Universidade de Aveiro
2021

Miguel Diogo
Ferraz Araújo

# Agentes com Aprendizagem Automática para Jogos de Computador

# Machine Learning Agents for Computer Games

Universidade de Aveiro
2021

**Miguel Diogo Ferraz Araújo**

# Agentes com Aprendizagem Automática para Jogos de Computador

# Machine Learning Agents for Computer Games

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor José Nuno Panelas Nunes Lau, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Luís Filipe de Seabra Lopes, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

**agradecimentos / acknowledgements**

Quero agradecer aos orientadores, Professor Nuno Lau e Professor Luís Seabra Lopes, pelo excelente acompanhamento que me deram durante a realização desta dissertação. Assim como ao colaborador, Professor Diogo Gomes, pela ajuda prestada. Quero agradecer também aos meus pais e à minha irmã pela motivação que me deram. Por fim, quero agradecer a todos os meus amigos pelo seu apoio, em especial ao Ruben Filipe pelas diversas conversas que tivemos sobre o tema desta dissertação.

I want to thank the coordinators, Professor Nuno Lau and Professor Luís Seabra Lopes, for the excellent support they gave me during this dissertation. As well as the collaborator, Professor Diogo Gomes, for the help provided. I also want to thank my parents and my sister for the motivation they gave me. Finally, I would like to thank all my friends for their support, especially Ruben Filipe for the various conversations we had on the subject of this dissertation.

**palavras-chave**

Aprendizagem Automática, Aprendizagem por Reforço, Aprendizagem Profunda, Aprendizagem Profunda por Reforço, Agentes, Jogos de Computador

**resumo**

Nos últimos anos, novos algoritmos de Aprendizagem por Reforço foram desenvolvidos. Estes algoritmos usam Redes Neuronais Profundas para representar o conhecimento do agente. Após ultrapassarem marcos anteriores da Inteligência Artificial (AI), como o Xadrez e o Go, esses métodos de Aprendizagem Profunda por Reforço (DRL) foram capazes de superar o nível humano em jogos muito complexos como o Dota 2, onde é necessário um planeamento a longo prazo e nos quais equipas profissionais de jogadores humanos treinam diariamente para ganhar competições de desportos eletrónicos. Estes algoritmos começam do zero, não usam exemplos de comportamento humano e podem ser aplicados em vários domínios. Aprendendo pela experiência, novos e melhores comportamentos foram descobertos, indicando um grande potencial nestes algoritmos. No entanto, eles exigem muito poder computacional e tempo de treino.

Os jogos de computador são utilizados numa disciplina de AI da Universidade de Aveiro como domínio de aplicação dos conhecimentos de AI adquiridos pelos alunos. Os alunos devem desenvolver agentes de software para esses jogos e tentar obter as melhores pontuações. O objetivo desta dissertação é desenvolver agentes usando as mais recentes técnicas de DRL e comparar o seu desempenho com o dos agentes desenvolvidos pelos alunos.

Para começar, os agentes com DRL foram desenvolvidos para um jogo mais simples como o Jogo do Galo, onde várias opções de aprendizagem foram abordadas até ser criado um agente robusto capaz de jogar contra vários oponentes.

Posteriormente, foram desenvolvidos agentes com DRL capazes de jogar a versão do Pac-Man utilizada na disciplina da Universidade de Aveiro, no ano letivo de 2018/19, através da realização de diversas experiências onde os parâmetros utilizados no processo de aprendizagem foram modificados de forma a obter melhores pontuações.

O agente desenvolvido, que obteve a melhor pontuação, consegue jogar em todas as configurações de jogo utilizadas na avaliação da disciplina e alcançou o top 7 das classificações, entre mais de 50 agentes desenvolvidos por alunos que utilizaram estratégias embutidas no código com algoritmos de pesquisa.

**keywords**

Machine Learning, Reinforcement Learning, Deep Learning, Deep Reinforcement Learning, Agents, Computer Games

**abstract**

In recent years, new Reinforcement Learning algorithms have been developed. These algorithms use Deep Neural Networks to represent the agent's knowledge. After surpassing previous Artificial Intelligence (AI) milestones, such as Chess and Go, these Deep Reinforcement Learning (DRL) methods were able to surpass the human level in very complex games like Dota 2, where long-term planning is required and in which professional teams of human players train daily to win e-sports competitions. These algorithms start from scratch, do not use examples of human behavior, and can be applied in various domains. Learning from experience, new and better behaviors were discovered, indicating a lot of potential in these algorithms. However, they require a lot of computational power and training time.
Computer games are used in an AI course at the University of Aveiro as an application domain of the AI knowledge acquired by students. The students should develop software agents for these games and try to get the best scores. The objective of this dissertation is to develop agents using the latest DRL techniques and to compare their performance with the agents developed by students.
To begin with, DRL agents were developed for a simpler game like Tic-Tac-Toe, where various learning options will be addressed until a robust agent capable of playing against multiple opponents is created.
Then, DRL agents capable of playing the version of Pac-Man used in the University of Aveiro course, in the 2018/19 academic year, were developed through the realization of various experiments where the parameters used in the learning process were modified in order to obtain better scores.
The developed agent, that obtained the best score, is able to play in all game configurations used in the evaluation of the course and reached the top 7 ranking, among more than 50 agents developed by students that used hard-coded strategies with pathfinding algorithms.

# Contents

# List of Figures

# List of Tables

# Acronyms

| | |
|---|---|
| **IEETA** | Institute of Electronics and Telematics Engineering of Aveiro |
| **IRIS** | Intelligent Robotics and Systems |
| **AI** | Artificial Intelligence |
| **AGI** | Artificial General Intelligence |
| **ML** | Machine Learning |
| **RL** | Reinforcement Learning |
| **SL** | Supervised Learning |
| **UL** | Unsupervised Learning |
| **ANN** | Artificial Neural Network |
| **DNN** | Deep Neural Network |
| **ReLU** | Rectified Linear Unit |
| **MLP** | Multilayer Perceptrons |
| **GD** | Gradient Descent |
| **SGD** | Stochastic Gradient Descent |
| **Adam** | Adaptive Moment Estimation |
| **DRL** | Deep Reinforcement Learning |
| **MDP** | Markov Decision Process |
| **CNN** | Convolutional Neural Network |
| **DQN** | Deep Q-Network |
| **DDQN** | Double Deep Q-Network |
| **PER** | Prioritized Experience Replay |
| **DNA** | Dueling Network Architecture |
| **D3QN** | Dueling Double Deep Q-Network |
| **POMDP** | Partially Observable Markov Decision Processes |
| **A2C** | Advantage Actor-Critic |
| **A3C** | Asynchronous Advantage Actor-Critic |
| **TRPO** | Trust Region Policy Optimization |
| **KL** | Kullback–Leibler |
| **PPO** | Proximal Policy Optimization |
| **MCTS** | Monte Carlo Tree Search |
| **GPU** | Graphics Processing Unit |
| **CPU** | Central Processing Unit |
| **TPU** | Tensor Processing Unit |
| **NFQ** | Neural fitted Q-iteration |
| **LSTM** | Long short-term memory |

CHAPTER 1

# Introduction

In this dissertation, recent Machine Learning (ML) algorithms will be used to develop software agents for computer games. These techniques combine Reinforcement Learning (RL) with Deep Neural Networks (DNNs).

## 1.1 Motivation

Games have been used to test Artificial Intelligence (AI) algorithms, since the beginning of studies in this area, as they are well-defined problems that often address some aspects of real-world challenges, and are designed to force the player to make complicated decisions. Also, before being applied in the real world, the algorithms can be evaluated a lot faster, due to iteration speed.

The Intelligent Robotics and Systems (IRIS) group of the Institute of Electronics and Telematics Engineering of Aveiro (IEETA) has several lines of research related to games, in which the way to solve complex tasks is learned by Machine Learning (ML) instead of being programmed directly by humans.

