 Perceptron . . . 13
  - 2.2.2 Backpropagation . . . 14
  - 2.2.3 Multilayer Perceptron . . . 18
  - 2.2.4 Challenges and Solutions . . . 22
  - 2.2.5 Gradient Descent Variants . . . 25
  - 2.2.6 Convolutional Neural Network . . . 27
- 2.3 Deep Learning . . . 32
  - 2.3.1 Deep Q-networks . . . 33
  - 2.3.2 Proximal Policy Optimization Algorithms . . . 34

3 Methodology 39
- 3.1 Gym Environment . . . 39
  - 3.1.1 Mountain Car . . . 40
  - 3.1.2 Super Mario Bros . . . 42
- 3.2 Mountain Car Solution . . . 43
  - 3.2.1 Preprocessing . . . 44

3.2.2 Q-learning Implementation 44
3.2.3 DQN Implementation 45
3.3 Super Mario Bros. Solution 46
3.3.1 Preprocessing 46
3.3.2 Solving Mario 47

4 Results **48**
4.1 Mountain Car Environment 48
4.1.1 Q-learning 48
4.1.2 DQN 49
4.1.3 Performance Comparison 51
4.2 Super Mario Bros. Environment 52
4.2.1 Preprocessing 52
4.2.2 DQN 53
4.2.3 PPO 55
4.2.4 Comparison 57

5 Conclusion **58**
5.1 Suggestions 59

Bibliography **60**

A Methodology **67**

B Results **71**
B.1 Q-learning 71
B.2 Mario Results 75

List of Figures

2.1.1 Basic schema of the functioning of an RL algorithm (Bhatt, 2018). 6
2.2.1 Nonlinear model of a neuron k (Haykin, 2009) 13
2.2.2 Representation of (a) threshold function, and (b) logistic function (Haykin, 2009) 14
2.2.3 Design of a multilayer perceptron with two hidden layers (Haykin, 2009). . . 18
2.2.4 Representation of ReLU function (Liu, 2017). 23
2.2.5 Classical architecture of a CCN used to classify handwritten digits (Saha, 2018). 28
2.2.6 Computation of a CNN hidden layer. We slide the local receptive field by one neuron to the right between (a) and (b) (Nielsen, 2015) 28
2.2.7 Representation of the first hidden layer composed of three different features map (Nielsen, 2015). 29
2.2.8 Example of pooling operation of size 2×2 (Yingge et al., 2020). 30
2.2.9 Complete architecture of a CNN taking as input a 28×28 pixel image, using three 5×5 kernels to detect features and a max-pooling layer applied to 2×2 regions, across each of the 3 feature maps (Nielsen, 2015). 31
2.2.10 2-D convolution example without kernel-flipping (Goodfellow et al., 2016). . 32
2.3.1 Surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right) (Schulman et al., 2017). . 37

3.1.1 Frame of the Mountain Car environment. 41
3.1.2 Frame of Super Mario Bros. 42

4.1.1 Distribution of the rewards obtained over 100 consecutive episodes of the Mountain Car environment by the models trained with Q-learning algorithm. 49
4.1.2 Distribution of the rewards obtained over 100 consecutive episodes of the Mountain Car environment by the models trained with DQN algorithm. . . 50
4.1.3 Policy comparison for the best Q-learning model with the best DNQ model. 51
4.1.4 Smoothing policy with square length value of 5. 52
4.1.5 Distribution of the rewards obtained with the smoothed policies. The models correspond respectively to a square length of 1, 3, 5, 7, 9. 52
4.2.1 Image obtained by the preprocessing of the observations. 53
4.2.2 Distribution of the rewards obtained after 50 simulations for each model trained with DQN for the Mario environment. 54
4.2.3 Analyze of the rewards obtained over 50 Mario episodes. The boxplots correspond respectively to a simulation after 1, 2, 3 and 4 millions steps training. 56

4.2.4 Distribution of the rewards obtained after 50 simulations for each model trained by PPO for the Mario environment 56
4.2.5 Distribution of the rewards obtained over 5000 simulations for the best model trained with PPO for the Mario environment. 57

B.1.1 Model 1: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 71
B.1.2 Model 2: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 71
B.1.3 Model 3: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 72
B.1.4 Model 4: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 72
B.1.5 Model 5: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 72
B.1.6 Model 6: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 73
B.1.7 Model 7: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 73
B.1.8 Model 8: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm. 73
B.1.9 Policies obtained by smoothing the model 8. 74
B.2.1 Example of a stuck model. 75

List of Tables

3.1.1 Observation space for Mountain Car problem. 41
3.1.2 Action space for Mountain Car problem. 41
3.1.3 Possible list of actions. 43
3.2.1 Default architecture for the CNN implemented in Stable Baselines3. 46

4.1.1 Model's parameters used for Q-learning. 48
4.1.2 Mean and standard deviation obtained over 100 consecutive episodes in the Mountain Car environment by the models trained with Q-learning algorithm. 49
4.1.3 Parameters' values for each model trained by the DQN algorithm. 50
4.1.4 Mean and standard deviation obtained over 100 consecutive episodes in the Mountain Car environment by the models trained with DQN algorithm. . . 51
4.1.5 Mean and standard deviation obtained over 100 consecutive episodes of the Mountain Car environment by the smoothed policies. 52
4.2.1 Description of the preprocessed environments. 53
4.2.2 Model's parameters used to solve Mario with DQN. 53
4.2.3 Mean and standard deviation obtained over 50 consecutive episodes of the Mario environment by the models trained with DQN algorithm. 54
4.2.4 Model's parameters used to solve Mario with PPO. 55
4.2.5 Mean and standard deviation obtained over 50 consecutive episodes of the Mario environment by the models trained with PPO algorithm 56
4.2.6 Mean, standard deviation, maximun reward and number of flags reached obtained over 5000 consecutive episodes of the Mario environment by the best models trained with PPO algorithm. 57

A.0.1 Information presented in the dictionary returned by the step function for the Mario environment. 67
A.0.2 Type of possible environments in gym-super-mario-bros package of Kauten (2018b). 68
A.0.3 Possible arguments for the DQN function from Stable Baseline3 (Raffin et al., 2021). 69
A.0.4 Possible arguments for the PPO function from Stable Baseline3 (Raffin et al., 2021). 70

Chapter 1

Introduction

1.1 Overall Picture

With the constant growth of the digital world, a lot of data are easily accessible. Coupled with the increasing computing power, it leads to the emergence of Artificial Intelligence (AI) including deep learning techniques. Deep learning is part of the machine learning field, but unlike classical algorithms, an AI trained by deep learning can continue to learn by having access to more data.

Artificial neural networks are the core of deep learning. It is a technology inspired from biological neural networks present in the human brain. Like in the brain, neurons are interconnected to one another in the different layers of the network. The most simple neural network consists of one neuron that receives an information, processes it and outputs a result. These models can be scaled to millions of neurons present in different layers of the network. If a network has more than three layers (including the input and output), we call this a deep neural network, from which the term deep learning is derived (Kavlakoglu, 2020).

The applications of deep learning are quite extensive. An important DL application is the ability to recognize objects on images. This property can be used in many different fields like robotics by giving eyes to robots, in agriculture by differentiating weeds from crops when applying herbicide, or even in the medical field to class skin cancers.
Besides image recognition, deep learning can be used to ensure predictions in many different ways. For example, Jumper et al. (2021) have developed a model that predicts the 3D structure of different proteins by looking only at their amino acid sequence. This protein folding problem is at the center of research for more than 50 years.

Another aspect of deep learning is called the *Deep Reinforcement Learning* (RL), which is at the core of this master thesis. In RL, an agent learns by trial and error. After each action performed by the agent, we reward or penalize him by a certain amount.
Deep RL is still a modern field in constant expansion, but we can already notice some significant applications. One of the most famous is AlphaGo, which "is the first computer program to defeat a Go world champion" (DeepMind, 2022). RL is additionally used for self-driving vehicles, from cars to boats. In the military field, the 6th generation of fighters currently in development should be able to perform unmanned missions.

1.2 Research Questions

The core objective of this master thesis is to discover the Deep Reinforcement Learning methodology. The most intuitive way is to construct an AI for a video game.
There exist many different algorithms that can be used to develop an AI. The more basic is **Q-learning** that was developed by Watkins (1989). Later, with the development of neural networks and increase in computing power, more sophisticate algorithms using neural networks were constructed. In this work, we studied two methods, **Deep Q-learning** (DQN) developed by Mnih et al. (2013) and Mnih et al. (2015), and one of the most advanced and efficient methods, the **Proximal Policy Optimization** (PPO) constructed by Schulman et al. (2017).

We have applied both algorithms to two games in order to compare their performance. The first one is called **Mountain Car**. It is a very simple game with only two inputs. This game is used in many papers to explain RL. The second is the famous **Super Mario Bros** on NES. This game is much more complex than the first, the inputs being the frames of the game. Both these games are already implemented in a Python package called **Gym** developed by OpenAI (Brockman et al., 2016). The goal of Gym is to provide a common benchmark to evaluate different RL algorithms.

In order to compare the performances of the algorithms, we will answer the following questions:

- Are we able to solve the Mountain Car environment using the Q-learning, which is the more basic algorithm? Do different values for the hyperparameters influence the results? Are we capable to optimize the best policy after training?
- Does a more complex algorithm like DQN perform better in the Mountain Car environment? What are the best hyperparameter values for this algorithm? What are the pro and cons of such a model in a simple environment?
- Among algorithms using neural networks, which one between DQN and PPO performs the best in a more complex environment like Super Mario Bros? How does the choice of hyperparameter values influence the results? Are we able to solve the first Super Mario Bros level?

1.3 Summary of Results

In this section, we present a brief summary of the results found during this master thesis.

Q-learning in Mountain Car: We achieve quite easily the training of an agent with Q-learning to solve the Mountain Car problem. However, this environment is considered solved if we reach an average cumulative reward of -110 over 100 consecutive episodes. The best agent trained with Q-learning only achieved an average cumulative reward of -130.
The choice of parameters strongly influences the results. The best model is constructed by using 0.05 as learning rate, a discount rate of 0.99, a Q-table of dimension 40×40 and is trained during 40,000 episodes.
With this best model, we were able to construct an "image" of the policy, which shows which actions are taken in a given situation. Some actions inside a cluster are diverging, we could say there is a huge variance in the Q-table. To prevent that, we have "smoothed" the table.

After this "smoothing" the average cumulative reward was equal to -105, which is enough to consider the environment solved.

DQN in Mountain Car: We have found that agents trained by DQN perform better than the ones trained by Q-learning. The best agent has obtained an average cumulative reward of -99, solving the environment. Just like for the Q-learning, choosing the right hyperparameter values influences the result. An agent trained with the default values received an average cumulative reward of -137.
The primary advantage of DQN is its performances compared to the Q-learning. We can however note that finding the right parameters can be difficult. The training times for the two algorithms were globally the same, but we can achieve some good results with the Q-learning in much fewer training episodes compared to DQN. Finally, the DQN algorithm is more complicated to understand and to "see" the learning due to its neural network. It is more of a "black box" than the Q-learning.

Super Mario Bros environment: Solving this environment was the most challenging part of this master thesis. For both DQN and PPO, we have trained various models with different hyperparameter values. The best model trained by DQN achieved an average result over 50 consecutive episodes of 864. For PPO, we globally obtain the same performances with an average reward of 868.
As before, choosing the right hyperparameter values influences the rewards. However, we have tested different methodologies for preprocessing the environment and found that models receiving as inputs a scaled image of 84×84 pixels perform just as well or even better than models taking as inputs the complete 240×250 pixels image. Moreover, they are trained more quickly, 21 hours compared to 44 hours.
Despite both algorithms producing the same average rewards, the models trained by PPO produced more outliers, meaning that the agent goes more further in the level compared to an agent trained by DQN. We have even found that agents trained by PPO can sometime finish the first level and starting the second. The probability of occurring being close to 1 every 500 episodes.

1.4 Structure of the Thesis

Chapter 2 The theoretical concepts used during this master thesis are explained in this chapter. We initially start to explain Reinforcement Learning and present the Q-learning algorithm. Subsequently we define and mathematically develop how neural networks work, from their basic form like the perceptron to more complex networks like convolution neural networks. Finally, we will go into the deep learning part, where we explain the DQN and PPO algorithms.

Chapter 3 In this chapter, we present the methodology that was used to obtain the results. We begin by presenting the Gym environment and its basic functions. We also present the Mountain Car and Super Mario Bros environments. Afterwards, we explain how we have solved the Mountain Car environment with the Q-learning algorithm and DQN. Ultimately, we describe the preprocessing techniques employed to solve Mario and the methodology used to obtain the results for both DQN and PPO applied to Mario.

Chapter 4 The results are displayed in this chapter. Firstly, we present the results obtained on the Mountain Car environment with Q-learning and DQN. We compare the rewards and present a way to obtain better results by "smoothing" the Q-table for the Q-

learning algorithm. Subsequently, we solve the Super Mario Bros environment. We start by explaining the preprocessing methodologies used, then show the results obtained with DQN and PPO. Lastly, we investigate the probability of the most efficient agent to complete the first level.

Chapter 5 We conclude this master thesis by this chapter where we review the results obtained, discuss the possible limitations of the methodology and provide some suggestions for future work.

Chapter 2

State of the Art

This chapter is dedicated to the description of the concepts used inside this thesis. We will start by describing the notion of reinforcement learning. We will subsequently discover and review the history of neural networks and finally use these two notions to define deep reinforcement learning. We assume that the reader has the basic mathematical and machine learning knowledge required to understand the topics described inside this thesis.

2.1 Reinforcement Learning

2.1.1 Introduction

This section is devoted to the explanation of Reinforcement Learning (RL). The descriptions presented are mainly inspired from François-Lavet et al. (2018) and Sutton and Barto (2018).

Reinforcement Learning can be seen as a field of machine learning. Machine learning is defined by Mitchell (1997) and Bishop and Nasrabadi (2006) as a study that provides methods, algorithms that will improve through experience, thanks to the use of data, to achieve some tasks. Roughly, these tasks can be separated into three different categories:

- *Supervised learning* which will perform a regression or classification based on labelled training data (Russell and Norvig, 2002).
- *Unsupervised learning* is a field that groups type of algorithms that will learn patterns from unlabelled data.
- *Reinforcement learning* or RL which "is the task of learning how agents ought to take sequences of actions in an environment in order to maximize cumulative rewards" (François-Lavet et al., 2018).

The primary goal of RL is to train an agent to learn a "desirable" behavior. To achieve that, the agent will interact with his environment. By performing this, the agent will gather experience which allows him to take the right decision to achieve a defined objective. Generally, the process will be iterative. The agent will perform an action A at time t. This action will modify the environment in which the agent evolves, making it pass into a new state S_{t+1}. A reward R_{t+1} is given to the agent depending on the result of his action A_t. We provide a positive reward if the new state gets closer to the defined objective, negative if it is the opposite. The goal of the agent will be to maximize the cumulative sum of these rewards. Figure 2.1.1 presents the basic idea of how RL algorithms work.

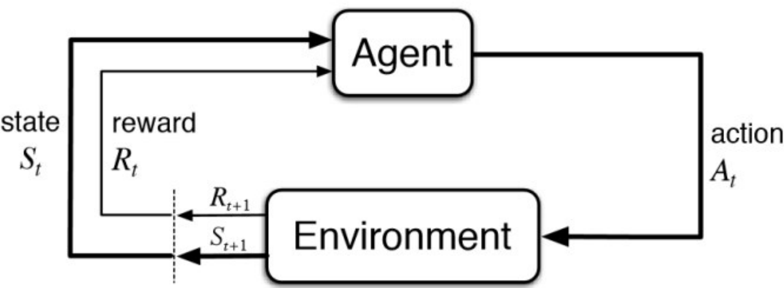


Figure 2.1.1: Basic schema of the functioning of an RL algorithm (Bhatt, 2018).

RL algorithms work by trial-and-error, as opposed to dynamic programming that needs to perceive its complete environment. In spite of this, both these techniques are involved in the modern definition of reinforcement learning. To explain what dynamic programming is, we need to come back to the late 1950s where the term "optimal control" was used to describe "the problem of designing a controller to minimize a measure of a dynamical system's behavior over time" (Sutton and Barto, 2018). To solve this problem, Richard Bellman and others employed the concepts of a dynamical system's state and a value function, which is currently called the Bellman equation. Dynamic programming refers to the ensemble of methods used to solve optimal control problems by solving the Bellman equation (Bellman, 1957a). The discrete stochastic version known as Markovian decision processes (MDP) was also introduced by Bellman (1957b).

2.1.2 Formal Definition

The RL environment will generally take the form of a Markov decision process. The first explanation of the control setting was proposed by Bellman (1957a) and extended to learning by Barto et al. (1983). It is globally the same scheme as we have described before, we merely add some notations that will be useful for the remainder of this master thesis. An agent in a given starting state $s_0 \in \mathcal{S}$ collects an initial observation $\omega_o \in \omega$. An action $a_t \in \mathcal{A}$ is taken by the agent at each time step t, leading to three results. The agent receives a reward $r_t \in \mathcal{R}$, the state of the environment moves to $s_{t+1} \in \mathcal{S}$, and the agent gathers a new observation $\omega_{t+1} \in \Omega$.

Markov Property

The environment is assumed to take the form of a Markov decision process, meaning it has to satisfy the Markov property. This property states that the future observations of the process depend only on the present observation and not of the past. The agent does not look at the history of the process. In this case, the process has to satisfy:

- $\mathbb{P}(\omega_{t+1}|\omega_t, a_t) = \mathbb{P}(\omega_{t+1}|\omega_t, a_t, ..., \omega_0, a_0)$ and,
- $\mathbb{P}(r_t|\omega_t, a_t) = \mathbb{P}(r_t|\omega_t, a_t, ..., \omega_0, a_0)$

Following Bellman (1957b), a Markov Decision Process (MDP) is a 5-tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ represents the transition function, i.e. a set of conditional transition probabilities between states
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ is the reward function. $\mathcal{R}$ represents the possible set of values taken by the reward
- $\gamma \in [0, 1]$ is a discount factor. It is used to weight the returns obtained by the reward function. This factor is more precisely described in the following sections.

If the state and action space are finite, we call this process a finite Markov decision process. In a MDP, the system is also fully observable, which means that the observation is the same as the state of the environment at each time step t: $\omega_t = s_t$. The probability of shifting to s_{t+1} is given by the state of the transition function $T(s_t, a_t, s_{t+1})$. The reward is computed thanks to the reward function $R(s_t, a_t, s_{t+1}) \in \mathcal{R}$.

Policies

The way an agent selects an action in a given state is specified by a policy π. This policy can be deterministic or stochastic. In the first case, depending on the state s, the policy will always give the same action a. This policy is described as $\pi(s) : \mathcal{S} \to \mathcal{A}$.
In the stochastic case, the action a chosen for a particular state s is determined in a probabilistic manner. This is defined as $\pi(s, a) : \mathcal{S} \to \mathcal{A}$, where $\pi(s, a)$ denotes this probability.

Expected Returns

The goal of the agent will be to maximize the cumulative rewards received in the long run. More formally, we will say that the agent seeks to maximize the *expected return.* In the simplest case, the return can be defined as the sum of the rewards: $G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$ where T is the final time step. This approach is possible if there exist a terminal state at which each episode can end. However, in many cases, there is no terminal state and the agent-environment interaction will go without limit. If we follow the previous formula, the final step will be $T = \infty$, meaning that the return that we try to maximize could be infinite. To prevent that, we add a discount parameter $\gamma \in [0, 1]$. The agent will now try to select the actions such that the sum of discounted rewards is maximized. This discounted return is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The smaller γ, the more weight is given to the earlier rewards. The agent will take actions that provide high rewards on a short horizon. The opposite is true for a γ close to one.

Value Functions

Value functions are at the center of RL algorithms. They represent "an estimate of "how good" it is for the agent to be in a given state" (Sutton and Barto, 2018). The term "how good" is represented by the expected return, which depend on the actions, hence the policy

followed by the agent. In a MDP, we define $v_\pi(s)$ the value of a state s under a policy π as the expected return starting at state s following the policy π, or:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right]$$

This function is called *state-value function* for policy π.

The same function that defines the value of taking an action a in state s under policy π is defined as $q_\pi(s, a)$:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.1.1}$$

This function is called *action-value* or *Q-value* function for a policy π.

These value functions used in reinforcement learning and dynamic programming fulfill a fundamental property, they satisfy particular recursive relationships. The equation 2.1.2c defines the relation between the value of a state and the values of its successor states. This equation is called the *Bellman equation for* v_π. It will average over all the possible states, weighting them by its probability of occurring.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] && \text{(2.1.2a)} \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right] \\
&= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)\left[r + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'\right]\right] && \text{(2.1.2b)} \\
&= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \ \ \forall s \in S && \text{(2.1.2c)}
\end{aligned}$$

By definition, the expected value of a variable can be defined as $\mathbb{E}(\mathbb{X}) = \sum_i \mathcal{P}(X = x_i)x_i$. In this case, the probability is defined by $\pi(a|s)p(s',r|s,a)$ where $\pi(a|s)$ is the probability of taking the action a in state s and $p(s',r|s,a)$ is the transition probability of going to state s' with reward r by being in state s and taking action a. These probabilities are used to weight the quantity in bracket. By taking the sum over a, s' and r, we compute all possibilities, meaning we compute the expectation.
The term $\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_{t+1} = s'$ in equation 2.1.2b is substituted by $v_\pi(s')$ as the equation 2.1.2a shows.