Furthermore, achieving human-level intelligence and the discovery of beneficial knowledge, by safely building Artificial General Intelligence (AGI), have been the focus of companies like DeepMind and OpenAI. They developed many new approaches in recent years, overtaking human knowledge on very complex games like Go, StarCraft II, and Dota 2, with algorithms that started without any examples of human behavior.

An agent that can learn without examples of optimal behavior, must interact with the environment and receive some kind of feedback in order to learn how to make decisions. This area of ML is called Reinforcement Learning (RL), and unlike other paradigms, it does not need a dataset, as data is generated by the agent experience. The recent approaches developed by the mentioned companies are grouped in the Deep Reinforcement Learning (DRL) area that uses Deep Neural Networks (DNNs) to represent the RL agent's knowledge.

## 1.2 Objectives

The University of Aveiro has an AI course [1] where computer games, developed by the teaching staff, are used to test the agents developed by students. Typically, the students program directly their strategy and do not use ML.

The final goal of this dissertation is to develop DRL agents, using state-of-the-art algorithms, for the same games that have been used in the university course, for which students have been developing strategies over the years, and then, verify if the learned strategies can reach the performance of the ones developed by students.

Therefore, the main objectives of this dissertation are the following:

- Develop agents using DRL techniques for a simpler game, such as Tic-Tac-Toe, before moving on to an AI course game like Pac-Man.
- Create a custom Gym [2] environment for the Tic-Tac-Toe game, in order to facilitate the integration with DRL libraries.
- Adapt the AI course game version of Pac-Man to a custom Gym environment where it is possible to use the same DRL algorithms and libraries used in Tic-Tac-Toe.
- Tune the learning strategy of the developed agents, so they can achieve better results.
- Properly test the developed agents and compare them with other reference agents.

## 1.3 Outline

This dissertation is divided into 4 chapters. In this introductory chapter 1, the motivations and objectives of this work were described. Chapter 2 presents the used DRL algorithms, including their deep learning and RL bases, as well as the tools used to implement them. In chapter 3, the various stages of the creation of the agents for both games, Tic-Tac-Toe and Pac-Man, are described. In chapter 4 the conclusions about the previous chapters are described, as well as the future work that could continue the tasks completed in this dissertation.

[1] See `www.ua.pt/en/uc/12287` for more details on the course.
[2] `https://gym.openai.com/`

CHAPTER 2

# Deep Reinforcement Learning (DRL)

Machine Learning (ML) is a branch of Artificial Intelligence (AI) where algorithms automatically learn to perform a task through a set of data. Depending on the data format we can separate the ML algorithms into three categories.

If the data is already labeled, we can train a Supervised Learning (SL) algorithm to learn a model that, given an unknown input, returns the correct label. A common example is to identify which digit from 0 to 9 is represented in a grayscale image with a handwritten digit.

If the data is not labeled, we can use an Unsupervised Learning (UL) algorithm to discover patterns or group the data by some discovered categories.

In this work we are going to focus on Reinforcement Learning (RL) since these algorithms allow us to train agents for the game environments we are interested in, without needing to use a dataset. The data is generated by the agent experience, and it learns by receiving feedback from the environment, as will be explained in section 2.2.

Section 2.1 will cover Deep Neural Networks (DNNs), since they are used in recent RL algorithms, forming the Deep Reinforcement Learning (DRL) family of algorithms. Some of them were used in this dissertation, such as Deep Q-Network (DQN) and Proximal Policy Optimization (PPO). Finally, section 2.3 covers the tools used to implement the mentioned algorithms.

## 2.1 Deep Learning

Deep learning is a technique that uses Deep Neural Networks (DNNs) to train ML algorithms to perform a given task. DNNs are a type of Artificial Neural Networks (ANNs), so in section 2.1.1 we present the basic ANNs, in section 2.1.2 the DNNs are described, and finally in section 2.1.3 we present the Convolutional Neural Networks (CNNs).

### 2.1.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a structure composed by artificial neurons. The first type of artificial neuron was called Perceptron and was inspired in the brain nervous activity [1]. The Perceptron is a function that receives several inputs $x_1, x_2, x_3, \cdots, x_n$ and returns only one binary value $y$ as output [2]. The perceptron's output can be calculated as:

$$\text{y} = \begin{cases} 0 & \text{if } \sum_{j=1}^{n} w_j x_j + b \leq 0 \\ 1 & \text{if } \sum_{j=1}^{n} w_j x_j + b > 0 \end{cases} \tag{2.1}$$

To express the importance of each input there is an associated weight $w$, as shown in Fig. 2.1. The bias $b$ can be seen as a threshold for activating the perceptron, for example, if the bias is very negative it will be difficult for the neuron to output 1.

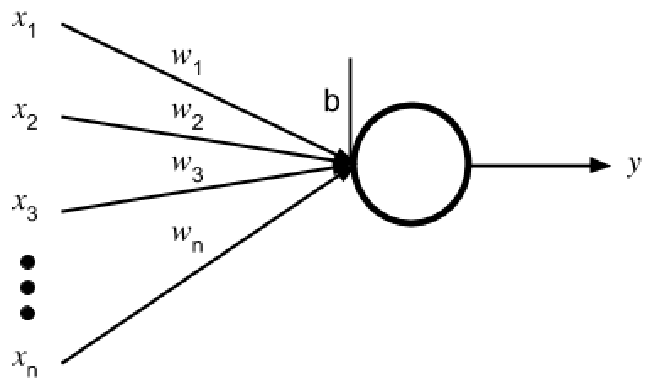


**Figure 2.1:** Artificial Neuron.

An ANN is composed of layers of neurons, where each neuron receives the outputs of all neurons in the previous layer, and outputs a number to the next layer. Taking the example of identifying handwritten digits, mentioned in the introduction to this chapter, the number of neurons in the first layer would be equal to the number of pixels in the input images. Then, there might or not be a hidden layer, and finally the last layer would have the number of neurons equal to the number of labels in the problem. In this case there are 10 different digits to identify, each one with a corresponding neuron. One of the neurons would be active, meaning the image contains the corresponding digit.

The weights and bias are parameters that can be adjusted in order to enable the ANN to match the inputs to the respective outputs. For the ANN to learn them, it is necessary that when a small change is made to these parameters, only a small corresponding change occurs in the ANN output as illustrated in Fig. 2.2.

The problem with the perceptron is precisely that when a small change is made in its weights or bias, it can change its output completely, for example, from 0 to 1. ANNs do not use perceptrons, but rather sigmoid neurons, since they can use non-binary output values.

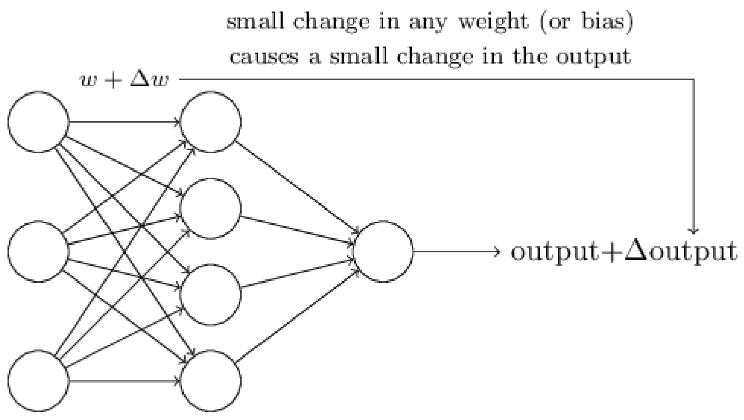

small change in any weight (or bias) causes a small change in the output
$w + \Delta w$
output+$\Delta$output

**Figure 2.2:** ANN's learnable parameters. Source: [2]

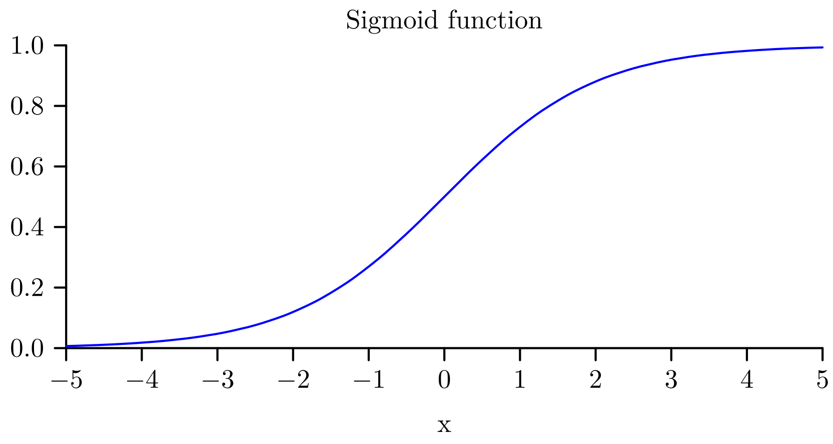


**Figure 2.3:** Sigmoid function. Equation 2.3

These neuron's output can be calculated as:

$$y = \sigma(\sum_{j=1}^{n} w_j x_j + b) \tag{2.2}$$

where $\sigma$ is a nonlinear activation function, which in this case is the sigmoid function (Equation 2.3) that smooths the output of the neuron in a value between 0 and 1, as represented in Fig. 2.3.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The ANN training is an optimization process that seeks to find the minimum of a loss function, by modifying the weights and biases of the ANN. The loss function is the error between the true values (labels) in the training data set, and the values calculated by the ANN. A popular loss function is the Mean Squared Error (MSE) that averages the squared differences between the network outputs and the labels as following:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - N(x_i; \theta))^2 \tag{2.4}$$

where $\theta$ are the weights and biases, $N(x_i; \theta)$ is the network output for the input $x_i$, and $y_i$ is the corresponding label.

The gradient of the loss function, with respect to the weights and biases of the network, is a vector that indicates the direction of the steepest increase in the loss function, in the space of the weights and biases. Its opposite direction indicates the direction that minimizes the loss function.

The method used to calculate the gradient of the loss function is called backpropagation. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule. The gradient is calculated one layer at a time, beginning in the last layer to avoid redundant calculations of intermediate terms in the chain rule.

There are several optimizers to do the backpropagation calculus, and propagate the gradient to the ANN's weights and biases. The most popular is the Gradient Descent (GD). In its basic variant, it calculates the gradient of the loss function for the entire training dataset, and uses its opposite direction to update the network parameters $\theta$ as following:

$$\Delta\theta = -\alpha \nabla_\theta L(\theta) \tag{2.5}$$

where $\alpha$ is the *learning rate* that determines the size of the gradient step, that is, how much the parameters will be updated in the gradient opposite direction.

Using backpropagation for every single training example, in order to calculate the desired changes, and then averaging them to calculate the gradient is very slow. It is common to use Stochastic Gradient Descent (SGD), that shuffles the training data and divides it in several mini-batches, to compute a GD step for each of the mini-batches.

There are other optimizers that explore more complex strategies to calculate the gradient and update the parameters. For example, Momentum [3] adds the previous update to the current update in order to accelerate the convergence to a minimum, and Adaptive Moment Estimation (Adam) [4] combines Momentum with the use of different *learning rates* for each of the parameters. The mentioned optimizers are implemented in the TensorFlow library used in this dissertation, that will be presented in section 2.3.1.

### 2.1.2 Deep Neural Networks

Deep Neural Networks (DNNs) are a subcategory of ANNs that have more than one hidden layer, which makes them have many more adjustable parameters.

To give you an idea, let's assume that the images with the handwritten digits are 28 by 28 pixels. So, the first layer will have 784 neurons. The last layer will have 10 neurons as already mentioned, in this case outputting numbers between 0 and 1 that represent the probabilities of being each of the 10 possible digits. Supposing that we use, for example, two hidden layers with 32 neurons each, we will have $32 \times 784 + 32 \times 32 + 10 \times 32 = 26432$ weights, and $32 + 32 + 10 = 74$ biases, which in total makes 26506 parameters to learn.

DNNs can get even bigger with many more layers and parameters to tune. Increasing the number of layers, the gradient of the loss function can get very attenuated to the first

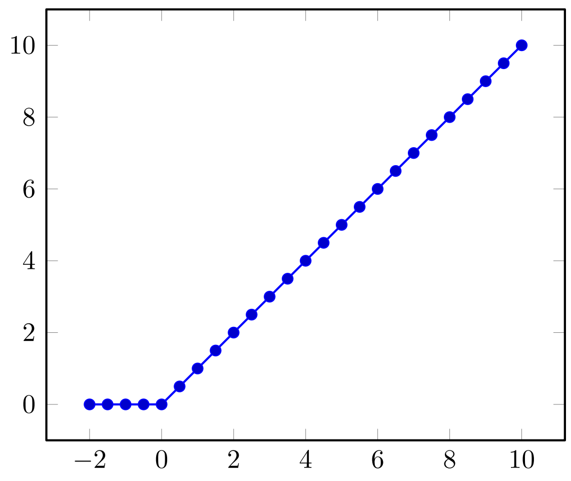


**Figure 2.4:** ReLU function. Equation 2.6

layers since its propagation is made from the last layer and the sigmoid gradient tends to 0 as the absolute value of $x$ increases. For this reason, it is more common to use the ReLU [5] (Equation 2.6) as an activation function, represented in Fig. 2.4, as its gradient is a constant value, for $x$ values larger than 0, not attenuating the gradient of the loss function.

$$f(x) = \max(0, x) \tag{2.6}$$

These networks are able to learn complex tasks, difficult to solve writing an algorithm by hand. They are sometimes called Multilayer Perceptrons (MLP), for historical reasons, despite being made of neurons with activation functions like ReLU or sigmoid, instead of perceptrons [2].

### 2.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a subcategory of DNNs that are specific for 2D data, where locality information is important. The first hidden layers of these networks are convolutional. A convolutional layer can detect multiple localized characteristics (features) of the input image. To do so, each neuron of the convolutional layer is connected only to a window of the previous layer neurons.

In the example illustrated in Fig. 2.5, the input image is $28 \times 28$ pixels and the filter (window) size is $5 \times 5$. Since the filter size is the same for all the neurons in the convolutional layer, and the filter will slide by one pixel at a time, this layer will have $24 \times 24$ neurons. If the slide of the filter was from 2 pixels between each neuron, the convolutional layer would have only 12 neurons. The slide size is called stride. Both the filter size and the stride are hyper-parameters that can be tuned.

Weights and biases are shared by all the neurons in the layer. So, in this case, the filter is defined by only $5 \times 5$ weights, and a bias. Therefore, all neurons in the filter can detect the same feature, but in different locations of the input, and the filter will be able to identify a feature in the entire image, for example, a vertical line.