Thanks to these functions, we can determine the optimal policy that can be used to solve the RL task. "A policy π will be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all the states" (Sutton and Barto, 2018). This is translated as $\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. This is call an *optimal policy* denoted as π^*.
We can also define the *optimal state-value function* v^*:

$$v^*(s) = \max_\pi v_\pi(s), \ \ \forall s \in S$$

The *optimal action-value function* q^* is defined as:

$$q^*(s,a) = \max_{\pi} q_\pi(s,a), \ \ \forall s \in S, \ a \in A(s)$$

This function can be expressed in term of v^*:

$$q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a\right]$$

The optimal state-value function v^* is a value function for a policy, meaning it has to fulfill the self-consistency condition given by the Bellman equation 2.1.2c. The equation 2.1.3 is called the *Bellman optimality equation.* "It expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state" (Sutton and Barto, 2018). This function represents the maximum value obtained for a state s. We follow the same development used to find the equation 2.1.2c except that this time, a is fixed as the action that bring the maximum value in a state s.

$$\begin{aligned}
v^*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r + \gamma v^*(s')\right] \qquad (2.1.3)
\end{aligned}$$

The Bellman optimality equation for q^* is:

$$\begin{aligned}
q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q^*(s',a')\right] \qquad (2.1.4)
\end{aligned}$$

In practice, finding v^* thanks to the equation 2.1.3 lead to one unique solution for finite MDP. Solving this equation is like solving a system of N equations corresponding to N states with N unknowns. If the dynamics of the environment are known, i.e. if we understand how one state transition to another, which is represented by $p(s',r|s,a)$, we could in principle solve it by using methods to solve system of nonlinear equations. The same goes for q^*.
Once we get v^*, the optimal policy is found by choosing the action a for each state s that returns the maximum value in v^*. It represents a one-step search.
For q^* it is even easier, the one-step search is unnecessary. The agent has just to find the action that maximizes $q^*(s,a)$ for each state s. This function represents the optimal expected long-term returns for each state-action pair. This method is computationally more expensive than only finding value for the states, but it allows to select the optimal actions without having to know the possible successor states and their values, i.e. we do not need to know the environment's dynamics.

2.1.3 Solving Methods

One way to obtain estimates of $v^*(s)$ and $q^*(s,a)$ is to employ Monte Carlo methods. These methods obtain an estimate of the function by running several simulations and averaging the results (Kroese and Rubinstein, 2012). We will utilize this technique later to perform a Q-value algorithm. However, we will see that this method can not be applied in all situations due to computational requirements.

Dynamic Programming

Before presenting Monte Carlo methods, we will first describe Dynamic Programming (DP) methods. These methods refer to a collection of algorithms that can be used to compute the optimal policies seen before given an ideal model of the environment like a MDP (Sutton and Barto, 2018). These methods are at the base of RL, but their practical use is limited since the whole environment need to be known and due to their computational complexity.

We start by assuming a finite MDP, meaning that its state, action and rewards sets, S, $A(s)$ and R for $s \in S$ are finite and that the dynamic of the environment is given by a set of probabilities $p(s', r|s, a)$. The primary goal will be to use the value functions v^* of formula 2.1.3 and q^* of formula 2.1.4 to search the good policies for a given problem.

To determine the optimal policy, we can use the *value iteration* algorithm. Inside this one, the value v_k is computed at each iteration with the following formula:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right] \end{aligned}$$

for all $s \in S$. The sequence v_k can be shown to converge toward v^* for any starting v_0. However, it will require an infinite number of iterations to converge exactly to v^*. In practice, we will terminate the algorithm when the value function changes by a small amount. Once we have the "optimal" value function, we employ it to compute a deterministic policy $\pi \simeq \pi^*$. The idea will be to consider changes at all states and to all possible actions and to choose the action a that appears to be the best according to $q_\pi(s,a)$. In practice, it follows the equations:

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\right] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \end{aligned}$$

where $\arg\max_a$ is the value of a chosen to maximize the expression following. The pseudo code 1 below explains how the value iteration algorithm works (Sutton and Barto, 2018).

Q-learning Algorithm

The development of an off-policy temporal difference algorithm knows as *Q-learning* by Watkins (1989) represented a major advance for the RL community. Before explaining this algorithm, we first need to describe what *on-policy*, *off-policy* and *temporal difference* mean.

To understand the concept of *on-policy* and *off-policy*, we need to discuss a challenge first that is typical to RL, which is the trade-off between *exploration* and *exploitation* (Cohen

Algorithm 1 Value iteration algorithm

Initialize array V arbitrarily (representing the value for all states)
repeat
  $\Delta \leftarrow 0$
  for $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a) [r + \gamma v_k(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
return $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a) [r + \gamma v_\pi(s')]$

et al., 2007). To maximize his rewards, the agent will tend to select actions that he has already performed in the past, and that has been proved effective. However, in order to be able to discover these "good" actions, the agent has to try new ones. "The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must investigate a variety of actions and progressively favor those that appear to be best" (Sutton and Barto, 2018). A policy must follow an *optimal* behavior to learn action values, but they also need to behave non-optimally in order to explore all actions.

- In the *on-policy* approach, the agent attempts to improve the same policy that is used to select the actions. The exploration vs exploitation dilemma is solved by including randomness. It means that random actions can be selected with probability ϵ to ensure exploration. We also call this type of policy a ϵ-greedy policy.
- For the *off-policy* approach, we employ two policies. The first, the *target policy*, is used for function estimation and improvement. The second is there to generate the data use to compute the first policy and ensure the exploration. It is called the *behavior policy.*

Temporal difference learning (TD) represents a combination of Monte Carlo methods and dynamic programming. It learns from raw experiences without having to model the environment dynamics like Monte Carlo. And like dynamic programming, TD methods update the estimates on the basis of other learned estimates without waiting for the final outcome.
To explain, we will compare Monte Carlo and TD methods to get the prediction of v_π. Both methods update their estimate of v of v_π for non-terminal states S_t (Sutton and Barto, 2018). In the Monte Carlo method, we wait to know the return obtained by the visit at time t, then we update $V(S_t)$ with this return using the equation:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$

where G_t is the actual return at time t, and α is a constant step-size parameter. This method must wait to the end of an episode to know the value of G_t, and then update $V(S_t)$.
In temporal difference methods, we wait only until the next step. We update $V(S_t)$ at time $t+1$ by using the reward R_{t+1} and the estimate $V(S_{t+1})$ following the equation:

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

These two methods are linked. The Monte Carlo use an estimate of the equation 2.1.5a whereas the TD use an estimate of the equation 2.1.5b. For the MC, the expected value is not known, a sample return is used as estimate instead of the real expected return. For the TD, $v_\pi(S_{t+1})$ is not known. We use as estimate $V(S_{t+1})$.

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] && \text{(2.1.5a)}\\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right]\\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right]\\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\right] && \text{(2.1.5b)}
\end{aligned}
$$

Now that we have defined the concept, we can describe the *Q-learning* algorithm developed by Watkins (1989). This algorithm learns the action-value function Q as a direct approximation of q^*, the optimal action-value function, independently of the policy followed. The function is computed as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma\max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.1.6}$$

where α is the learning rate or step size. It determines how much new information override old information. If it is equal to zero, the agent will learn nothing and the closer it gets to one, the more the agent considers recent information. The γ is the same discount factor as explained before.
The algorithm 2 presents how the Q-learning method works (Sutton and Barto, 2018).

Algorithm 2 Q-learning algorithm

```
Initialize Q(s, a) arbitrarily, set Q(terminal − state, ·) = 0
for each episode do:
    Initialize S
    repeat for each step of episode:
        Select A from S using policy derived from Q
        Take action A, get R, S'
        Q(S_t, A_t) ← Q(S_t, A_t) + α [R_{t+1} + γ max_a Q(S_{t+1}, a) − Q(S_t, A_t)]
        S ← S'
    until S is terminal
```

Watkins and Dayan (1992) have proved the convergence of this algorithm to the optimal value function q^* under the following conditions:

- discretization of the state-action pairs
- all pairs continue to be updated, i.e. all actions are repeatedly sampled in all states to ensure a correct exploration

In this basic version, the algorithm will save $Q(S, A)$ as a table with one entry for every pair of state-action. However, this is inapplicable to many problems due to high-dimensional

state-action space. For example, the game of backgammon has 10^{20} states (Sutton and Barto, 2018) which is therefore impossible to solve with classic Q-learning method. To try to solve problems with many various states, scientists have come with the idea of approximating the target functions by an artificial neural network.

2.2 Neural Networks

In this section, we define and review the history of artificial neural networks. Artificial neural networks are inspired by the neurons present in the biological brain. An artificial neuron receives a signal in the form of some digits, processes it and output a result thanks to a non-linear transformation function. Typically, neurons are organized in layers that perform different transformations of their inputs. These inputs have a weight that can be adjusted through a learning process. Information travel from the first layer, called the input layer, could traverse some additional layers and ultimately reach the output layer.

This section is mainly based on Haykin (2009). Some elements described in video made by Machine Learnia (2021) are also used to acquire the fundamental idea behind neural networks.

2.2.1 Perceptron

The invention of neural networks is mainly due to Rosenblatt (1958) who has proposed the perceptron. It is the most basic form of neural network used to classify *linearly separable* patterns. It consists of a single neuron with bias and weights that are adjusted through an algorithm developed by Rosenblatt (1958, 1961). Figure 2.2.1 represents how the neuron works, where $x_1, x_2, ..., x_m$ are the inputs, $w_{k1}, w_{k2}, ..., w_{km}$ represent the weights of the neuron k and b_k is the bias apply on the neuron k.

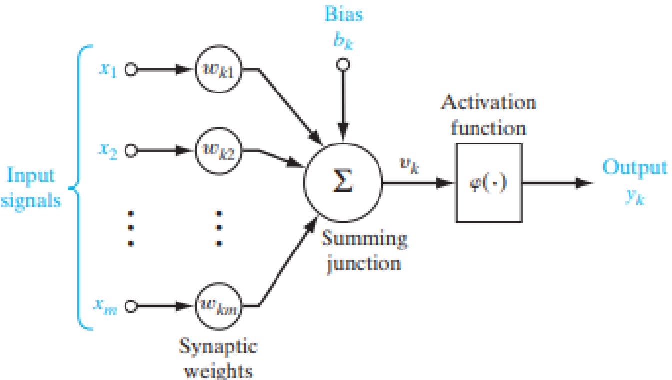


Figure 2.2.1: Nonlinear model of a neuron k (Haykin, 2009)

.

Mathematically, we can represent this neuron k as:

$$z_k = \sum_{j=1}^{m} w_{kj} x_j + b_k \tag{2.2.1}$$

and

$$a_k = \varphi(z_k) \tag{2.2.2}$$

where z_k represents the linear combination of the input signal plus the bias, $\varphi(\cdot)$ is the *activation function* who will transform the combination v_k into the two different classes. There are principally two types of activation functions, threshold functions and sigmoid functions. Modern algorithms can utilize other types of activation functions (ReLU). They will be described later.

- **Threshold function** is called a *Heaviside function* with a corresponding decision defined by:

$$\varphi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2.2.3}$$

  Neurons using this type of activation function are also referred as the *McCulloch-Pitt model* due to the previous work achieved by McCulloch and Pitts (1943).

- **Sigmoid function** represents the most common form of activation function that we can find in a neural network. These functions take the form of an "S" on a graph. An example of such a function is the *logistic function* defined by:

$$\varphi(z) = \frac{1}{1 + \exp(-az)} \tag{2.2.4}$$

  where a is the slope parameter of the function. This function will assume a continuous range of value between zero and one. The main advantage of this function is its differentiability, compared to threshold function. We use this property later in this section.

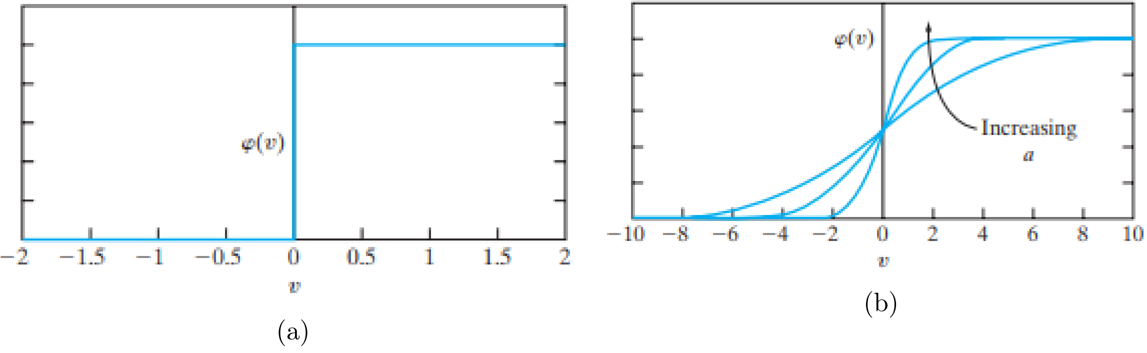


Figure 2.2.2: Representation of (a) threshold function, and (b) logistic function (Haykin, 2009)

.

2.2.2 Backpropagation

Before using a neural network to achieve some tasks, we first need to train the model. In case of a neural network, it means adjusting the weights w and the bias b. The popular method to train a model is called the backpropagation algorithm. This technique was first used by Rumelhart et al. (1986) in neural network framework and developed by Rumelhart

et al. (1985) [1]. This concept was not new. The basis is derived by Kelley (1960) and Bryson (1961). A simpler solution to the one found before is presented by Dreyfus (1962) based on the chain rule. The first implementation on a computer is made by Linnainmaa (1970). This implementation represents a general method for automatic differentiation.

The backpropagation algorithm can be differentiated in two phases:

- **Forward phase:** the weights are fixed and the input signal is propagated through the network until reaching the output.
- **Backward phase:** we compare the output received to the desired response. This comparison is performed by computing the error. This error is then propagated through the network in the backward direction. During this propagation, weights are adjusted according to some rules.

The cost function associated to the network is called a loss function. It maps the value of multiple variables into a real number. In backpropagation framework, the cost function calculates the difference between the outputs of the network and their expected values. This function must fulfil some assumption to be used in backpropagation algorithm (Nielsen, 2015). First, for n training examples x and an error function E_x, it can be noted as an average $E = \frac{1}{n}\sum_x E_x$. The second is it can be written as a function of the outputs from the neural network.

The weight adjustments are done via a gradient descent method following this formula:

$$W_{t+1} = W_t - \alpha \frac{\partial E}{\partial W_t} \tag{2.2.5}$$

where E is the loss function, W_t are the weights at time t and $\alpha > 0$ is the learning rate. The core objective of the backpropagation algorithm is to compute the partial derivative of the error E by regard to the weights W.

To get the intuition behind the backpropagation algorithm and define the equation needed to achieve it, we start at the basis with an example consisting of two inputs x_1, x_2, one neuron and m training data. The method and equations come from Machine Learnia (2021), which follows Rumelhart et al. (1985) and Haykin (2009).

For this example with two variables and one neuron, we can write the neuron from the equation 2.2.1 as:

$$z(x_1, x_2) = w_1 x_1 + w_2 x_2 + b \tag{2.2.6}$$

The sigmoid function is the logistic function defined in equation 2.2.4 with slope $a = 1$. This sigmoid function represents the probability that one training example belong to one class. This classification problem can be represented by a Bernoulli law:

$$\mathcal{P}(Y = y) = \varphi(z)^y \times (1 - \varphi(z))^{1-y} \tag{2.2.7}$$

We use this equation to define the loss function. In this case, we will use the maximum likelihood function. More specifically, we compute the minimum of the negative log likelihood.

[1] Despite the date of the second paper is before the first, the second paper is a generalization of the backpropagation technique. "We describe a new learning procedure, back-propagation, for networks of neurone-like units" (Rumelhart et al., 1986). "This paper presents a generalization of the perception learning procedure for learning the correct sets of connections for arbitrary networks" (Rumelhart et al., 1985). This temporal issue is probably linked to the time taken to publish the articles.

The loss function E will be:

$$E = -\frac{1}{m}\sum_{i=1}^{m} y_i \log(\varphi(z_i)) + (1 - y_i)\log(1 - \varphi(z_i)) \quad (2.2.8)$$

As explain before, the goal is to adjust the weights in the network thanks to the formula 2.2.5. We compute $\frac{\partial E}{\partial W}$ by chain rule [2]:

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial \varphi(z)} \times \frac{\partial \varphi(z)}{\partial z} \times \frac{\partial z}{\partial W_1} \quad (2.2.9)$$

By computing the partial derivatives, we found:

i)

$$\frac{\partial E}{\partial \varphi(z)} = -\frac{1}{m}\sum \frac{y}{\varphi(z)} - \frac{1-y}{1-\varphi(z)} \quad (2.2.10)$$

ii)

$$\frac{\partial \varphi(z)}{\partial z} = \varphi(z)(1 - \varphi(z)) \quad (2.2.11)$$

To compute this equation, we need a differentiable activation function, hence the use of the sigmoid function defined in equation 2.2.4. We will see that ReLU activation function, which are not differentiable in zero, can also be used.

iii)

$$\frac{\partial z}{\partial W_1} = x_1 \quad (2.2.12)$$

By putting all these equations inside the equation 2.2.9, we found that:

$$\begin{aligned}\frac{\partial E}{\partial W_1} &= -\frac{1}{m}\sum \left(\frac{y}{\varphi(z)} - \frac{1-y}{1-\varphi(z)} \times \varphi(z)(1-\varphi(z)) \times x_1\right) \\ &= -\frac{1}{m}\sum_{i=1}^{m}(y_i - \varphi(z_i))x_{1i}\end{aligned}$$

The equation for $\frac{\partial E}{\partial W_2}$ and $\frac{\partial E}{\partial b}$ are nearly the same:

$$\frac{\partial E}{\partial W_2} = -\frac{1}{m}\sum_{i=1}^{m}(y_i - \varphi(z_i))x_{2i}$$

$$\frac{\partial E}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}(y_i - \varphi(z_i))$$

[2] "In Leibniz's notation, if a variable z depends on the variable y, which itself depends on the variable x (that is, y and z are dependent variables), then z depends on x as well, via the intermediate variable y. In this case, the chain rule is expressed as" (Wikipedia contributors, 2022):

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

We can generalize the model (still one neuron) to an indeterminate number of variables (inputs) by vectorizing the equations. To achieve that, we define the matrix $X \in \mathbb{R}^{m \times n}$ where m is the number of observations and n the number of variables. The vector of label is represented by $Y \in \mathbb{R}^{m \times 1}$.
The vectorization of the neuron v will be $V \in \mathbb{R}^{m \times 1}$:

$$Z = XW + b$$

where $W \in \mathbb{R}^{n \times 1}$ and $b \in \mathbb{R}^{m \times 1}$.
From this one, the activation function become $A = \varphi(Z) \in \mathbb{R}^{m \times 1}$.
Finally, the weights are updated following:

$$W = W - \alpha \frac{\partial E}{\partial W}$$

with $\frac{\partial E}{\partial W} \in \mathbb{R}^{n \times 1}$. The gradient can also be expressed as:

$$\frac{\partial E}{\partial W} = -\frac{1}{m} X^{\intercal}(A - Y)$$

where the matrix X is transposed because $X \in \mathbb{R}^{m \times n}$ and $(A - Y) \in \mathbb{R}^{m \times 1}$.
For the bias, the descent is expressed as:

$$b = b - \alpha \frac{\partial E}{\partial b} \tag{2.2.13}$$

where $\frac{\partial E}{\partial b} = -\frac{1}{m} \sum (A - Y)$, which provide a real number.

With all these formulae, we can train a neural network. The pseudo algorithm uses to train a neural network is described below. The number of iterations to train the neural network is not well defined. Following Kramer and Sangiovanni-Vincentelli (1988), we can say that the algorithm as converged when "the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold". Haykin (2009) proposes another stopping criterion by considering the absolute rate of change of the squared error, and stopping the algorithm when this value is below a sufficiently small threshold.

Algorithm 3 Training a basic neural network

Take $X \in \mathbb{R}^{m \times n}$ as input, $Y \in \mathbb{R}^{m \times 1}$ as label
Initialise weights and bias
repeat:
  Compute output of the neuron: $Z = XW + b$
  Compute activation value: $A = \varphi(Z)$
  Compute loss function E thanks to activation value A and labels Y:
    $E = -\frac{1}{m} \sum_{i=1}^{m} y_i \log(A_i) + (1 - y_i) \log(1 - A_i)$
  Compute gradients: $\frac{\partial E}{\partial W} = -\frac{1}{m} X^{\intercal}(A - Y)$ and $\frac{\partial E}{\partial b} = -\frac{1}{m} \sum (A - Y)$
  Update weights and bias: $W = W - \alpha \frac{\partial E}{\partial W}$ and $b = b - \alpha \frac{\partial E}{\partial b}$
until change of error $E <$ small value

2.2.3 Multilayer Perceptron

In the previous section, we have defined the basic perceptron, which represents a single-layer network. This type of network is limited to the classification of linearly separable patterns. The *multilayer perceptron* have been invented to overcome this limitation.

This type of network conserves the fundamental idea behind the perceptron, except that it can be composed of one or more hidden layers. These layers are *hidden* from the input and output layers. The number of neurons per layer can also change. Figure 2.2.3 shows an example of a multilayer perceptron.