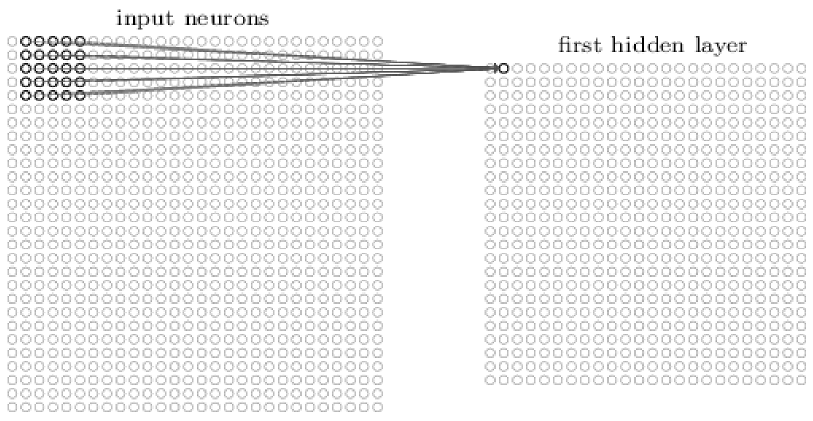


**Figure 2.5:** Convolutional layer with a 5 × 5 filter. Source: [2]

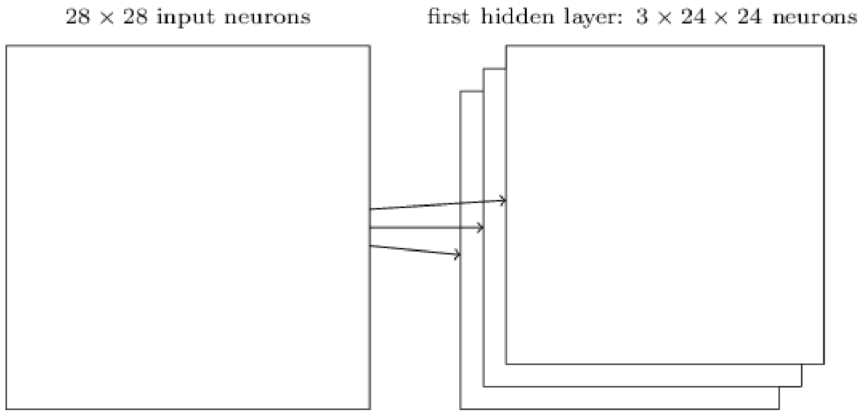


**Figure 2.6:** Convolutional layer with 3 filters. Source: [2]

A convolutional layer can contain several filters (all with the same size and stride), in order to identify several characteristics, as illustrated in Fig. 2.6. Usually convolutional layers have more than 3 filters. Examples of networks with several convolutional layers, each one with 32 or 64 filters, will be presented in section 2.2.2 of Reinforcement Learning (RL).

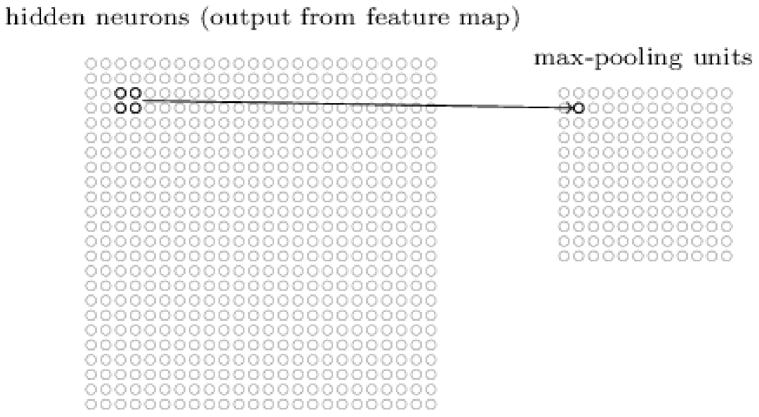


**Figure 2.7:** Max-pooling layer over a 2 × 2 region. Source: [2]

Typically CNNs use another type of layers right after the convolutional layers, the pooling layers. Their job is to simplify the output of the convolutional layers, by condensing the information. There are various techniques to use in pooling layers, for example, using max-pooling, each neuron will output the maximum activation in a given input region, as illustrated in Fig. 2.7.

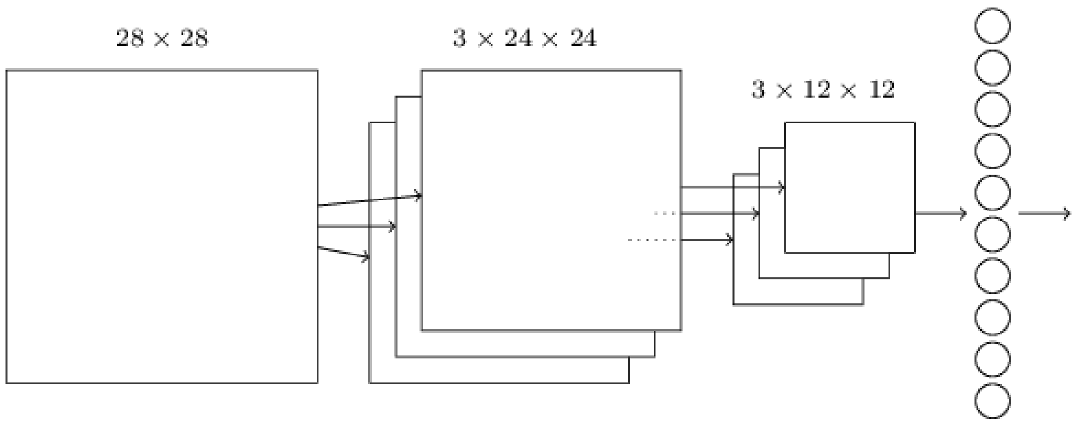


**Figure 2.8:** Full CNN architecture. Source: [2]

The final layer of a CNN is usually a normal fully connected layer, in order to output the desired format. Multiple fully connected layers can be stacked at the end of a CNN.

A simple example of a complete CNN architecture is illustrated in Fig. 2.8 where all the different mentioned layers are put together to solve the problem of identifying handwritten digits. The $28 \times 28$ input image is fed to a convolutional layer with 3 filters of size $5 \times 5$, which results in a $3 \times 24 \times 24$ layer. Next, a max-pooling layer is applied to $2 \times 2$ regions, resulting in a $3 \times 12 \times 12$ layer. Lastly, a fully connected layer with 10 neurons is applied to all the 432 previous resulting neurons, in order to output a result for each of the 10 possible digits.

In the Deep Reinforcement Learning (DRL) algorithms that will be presented in the next section, the authors do not use pooling layers in the CNN architectures. That is probably because the input contains very detailed information that is not worth losing. For example, if the input image is a map with walls and bombs, maybe it is not worth to condense the information in order to have more performance, because crucial information will be lost.

## 2.2 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of ML where algorithms that use Deep Neural Networks (DNNs) have been recently developed and obtained surprising results, forming the Deep Reinforcement Learning (DRL) field. In section 2.2.1 the basic concepts of the RL field are presented, and in the following sections we present 3 different categories of DRL algorithms. In section 2.2.2 we present algorithms that use value function approximators, in section 2.2.3 the policy gradients algorithms are presented, and finally, in section 2.2.4 we present algorithms that are based on the environment model.

### 2.2.1 Background of Reinforcement Learning

Humans, as well as other intelligent beings, learn from interacting with the world, experiencing different situations and discovering outcomes from their actions. These interactions can have long-term influences, which are often not obvious. This is the paradigm addressed in Reinforcement Learning (RL), where an agent learns to make decisions from interacting with the environment, without examples of optimal behavior. To make this possible, the agent needs to receive some kind of feedback from the environment to know how well it is doing. The numerical feedback signal the agent receives from the environment after taking an action, at time step $t$, is called the reward and is denoted as $r_{t+1}$. The goal of the agent is to maximize the sum of future rewards [6].

*Markov Decision Process*

A Markov Decision Process (MDP) is a straightforward way to represent the problem of learning from interactions, as it defines the main concepts needed. The agent is the decision-maker, that interacts with the environment through actions. In each time step, the agent observes the state of the environment, picks an action, and receives the next state and reward from the environment as shown in Fig. 2.9.