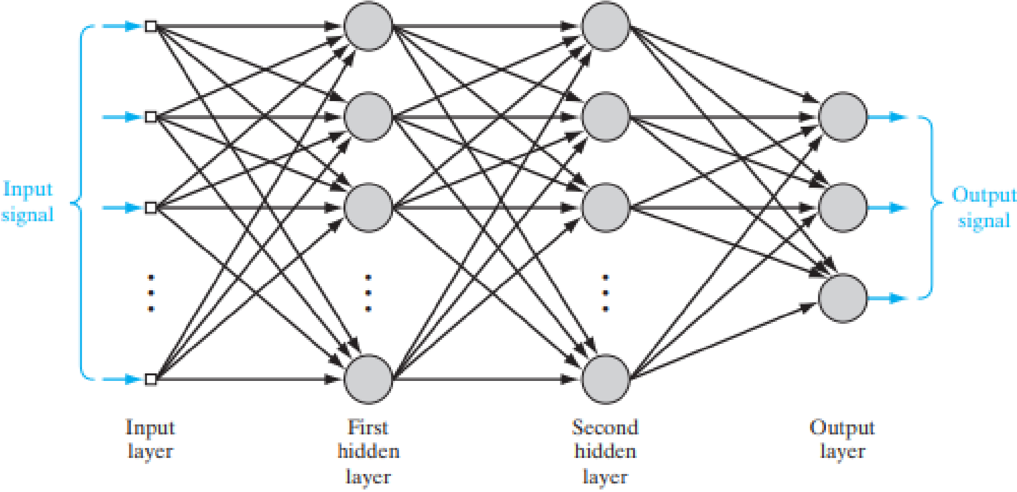


Figure 2.2.3: Design of a multilayer perceptron with two hidden layers (Haykin, 2009).

Despite being more complex than a single-layer perceptron, the training of a multilayer perceptron can also be achieved through backpropagation. Before explaining all the equations, we first need to perceive how the inputs are propagated inside such a network. To keep simple, we imagine a network with two neurons as input, one hidden layers composed of two neurons and one output neuron.

As before, the inputs propagate through the weights to reach the neuron in the first layer. In each neuron, we will compute the value z following the equation 2.2.1. Once we have a z-value, we can compute the activation value using an activation function, in this case the logistic. We obtain the equations:

$$\text{Neuron 1} = \begin{cases} z_1 = w_{11}x_1 + w_{12}x_2 + b_1 \\ a_1 = \frac{1}{1+\exp -z_1} \end{cases}$$

$$\text{Neuron 2} = \begin{cases} z_2 = w_{21}x_1 + w_{22}x_2 + b_2 \\ a_2 = \frac{1}{1+\exp -z_2} \end{cases}$$

The weights are now defined by w_{ij} where i refer to the starting neuron and j the ending neuron.
The activation values a_1 and a_2 are now propagated through other weights to reach the output layer, where they will serve as input to compute, once again, the z values, which is used to compute the new activation value of the output. In mathematical terms:

$$\text{Neuron } 1^{[2]} = \begin{cases} z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + b_1^{[2]} \\ a_1^{[2]} = \frac{1}{1+\exp -z_1^{[2]}} \end{cases}$$

A new superscript appears in the notation. It indicates where in the NN architecture the weight is located. For example, $w_{21}^{[2]}$ represents the weight located between the hidden layer (second layer in the network after the input layer) and the output layer, representing the link between the neuron 2 from the hidden layer to the neuron 1 of the output layer.
As before, we can vectorize these equations to get the matrix of z-values and activation values a. the results are the equations:

$$\begin{aligned} \text{Layer } 1 &= \begin{cases} Z^{[1]} = W^{[1]} \cdot X + b^{[1]} \\ A^{[1]} = \frac{1}{1+\exp -Z^{[1]}} \end{cases} \\ \text{Layer 2 (output layer)} &= \begin{cases} Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]} \\ A^{[2]} = \frac{1}{1+\exp -Z^{[2]}} \end{cases} \end{aligned} \tag{2.2.14}$$

Thanks to $A^{[2]}$, we can compute the loss function E.

$$E = -\frac{1}{m} \sum y \log(A^{[2]}) + (1-y) \log(1 - A^{[2]}) \tag{2.2.15}$$

Finally, this loss function is used to derive the gradients. In this example, the gradients are defined by the chain rule:

$$\frac{\partial E}{\partial W^{[2]}} = \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \tag{2.2.16}$$

$$\frac{\partial E}{\partial b^{[2]}} = \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \tag{2.2.17}$$

$$\frac{\partial E}{\partial W^{[1]}} = \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A[1]}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \tag{2.2.18}$$

$$\frac{\partial E}{\partial b^{[1]}} = \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A[1]}{\partial Z^{[1]}} \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \tag{2.2.19}$$

The demonstrations and the final equations for the gradients are inspired from Machine Learnia (2021). We can see in the chain rule equations that there is always a common part between $\frac{\partial E}{\partial W}$ and $\frac{\partial E}{\partial b}$. We define this common part as:

$$\begin{aligned} dZ2 &= \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\ dZ1 &= \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A[1]}{\partial Z^{[1]}} \end{aligned} \tag{2.2.20}$$

We first start with $dZ2$. By replacing $\frac{\partial E}{\partial A^{[2]}}$ by the value already defined in the equation 2.2.10 and $\frac{\partial A^{[2]}}{\partial Z^{[2]}}$ by the value in the equation 2.2.11, we found after some simplifications:

$$\begin{aligned} dZ2 &= \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\ &= \frac{1}{m} \sum \left(\frac{-y}{A^{[2]}} + \frac{1-y}{1-A^{[2]}}\right) \times A^{[2]}(1 - A^{[2]}) \\ &= \frac{1}{m} \sum A^{[2]} - y \end{aligned}$$

For notational simplicity, we drop the term $\frac{1}{m}\sum$. This term will be reused later to simplify some equations. The value of $dZ2$ is therefore equals to:

$$dZ2 = A^{[2]} - y \tag{2.2.21}$$

One can notice that we have a problem with the dimensions. $dZ2 \in \mathbb{R}^{n^{[2]}\times m}$ whereas $A^{[2]} \in \mathbb{R}^{n^{[2]}\times m}$ and $y \in \mathbb{R}^{1\times m}$. To solve that, we apply Broadcasting. It is a technique used when "arrays with different sizes cannot be added, subtracted, or generally be used in arithmetic. A way to overcome this is to duplicate the smaller array so that it is the dimensionality and size as the larger array" (Bronlee, 2018). y will so be duplicated to match $A^{[2]}$.

Now that we have defined $dZ2$, we can replace it in the equation 2.2.16. $\frac{\partial Z^{[2]}}{\partial W^{[2]}}$ is computed via $Z^{[2]}$ in the equation 2.2.14.

$$\begin{aligned}\frac{\partial E}{\partial W^{[2]}} &= \frac{1}{m}\sum dZ2 \times \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\ &= \frac{1}{m}\sum dZ2 \times A^{[1]}\end{aligned}$$

We still have a problem with the dimension of the matrix. $\frac{\partial E}{\partial W^{[2]}}$ is in dimensions $(n^{[2]}\times n^{[1]})$. The dimensions of $dZ2$ was defined before as $(n^{[2]} \times m)$ and $A^{[1]}$ has dimensions $(n^{[1]} \times m)$. To obtain the correct dimensions for $\frac{\partial E}{\partial W^{[2]}}$, we transpose $A^{[1]}$ and take the dot product. The sum is implicitly included in the dot product, giving as final equation:

$$\frac{\partial E}{\partial W^{[2]}} = \frac{1}{m} dZ2 \cdot A^{[1]^{\intercal}}$$

To compute $\frac{\partial E}{\partial b^{[2]}}$, we substitute $dZ2$ in the equation 2.2.17 and find the partial derivative of $Z^{[2]}$ by regard to $b^{[2]}$, which is computed via $Z^{[2]}$ in the equation 2.2.14. We obtain:

$$\frac{\partial E}{\partial b^{[2]}} = dZ2 \times \frac{\partial Z^{[2]}}{\partial b^{[2]}} \quad = dZ2 \times 1$$

Once again we have a problem with the dimensions. $\frac{\partial E}{\partial b^{[2]}}$ has dimensions $(n^{[2]} \times 1)$ and $dZ2$ has dimensions $(n^{[2]} \times m)$. To solve that, we will use the sum that we have keep aside. If we take the sum following the first axis (sum of the columns), we change the dimensions of $dZ2$ from $(n^{[2]} \times m)$ to $(n^{[2]} \times 1)$. It is represented in the equation by the $\sum_{axis\,1}$. The final equation became:

$$\frac{\partial E}{\partial b^{[2]}} = \frac{1}{m}\sum_{axis\,1} dZ2 \tag{2.2.22}$$

Now that we have found the gradients for the second layer, we need to compute the ones related to the first layer. To achieve that, we start by computing $dZ1$ following the equation 2.2.20. We can see that $dZ2$ can be substituted inside this one. The remaining term as computed by looking at the equations 2.2.14. It gives us:

$$\begin{aligned}dZ1 &= \frac{\partial E}{\partial A^{[2]}} \times \frac{\partial A^{[2]}}{\partial Z^{[2]}} \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A[1]}{\partial Z^{[1]}} \\ &= dZ2 \times \frac{\partial Z^{[2]}}{\partial A^{[1]}} \times \frac{\partial A[1]}{\partial Z^{[1]}} \\ &= dZ2 \times W^{[2]} \times A^{[1]}(1 - A^{[1]})\end{aligned}$$

By looking at the dimensions, we see that $dZ1 \in (n^{[1]} \times m)$, $dZ2 \in (n^{[2]} \times m)$, $w^{[2]} \in (n^{[2]} \times n^{[1]})$ and $A^{[1]}(1 - A^{[1]}) \in (n^{[1]} \times m)$. We arrange the equation by putting $W^{[2]}$ in front of the equation, transposing it and taking its dot product with $dZ2$. The final equation is:

$$dZ1 = W^{[2]\intercal} \cdot dZ2 \times A^{[1]}(1 - A^{[1]})$$

It is important to note that the multiplication ($\times$) is not a dot product between the matrix but a simple term to term multiplication.

With this value for $dZ1$, we can compute the gradients defined in the equations 2.2.18 and 2.2.19. Once again, the partial derivation is achieved by looking at $Z^{[1]}$ in the equations 2.2.14. It gives us for $\frac{\partial E}{\partial W^{[1]}}$:

$$\begin{aligned}\frac{\partial E}{\partial W^{[1]}} &= dZ1 \times \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ &= dZ1 \times X\end{aligned}$$

To match the dimensions $(n^{[1]} \times n^{[0]})$ of $\frac{\partial E}{\partial W^{[1]}}$, we need to transpose X which has dimensions $(n^{[0]} \times m)$ in order to take the dot product of $dZ1$ (dimensions $(n^{[1]} \times m)$)). We also need to add the term keep aside from $dZ2$, which produces as final result:

$$\frac{\partial E}{\partial W^{[1]}} = \frac{1}{m} dZ1 \cdot X^{\intercal}$$

For $\frac{\partial E}{\partial b^{[1]}}$, we obtain:

$$\begin{aligned}\frac{\partial E}{\partial b^{[1]}} &= dZ1 \times \frac{\partial Z^{[1]}}{\partial b^{[1]}} \\ &= dZ1 \times 1\end{aligned}$$

When we look at the dimensions, we observe the same issue as the one observed when computing $\frac{\partial E}{\partial b^{[2]}}$. We apply the same methodology to solve it, the final equation is:

$$\frac{\partial E}{\partial b^{[1]}} = \frac{1}{m} \sum_{axis1} dZ1$$

The last step is to generalize the previously elaborated equations to neural networks of any given architecture, independently of its number of layers and neurons. We have seen that during the forward propagation, the next value of Z will depend on the previous value of A. During the back propagation, some values also depend on the previous values of another layer. For this reason, we define the *layer* $l \in [1, L]$ where L is the number referring to the last layer (the output layer).

For each layer l, we have m training examples and $n[l]$ number of neurons in the layer l:

a) **Parameter initialization:** we set random values for the weights and bias but they have to respect the following dimensions:

$$W^{[l]} \in \mathbb{R}^{n[l] \times n[l-1]} \tag{2.2.23}$$

$$b^{[l]} \in \mathbb{R}^{n[l] \times 1} \tag{2.2.24}$$

b) **Forward propagation:** in order to simply the formulas, we set that $X = A^{[0]}$

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]} \tag{2.2.25}$$

$$A^{[l]} = \frac{1}{1 + \exp -Z^{[l]}} \tag{2.2.26}$$

c) **Back propagation:**

$$dZ^{[L]} = A^{[L]} - y \tag{2.2.27}$$

$$dZ^{[l-1]} = W^{[l]\intercal} \cdot dZ^{[l]} \times A^{[l-1]}(1 - A^{[l-1]}) \tag{2.2.28}$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]\intercal} \tag{2.2.29}$$

$$db^{[l]} = \frac{1}{m} \sum_{axis\,1} dZ^{[l]} \tag{2.2.30}$$

where the first equation stand for the last layer L while setting $X = A^{[0]}$.

d) **Parameter update:**

$$W^{[l]} = W^{[l]} - \alpha \times dW^{[l]} \tag{2.2.31}$$

$$b^{[l]} = b^{[l]} - \alpha \times b^{[l]} \tag{2.2.32}$$

where α is the learning rate.

With all these formulas, one can implement its own neural network with the number of layers and neurons of one's preference.

2.2.4 Challenges and Solutions

As seen before, the single layer perceptron can only be used to classify linearly separated classes. If we want to differentiate non-linear classes, we have to use a multilayer perceptron. By adding more neurons and more layers, a MLP can approximate any functions and so solve any problems. It is called an *universal function approximator* that is proved by the *universal approximation theorem* (Hornik et al., 1989). This proof does not indicate the number of neurons and layers that is required to achieve this approximation, as well as the weights and learning parameters.

Due to this property, one can be tempted to put a lot of neurons and hidden layers in the architecture of his neural network. However, some problems could occur.

The more obvious is the ***computation time***. As we increase the number of parameters to compute, the time required to perform all the computations will increase tremendously. We are clearly limited by the hardware limitations. It is for this reason that despite being known since 1950s, the revolution of deep learning only took place relatively recently. In 2012, we saw the performance of MLP increase dramatically due to the increasing number of layers inside the networks. This was possible thanks to the growing among of data and computing power that become available in 21th century (Lee, 2019).

One way to optimize this computation time is to increase the learning rate α. If this rate is set too small, it produces a long training process and the optimizer could get stuck in (too) local minimum. On the other hand, if it is too large, we could obtain a suboptimal solution or an

unstable training process (Brownlee, 2019). One solution is to change the learning rate during the training process. We typically start with a high learning rate that decreases during the process. We can additionally use the *adaptive control of the learning rate* that was discovered by Murata (1998).

The second problem, appeared around the year 2000, is the ***vanishing gradient problem***. This issue is encountered in neural networks trained via backpropagation using gradient methods. In these MLP, the weights are updated proportionally to the gradient value. It can happen that this value is too small and gradually decreases during backpropagation. This can lead to weights not being updated or, even worse, to terminate the learning process (Basodi et al., 2020). Despite not being explained in this master thesis, the vanishing gradient problem is a strong issue for recurrent neural networks (RNN). Several methods can be employed to avoid the vanishing gradient problem like using ReLu activation functions, batch normalization and residual neural network, who are a type of deep network that use connections or shortcuts to jump over some layers (He et al., 2016).

A) **Rectified Linear Unit (ReLU)** is a type of activation function that is defined by the positive part of its arguments: $f(x) = x^+ = \max(0, x)$ where x is the input to a neuron (Liu, 2017; Brownlee, 2019). This activation function is typically used for hidden node due to their semi-linear behaviour that provides an easy learning. For output nodes, we prefer other functions like the sigmoid. Figure 2.2.4 represents the ReLU function.

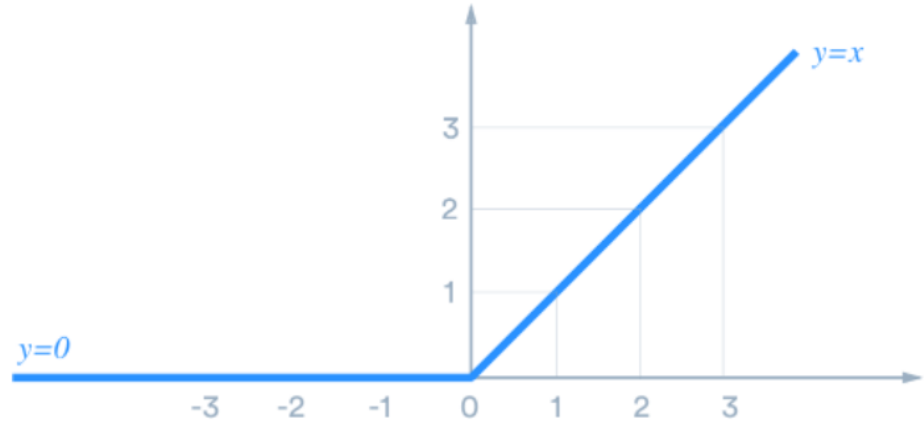


Figure 2.2.4: Representation of ReLU function (Liu, 2017).

The particular advantages of this function are:

1) *Computational simplicity:* this function does not require computing an exponential, like the sigmoid function. "Computations are also cheaper: there is no need for computing the exponential function in activations" (Glorot et al., 2011).
2) *Sparsity:* refers to the fact that negative inputs will have a ReLU value equal to zero. Neurons in a neural network should not always be activated depending on the input. For example, "in a model detecting cats in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is about

a building" (Liu, 2017). That allows the network to be faster to train as they are fewer neurons to compute.

3) *Gradient propagation:* this function has fewer vanishing gradient due to its only positive values. The saturation can occur from only one side, compared to the sigmoid function who accepts negative values (Glorot et al., 2011).

Although this function has many advantages, some problems can occur. Firstly, the function is undifferentiable in zero. This can be easily solved by arbitrarily setting the value of its derivative to 0 or 1. The second, called the dying ReLU problem, is more challenging. It is a form of vanishing gradient that happen when a neuron is pushed in inactive states for most inputs. In this case, no gradients flow backward through the neuron. The neuron is stuck in this state and "dies". If a significant number of neurons in the network are stuck in this state, the training process is no more efficient and the network could not perform well (Lu et al., 2019).
A way to avoid the dying ReLU problem is to use a variation of the ReLU function based on the following equation (Van Oirbeek, 2020):

$$g(z_i, \alpha_i) = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- **Absolution value rectification** is computed by fixing α_i to -1, s.t. $g(z_i) = |z_i|$
- **Leaky ReLU** by fixing α_i to a small value. In this case, the output for a negative value of z_i will be different from zero, allowing the gradient to pass through the neuron.

B) **Batch normalization** is a method proposed by Ioffe and Szegedy (2015) that makes the neural network faster and more stable thanks to the normalization of the layer's inputs by scaling and centering. In deep neural network, each layer has inputs with corresponding distribution. These distributions are affected during the learning process by the randomness in the parameter initialization and input data. The effect of these sources of randomness on the distributions is called the *internal covariance shift*. In addition to reduce this internal covariance shift, batch normalization allows using higher learning rate without vanishing or exploding gradients. Last but not least, some regularizing effect can be observed, improving the network generalization property which mitigates the overfitting.
Despite all these advantages, there are some undesirable properties like the dependence to the batch size and interaction between examples. Recently, Brock et al. (2021) have developed the Normalizer-Free Nets that achieved strong performance on image classification while being faster to train.

One last problem that could arrive when increasing the size of the network is ***overfitting***. It is a recurrent issue in machine learning where "the model performs perfectly on training set, while fitting poorly on testing set. This is due to that overfitted model has difficulty coping with pieces of the information in the testing set, which may be different from those in the training set" (Ying, 2019).
The simple solution will be to reduce the size of the network, but the overall performance could decrease. Luckily, other techniques can reduce overfitting. They are called *regularization* techniques. As we have seen before, *batch normalization* is one of them. Another technique wildly used is known as the *weight decay* or *L2 regularization* (Nielsen, 2015). The idea is to

add an extra term to the cost function, called the regularization term. For example, the loss function defined in the equation 2.2.15 become:

$$E = -\frac{1}{m}\sum y\log(A) + (1-y)\log(1-A) + \frac{\lambda}{2m}\sum_{w} w^2$$

where $\lambda > 0$ is the regularization parameter and w are the weight values.
The goal of this function is to make the network prefer to learn small weights. "The smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. That makes it difficult for a regularized network to learn the effects of local noise in the data" (Nielsen, 2015). If the network includes large weights, it tends to model noise present in the training data, which is something we want to avoid in order to achieve a decent generalization of the model.

2.2.5 Gradient Descent Variants

So far, we have determined the classical way to compute the gradient descent during back-propagation. However, there are primarily three variants of gradient descent techniques. For each technique, the amount of data used to compute the loss function varies. A trade-off between the accuracy of the parameter update and the time it takes to perform an update is made (Ketkar and Santana, 2017).

Before going further into the description of the gradient techniques, we correctly define the difference between several terms commonly used in deep learning. These definitions come from Brownlee (2018):

1) ***Sample*** is a single row of data. It contains inputs that feed the algorithm and labels used to compare the predictions and compute the errors. It is our x and y.

2) ***Batch size*** is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. It can be viewed as a for-loop over the samples before updating the weights of the model. If the dataset can not be divided by the batch size, the final batch will be composed of fewer examples.

3) ***Epoch*** is a hyperparameter that defines the number of times that the learning algorithm will work through the entire training dataset. One epoch means that each sample has had the opportunity to update the weights of the model. An epoch can include one or more batches.