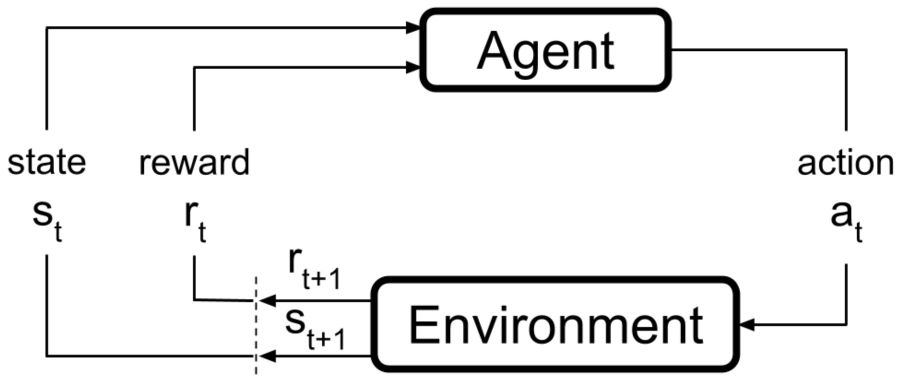


**Figure 2.9:** The agent-environment interaction in a MDP. Source: Adapted from [6, Fig. 3.1]

The sum of future rewards, from time step $_{t+1}$ until the last time step $T$, is called the return, denoted $R_t$:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T = \sum_{k=0}^{T} r_{t+k+1} \tag{2.7}$$

This concept works for episodic tasks, but some tasks can be continuous where $T = \infty$. So, it is common to use a discounting factor $\gamma$, defining the return as:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.8}$$

This notion of discounted return not only is guaranteed to converge for all kinds of tasks but also makes the agent prioritize immediate rewards over future rewards since future rewards will be more heavily discounted, as $\gamma$ is usually between 0 and 1.

*Value Functions*

To estimate how good an action or a state is, we can use the notion of expected return and calculate value functions that represent these estimates. But since the rewards the agent can expect in the future depend on the actions it will perform, the value functions are defined in relation to policies, which represent particular ways of acting [6].

A policy represents the probabilities of taking actions given a state. An agent following policy $\pi$ as the probability $\pi(a|s)$ of taking action $a$ under state $s$.

To evaluate a state $s$, we can define the value function $V_\pi(s)$ that represents the expected return starting in $s$ and following policy $\pi$ thereafter:

$$V_\pi(s) = E_\pi\left[R_t \mid s_t = s\right] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right] \tag{2.9}$$

Similarly, it is possible to calculate $Q_\pi(s,a)$ that represents the value of the expected return after taking action $a$, starting from $s$, and following policy $\pi$ thereafter:

$$Q_\pi(s,a) = E_\pi\left[R_t \mid s_t = s, a_t = a\right] \tag{2.10}$$

The goal of a RL algorithm is to improve its policy until the optimal policy is found. A policy that has greater or equal expected return $V_\pi(s)$ for all states, compared to all other policies, is considered an optimal policy. Associated with the optimal policy is an optimal state-value function $V_*(s)$ which gives the maximum expected return achievable, by any policy $\pi$, for each state. Similarly, the optimal policy has an associated optimal action-value function $Q_*(s,a)$ that gives the maximum expected return achievable, by any policy $\pi$, for each state-action pair.

*Dynamic Programming*

When the model of the environment is known by the agent, it is possible to know how the state will change after applying a certain action. In those cases, Dynamic Programming methods can be used, such as Policy Iteration and Value Iteration, that iteratively update the value for each state until they converge to $V_*(s)$. In the case of Value Iteration, the optimal policy can be deducted after the convergence, by using the model of the environment to evaluate which action maximizes the value of each state. However, if the model is not known, it is not possible to deduce the policy from the state-value function, because the agent does not know how the state will change after applying an action, thus it can not choose the optimal action. Therefore, it is common to use the action-value function $Q_*(s,a)$ instead of the state-value for model-free algorithms, as the agent can pick the action that has the largest q-value for the state it is currently in.

*Q-Learning*

The most popular RL algorithm is Q-Learning [7], which finds the optimal policy by learning the optimal q-values for all state-action pairs. This is done by iteratively updating the q-values after every completed transition in the environment by:

$$Q\left(s_t, a_t\right) = (1-\alpha)\, Q\left(s_t, a_t\right) + \alpha\left(r_{t+1} + \gamma \max_{a'} Q\left(s_{t+1}, a'\right)\right) \tag{2.11}$$

where $\alpha$ is the *learning rate* that controls how much the old value $Q\left(s_t, a_t\right)$ will be updated to the target value $r_{t+1} + \gamma \max_{a'} Q\left(s_{t+1}, a'\right)$. The target value includes the immediate reward, plus the greater discounted expected return for the state $s_{t+1}$ that action $a_t$ led to.

This is a model-free algorithm, as it does not need to know the model of the environment to choose the best action, it can simply pick the action that has the highest q-value for the current state.

*Exploration vs Exploitation*

If the action we pick is always the one that has the highest q-value for the state we are in, this is called a greedy policy and is not a good idea because we can end up committing always for the same actions and therefore the same states, missing some better possibilities. In RL, this trade-off is often called Exploration vs Exploitation and addresses the fact that the value of information gained by taking a non-optimal action, can be larger than the value of taking the greedy action. The more common approach is to use an $\epsilon$-greedy strategy where epsilon is the probability of taking a random action, and it can decay from 1 to some small number, during a certain number of time steps, forcing the agent to explore in the beginning and almost only exploit at the end of its lifetime.

### 2.2.2 Value Function Approximation

In environments where the state space is too big, it is not possible to store a value for each state, or even for all q-values as we will need $S \times A$ values, where $S$ is the number of states, and $A$ is the number of actions. So, it is possible to use a function approximator, parameterized by some number $P$ of parameters, that try to represent the behavior of the state-value function or the action-value function, and therefore reduce memory used, if $P < S \times A$.

RL frequently uses differentiable approximators, such as Artificial Neural Networks (ANNs) or Linear Combinations, as we can calculate the gradient and know how much changes in the weights affect the output. Another big advantage of using function approximators is that they are able to generalize to states never seen before using known states information. However, when a nonlinear function, such as a ANN, is used to approximate the optimal action-value function $Q_*\left(s, a\right)$, it may be unstable and diverge [8].

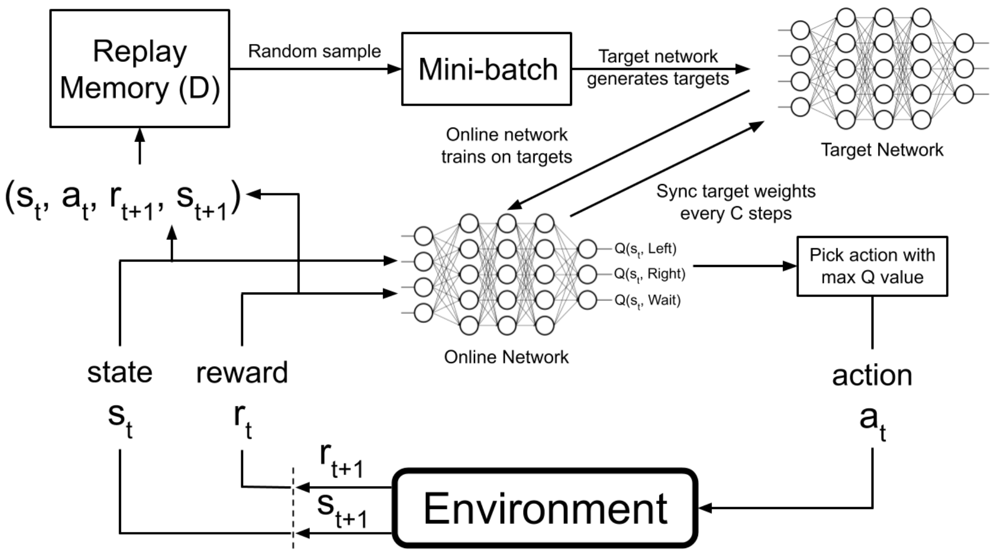


**Figure 2.10:** Deep Q-Network (DQN) algorithm.

*Deep Q-Network*

The authors of the Deep Q-Network (DQN) algorithm [8] point out some reasons, such as the fact that subsequent observations are highly correlated, and the policy may change drastically with small updates to the q-values and therefore change the data the agent will experience in the future. So, they used a biologically inspired technique that, at each time step, stores the transition $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in a large data set of recent experiences $\mathcal{D}_t = \{e_1, \ldots, e_t\}$ and trains over a random mini-batch of those, preventing the agent from learning only from recent high correlated experiences and getting smoother changes in the data distribution. Another reason for the ANN instability is the correlation between the target values and the current values since they are both generated by the same function approximator, so the authors proposed the use of a second ANN, called the target network, that decreases these correlations and makes divergence less likely. The target network is used to generate targets $y_i$ for each sampled experience $e_i = (s, a, r, s')$ as