We can view the training process as a for-loop over the number of epochs, within this loop another for-loop is nested that iterate over each batch of samples, where one batch is composed of the batch size number of training samples. After each batch the parameters are updated and after all the epoch, the algorithm end. The algorithm can stop before running through all epoch, in this case, it typically stop after a complete epoch is finished, and not in the middle of one.

Now that these concepts are clear, we can explain the difference between the gradient descent techniques.

A) **Batch gradient descent** represents the original technique we have seen so far. All the training data are taken into consideration to compute the loss function. When we have

computed all the gradients from the training examples, we average them and use this mean to update the parameters. In this algorithm, the batch size equal the size of the dataset, meaning the parameters are updated at each epoch.
In most cases, this algorithm represents a non-viable option due to large dataset. We might run out of memory to load the entire batch.

B) **Stochastic gradient descent** updates the weights inside the model by using one sample at a time. It is similar to say we use a batch size equal to 1. The consequences are less memory used and a faster learning process. However, due to the frequent update of the weights, the loss function fluctuates a lot, despite reaching a minimum, it continues to oscillate around this one (Patrikar, 2019). To summarize it, for one epoch, we (1) select an example, (2) feed the neural network, (3) compute the gradients, (4) update the weights, (5) repeat steps 1-4 for all examples in the training dataset.

C) **Mini-batch gradient descent** represents a compromise between the two techniques seen before. We use a batch size greater than 1 and smaller than the entire dataset. It is considered as a good trade-off between a low variance and computation time. We implement the same algorithm as for the stochastic technique, but by replacing one training example by a batch of training example.
The batch size depends on the problem but generally speaking it is chosen "between 1 and a few hundreds, e.g. B = 32 is a good default value, with values above 10 taking advantage of the speed-up of matrix-matrix products over matrix-vector products" (Bengio, 2012). Moreover, it is recommended but not mandatory to select a number in range of powers 2 like 16/32/64/... as batch size, as it accommodates best from computation perspective (Ketkar and Santana, 2017).

D) **Adam**, derived from adaptive moment estimation, is a method proposed by Kingma and Ba (2014). It is an optimization of the stochastic gradient. This method combines the AdaGrad method of Duchi et al. (2011) and the RMSProp of Tieleman and Hinton (2012). It computes an exponential moving average of the gradient and the squared gradient, while adding some parameters to control the decay rates of these moving averages.
Concretely, at time step t, we get the gradients g_t. With these gradients we update m_t, which is the biased first moment estimate and v_t, the biased second raw moment estimate.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

These biased estimates are then used to compute the bias-corrected first and second moment estimate.

$$\hat{m_t} = \frac{m_t}{(1 - \beta_1^t)}$$
$$\hat{v_t} = \frac{v_t}{(1 - \beta_2^t)}$$

Finally, we update the parameters of the model using the following formula:

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon} \tag{2.2.33}$$

Some notes about the equations. First, all operations on vectors are element-wise, it is not a dot product. α represents the learning rate, β_1 the exponential decay rate of the first moment estimate, β_2 the exponential decay rate of the second moment estimate, and ϵ is a small value to prevent any division by zero. Following Kingma and Ba (2014), good default settings for these values are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The value for epsilon can change depending on the problem. For example, TensorFlow suggests that "when training an Inception network on ImageNet a current good choice is 1.0 or 0.1" (Abadi et al., 2015).

2.2.6 Convolutional Neural Network

Convolutional neural networks (CNN) are a type of neural network that has been developed by Fukushima and Miyake (1982). They are frequently applied to data having a grid-like topology, like images that can be described as a 2D grid of pixels. Their name comes from the term convolution which is a mathematical operation. Most of the information present inside this section comes from Goodfellow et al. (2016), Nielsen (2015) and Udyavar (2017).

Description

Multi Layer Perceptrons seen so far are unsuited for working with images. These networks do not take into account the spatial architecture of the image. Pixels close or far to each other are treated the same way. The goal of convolutional networks is to take advantage of this spatial configuration. In these types of network, one part is composed of convolutional layers that are used to do feature extraction and the other part is a classical fully connected layer, as seen so far, used to perform the classification. The primary advantage of CCNs is that it improves the computation speed by reducing the number of inputs used in the fully connected layer by selecting the important features present in the image. Figure 2.2.5 shows how the classical CCN works.

CCNs are based on three basic ideas: local receptive fields, shared weights and pooling (Nielsen, 2015).

1. **Local receptive fields:** in MLPs, we represented the input layer as a vertical line of neurons. In CNNs, the inputs are usually images, which can be described by a width and height of pixels, for example 28×28 square of neurons. The values taken by the neuron correspond to the pixel's value. Instead of connecting every input to every hidden neuron, we make connections in a small, localized region of the image. This region is also called a *local receptive fields* or *kernel*. For example, if we take a kernel of dimension 5×5, we connect 25 inputs neurons to one hidden neuron, meaning the hidden neuron has 25 weights coming from the input neuron and one bias. The value of the hidden neuron is computed by convolution, see Section 2.2.6 for more details on the computation. Once we have computed the value of the hidden neuron, we slide the local receptive field to the right and compute the value for the new hidden neuron. The amount by which the filter slide is referred to as the *stride* (Udyavar, 2017). We generally take a stride equal to one. If we consider our example of an image with 28×28 pixel as input, and a local

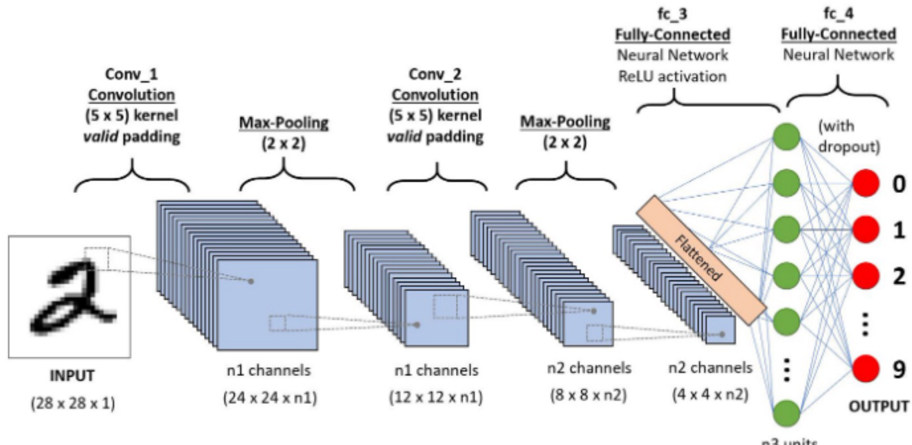


Figure 2.2.5: Classical architecture of a CCN used to classify handwritten digits (Saha, 2018).

receptive field of 5×5, we obtain 24×24 neurons in the hidden layer. This comes from the fact that we can only move the local receptive field by 23 neurons to the right (or down) before colliding with the border of the image. Figure 2.2.6 shows how this first step works.

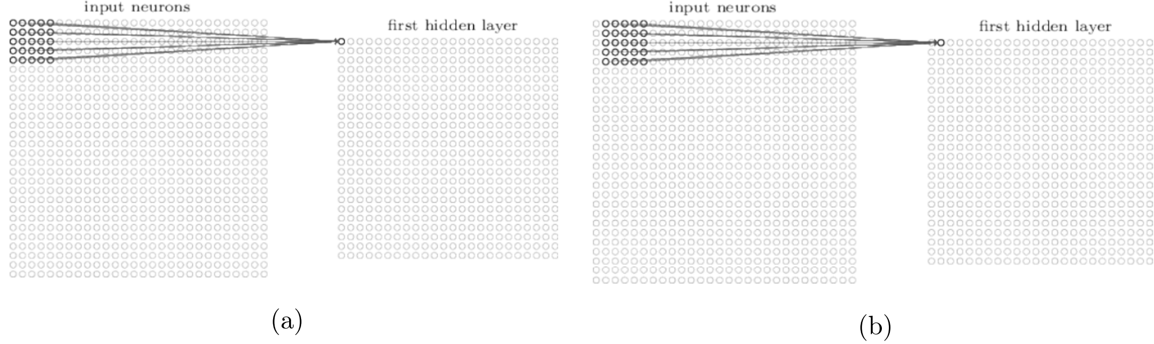


Figure 2.2.6: Computation of a CNN hidden layer. We slide the local receptive field by one neuron to the right between (a) and (b) (Nielsen, 2015)

2. **Shared weights and biases:** We have mentioned before that each hidden neuron in our example have 5×5 weights and one bias. These weights and biases are the same for all the hidden neurons computed during this step. We can say that the j, k-th hidden neuron has as output (Nielsen, 2015):

$$\varphi\left(b + \sum_{l=1}^{5} \sum_{m=1}^{5} w_{l,m} a_{j+l,k+m}\right)$$

where φ is an activation function, b the bias, $w_{l,m}$ an 5×5 array of shared weights and $a_{x,y}$ the input activation at position (x, y).
This sharing property means that all neurons in a hidden layer detect the same feature,

but at a different location on the image. A feature represents a kind of input pattern detected by the hidden neuron. Examples of features are edges, corners, circles, etc. The ability to detect the same feature all over the image allows CCNs to have a translation invariance of the image. "Translation invariance is the ability to ignore positional shifts, or translations, of the target in the image. A cat is still a cat regardless of whether it appears in the top half or the bottom half of the image" (Soni, 2019).
CNNs are not in their basic design invariant to rotation. The easiest way to approach an invariance in rotation is simply by augmenting the training data by rotating the image (Chidester et al., 2018). For the record, new formulations of convolutional layers like a spatial transform layer (Jaderberg et al., 2015) and a deformable convolutional layer (Dai et al., 2017) can aid in learning the rotation invariance.
The map from the input layer to the hidden layer is often called a *feature map*, defined by *shared weights*. These weights are also called *kernel* or *filter*.
Convolutional layers are typically composed of more than one feature map to achieve a proper image recognition. The number of feature maps of which the convolutional layer consists, is called the *filter depth* (Udyavar, 2017). If we take back the example considered so far, and we apply three different kernels of 5×5 shared weights, we end with a first hidden layer composed of 3 features map of 24×24 neurons each. The first layer has a volume equal to $3 \times 24 \times 24$. This first hidden layer is capable of detecting three different kind of features in the image. This is represented in Figure 2.2.7.

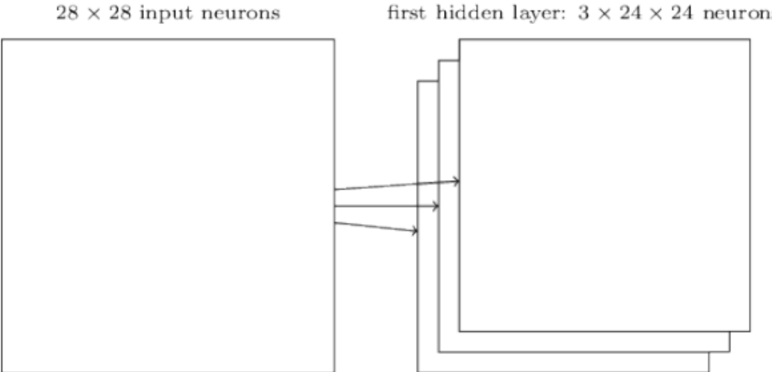


Figure 2.2.7: Representation of the first hidden layer composed of three different features map (Nielsen, 2015).

The dimensionality of the hidden layer, and therefore the number of neurons composing this layer, can be computed by the next formula. Thank to this formula, we can have an hint of the number of parameters composing the CNN, i.e. its complexity.

$$n_{neuron} = \frac{(W - F + 2P)}{S + 1} \tag{2.2.34}$$

where W is the number of neurons from the input layer, $F = height \times width \times depth$ is the volume of the filter, in our case the volume of the filter is $5 \times 5 \times 3 = 75$, S is the size of the stride, already defined before and finally, P is the padding.
The *padding* refers to the technique of adding a border of zero around the input layer to keep the same width and height across the layers inside the network. In our example,

if we add two borders of zero at the input layer, we end up with an input layer of size 32×32, which became and hidden layer of size 28×28 computed with a kernel of size 5×5.

The most significant advantage of the shared weights is that it drastically decrease the number of weights that need to be computed as compared to a fully connected layer. If we take back the example, we have 5×5 weights plus one bias for each feature map. If we imagine a first layer with 20 feature maps, we have $20 \times 26 = 520$ parameters to compute inside the CNN. If we design a fully connected neural network with $28 \times 28 = 784$ input neuron and 30 hidden neurons, we obtain 784×30 weights plus 30 bias, meaning 23,550 parameters to compute. The convolutional layer has in this case 40 times less parameters than the fully-connected layer.

3. **Pooling layers:** A convolutional neural network also contains pooling layers that are typically used after the convolutional layer. The idea of a pooling layer is to simplify the information received from the output of the convolutional layer.

   This type of layer takes each feature map output from the convolutional layer and computes a condensed feature map. This simplification is performed by taking a region of a certain size, for example 2×2 and applying a certain operation on the input values to output only one value. They are two types of commonly used operation, i.e. max or average. Figure 2.2.8 presents a pooling operation of size 2×2.

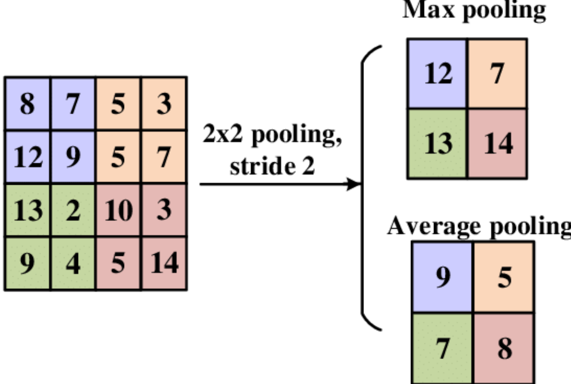


Figure 2.2.8: Example of pooling operation of size 2×2 (Yingge et al., 2020).

   By using max-pooling, the network knows if a given feature is discovered anywhere in a region of the image. We do not need to retain the exact position because once a feature is found, we merely need to know its rough location relative to other features. "A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers" (Nielsen, 2015). In our example, the pooling layer takes as an input the $3 \times 24 \times 24$ neurons from the convolutional layer and uses a 2×2 max pooling to produce a $3 \times 12 \times 12$ hidden feature layer.

Now that we have described all the steps used in a convolutional network, we can put them together and construct a complete convolutional neural network. The last step is to add a fully connected layer that outputs the values we require to classify the images. In the example, we classify hand-written digits from 0 to 9. The last output layer will therefore be

a fully connected layer with 10 neurons, each neuron being connected to all neurons present in the last step of the convolutional part (i.e. after pooling). Figure 2.2.9 shows the complete architecture of the convolutional neural network.

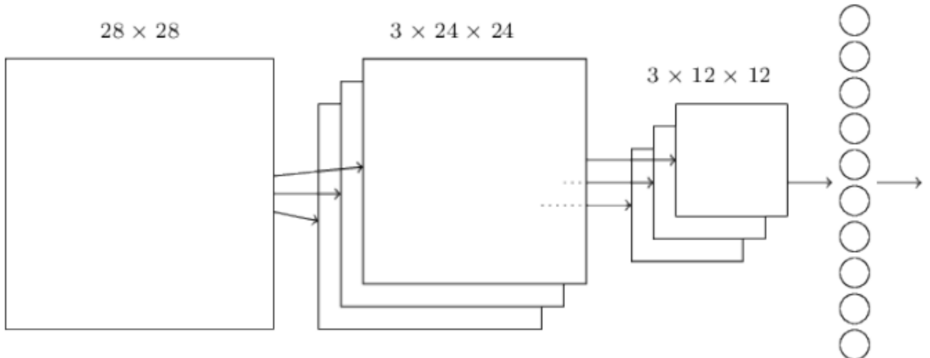


Figure 2.2.9: Complete architecture of a CNN taking as input a 28 × 28 pixel image, using three 5 × 5 kernels to detect features and a max-pooling layer applied to 2 × 2 regions, across each of the 3 feature maps (Nielsen, 2015).

Convolution Operator

The convolution operator smooths a function by using a weighting function $w(a)$ where a is an observation. If x and w are defined only on the integer t, the discrete convolution is defined by the second equation bellow:

$$s(t) = \int x(a)w(t-a)da = (x * w)(t)$$

$$s[t] = (x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t-a]$$

In convolutional network terminology, the function x refer to the *input*, the second argument $w(a)$ is known as the *kernel* and the output is referred as the *feature map*.
Convolution are often used on more than one axis, generally two if we refer to an image. The equation become, for an image I as input and a two-dimensional kernel K:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n]$$

Many libraries will implement this function by without flipping the kernel and call this operation the *cross-correlation*. We can also iterate over I or K due to the commutative property of the convolution:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n]$$

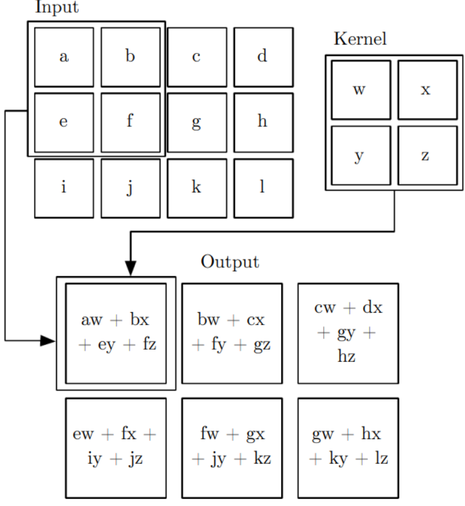


Figure 2.2.10: 2-D convolution example without kernel-flipping (Goodfellow et al., 2016).

Training of CNN

Convolutional neural networks are trained by backpropagation like classical MLPs. We have already explained in Section 2.2.6 how the forward propagation in CNNs works. Like MLPs, when we obtain the result, we compute the loss function and, thanks to this loss function, we move on to the gradient descent.

The formula used inside this backpropagation for CNNs are beyond the scope of this thesis. In fact, the majority of papers or books that explain CCNs do not cover the mathematical details behind the backpropagation. Moreover, CNNs are already implemented in the classical packages used to perform data sciences task like TensorFlow (Abadi et al., 2015) or PyTorch (Paszke et al., 2019).

However, if one is curious about the mathematics and methodology behind backpropagation for convolutional neural networks, we highly recommend reading the article write by Solai (2018) which provides a clear explanation based on the more detailed explanations of Jefkine (2016).

2.3 Deep Learning

This section describes the algorithms that used during this thesis. As we will describe in the succeeding chapter dedicated to the methodology, we work on Atari 2600 game, meaning we typically learn from pixels. In this section, we present algorithms that use deep neural networks to achieve their objective of learning. These algorithms globally follow the process described in the Section 2.1, i.e. an agent performs an action that impacts its environment, the environment transitions to a new state and the agent receives a reward for his action. The biggest change resides in the way we memorize the policy applied by the agent. In the Section 2.1.3 based on Q-learning algorithm, the policy was stored inside a Q-table. This solution is unworkable if there are too many possible different states and actions. Deep learning algorithms correct this problem by mapping the inputs states to (action, Q-value) pair inside a neural network.

2.3.1 Deep Q-networks

the Deep Q-network (DQN) algorithm is introduced by Mnih et al. (2013) and Mnih et al. (2015) and was able to obtain a strong performance for a variety of Atari games by learning directly from the pixels. In this algorithm, they use a CNN that receives the pixel of the game as input and produces a Q-value for each possible action performed by the agent. Using a nonlinear function approximator for the action-value function presents a difficulty in reinforcement learning as it is known to be unstable or diverge (Tsitsiklis and Van Roy, 1996). This instability is produced by several causes: (i) the correlation present in the sequence of observations, (ii) minor updates to the Q-function may change the policy in huge way, changing the data distribution, and (iii) the correlations between the action-values and the target values.

To address these changes, Mnih et al. (2015) use an experience replay mechanism that randomizes over the data in order to remove the correlation present in the observation sequences. In order to reduce the correlation with the target values, they use two different neural networks containing the same architecture, but with different weights. Every C step, they update the *target network* with the weights of the *main network* or *Q-network*. C is an arbitrary number that can change depending on the problem. Ohnishi et al. (2019) provides a comparison of different values of C used for different variation of DQN. A typical value for C is between 100 and 1000.

The algorithm works like this (Mnih et al., 2015):

- Use a deep convolutional neural network to parameterize an approximate value function $Q(s, a; \theta_i)$ where θ_i are the weights of the Q-network at iteration i, state s and chosen action a.
- The experience replay is achieved by storing the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t in a dataset $D = e_1, ..., e_t$.
- Apply Q-learning updates on samples (known as *mini-batches*) of experiences selected from the pool of stored samples D. The samples are drawn uniformly at random following $U(D) \sim (s, a, r, s')$.
- The Q-learning update at iteration i is achieved with the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right] \tag{2.3.1}$$

  where γ is the discount factor, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the parameters of the target network used to compute the target at iteration i. The parameters of the target network are updated every C steps and keep constant between them.

  This equation can be considered as the mean squared error of the Bellman equation seen in Section 2.1 where the optimal target values $r + \gamma \max_{a'} Q^*(s', a')$ are substituted with approximate target values $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$.

  The differentiation of the loss function with respect to the weights is:

$$\frac{\partial L(\theta_i)}{\theta_i} = \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right) + \theta_i Q(s, a; \theta_i) \right]$$

The full expectation of the gradient is often not calculated. Instead, the loss function is optimized by stochastic gradient descent using the Q-learning algorithm (Watkins, 1989). The weights are updated at every time step, which replaces the expectation by a single sample. θ_i^- is also set equal to θ_{i-1}.

Training algorithm for deep Q-networks is presented in the algorithm 4 (Mnih et al., 2015). We can notice that the agent follows an $\epsilon-$greedy policy to select and execute the actions. With this strategy, the agent follows the greedy strategy with probability $1-\epsilon$ and selects a random action with probability ϵ. In their experiments, Mnih et al. (2015) linearly decreased ϵ from 1 to 0.1 during the first million frames, after which they fixed it at 0.1. More specifically, they have trained the model on 50 million frames and used a replay memory of the 1 million most recent frames. Finally, they have preprocessed the frames to reduce the number of inputs. We will go into the detail, but this preprocessing step is presented in the algorithm 4 by the function ϕ.

Algorithm 4 Training a deep Q-learning model with experience replay

```
Initialize replay memory D with capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q̂ with weights θ⁻ = θ
for episode=1,M do:
    Initialize sequence s_1 = x_1 and preprocessed sequence φ_1 = φ(s_1)
    for t=1,T do:
        if x ~ Unif(0,1) < ε then:
            select random action a_t
        else:
            select a_t = argmax_a Q(φ(s_t), a; θ)
        Execute a_t and get reward r_t, image x_{t+1}
        Set s_{t+1} = s_t, a_t, x_{t+1} and preprocess φ_{t+1} = φ(s_{t+1})
        Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
        Sample random minibatch of transitions (φ_t, a_t, r_t, φ_{t+1}) from D
        if episode terminates at step j + 1 then:
            Set y_j = r_j
        else:
            Set y_j = r + γ max_{a'} Q̂(φ_{j+1}, a'; θ_i⁻)
        Compute loss function (y − Q(φ_j, a_j; θ_i))^2
        Perform gradient descent with respect to parameters θ
        Every C steps reset Q̂ = Q
```

2.3.2 Proximal Policy Optimization Algorithms

This is a policy gradient method developed by Schulman et al. (2017) that samples data through interaction with the environment and optimizes a function using stochastic gradient ascent. This method is an improvement of the trust region policy optimization (TRPO) created by Schulman et al. (2015). Despite being efficient and reliable, TRPO is complicated and not compatible with architectures that include noise or parameter sharing between the policy and the value function.

Policy proximal optimization (PPO) achieves the performance of TRPO while using only first

order optimization. The objective function contains clipped probability ratios that produce a pessimistic estimate of the performance of the policy. This clipping is described later. This section is principally based on the work of Schulman et al. (2017). Arxiv Insights (2018) provides also clear explanations of the algorithm in a condensed video.

Policy Gradient Methods

"Policy gradient methods are a type of reinforcement learning techniques that rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent" (Peters and Bagnell, 2010). An estimator of the policy gradient is calculated then plugged into a stochastic gradient ascent algorithm [3]. The gradient estimator is commonly defined as:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. This advantage function is described later in this section. This estimator $\hat{g}$ can be obtained by differentiating the loss function:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

Traditional methods perform multiple steps of optimization on this loss L^{PG} using the same trajectory. This generally leads to destructively large policy updates that make the model unstable.

Trust Region Methods

To avoid these large policy updates, TRPO (Schulman et al., 2015) maximizes an objective function subject to a constraint on the size of the policy update. Mathematically [4],

$$\begin{aligned} \max_\theta \quad & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \\ \text{subject to} \quad & \hat{\mathbb{E}}_t [KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \ \leq \delta \end{aligned}$$

where θ_{old} is the vector of policy parameters before the update. To approximately solve these equations, one must use the conjugate gradient algorithm after making a linear approximation to the objective function and a quadratic approximation to the constraint.

Instead of using this constraint, the theory of TRPO suggest to use a penalty. The unconstrained optimization problem then becomes:

$$\max_\theta \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$$

where β is a coefficient to fix. TRPO use the constrained problem defined above because choosing a single value for β that performs well across different problems is complicated.

[3] A gradient ascent algorithm is globally the same as a gradient descent method explained previously, except that then rather of minimizing a function, we maximize it (De Luca, 2020).

[4] Inside these equations, KL refers to the Kullback-Leibler divergence is a measure of how one probability distribution P is different from a second (Kullback and Leibler, 1951).

Clipped Surrogate Objective

In TRPO, the "surrogate" objectives is

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{2.3.2}$$

where we have set $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. CPI refers to conservative policy iteration (Kakade and Langford, 2002). As explained before, without constrains, maximizing L^{CPI} leads to large policy updates. The idea of clipping [5] the objective is to penalize policy changes that move $r_t(\theta)$ away from 1. The objective then becomes:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min \left(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2.3.3}$$

where ϵ is a hyperparameter generally fixed at 0.2. This function takes the minimum of the L^{CPI} defined in equation 2.3.2 and the modified surrogate objective where the probability ratio is clipped in the interval $[1-\epsilon, 1+\epsilon]$. Taking the minimum assures a lower bound, i.e. a pessimistic bound for the objective function. The idea behind this function will be described in the following section.

PPO Algorithm

The objective of this algorithm is to maximize a loss function partially constructed from the equation 2.3.3. This maximization is achieved through stochastic gradient ascent.

Mnih et al. (2016) have popularized a style of policy gradient implementation that runs the policy for T time steps and uses the collected data to perform the update. The PPO algorithm is implemented in the same way. This type of implementation requires an advantage estimator $\hat{A}_t$ that is defined as:

$$\hat{A}_t = r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) - V(s_t) \tag{2.3.4}$$

This advantage estimator takes the difference of two terms. The first is the discounted reward that are globally $\sum_{k=0}^{\infty} \gamma^k r_{t+k}$ where γ is the discounted parameter. This first term is computed by running the policy on the T time steps. The second is the baseline estimate $V(s_t)$. This baseline estimate is the value function. It provides an estimate of the discounted return that we expect to get at the state s_t. This function is updated during training.
The advantage function $\hat{A}_t$ allows us to know if we would have obtained a better result by performing a given action compared to what we had estimated before. If $\hat{A}_t$ is positive, we obtain a positive gradient, meaning we increase the probabilities of selecting this sequence of actions in the future.

In the PPO algorithm, the loss function is clipped to avoid big updates of the parameters that could destroy the policy. Figure 2.3.1 presents how this clipping technique works in the algorithm. We can differentiate two cases.

- If $\hat{A}_t$ is **positive**, the discounted returns obtained are bigger than expected by the value function $V(s_t)$. In this case, we want to encourage these actions, meaning that the loss

[5] Clipping is defined by (Harris et al., 2020) as "given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1".

function should be positive. In order to avoid having too much change in the weights of the network, the value of the loss function is clipped if $r > 1 + \epsilon$, i.e. if the probability of selecting the action a_t in state s_t for the actual policy is much higher than in the old policy. We do not want it to diverge too much from the old policy to avoid instability.

- In the **negative** case, the discounted returns are smaller than expected. As we do not want to repeat these actions, the loss function should be negative. If r_t is smaller than $1 - \epsilon$, the curve flattens to avoid reducing these action probabilities to zero. If r_t is big, the selected action is much more probable than before, but this will lead to a negative advantage function. In this case, we want to undo the last gradient step thanks to a negative L^{CLIP} that produces a negative gradient. This negative gradient decreases the probabilities of selecting this action in this case. Furthermore, this is the only region where the unclipped part of the objective function 2.3.3 has a lower value than the clipped part, which is returned by the minimization operator.

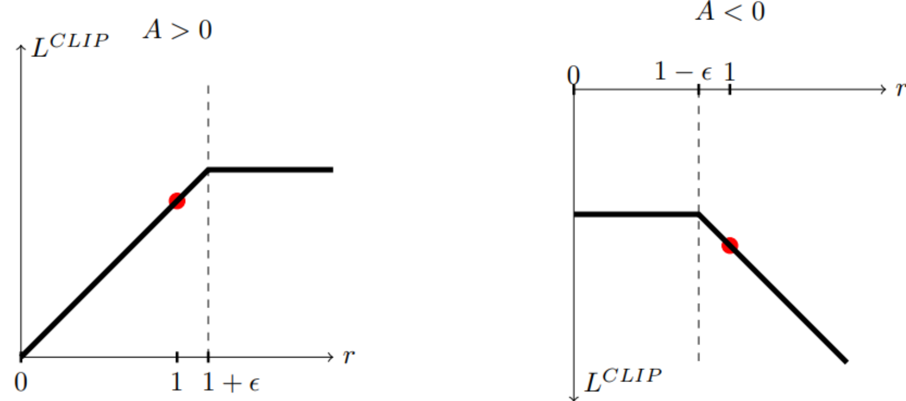


Figure 2.3.1: Surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right) (Schulman et al., 2017).

The PPO algorithm can be viewed as an Actor-Critic method (Konda and Tsitsiklis, 1999) where the Actor corresponds to the policy used to choose the actions performed by the agent. The Critic is represented by the value function $V(s)$. The neural network architecture used for this algorithm shares parameters between the policy (the Actor) and the value function (the Critic). Therefore, the loss function should combine the policy surrogate presented in the equation 2.3.3 and a value function error term. An entropy bonus is added to both terms to ensure sufficient exploration, as suggested by Williams (1992) and Mnih et al. (2016). The loss function maximized at each iteration becomes:

$$L_t(\theta) = \hat{\mathbb{E}}_t \left[L^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.3.5}$$

where c_1, c_2 are coefficients used to balance the two terms, S represents the entropy bonus and L^{VF} is the squared-error loss $\left[V_\theta(s_t) - V_t^{targ} \right]^2$.

The pseudo algorithm 5 presents how the PPO algorithm works. At each iteration, each of the N actors (that act in parallel) collects T time steps of data. The surrogate loss described in the equation 2.3.3 is then constructed with these NT time steps of data, and optimized with mini-batch SGD or Adam, for K epochs.

Algorithm 5 PPO

for iteration=1,2,... **do**:
  for actor=1,2,...,N **do**:
    Run policy $\pi_{\theta_{old}}$ in environment for T time steps
    Compute advantage estimates $\hat{A}_1$,..., $\hat{A}_T$
  Optimize surrogate L wrt θ, with K epochs and mini-batch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$

Chapter 3

Methodology

In this chapter we present the methodology used to achieve the results showed in the following chapter. The goal of this thesis is to discover deep reinforcement learning. To achieve that, we compare different learning models to different situations and analyse which model perform the best according to the situation.

Before training an agent, we need to define the environment inside which it will be trained. We use two different games that primarily differ by their complexity. The first game is called *Mountain Car* and the second is the well know *Super Mario Bros*. Both these games are implemented in a *Gym environment* developed by OpenAI.

3.1 Gym Environment

Gym is a Python library developed by OpenAI that provides diverse environments that are easy to set up. The primary goal is to standardize the way environments are defined in AI research in order to be able to replicate results and provide a common benchmark to compare algorithms (Brockman et al., 2016; Gershgorn, 2016). The Gym documentation, where most of the explanations provided inside this section come from, is open source and hosted by Farama Foundation (2022) on git.

A Gym environment follows the classical reinforcement learning pattern described in the previous chapter: an agent performs an action in the environment, then we collect the observation corresponding to the new state of the environment and the reward.
In Gym, several key functions are used to perform this cycle:

- ***step(action)*** where *action* represents the action performed by the agent. This function returns an object *observation*, which contains the new state of the environment, a *reward*, the amount of reward returned by conducting the action, a boolean value *done*, that signals if the episode has ended, and finally, a dictionary *info* that contains different information linked to the game.

- ***reset()*** is a function that resets the environment to an initial state and returns the initial observation

- ***render()*** allows to continuously render the current state of the game for human comprehension. It shows the game in action.

Each game implemented in Gym has two attributes named *action_space* and *observation_space* that describe the possible set of actions taken by the agent and the possible domain of observations returned by the game. The possible types of spaces available in Gym are:

- **Box**: a n-dimensional continuous space.
- **Discrete**: a discrete space that generally takes [0,1,...,n-1] as possible values for the observations or actions.
- **Dict**: a dictionary of simple spaces.
- **Tuple**: a tuple of simple spaces.
- **MultiBinary**: a n-shape binary space.
- **MultiDiscrete**: a series of *discrete action spaces* with a different number of actions in each element.

Wrappers are the last convenient implementation in Gym. These allow a user to modify an existing environment without altering the underlying code. They make the environments more modular such that they can be chained to combine their effects. There are three principal types of wrapper:

- **ActionWrapper** transforms actions before applying them to the base environment.
- **ObservationWrapper** transforms observations that are returned by the base environment.
- **RewardWrapper** transforms the rewards that are returned by the base environment.

One can use these classes of wrappers to create his/her own wrapper that inherits from the classes described above. Most of the time, common wrappers are already implemented by Gym, as the ones used in the following sections.

3.1.1 Mountain Car

Mountain Car is the first environment used to compare the RL algorithms. This game is part of the *Classical Control environment* implemented by Gym. These environments are considered as the simplest to solve due to their limited number of actions and their low complexity. Mountain Car is a deterministic MDP that was first described by Moore (1990). It consists of a car placed at a bottom of a sinusoidal valley. The only possible actions are to accelerate in one of the two directions, or do nothing. The goal is to reach the top of the right mountain. However, the car can not achieve that by only accelerate to the right. It has to create a momentum by alternate between the direction to accelerate at the right moment. Figure 3.1.1 presents a starting state of the game.

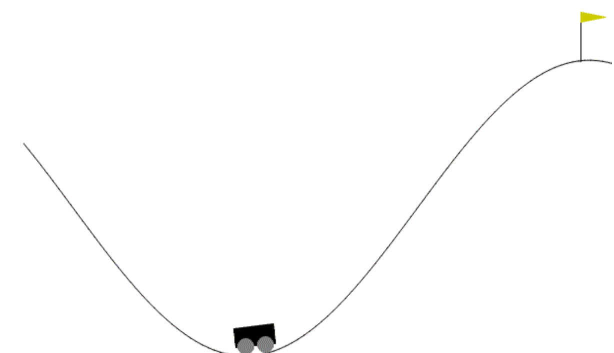


Figure 3.1.1: Frame of the Mountain Car environment.

The environment is constructed by running the function *gym.make("MountainCar-v0")* and is constrained by the following information.

- **Observation space:** a ndarray with shape (2,) that corresponds to:

Index	Description	Domain	
		Low	High
0	Position of the car along the x-axis	-1.2	0.6
1	Velocity of the car	-0.07	0.07

Table 3.1.1: Observation space for Mountain Car problem.

- **Action space:** 3 discrete deterministic actions described by:

Value	Observation
0	Accelerate to the left
1	Don't accelerate
2	Accelerate to the right

Table 3.1.2: Action space for Mountain Car problem.

- **Transition dynamics:** given an action, the velocity and position are computed from

$$velocity_{t+1} = velocity_t + (action - 1) \times force - cos(3 \times position_t) \times gravity$$
$$position_{t+1} = position_t + velocity_{t+1}$$

  where $force = 0.001$ and $gravity = 0.0025$. The collisions at either end are inelastic, and the velocity is set to zero if the car hit the left wall. The position and velocity are clipped by the values presented inside Table 3.1.1.

- **Reward:** the goal is to reach the flag on the top right as quickly as possible. To represent that, the agent is penalized with a reward equal to -1 at each time step.

- **Starting state:** the position is assigned following a uniform distribution between [-0.6,-0.4] and the velocity is set to zero.

- **Episode termination:** the episode end if either the position of the car is greater than or equal to 0.5, meaning the car has reached the flag, or if the length of the episode is 200, meaning the car did not reach the flag. In this last case, the cumulative reward is equal to -200.
- **Solving condition:** according to OpenAI (2020) this environment is considered solved if we reach an average reward of -110.0 over 100 consecutive trials[1].

3.1.2 Super Mario Bros

The second game handled during this thesis is the well known *Super Mario Bros*. We use the OpenAI Gym environment implemented by Kauten (2018b) that provides different levels of Super Mario Bros. and Super Mario Bros. 2 from the Nintendo Entertainment System (NES) using the nes-py emulator (Kauten, 2018a).

The goal is to reach the end of a level without dying. There are monsters that can appear or holes in the ground. If Mario does not reach the end of the level by a certain time, the game is also lost.

Compared to the previous game, we do not enter the physical details of the game, like the velocity of Mario etc.. Instead, we focus more on the environment and action spaces, as well as the reward.

- **Environments:** in this implementation of Mario, there are eight worlds, each being composed of four stages. We possess three lives to complete the 32 stages of the game. However, we will see that completing the first stage is already complicated.
  Six different versions of Super Mario Bros are implemented. The one we employ represents the standard version that is called is *"SuperMarioBros-v0"*. The other versions and the way the game is rendered can be found in Table A.0.2 located in the Appendix A.
  The observation space of the "standard" version has a shape (240, 256, 3) where the first two arguments correspond to the height and width in pixels of the screen. The third correspond to color. We have one value of pixels for the Red color, one for the Green and one for the Blue (RGB code). The pixel's values can range from 0 to 255. A frame of the game is presented in Figure 3.1.2.

Figure 3.1.2: Frame of Super Mario Bros.

[1] There are no clear explanations about this threshold, but it is the commonly admitted value when we look at the OpenAI gym leaderboard.

- **Actions:** by default, the environment uses the full NES action space of 256 discrete actions, but we can reduce it by using one of the following action lists presented in Table 3.1.3 and use it in a wrapper named **JoypadSpace** imported from *nes_py.wrappers*.

Value	RIGHT_ONLY	SIMPLE_MOVEMENT	COMPLEX_MOVEMENT
0	Noop[1]	Noop[1]	Noop[1]
1	right	right	right
2	right + A	right + A	right + A
3	right + B	right + B	right + B
4	right + A + B	right + A + B	right + A + B
5		A	A
6		left	left
7			left + A
8			left + B
9			left + A + B
10			down
11			up

[1] "Noop" stand for do nothing.

Table 3.1.3: Possible list of actions.

- **Reward:** is designed to move the most on the right, as fast as possible and without dying. To achieve that, three variables compose the function:

  1. **V** which represents the difference of the agent's x position between two states, i.e. $V = x_1 - x_0$ where x_0 is the position of the agent on the x-axis before making a step, and x_1 is the position after the step.
  2. **C** is the difference in the game clock between frames. This penalty is in place so that the agent finishes the level as quickly as possible.
  3. **D** is a death penalty that penalizes the agent for dying in a state. It takes as value zero if the agent is alive. If the agent dies, it takes -15.

  The total reward obtains at each step is the addition of the different parts, i.e. $r = V + C + D$ and is clipped in the range [-15,15].

- **Info:** this is a dictionary returned by the *step* function that provides some information about the current state of the game. We do not use this dictionary in this thesis, but its composition can be found in Table A.0.1 of the Appendix A.

3.2 Mountain Car Solution

To solve this environment, we have implemented two algorithms. The first is the basic *Q-learning* described in the Section 2.1.3. For the second, we have used a more sophisticated known as *Deep Q-network* and described in the Section 2.3.1.

3.2.1 Preprocessing

Before applying a Q-learning algorithm to the Mountain Car environment, we first need to discretize the observations send by the game. Indeed, as explained in the Section 2.1.3, the Q-leaning algorithm use a Q-table that contains Q-values for all possible states and actions. The Mountain Car environment observation space are coded in float32, meaning the number have 8 decimal digits, which provide 180,000,000 possibilities for the position of the car along the x-axis and 14,000,000 possible velocities. Therefore, the size of the Q-table should be $180,000,000 \times 14,000,000 \times 3 = 7.56 \times 10^{15}$ which is obviously too large for classical computers (the $\times 3$ comes from the number of possible actions). To overcome this problem, we discretize the observation values by using the following equations:

$$size_{interval} = \frac{observation_{high} - observation_{low}}{size_{wanted}}$$
$$state_{discrete} = \frac{state_{continuous} - observation_{low}}{size_{interval}}$$

where $size_{wanted}$ represents an array corresponding to the dimensions wanted for the Q-table, and $observation_{high}$ / $observation_{low}$ correspond to the maximal and minimal values returned by the observation space (see Table 3.1.1 to get the values).

3.2.2 Q-learning Implementation

Our implementation globally follows the pseudo code 2 presented in the Section 2.1.3. The code is also inspired from Hayes (2019). The pseudo code of our implementation can be found below.

Algorithm 6 Implementation of Q-learning algorithm

Set training length M
Set $discrete_array = [x, y]$ where x,y are defined values
Initialize $Qtable$ of shape $(x, y, 3)$
Initialize $env = gym.make("MountainCar - v0")$
Initialize $epsilon$
for $episode = [1, M]$ **do**:
  $state \leftarrow discretize(env.reset())$
  $done \leftarrow False$
  while not True **do**:
    if $unif(0, 1) < 1 - epsilon$ **then**
      $action \leftarrow argmax(Qtable[state])$
    else
      $action \leftarrow random(0, 1, 2)$
    $newstate, reward, done, info \leftarrow env.step(action)$
    $newstateD \leftarrow discretize(newstate)$
    if $done$ and $newstateD[0] >= 0.5$ **then**
      $Qtable[newstateD[0], newstateD[1], action] \leftarrow reward$
    else
      Compute Qvalue with equation 2.1.6
    $state \leftarrow newstateD$
  if decay epsilon **then** ▷ If want to decrease the *epsilon* value
    $epsilon- =$ decay value

3.2.3 DQN Implementation

For the Deep Q-network algorithm, we have used the function already implemented by Stable Baselines3, which is a set of reliable implementations of reinforcement learning algorithms in PyTorch developed by Raffin et al. (2021). They provide the vanilla implementation of Deep Q-Learning based on the paper of Mnih et al. (2013).