$$y_i^{DQN} = r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) \tag{2.12}$$

where $\theta_i^-$ are the target network weights that are periodically updated with the weights $\theta_i$ of the online network, only every $C$ time steps, staying fixed between updates thus making the target network more stable. So, the Q-network is trained in each time step as described in Fig. 2.10, by sampling a uniformly random mini-batch from the replay memory $\mathcal{D}$ and optimizing by Gradient Descent (GD) the following sequence of loss functions:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(\mathcal{D})}\left[\left(y_i^{DQN} - Q(s, a; \theta_i)\right)^2\right] \tag{2.13}$$

There are other stable methods for training ANNs in RL, like Neural fitted Q-iteration (NFQ) [10], but they are not sufficiently efficient to be used with large ANNs. In DQN Atari

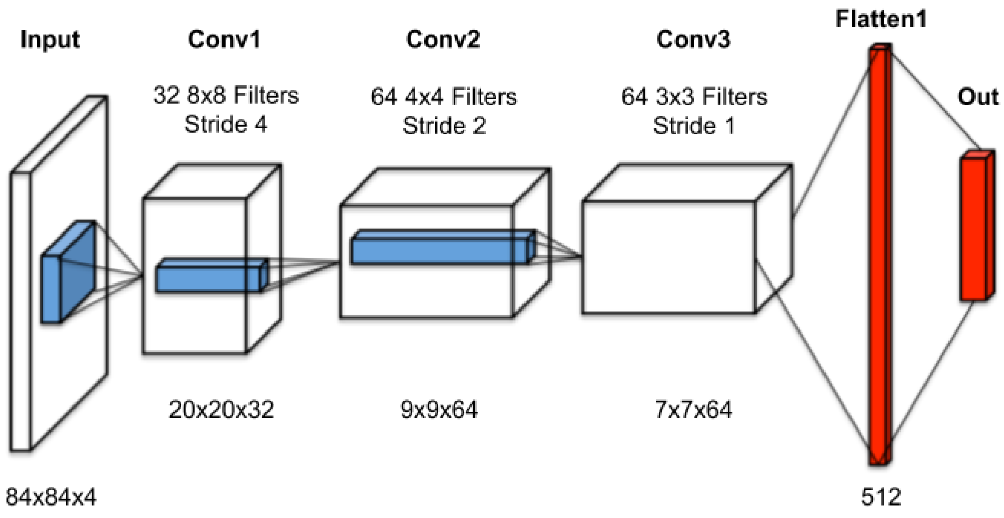


**Figure 2.11:** Deep Q-Network (DQN) architecture. Three convolutional layers and one fully connected (flatten) layer. Source: Adapted from [9]

experiments, DeepMind researchers were able to use a CNN that contains three convolutional hidden layers and one fully connected hidden layer as illustrated in Fig. 2.11. The first convolutional hidden layer applies 32 filters of size $8 \times 8$ with stride 4, the second hidden layer convolves 64 filters of $4 \times 4$ with stride 2, and the third convolutional layer applies 64 filters of $3 \times 3$ with stride 1. The fully connected hidden layer consists of 512 neurons. All hidden layers are followed by the ReLU activation function. The output layer is a fully connected linear layer with a q-value output for each valid action.

DQN algorithm was evaluated in a set of games from the Atari 2600 console. The same parameters and network architecture were used for all tasks, only receiving the video input images from the simulator, without any predefined features. So, the input of the CNN is the last four preprocessed frames as a $84 \times 84 \times 4$ image, and the number of outputs can go from 4 to 18, depending on the game. The agents were able to outperform human professionals in 29 of the 49 tested games and have successfully learned strategies that humans follow, without using any human data in the learning process, and only receiving the image pixels as input.

*Double Deep Q-Network*

Later in 2015, DeepMind's team incorporated the idea of Double Q-learning [11] into the DQN, creating the Double Deep Q-Network (DDQN) [12]. In Q-learning, the same value function is used to select and evaluate the action for the target value, which can lead to over-optimistic value estimates. Double Q-learning addresses this issue by decoupling the selection of an action from its evaluation, using a different function for each step. So in DDQN the online network $\theta_i$ is used to select the action while the target network $\theta_i^-$ is used to evaluate it, resulting in the following targets:

$$y_i^{DDQN} = r + \gamma Q\left(s', \arg\max_{a'} Q\left(s', a'; \theta_i\right); \theta_i^-\right) \tag{2.14}$$

Using target $y_i^{DDQN}$ instead of target $y_i^{DQN}$ in the DQN loss calculation, we get the

DDQN algorithm, which was also evaluated in the Atari environments, and ended up finding better policies.

*Prioritized Experience Replay*

In both DQN and DDQN, sparse reward games were the most difficult as the agent is unlikely to reach a rewardable state, and this transition will also be easily hidden among all other redundant ones. To help overcome this problem of some transitions having more information to learn than others, aggravated by sparse reward tasks, a mechanism called Prioritized Experience Replay (PER) was introduced by the DeepMind's team [13].

With PER the stored experiences have an associated priority that represents the learning value of a transition, liberating the agent from having the probabilities of considering transitions proportional to the frequency that they were experienced. The priority of each experience $e_i$ is proportional to the error given by $L_i$ which indicates how unexpected was the transition and is converted to a normalized probability of choosing that experience when sampling a batch. Introducing sampling priorities to the replay buffer $\mathcal{D}$, changes the distribution on favor of the higher prioritized experiences, making the network overfit to them. To address this, the update step is scaled down by a weighting factor, according to the probability they were sampled with. In this way, the DDQN algorithm combined with PER was able to overtake the state-of-the-art results in the Atari benchmark.