This deep reinforcement learning algorithm can accept three different policies as input depending on the environment on which we are training.

- **MlPolicy** which is used when we get discrete values as input. This policy is used to solve the Moutain Car problem. In this case, the algorithm uses an MLP with default architecture: one input layer where the number of neurons depends on the number of inputs, two hidden layers with 64 neurons each, and one output layer with a number of neurons corresponding to the number of possible actions. In the Mountain Car environment, it is an MLP with an architecture corresponding to $(2 \times 64 \times 64 \times 3)$.

- **CnnPolicy** corresponds to a policy using image as input. This policy is unused for Mountain Car environment, but we will employ it to solve the Super Mario Bros. environment. This policy uses a CNN with a default architecture corresponding to the one used by Mnih et al. (2015). This architecture is described in the table below, where *nchannel* corresponds to the third dimension of the image, i.e. equal 3 if we use RGB image, equal 1 if the images are gray scaled.

Id layer	1	2	3
Type of layer	2D convolutional	2D convolutional	2D convolutional
Nb feature map input	*nchannel*	32	64
Nb feature map output	32	64	64
Kernel size	8	4	3
Stride	4	2	1
Padding	0	0	0
Activation function	ReLu	ReLu	ReLu

Table 3.2.1: Default architecture for the CNN implemented in Stable Baselines3.

The CNN is then flatten and linked to a classical MLP with one hidden layer composed of 512 neurons and one output layer corresponding to the possible actions in the environment.

- **MultiInputPolicy** is the last policy implemented. It is not developed because we do not use it in this master thesis.

Finally, the parameters accepted by the *DQN* function are presented with their default values in Table A.0.3 showed in the Appendix A.

3.3 Super Mario Bros. Solution

To solve Super Mario Bros., we have used two different algorithms, the Proximal Policy Optimization described in Section 2.3.2 and the Deep Q-Network previously used to solve the Mountain Car problem.

3.3.1 Preprocessing

As for the Mountain Car problem, we need to do some preprocessing before trying to solve this environment. It is even mandatory for environments with images as inputs. In Gym most of the preprocessing is achieved thanks to wrappers (see Section 3.1). In RL, the environment is generally modified as follows:

1. **Grayscales** images thanks to the *GrayScaleObservation* wrapper from Gym. It allows reducing the three dimensions of the image due to the colors to only one gray dimension.

2. **Resize** observations with the *ResizeObservation* wrapper that resize the dimension of the image to a defined dimension (typically 84×84 pixel).

3. **Stacking frame** stacks a defined number of frames (mainly 4). The idea is to provide the intuition of movement to the agent. If the agent receive just one frame, it is a frozen image at a certain time, the agent can not know in which direction Mario is going. If we stack 4 frames, we can provide the idea of movement. This stacking is achieved thanks to the *VecFrameStack* from Stable Baseline.

Another commonly used wrapper from Gym is the *atari_processing* that follows the guidelines provided by Machado et al. (2018). This wrapper will: i) skip certain number of frames (4 defaults), ii) Max-pooling over the most recent two observations from the frame skips, iii)

resize to a 84×84 square of pixels, iv) grayscale the observations, and v) scale the pixel values between zero and one.

3.3.2 Solving Mario

The algorithms used to solve the Mario environment are the Proximal Policy Optimization (PPO) and the Deep Q-Network (DNQ). The implementation of the last one is already defined in the previous Section 3.2.3. We have used the same implementation of the algorithm used to solve the Mountain Car environment. However, there are small changes: i) the hyperparameters and, ii) the policy used. In the Mario environment, we receive as inputs images preprocessed. Therefore, we can not use the *MlPolicy* previously described. We change it to use the *CnnPolicy*, explained in the previous section.

For the PPO implementation, we have used the algorithm present in the Stable Baseline3 package, which is based on the original work of Schulman et al. (2017). The detail of the parameters accepted by the function and their default values can be found in Table A.0.4 in the Appendix A. Some modifications of the original code were made. If one wants to replicate the exact implementation made in Stable Baseline3, 37 details are presented in a blog by Huang et al. (2021).

The methodology applied to solve the problem is the same for the two algorithms.

1. **Preprocessing** is performed on the environment. The preprocessing used is detailed in the next chapter.

2. **Model creation:** a model is then specified thanks to the *PPO* or *DQN* function from Stable Baseline3. Some hyperparameters can be specified.

3. **Training** is done through the **learn** function applied to the model previously constructed. This training is done during a certain number of time steps, which is analyzed in the following chapter.

4. **Evaluation** is performed at the end of the training.

Ultimately, the computing is done on a GPU (Graphic Processing Unit) using CUDA (Computed Unified Device Architecture) that is developed by Nvidia (Nvidia Developer, 2022). CUDA allows speeding up computing by employing the thousands of GPU cores in parallel. The computer on which training is made is composed by an Intel Core i7 920 with 2.66GHz as CPU, an Nvidia Geforce GTX 1050 Ti as GPU and 12Go of RAM.

Chapter 4

Results

This chapter presents the results obtained with the methodology presented in the chapter 3.
First, we compare the performances of the Q-learning models with the DQN algorithm for the Mountain Car environment. For each algorithm, we have trained various models with different parameters to identify which one perform the best.
Subsequently, we analyse the performances of DQN and PPO models trained for the Super Mario Bros environment.

4.1 Mountain Car Environment

4.1.1 Q-learning

The results are obtained using the algorithm 6 described in the Section 3.2.2. We have constructed eight models with different values as parameters to try to obtain the best model. The parameters and their values can be found in Table 4.1.1.

Model	**Learning rate**	**Nb episodes training**	**Discount**	**Epsilon start**	**Epsilon end**	**Episode end decay**	**Q shape**	**Type decay**
1	0.1	10000	0.9	1	0.01	10000	[20,20]	Exponential
2	0.1	10000	0.9	1	0.01	10000	[20,20]	Linear
3	0.2	10000	0.9	1	0.01	8000	[20,20]	Exponential
4	0.1	10000	0.99	1	0.1	8000	[20,20]	Exponential
5	0.1	10000	0.9	0.5	0.01	8000	[20,20]	Linear
6	0.1	5000	0.9	1	0.01	5000	[20,20]	Exponential
7	0.05	40000	0.9	1	0.01	34000	[40,40]	Exponential
8	0.05	40000	0.99	1	0.01	34000	[40,40]	Exponential

Table 4.1.1: Model's parameters used for Q-learning.

Inside this table, the parameter *episode end decay* represents the episode at which we want the epsilon to reach its end value. After this threshold, the epsilon stays constant with a value equivalent to its ending value. The *Q shape* describes the size of the Q-table obtained by the discretization presented in the Section 3.2.1. Finally, the *type decay* depicts how we reduce the epsilon's value, i.e. linearly or exponentially.

After completing the training of each model, we have employed them to obtain the average reward over 100 consecutive episodes. This metric is used to find which model performs the best. The results can be found in Figure 4.1.1 bellow, where the triangles represent the mean

rewards. Table 4.1.2 shows the mean and standard deviation of each model. The evolution of the rewards and epsilon over the training episodes can be found in the Appendix B.1.
As a reminder, the negative cumulative reward come from the way we reward the agent, -1 at each step performed until reaching the flag or performing 200 steps, which end the episode (see Section 3.1.1).

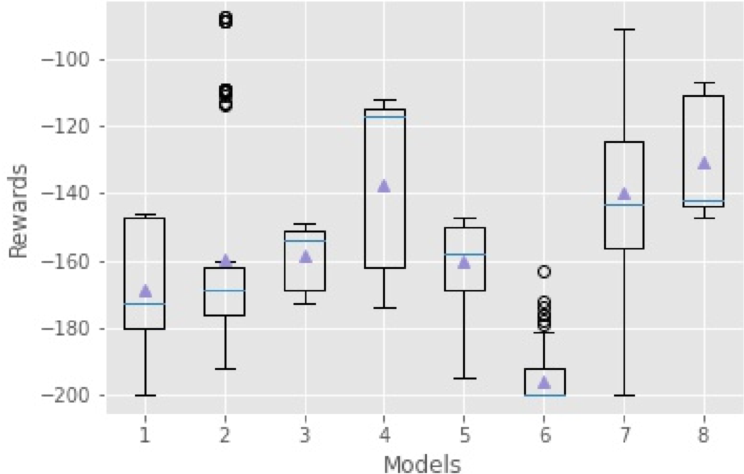


Figure 4.1.1: Distribution of the rewards obtained over 100 consecutive episodes of the Mountain Car environment by the models trained with Q-learning algorithm.

	Model 1	**Model 2**	**Model 3**	**Model 4**	**Model 5**	**Model 6**	**Model 7**	**Model 8**
mean	-168.81	-159.71	-158.75	-137.34	-160.33	-195.73	-139.86	-130.84
std	18.49	29.51	9.02	24.67	12.38	7.68	23.9	15.64

Table 4.1.2: Mean and standard deviation obtained over 100 consecutive episodes in the Mountain Car environment by the models trained with Q-learning algorithm.

The model 8 is the best model on average. It was constructed by taking the best parameter's value of the other models. For example, we see that the model 4 performs better than the others, which seems to be due to a higher discount value. By comparing the model 1 and 2, we see that decaying linearly the epsilon value instead of exponentially lead to higher rewards. However, after some tests, the model 8 performs better with exponential value. Finally, taking a (40,40) dimension for the Q-table, increasing the number of training episodes and decreasing the learning rate also seem to improve the average reward.

4.1.2 DQN

For the DQN results, we have also compared different models to see which parameters are the best. Finding good parameters for this type of algorithm can be difficult. However, Raffin (2020) has developed a training framework using Stable Baselines3 (Raffin et al., 2021) that provides scripts for training, evaluating agents and tuning parameters. It also provides a collection of already tuned hyperparameters for common environments, like Mountain Car.

Table 4.1.3 presents the different hyperparameter values used by the models. The values inside the model 1 correspond to the tuned value present in the Zoo framework. The model 2 corresponds to the default values used by the *DQN* function from Stable Baseline3. For the model 3 we have increased the learning rate. Ultimately, the model 4 represents a small improvement of the model 1 with a lower learning rate and higher training time (time steps training).

	Model 1	**Model 2**	**Model 3**	**Model 4**
Learning rate	0.004	0.0001	0.04	0.001
Batch size	128	32	64	128
Buffer size	10000	1000000	5000	10000
Learning start	1000	50000	1000	1000
Gamma	0.98	0.99	0.98	0.98
Target update interval	600	10000	600	600
Train freq	16	4	16	16
Gradient step	8	1	8	8
Exploration fraction	0.2	0.1	0.2	0.2
Exploration final eps	0.07	0.05	0.07	0.07
Policy kwargs	net_arch=[256,256]	None	net_arch=[256,256]	net_arch=[256,256]
Time steps training	200,000	200,000	200,000	400,000

Table 4.1.3: Parameters' values for each model trained by the DQN algorithm.

The description of each parameter can be considered in Table A.0.3 inside the Appendix A. The "net_arch=[256,256]" added to the *Policy kwargs* define the MLP used, i.e. two hidden layers with 256 neurons each.

After training all the models, we have simulated 100 episodes and collected the rewards. The boxplots of the rewards by models can be found in Figure 4.1.2. Table 4.1.4 shows the mean and standard error of the rewards by models.

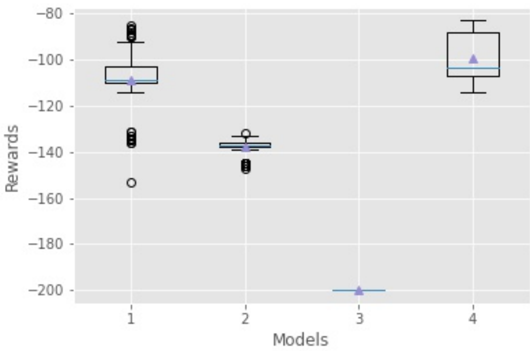


Figure 4.1.2: Distribution of the rewards obtained over 100 consecutive episodes of the Mountain Car environment by the models trained with DQN algorithm.

We clearly see that correctly tuned the hyperparameters is important if we want to obtain good results. A too large learning rate, like the model 3, leads to an unstable model that can not solve the environment. Despite already be tuned by Raffin (2020) in the Zoo framework, we see we can still improve the results by changing a bit the hyperparameters, especially by increasing the training time and decreasing a bit the learning rate.

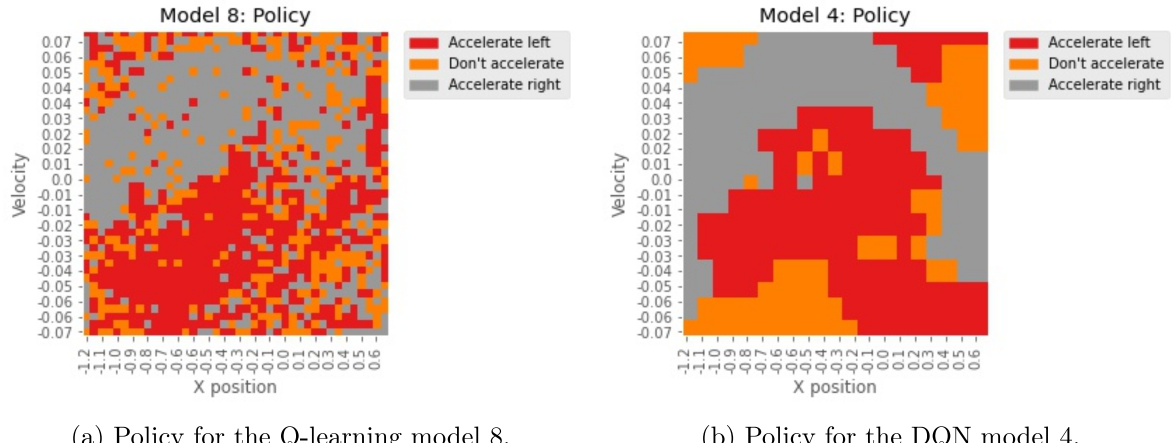
	Model 1	Model 2	Model 3	Model 4
mean	-108.55	-137.86	-200	-99.33
std	13.52	3.59	0	9.94

Table 4.1.4: Mean and standard deviation obtained over 100 consecutive episodes in the Mountain Car environment by the models trained with DQN algorithm.

4.1.3 Performance Comparison

When we compare the performance of the Q-learning algorithm with the DQN, we clearly see that DQN algorithm performs the best inside this environment, with an average reward for the best model over 100 consecutive episodes of -99.33, compared to -130.84 for the Q-leaning. The model trained with DQN has successfully solved the environment according to the criterion of having an average reward over 100 episodes higher than -110.

Figure 4.1.3 presents the policy of the best Q-learning model and DQN model.



(a) Policy for the Q-learning model 8.

(b) Policy for the DQN model 4.

Figure 4.1.3: Policy comparison for the best Q-learning model with the best DNQ model.

We can observe that the best policy obtained by Q-learning is highly heterogeneous. Among a cluster of observation (x position, velocity), an action selected can be inconsistent from the other actions present inside this cluster. This is represented by having, for example, a red square inside a group of grey squares. We do not encounter this problem with the policy created by the DQN. The policy is smooth.

We have tried to smooth the Q-learning policy to see if it leads to higher results. It was achieved by counting the number of occurrences of an action in a square centered on the tuple (pixel if we look at the image) we want to smooth. It is globally the same methodology as a convolution. We move a square across the Q-table, and count the number of times an action is present inside this square. The value at the center of this square is then replaced[1] by the most present action in the square. Figure 4.1.4 shows the outcome for a smoothing achieved with a square of length 5.

We have used 3, 5, 7 and 9 as square's length to smooth the best policy constructed by the model 8 trained by Q-learning. After that, we have evaluated the performances of the smoothed policies with the same methodology as before. The results are presented in Figure

[1] We replace its value in another empty Q-table to avoid employing pixels already smoothed.

4.1.5 and Table 4.1.5. The policies obtained with other square length values are presented in Figure B.1.9 in the Appendix B.1.

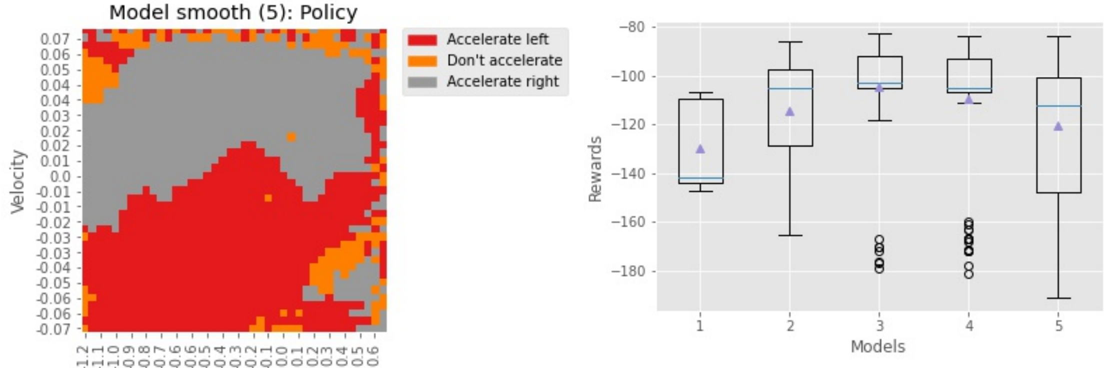


Figure 4.1.4: Smoothing policy with square length value of 5.

Figure 4.1.5: Distribution of the rewards obtained with the smoothed policies. The models correspond respectively to a square length of 1, 3, 5, 7, 9.

	Policy smooth (1)	Policy smooth (3)	Policy smooth (5)	Policy smooth (7)	Policy smooth (9)
mean	-129.98	-114.67	-104.58	-109.38	-120.56
std	16.29	25.22	20.68	26.42	27.42

Table 4.1.5: Mean and standard deviation obtained over 100 consecutive episodes of the Mountain Car environment by the smoothed policies.

We can note that smoothing the policy allows obtaining a better average reward. By using a smoothing square length of 5, we obtain an average reward of -104.58, which is enough to consider the environment solved by the Q-learning algorithm. Despite solving the environment, we can notice that the standard deviation has increased a bit and some episodes take more time to reach the flag than before.

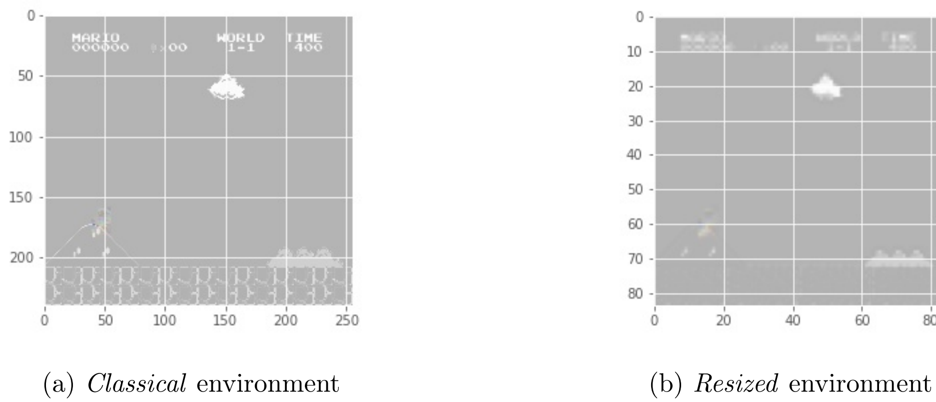
4.2 Super Mario Bros. Environment

4.2.1 Preprocessing

Before trying to solve the Mario environment, we have preprocessed the inputs received, as explained in the Section 3.3.1. We have created three different preprocessing methods applied to the Mario environment. We call the first the *classical environment* where the observations are greyscaled, we stack 4 frames and used the *simple movement* described in Table 3.1.3. The second prepossessing method is the *resized environment*, which is the same as the first one, but we resize the observation from 240×256 pixels to 84×84 pixels. The last is the *resized right environment*, which is a copy of the second environment, but we replace the possible actions by the *right only* action space. Table 4.2.1 summarizes the environments used. Examples of the results obtained after preprocessing are showed in Figure 4.2.1.

	Classical	**Resized**	**Resized right**
Greyscaled	Yes	Yes	Yes
Frames by stack	4	4	4
Image dimensions	[240;256]	[84;84]	[84;84]
Action space	SIMPLE MOVEMENT	SIMPLE MOVEMENT	RIGHT ONLY

Table 4.2.1: Description of the preprocessed environments.



(a) *Classical* environment (b) *Resized* environment

Figure 4.2.1: Image obtained by the preprocessing of the observations.

4.2.2 DQN

We have first started to try solving the Mario environment with the DQN algorithm. We have followed the same methodology as previously. We have designed various models, trained them for different times and simulated episodes to evaluate theirs performances.

Table 4.2.2 presents the different models designed and the parameters used. The remaining of the parameters were set at the default value presented in Table A.0.3.