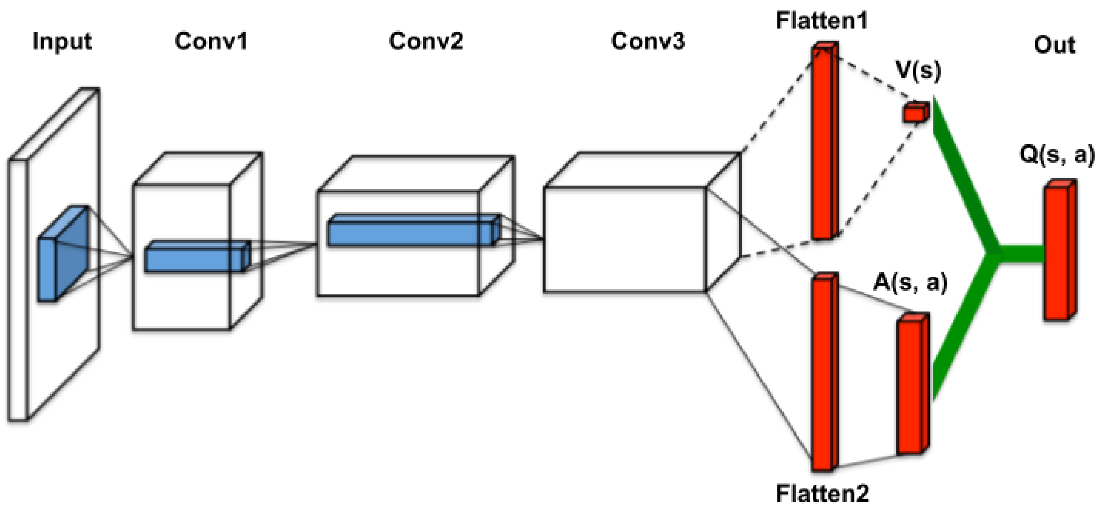


**Figure 2.12:** Dueling Network Architecture (DNA) architecture. Three convolutional layers and two separate streams of fully connected (flatten) layers. Source: Adapted from [9]

*Dueling Network Architecture*

Another approach was developed by the same company, where they propose the use of a different ANN architecture. In the Dueling Network Architecture (DNA) [9] the lower layers are convolutional as in DQN, but on top of those, are two separate sequences (streams) of fully connected layers, providing separate estimates of the value $V_\pi(s)$ and advantage $A_\pi(s,a)$ functions as illustrated in Fig. 2.12. The advantage function is given by $A_\pi(s,a) =$

$Q_\pi(s, a) - V_\pi(s)$ and represents how much better is an action compared to the other actions in a given state. These streams are then combined into a single output q-function, as in DQN, normalizing the q-values with the mean advantage values.

The great advantage of using DNA is in the training phase, as in every update of q-values, the value stream $V_\pi(s)$ is also updated, in contrary to the single-stream architecture where only the value for one action is updated. This leads to a more efficient learning of the state-value function, which is even more noticeable with an increasing number of actions. By combining Dueling Double Deep Q-Network (D3QN) with PER, the authors obtained even better results compared to the previous PER with DDQN baseline.

### 2.2.3 Policy Gradients

In the algorithms detailed in the previous section, the policy was generated by first approximating an action-value function $Q(s, a; \theta)$, parameterized by $\theta$, to estimate the future return of the possible actions, and then choosing the one with maximum value. In policy gradient algorithms the policy is learned directly, by optimizing a policy function $\pi(a \mid s; \theta)$, using parameters $\theta$, that maps states to actions [6].

An advantage of policy gradients algorithms is they can learn stochastic policies as the policy function outputs a probability distribution over the actions, given a state. In contrary of the policies generated by value functions that always output the same action for given a state, which are called deterministic policies. As a consequence, the exploration vs exploitation trade off is automatically handled by the policy gradient algorithms, as the agent will not always take the same action, allowing it to explore the action space. Additionally, in some environments, like Partially Observable Markov Decision Processes (POMDP), the optimal behaviour can be stochatic, as different states can be percieved by the agent as the same state, and maybe those states require different optimal actions.

Another advantage of policy gradients are the convergence properties that tend to be more stable as the policy updates are smoother. In value-based methods the policy can change drastically with small changes in the action-value estimates. And, as the policy is being parameterized directly, in the case of policy gradients, we have a guarantee to converge at least to a local optimum.

In addition, policy gradient algorithms are capable of handling continuous action spaces, as opposed to value function approximators. Also, in high dimensional action spaces, calculating the q-values for each action and then choosing the best one can be computationally hard, so the policy gradient algorithms also have the advantage of directly sampling an action of the probability distribution, without having to compute the maximum valued action.

*REINFORCE*

An example of a policy gradient method is the REINFORCE algorithm [14], that applies gradient ascent to update the parameters $\theta$ in the direction of the return $R_t$ as following:

$$\Delta\theta = \nabla_\theta \log \pi(a_t \mid s_t; \theta) R_t \tag{2.15}$$

As we have to wait for the end of an episode to know the true return and update the parameters, it is considered a Monte-Carlo method. The parameters will be adjusted depending on how well the end result of the episode was, which is an unbiased estimate of the expected future reward. But this method has high variance because for example if some actions were good, but others were bad and lead to a low return, the updated policy will avoid all the actions taken, including the good ones.

*Action-Value Actor-Critic*

In order to reduce the variance, it is possible to update the policy parameters more frequently, for example every step, if we use an estimate of the expected future reward using a value function approximator, like an ANN, and update the policy parameters $\theta$ in the direction of that estimate. These algorithms are called Actor-Critics as they use two sets of parameters, one set $\omega$ for the action-value function approximator (critic), and the other set $\theta$ for the policy function (actor) that will be updated in the direction of the critic as following:

$$\Delta\theta = \nabla_\theta \log \pi\left(a_t \mid s_t; \theta\right) Q\left(s_t, a_t; \omega\right) \tag{2.16}$$

*Advantage Actor-Critic*

It is common to subtract a baseline function $B(s)$ from the policy gradient [15], reducing the variance without changing the direction that maximizes the return, working as scaling factor. A common baseline function is the state value function $V(s)$, and when it is subtracted from the critic action-value function, it gives us the advantage function $A\left(s,a\right) = Q\left(s,a\right) - V\left(s\right)$ that represents how much better than the average is an action in a given state. Adjusting the policy parameters in the direction of the advantage function gives us the base of the Advantage Actor-Critic (A2C) algorithm, that applies the following updates to the parameters:

$$\Delta\theta = \nabla_\theta \log \pi\left(a_t \mid s_t; \theta\right) A\left(s_t, a_t; \omega\right) \tag{2.17}$$

To calculate the advantage function we could use two different sets of parameters for $Q_\pi\left(s,a\right)$ and $V_\pi\left(s\right)$, approximate these functions separately, and then combine the results to get the advantage function. But this approach adds a lot of complexity since many more parameters have to be learned. The common approach is to use the temporal difference error $\delta_t = r_{t+1} + \gamma V\left(s_{t+1}\right) - V\left(s_t\right)$ as an estimate of the advantage function. But, instead of using only the next state in the estimation, the agent receives up to $T$ rewards from the environment before updating the parameters as following:

$$\Delta\theta = \nabla_\theta \log \pi\left(a_t \mid s_t; \theta\right) \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V\left(s_{t+k}; \omega\right) - V\left(s_t; \omega\right) \tag{2.18}$$

where $k$ is limited by $T$ or a terminal state, representing how many timesteps of data the agent collected, tooking actions without updating the parameters.

*Asynchronous Advantage Actor-Critic*

The A2C algorithm is the synchronous version of Asynchronous Advantage Actor-Critic (A3C) [15] where different agents are executed in parallel on multiple instances of the environment. Each agent updates the global network asynchronously. The multiple agents collecting experience make the algorithm more sample efficient as policy gradients do not collect experiences to a replay buffer like DQN. In general, on policy gradients, after a batch of experience as been used to do a gradient update, the experiences are lost.

After the A3C paper was published, researchers created the synchronous implementation (A2C) that uses a coordinator that waits the parallel actors to finish their experience segment before updating the global parameters, and in the next iteration all agents start from the same policy. A2C algorithm was shown to perform better than A3C [16].

*Trust Region Policy Optimization*

The idea of Trust Region Policy Optimization (TRPO) [17] is to make sure that the updated policy does not move too far way from the current policy, to mitigate large policy updates that can be destructive. The objective function in advantage actor-critics can be described as:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta\left(a_t \mid s_t\right) \hat{A}_t\right] \tag{2.19}$$

which, due to derivation properties, is equivalent to maximizing:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t\right] \tag{2.20}$$

where $\theta_{\text{old}}$ is the vector of policy parameters before the update.