Model	Type env	Learning rate	Buffer size	Batch size	Discount	Target update	Training steps	Training time (min)
1	Classical	0.00025	30000	32	0.9	10000	1,200,000	
2	Classical	0.00025	5024	32	0.9	10000	800,000	440
3	Classical	0.00025	5024	32	0.9	10000	800,000 + 1,200,000	
4	Classical	0.0001	25000	256	0.99	5000	400,00	600
5	Classical	0.0001	25000	128	0.99	25000	2,800,00	
6	Classical	0.0001	10000	64	0.99	15000	2,000,000	
7	Resized	0.0001	10000	64	0.99	15000	2,000,000	480
8	Resized	0.0001	25000	64	0.99	25000	3,000,000	
9	Resized right	0.0001	25000	64	0.99	25000	1,200,000 + 4,800,000	1850
10	Resized right	0.0001	10000	32	0.99	15000	1,800,000	366
11	Resized right	0.0001	15000	64	0.99	25000	6,000,000	1544
12	Resized right	0.000001	512	64	0.99	5000	1,000,000	244

Table 4.2.2: Model's parameters used to solve Mario with DQN.

The model 3 and 9 were trained in two parts, first by doing 800,000 (1,200,000) time steps, then we have reloaded the models and pursue the training for 1,200,000 (4,800,000) more time steps.

For some models, we have also noted the training time.

To evaluate the different models, we have run 50 simulations. We have used fewer simulations than the Mountain Car environment because each episode for Mario is take much more time to complete. In order to shorten the evaluation time, we have implemented some small modifications. At first, for each training episode, Mario had three lives. During our evaluation steps, we stop the simulation after the loss of one live. Mario could also be stuck by a pipe and stay there till the end of the timer, which means 8000 steps. To prevent that, we stop the simulation if Mario stays at the same x-position for 1000 time steps.
Stopping the episode prematurely could lead to over rated some simulations due to the way the reward is designed. For the recall, the agent is penalized if he stays too long at the same position. An agent being stuck for 1000 time steps will obtain an higher cumulative reward than an agent stuck for 8000 time steps, despite both agent ending at the same x-position. Figure B.2.1 in Appendix B.2 presents this problem. In the end, models that are often stuck will perform better if we stop the simulation after 1000 time steps of being stuck, compared to their performance if we do not stop them. However, this does not change the overall performance. Models that are not stuck will perform better than models stuck, even if we stop the simulation before its natural end.

Figure 4.2.2 shows the distribution of the reward per model. The mean is indicated by the triangle. Table 4.2.3 presents the mean and standard error of the rewards for each model.

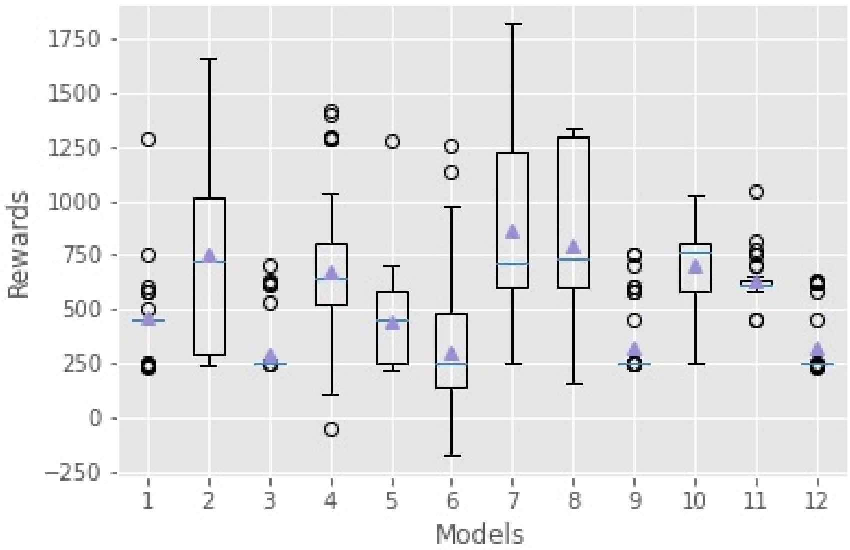


Figure 4.2.2: Distribution of the rewards obtained after 50 simulations for each model trained with DQN for the Mario environment.

	Model 1	**Model 2**	**Model 3**	**Model 4**	**Model 5**	**Model 6**	**Model 7**	**Model 8**	**Model 9**	**Model 10**	**Model 11**	**Model 12**
mean	458.54	752.7	292.14	675.9	446.66	305.42	864.64	798.9	316.7	706.36	635.28	319.58
std	154.58	422.95	123.74	354.49	198.85	345	362.94	399.97	151.67	191.51	85	145.71

Table 4.2.3: Mean and standard deviation obtained over 50 consecutive episodes of the Mario environment by the models trained with DQN algorithm.

The best model seems to be the model 7 with an average reward value of 864.64. We can notice that the model 3 and 9 are pretty bad. We think it is the consequence of the training in two times. The model 11 has the lower standard error and a decent average reward, it is probably due to its long training time (6,000,000 time steps). For the model 12, we have tried to decrease the learning rate and the buffer size, leading to very bad results. The model does not learn despite a long training time.

It is not obvious to draw a clear conclusion regarding the hyperparameters. It seems that the *resized* environment results in models with a higher average reward, as compared to models trained in a *classical* environment, as there are fewer inputs. The models are also train faster in *resized* environments. Finally, we can note that none of the models could reach the end of the first stage of Mario.

4.2.3 PPO

In the last part of this section, we use the PPO algorithm described in the Section 2.3.2 to solve the Mario environment.
As before, we have designed several models with different hyperparameter values. The specifications of the models can be found in Table 4.2.4.

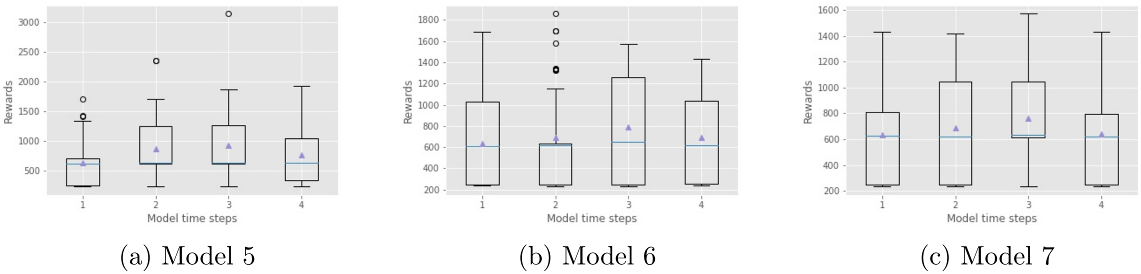
Model	Type env	Nstep	Batch size	Learning rate	Training steps	Training time (min)
1	Classical	4096	512	0.0001	1,600,000	1200
2	Resized	4096	1024	0.0001	3,000,000	850
3	Classical	256	64	0.00001	800,000	600
4	Resized right	256	16	0.0001	400,000	197
5	Resized right	1024	128	0.00001	4,000,000	1180
6	Resized right	512	64	0.000001	4,000,000	1270
7	Classical	512	64	0.000001	4,000,000	2650

Table 4.2.4: Model's parameters used to solve Mario with PPO.

This time, we have also added the training time in minutes for each model. As we can see, this training time can be huge, up to 44 hours for the model 7. Interestingly, the training time of this model is more than the double of the training time of model 6, in spite of having the same parameters and the same number of training steps. This is explained by the type of prepossessing method used. For model 7, we have used the *Classical* whereas for the model 6, we have used the *Resized right* (see Table 4.2.1 for more details).
The parameters chosen for the model 6 and 7 came from Renotte (2021). Inside his video, he creates an AI for Mario using the PPO algorithm, as we do. For these parameters and 4 million time steps, Mario achieves to pass the first level. We have tried to replicate this result inside this work.

Before showing the results obtained on 50 episodes, we have analyze the performance of some models that were trained during a varying number of time steps. This was achieved for model 5, 6 and 7 due to their longer training time steps. We have simulated 50 episodes after 1, 2, 3 and 4 million time steps and compared the rewards. The results are shown below.

In contrast of what one could think, the models do not perform better at 4 million steps. As a result, models 5, 6 and 7 used in the remainder of this work are the one obtained after 3 million steps of training.



(a) Model 5 (b) Model 6 (c) Model 7

Figure 4.2.3: Analyze of the rewards obtained over 50 Mario episodes. The boxplots correspond respectively to a simulation after 1, 2, 3 and 4 millions steps training.

The distribution of the rewards obtained over 50 consecutive episodes for all models can be found below.

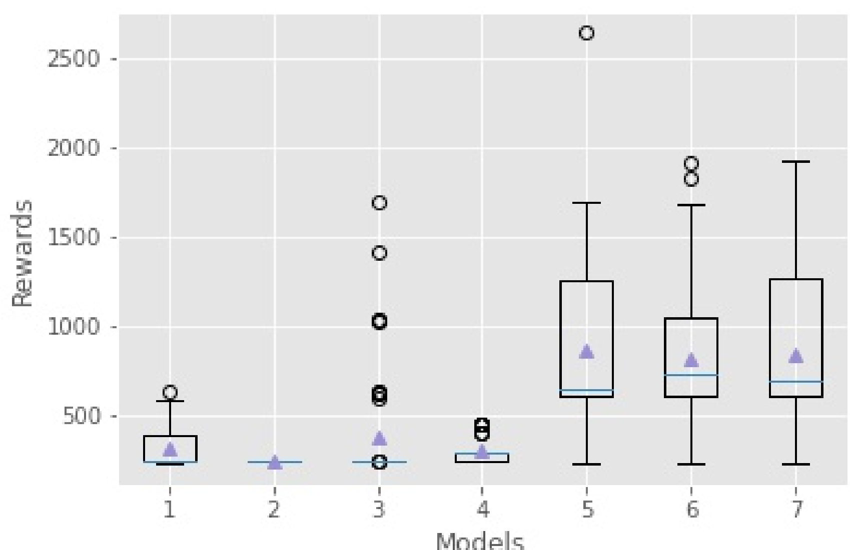


Figure 4.2.4: Distribution of the rewards obtained after 50 simulations for each model trained by PPO for the Mario environment

	Model 1	**Model 2**	**Model 3**	**Model 4**	**Model 5**	**Model 6**	**Model 7**
mean	316.42	252	386.96	310	868.02	818.32	847.6
std	124.46	0	321.59	68.69	467.52	443.19	423.05

Table 4.2.5: Mean and standard deviation obtained over 50 consecutive episodes of the Mario environment by the models trained with PPO algorithm

Model 5, 6 and 7 perform significantly better than the other models, probably due to their longer training time but also the right choice of hyperparameter values. We can also notice the length of the training time does not have such a strong effect (21 versus 44 hours) when using the resized environment.

4.2.4 Comparison

For both PPO and DQN, setting the hyperparameters right is primordial. We have seen that some models cannot learn from the environment when using bad values for their hyperparameters.

When we compare the best models constructed by both algorithms, we see that the PPO perform a little better, especially when we focus on the outlying rewards values. The ultimate goal of Mario is to reach the end of the level. Some episodes constructed with the models 5, 6 and 7 trained with PPO manage to reach the end of the level, which can be considered like the final goal of this work.

However, the probability of ending the level is quite small. We have simulated 5000 episodes for each model to identify which one reaches the end of the first level the most often. The results of these simulations are presented in Figure 4.2.5 and Table 4.2.6.

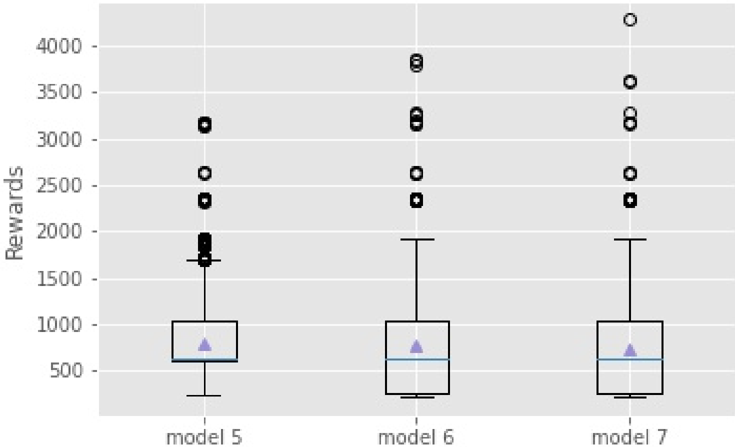


Figure 4.2.5: Distribution of the rewards obtained over 5000 simulations for the best model trained with PPO for the Mario environment.

	model 5	**model 6**	**model 7**
Mean	794.27	779.75	736.5
Std	404.99	452.75	444.01
Max	3186	3854	4279
Nb flags reached	9	11	7

Table 4.2.6: Mean, standard deviation, maximun reward and number of flags reached obtained over 5000 consecutive episodes of the Mario environment by the best models trained with PPO algorithm.

We can notice the model 5 has the highest mean reward, however, the principal objective is to reach the end of the level. This is achieved nine times by the model 5 and eleven times by the model 6. The model 6 is the best model among the three. We can note that the model 7 has also achieved the best run with a cumulative reward equal to 4279.

Chapter 5

Conclusion

This study intends to explore the Deep Reinforcement Learning methodology. To this end, we have implemented three different algorithms, the Q-learning, which is the most simple, the Deep Q-Network (DQN), which is an adaption of the Q-learning using neural networks, and finally the Proximal Policy Optimisation (PPO), which is one of the most advanced algorithms used in deep learning.

To test these algorithms, we have used two games. The first is called Mountain Car and is one of the most simple games implemented in Gym. It consists of a car placed at the bottom of a valley whose objective is to reach a flag situated at the top of a hill. There are only two inputs for this game and three possible actions. The second game is much more complicated. It is Super Mario Bros, also implemented in Gym. For this game, the inputs are the frames of the game, i.e. an 240×256 pixels image.

We have first started to solve the Mountain Car environment with Q-learning and DQN. For both algorithms, we have designed several models with different values as parameters.
Concerning the Q-learning, the most promising results were achieved by using a discount value of 0.99, an exponential epsilon decay, a learning rate equal to 0.05 and a Q-table of dimension 40×40. This model was trained during 40,000 episodes to achieve an average reward of -130.84. With this model, we have created an 2D image of the policy and tried to smooth this policy. After an adequate smoothing, we have obtained an average reward of -104.58, which was significantly better than previously. With this reward, the environment is considered solved by the Q-learning algorithm (average reward over -110).
We have used the implementation already developed by Stable Baselines3 (Raffin et al., 2021) for the DQN algorithm. The values of the best parameters were already found by Raffin (2020). We have used these ones to design the best DQN model. The only modification was to increase the number of training time steps to 400,000 and decrease the learning rate to 0.001. With these parameters, the model provides an average reward of -99.33, which solves the problem. The DQN algorithm performs much better than Q-learning.

After solving a simple environment, we try the methodology on a more complex game like Mario. First, we needed to preprocess the environment. We have used three preprocessing methodologies that we have compared. To summarize, the images are greyscalled and stacked by 4. In one environment, we keep the 240×256 pixels image. In the other two, we reduce this size to 84×84 pixels image.
After preprocessing, we have designed 12 models to determine the best parameters for DQN. We have found the best model was model 7, producing an average reward of 864. This model

uses the resized environment, with a learning rate equal to 0.0001, a buffer size equal to 10,000, a batch size of 64, a discount of 0.99 and is trained during 2,000,000 time steps. Despite a relative good performance, the model cannot complete the first level of Mario.
The same methodology is employed for the PPO algorithm. Among the investigated models, three models stand out from the crowd, the models 5, 6 and 7. All of them have been trained for 4,000,000 time steps. However, we have found that the models perform better after 3,000,000 steps of training, producing an average reward of approximately 850. We can notice that the model 7 was trained with an environment not resized compared to the others. The three give globally the same result, but the training of the model 7 has required 44 hours on our local machine, compared to 21 hours for the others.

Finally, during the simulation for the three best models trained with PPO, we have seen some rewards reaching 3000, which could signal the ending of the first level. To determine this frequency, we have run 5000 simulations for each model and discover that the model 5 successfully ends the first level nine times, the model 6 eleven times and the model 7 seven times. From this result, we can say that the model 6 is the best model constructed with 512 as buffer size (nstep), 64 as batch size, and a learning rate equal to 1e-6.

5.1 Suggestions

We end up this master thesis with some suggestions that could be worth to investigate. These suggestions primarily concern the Mario environment.

In the first place, we have seen that finding the best parameter values is complicated due to the extensive training time to achieve some results. A subsequent work could concern a way to optimize these values without having to train the models during a long time.

The training time for some models also present a direct limitation of this master thesis. When we look at models trained by companies, they have access to more computational power than we had, leading to a difficulty to find the best parameters.

The design of the neural networks present in the DQN and PPO correspond to the default design. It could be interesting to modify this design and observe how it impacts the performances of the models.

Finally, we have explored only model-free RL. We could investigate if using a model-based RL algorithm like the World Models (Ha and Schmidhuber, 2018) would lead to better results. We could also try to provide some demonstrations to the agent to accelerate its training time, like proposed by Brys et al. (2015).

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Arxiv Insights (2018). An introduction to policy gradient methods - deep reinforcement learning. https://www.youtube.com/watch?v=5P7I-xPq8u8ab$_{c}hannel = ArxivInsights$.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846.

Basodi, S., Ji, C., Zhang, H., and Pan, Y. (2020). Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3):196–207.

Bellman, R. (1957a). *Dynamic Programming*. Rand Corporation research study. Princeton University Press.

Bellman, R. (1957b). A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer.

Bhatt, S. (2018). 5 things you need to know about reinforcement learning. https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html. (Accessed on 04/25/2022).

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning*, volume 4. Springer.

Brock, A., De, S., Smith, S. L., and Simonyan, K. (2021). High-performance large-scale image recognition without normalization. In *International Conference on Machine Learning*, pages 1059–1071. PMLR.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Bronlee, J. (2018). A gentle introduction to broadcasting with numpy arrays. https://machinelearningmastery.com/broadcasting-with-numpy-arrays/. (Accessed on 05/12/2022).

Brownlee, J. (2018). Difference between a batch and an epoch in a neural network. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/: :text=at (Accessed on 05/13/2022).

Brownlee, J. (2019). Understand the impact of learning rate on neural network performance. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/. (Accessed on 05/13/2022).

Brys, T., Harutyunyan, A., Suay, H. B., Chernova, S., Taylor, M. E., and Nowé, A. (2015). Reinforcement learning from demonstration through shaping. In *Twenty-fourth international joint conference on artificial intelligence.*

Bryson, A. E. (1961). A gradient method for optimizing multi-stage allocation processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, volume 72, page 22.

Chidester, B., Do, M. N., and Ma, J. (2018). Rotation equivariance and invariance in convolutional neural networks. *arXiv preprint arXiv:1805.12301.*

Cohen, J., McClure, S. M., and Yu, A. (2007). Should i stay or should i go? how the human brain manages the trade-off between exploitation and exploration [proceedings paper]. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 362:933–42.

Dai, J., Qi, H., Xiong, Y., Li, Y., Zhang, G., Hu, H., and Wei, Y. (2017). Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773.

De Luca, G. (2020). What is the difference between gradient descent and gradient ascent? https://www.baeldung.com/cs/gradient-descent-vs-ascent. (Accessed on 05/19/2022).

DeepMind (2022). AlphaGo. https://www.deepmind.com/research/highlighted-research/alphago. (Accessed on 08/05/2022).

Dreyfus, S. (1962). The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

Farama Foundation (2022). Gym-docs. https://github.com/Farama-Foundation/gym-docs. (Accessed on 06/28/2022).

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., and Pineau, J. (2018). An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560.*

Fukushima, K. and Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer.

Gershgorn, D. (2016). Elon musk's artificial intelligence group opens a 'gym' to train a.i. https://www.popsci.com/elon-musks-artificial-intelligence-group-opens-gym-to-train-ai/. (Accessed on 06/28/2022).

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Ha, D. and Schmidhuber, J. (2018). World models.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Hayes, G. (2019). Use q-learning to solve the openai gym mountain car problem. https://gist.github.com/gkhayes/3d154e0505e31d6367be22ed3da2e955. (Accessed on 06/29/2022).

Haykin, S. S. (2009). *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, third edition.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Huang, S., Dossa, R. F. J., Raffin, A., Kanervisto, A., and Wang, W. (2021). The 37 implementation details of proximal policy optimization. https://ppo-details.cleanrl.dev//2021/11/05/ppo-implementation-details/. (Accessed on 07/11/2022).

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR.

Jaderberg, M., Simonyan, K., Zisserman, A., et al. (2015). Spatial transformer networks. *Advances in neural information processing systems*, 28.

Jefkine, K. (2016). Backpropagation in convolutional neural networks. https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/. (Accessed on 05/16/2022).

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589.

Kakade, S. and Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *In Proc. 19th International Conference on Machine Learning*. Citeseer.

Kauten, C. (2018a). Nes-py emulation system. https://github.com/Kautenja/nes-py.

Kauten, C. (2018b). Super Mario Bros for OpenAI Gym. https://github.com/Kautenja/gym-super-mario-bros.

Kavlakoglu, E. (2020). Ai vs. machine learning vs. deep learning vs. neural networks: What's the difference? https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks. (Accessed on 08/04/2022).

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954.

Ketkar, N. and Santana, E. (2017). *Deep learning with Python*, volume 1. Springer.

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

Konda, V. and Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, 12.

Kramer, A. and Sangiovanni-Vincentelli, A. (1988). Efficient parallel learning algorithms for neural networks. *Advances in neural information processing systems*, 1.

Kroese, D. P. and Rubinstein, R. Y. (2012). Monte carlo methods. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(1):48–58.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86.

Lee, T. B. (2019). How neural networks work—and why they've become a big business | ars technica. https://arstechnica.com/science/2019/12/how-neural-networks-work-and-why-theyve-become-a-big-business/. (Accessed on 05/13/2022).

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, University Helsinki, Finland.

Liu, D. (2017). A practical guide to relu. https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7. (Accessed on 05/13/2022).

Lu, L., Shin, Y., Su, Y., and Karniadakis, G. E. (2019). Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. (2018). Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562.

Machine Learnia (2021). Formation deep learning - youtube. =https://www.youtube.com/playlist?list=PLO$_f dPEVlfKoanjvTJbIbd9V5d9Pzp8Rw$.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill international editions - computer science series. McGraw-Hill Education.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

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):529–533.

Moore, A. W. (1990). Efficient memory-based learning for robot control. Technical report, University of Cambridge.

Murata, N. (1998). A statistical study of on-line learning. *Online Learning and Neural Networks. Cambridge University Press, Cambridge, UK*, pages 63–92.

Nielsen, M. A. (2015). *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA.

Nvidia Developer (2022). Cuda zone - library of resources. https://developer.nvidia.com/cuda-zone. (Accessed on 07/20/2022).

Ohnishi, S., Uchibe, E., Yamaguchi, Y., Nakanishi, K., Yasui, Y., and Ishii, S. (2019). Constrained deep q-learning gradually approaching ordinary q-learning. *Frontiers in neurorobotics*, 13:103.

OpenAI (2020). Mountaincar v0 wiki. https://github.com/openai/gym/wiki/MountainCar-v0. (Accessed on 06/28/2022).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Patrikar, S. (2019). Batch, mini batch & stochastic gradient descent. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. (Accessed on 05/13/2022).

Peters, J. and Bagnell, J. A. (2010). Policy gradient methods. *Scholarpedia*, 5(11):3698.

Raffin, A. (2020). Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

Renotte, N. (2021). Build an mario ai model with python | gaming reinforcement learning - youtube. =https://www.youtube.com/watch?v=2eeYqJ0uBKEab$_channel = NicholasRenotte$.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc Buffalo NY.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach.

Saha, S. (2018). A comprehensive guide to convolutional neural networks — the eli5 way. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Accessed on 05/15/2022).

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR.

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

Solai, P. (2018). How does backpropagation work in a cnn? https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c. (Accessed on 05/16/2022).

Soni, D. (2019). Translation invariance in convolutional neural networks. https://divsoni2012.medium.com/translation-invariance-in-convolutional-neural-networks-61d9b6fa03df. (Accessed on 05/16/2022).

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5 - rmsprop, coursera: Neural network for machine learning.

Tsitsiklis, J. and Van Roy, B. (1996). Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9.

Udyavar, N. (2017). Convolutional neural networks. https://github.com/nehal96/Deep-Learning-ND-Exercises/blob/master/Convolutional(Accessed on 05/15/2022).

Van Oirbeek, R. (2020). *Deep learning in insurance: Deep dive.* Louvain School of Statistics, Biostatistics and Actuarial Sciences, Course Material.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

Wikipedia contributors (2022). Chain rule — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Chain$_{r}uleoldid = 1087284020.[Online; accessed11 - May - 2022]$.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256.

Ying, X. (2019). An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022. IOP Publishing.

Yingge, H., Ali, I., and Lee, K.-Y. (2020). Deep neural networks on chip-a survey. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 589–592. IEEE.

Appendix A

Methodology

Key	Type	Description
coins	int	The number of collected coins
flag_get	bool	True if Mario reached a flag or ax
life	int	The number of lives remaining
score	int	The cumulative in-game score
stage	int	The current stage [1,...,4]
status	str	Mario's status ['small', 'tall', 'fireball']
time	int	The time left on the clock
world	int	The current word [1,...,8]
x_pos	int	Mario's x position in the stage (from the left)
y_pos	int	Mario's y position in the stage (from the bottom)

Table A.0.1: Information presented in the dictionary returned by the step function for the Mario environment.

Environment	Game	ROM	Example
SuperMarioBros-v0	Super Mario Bros.	standard	
SuperMarioBros-v1	Super Mario Bros.	downsample	
SuperMarioBros-v2	Super Mario Bros.	pixel	
SuperMarioBros-v3	Super Mario Bros.	rectangle	
SuperMarioBros2-v0	Super Mario Bros. 2	standard	
SuperMarioBros2-v1	Super Mario Bros. 2	downsample	

Table A.0.2: Type of possible environments in gym-super-mario-bros package of Kauten (2018b).

Name of parameters	Description	Default value
policy	The policy model to use	None
env	The environment to learn from	None
learning_rate	The learning rate, it can be a function of the current progress remaining (from 1 to 0)	0.0001
buffer_size	Size of the replay buffer	1,000,000
learning_starts	How many steps of the model to collect transitions for before learning starts	50,000
batch_size	Minibatch size for each gradient update	32
tau	The soft update coefficient ("Polyak update", between 0 and 1)	1
gamma	Discount factor	0.99
train_freq	Update the model every train_freq steps.	4
gradient_steps	How many gradient steps to do after each rollout	1
replay_buffer_class	Replay buffer class to use	None
replay_buffer_kwargs	Keyword arguments to pass to the replay buffer on creation.	None
optimize_memory_usage	Enable a memory efficient variant of the replay buffer at a cost of more complexity	False
target_update_interval	Update the target network every target_update_interval environment steps.	10,000
exploration_fraction	Fraction of entire training period over which the exploration rate is reduced	0.1
exploration_initial_eps	Initial value of random action probability	1.0
exploration_final_eps	Final value of random action probability	0.05
max_grad_norm	The maximum value for the gradient clipping	10
tensorboard_log	The log location for tensorboard	None
create_eval_env	Whether to create a second environment that will be used for evaluating the agent periodically	False
policy_kwargs	Additional arguments to be passed to the policy on creation	None
verbose	The verbosity level: 0 no output, 1 info, 2 debug	0
seed	Seed for the pseudo random generators	None
device	Device (cpu, cuda, ...) on which the code should be run.	"auto"
_init_setup_model	Whether or not to build the network at the creation of the instance	True

Table A.0.3: Possible arguments for the DQN function from Stable Baseline3 (Raffin et al., 2021).

Name of parameters	**Description**	**Default value**
policy	The policy model to use	None
env	The environment to learn from	None
learning_rate	The learning rate, it can be a function of the current progress remaining (from 1 to 0)	0.0003
n_steps	The number of steps to run for each environment per update	2048
batch_size	Minibatch size	64
n_epochs	Number of epoch when optimizing the surrogate loss	10
gamme	Discount factor	0.99
gae_lambda	Factor for trade-off of bias vs variance for Generalized Advantage Estimator	0.95
clip_range	Clipping parameter	0.2
clip_range_vf	Clipping parameter for the value function	None
normalize_advantage	Whether to normalize or not the advantage	True
ent_coef	Entropy coefficient for the loss calculation	0
vf_coef	Value function coefficient for the loss calculation	0.5
max_grad_norm	The maximum value for the gradient clipping	0.5
tensorboard_log	The log location for tensorboard	None
create_eval_env	Whether to create a second environment that will be used for evaluating the agent periodically	FALSE
policy_kwargs	Additional arguments to be passed to the policy on creation	None
verbose	The verbosity level: 0 no output, 1 info, 2 debug	0
seed	Seed for the pseudo random generators	None
device	Device (cpu, cuda, ...) on which the code should be run.	"auto"
_init_setup_model	Whether or not to build the network at the creation of the instance	True

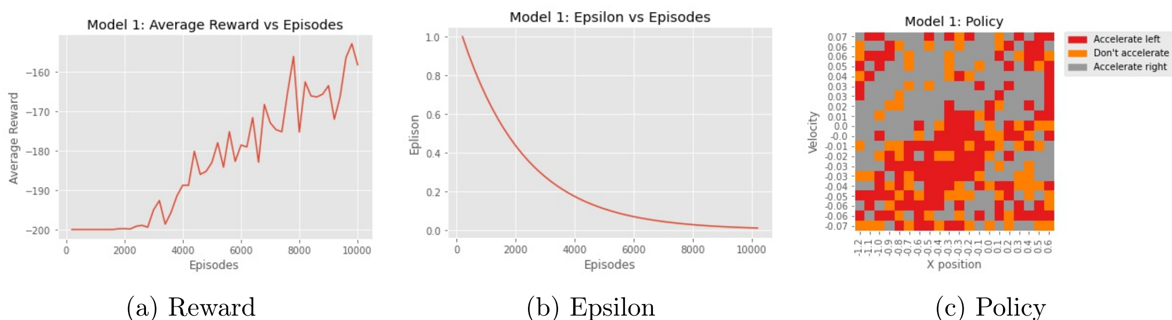
Table A.0.4: Possible arguments for the PPO function from Stable Baseline3 (Raffin et al., 2021).

Appendix B

Results

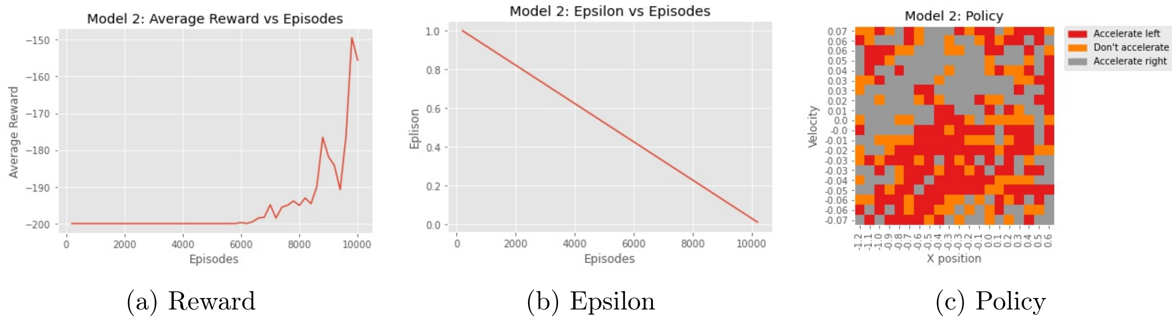
B.1 Q-learning

The images bellow present the evolution of the cumulative reward and the epsilon over the training episodes for each model obtained with the Q-leaning algorithm. The third images shows the final policy of the model, i.e. which action the model takes depending on the inputs.



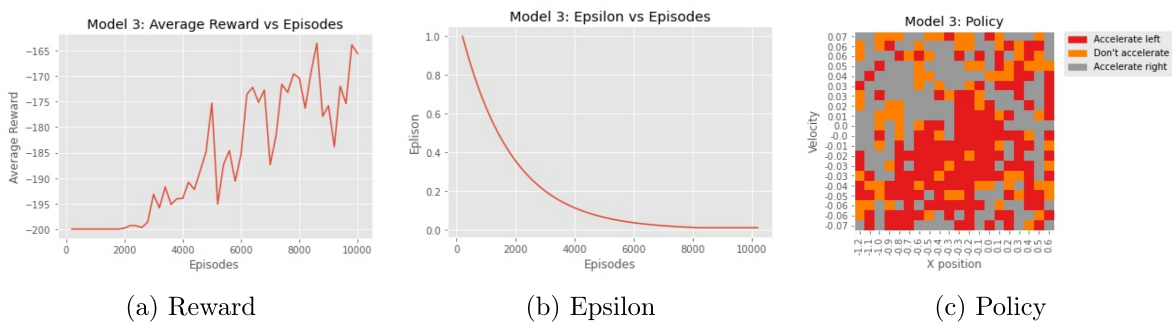
(a) Reward (b) Epsilon (c) Policy

Figure B.1.1: Model 1: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



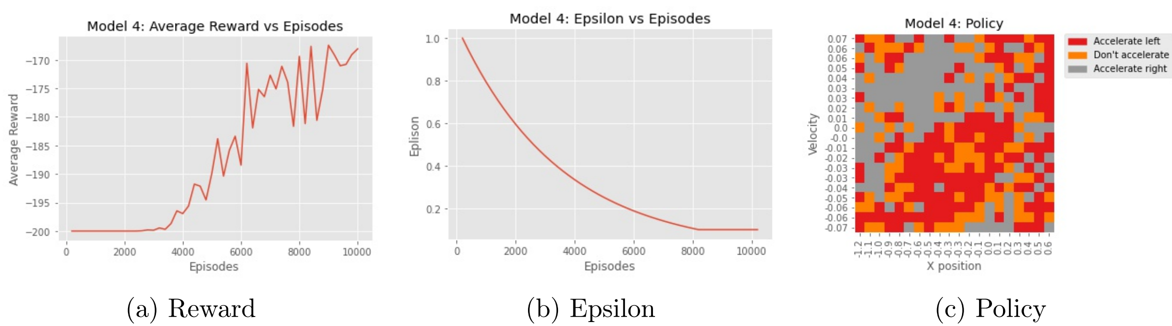
(a) Reward (b) Epsilon (c) Policy

Figure B.1.2: Model 2: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



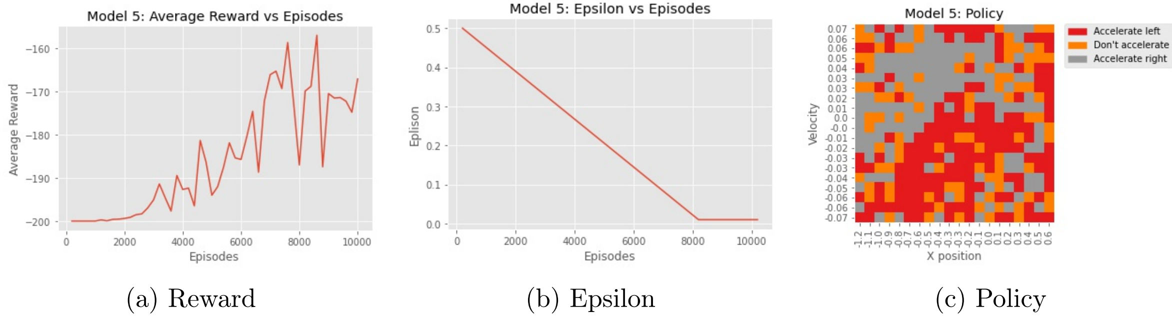
(a) Reward (b) Epsilon (c) Policy

Figure B.1.3: Model 3: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



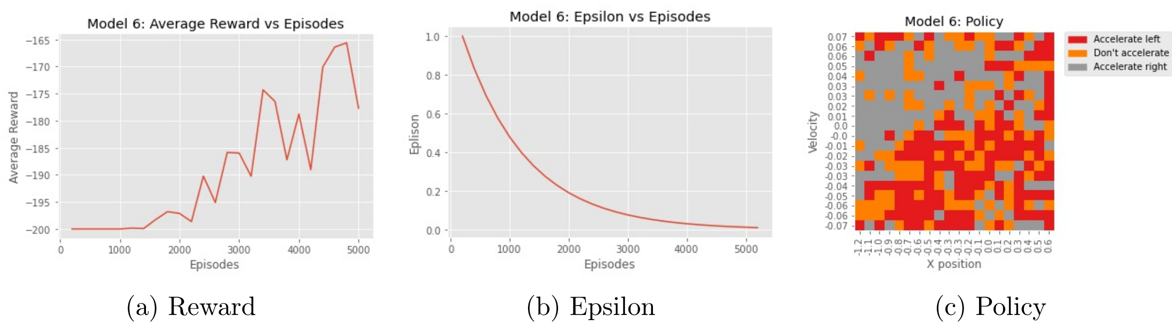
(a) Reward (b) Epsilon (c) Policy

Figure B.1.4: Model 4: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



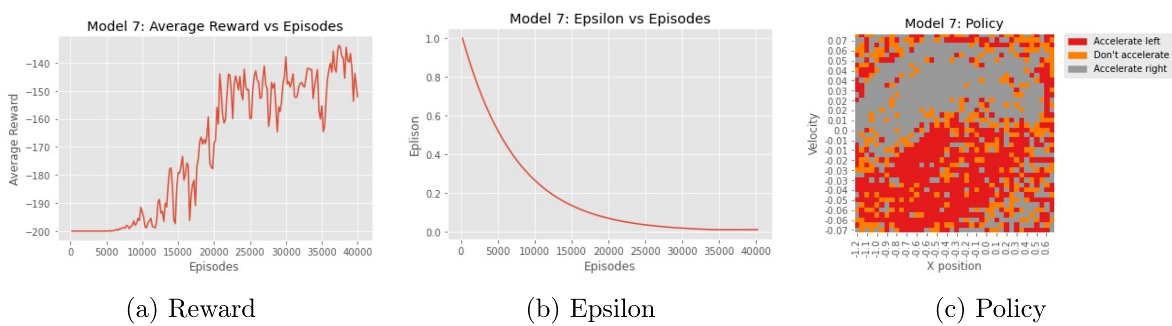
(a) Reward (b) Epsilon (c) Policy

Figure B.1.5: Model 5: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



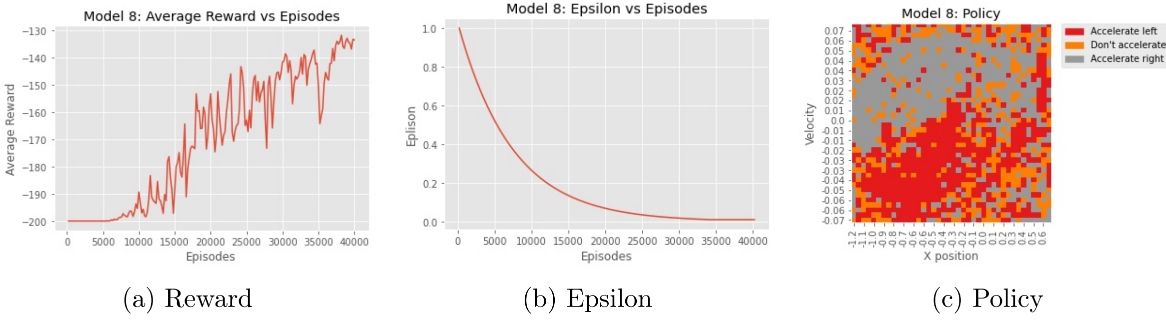
(a) Reward (b) Epsilon (c) Policy

Figure B.1.6: Model 6: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



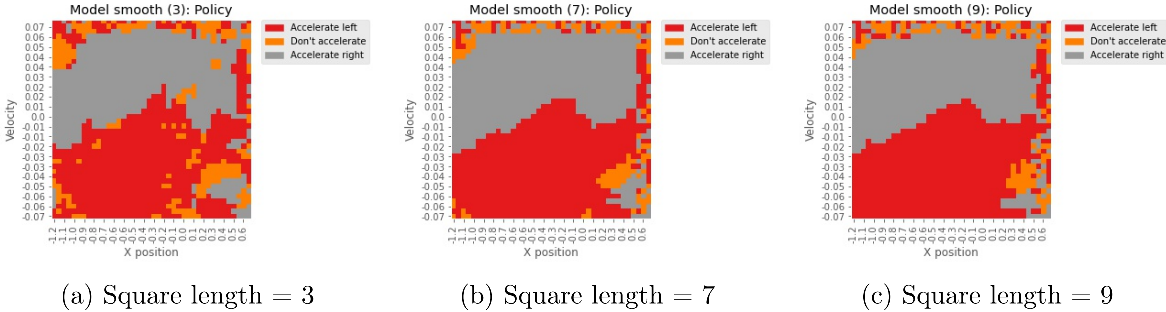
(a) Reward (b) Epsilon (c) Policy

Figure B.1.7: Model 7: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



(a) Reward (b) Epsilon (c) Policy

Figure B.1.8: Model 8: rewards evolution, epsilon evolution and final policy achieved with the Q-learning algorithm.



(a) Square length = 3 (b) Square length = 7 (c) Square length = 9

Figure B.1.9: Policies obtained by smoothing the model 8.

B.2 Mario Results

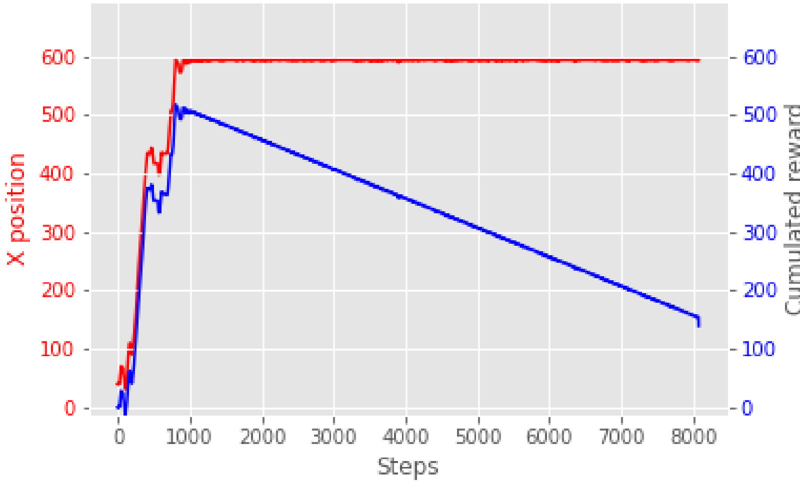


Figure B.2.1: Example of a stuck model.

This figure shows the evolution of the cumulative reward through time for a stuck model. We can see that the agent is stuck at the x-position 600. By staying stuck, the agent receives a negative reward that impacts his final cumulative reward. If we stop the simulation when the agent is stuck at the same place for 1000 time steps, his final reward will be equal to approximately 450, compared to 150 if we wait the natural end of the episode occurring after 8000 time steps. The consequence being a model that over-performs if we stop it prematurely.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté des sciences
Place des sciences, 2 bte L6.06.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/sc