The authors of TRPO maximized the optimization objective subject to a Kullback–Leibler (KL) constraint, so that the policy $\theta$ does not deviates to much from the old policy $\theta_{\text{old}}$, as following:

$$\begin{array}{ll} \underset{\theta}{\operatorname{maximize}} & \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t\right] \\ \text{subject to} & \hat{\mathbb{E}}_t\left[\mathrm{KL}\left[\pi_{\theta_{\text{old}}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right)\right]\right] \leq \delta \end{array} \tag{2.21}$$

*Proximal Policy Optimization*

A problem with TRPO is that the KL constraint adds an additional overhead to the optimization process. In Proximal Policy Optimization (PPO) [18] the constraint is included directly into the optimization objective.

Denoting the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$ we can define the TRPO objective, without a KL constraint, as following:

$$L^{TRPO}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta) \hat{A}_t\right] \tag{2.22}$$

The ratio $r_t(\theta)$ will be larger than 1 if the action is more probable now than it was in the old policy, and will be between 0 and 1 if the action is less likely now than it was before the last gradient step. With this notation we can define the clipped objective function used in PPO as:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[ \min \left( r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{2.23}$$

where $\epsilon$ is a hyperparameter, e.g $\epsilon = 0.2$. The authors modified the objective by taking the minimum between $L^{TRPO}$ and a clipped version of the probability ratio that encourages $r_t(\theta)$ to stay in the interval $[1-\epsilon, 1+\epsilon]$. Resulting in a lower (pessimistic) bound of the unclipped objective.

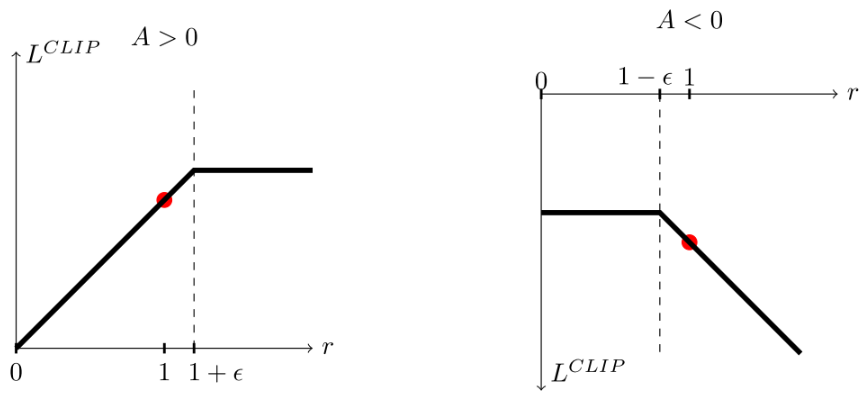


**Figure 2.13:** PPO clipped objective function. Source: [18, Fig. 1]

As illustrated in Fig. 2.13 (left), when the advantage function is positive, meaning the actions are better than expected, the objective function is clipped when $r_t(\theta)$ gets too high in order to not overdo the action update too much and limit the effect of the gradient update. Similarly, when the advantage function is negative, the objective function flattens when $r_t(\theta)$ goes near zero, as represented in Fig. 2.13 (right).

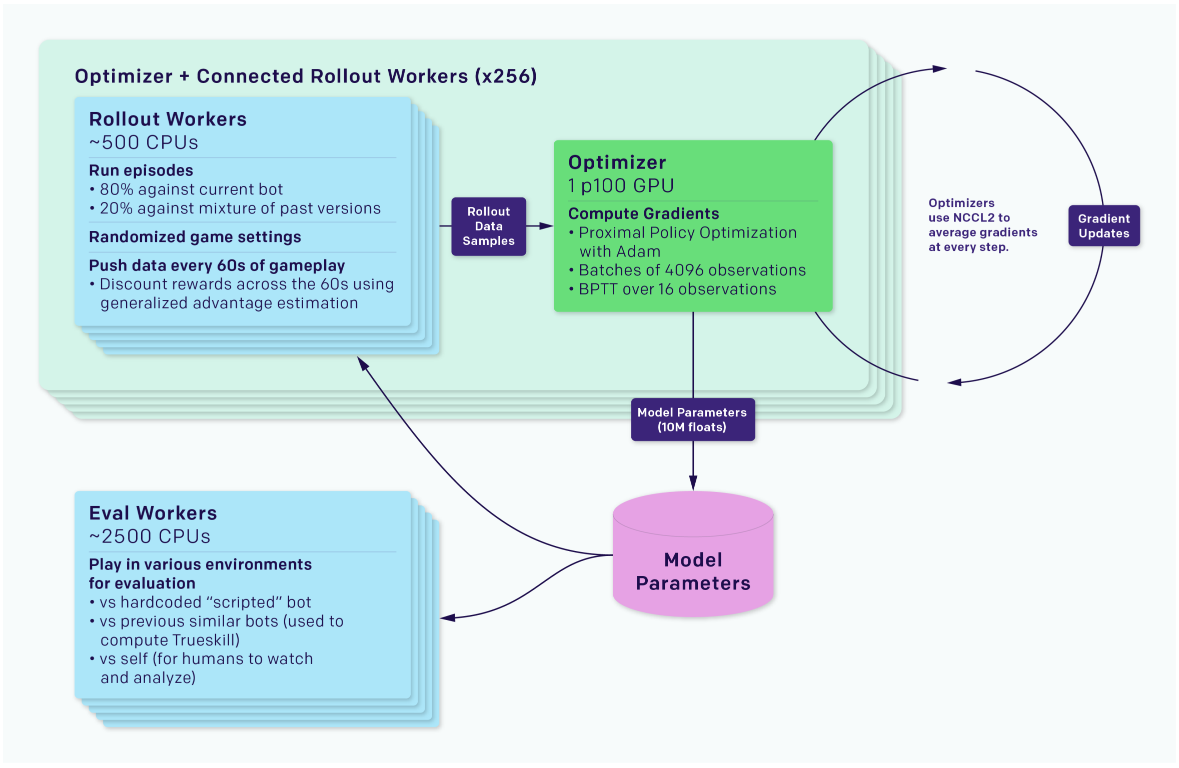
The PPO final objective function has two more terms in addiction to $L^{CLIP}$. One corresponding to the value function $V(s_t;\theta)$ error, used to calculate the advantage, and the other term is an entropy bonus to ensure sufficient exploration:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[ L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{2.24}$$

where S is the entropy bonus, $c_1$ and $c_2$ are coefficients (hyperparameters), and $L_t^{VF}$ is the value function approximator loss $\left(V_\theta(s_t) - V_t^{\text{targ}}\right)^2$.

As we can see, both policy function $\pi_\theta$ and value function $V_\theta$ are using the same set of parameters $\theta$, and this is due to the ANN architecture that shares some parameters between both functions, despite having two different heads that output different results. Because the input of both functions is the same it makes sense to have some shared parameters to handle things like feature extraction. This network architecture was already mentioned in the paper of the A2C/A3C algorithm [15].

As in the A2C algorithm, each iteration, each of the $N$ (parallel) agents collects $T$ timesteps of data. Then, the loss is built on these $N \times T$ data steps and the optimizer is executed.



Optimizer + Connected Rollout Workers (x256)
Rollout Workers
~500 CPUs
Run episodes
• 80% against current bot
• 20% against mixture of past versions
Randomized game settings
Push data every 60s of gameplay
• Discount rewards across the 60s using generalized advantage estimation
Rollout Data Samples
Optimizer
1 p100 GPU
Compute Gradients
• Proximal Policy Optimization with Adam
• Batches of 4096 observations
• BPTT over 16 observations
Optimizers use NCCL2 to average gradients at every step.
Gradient Updates
Model Parameters (10M floats)
Model Parameters
Eval Workers
~2500 CPUs
Play in various environments for evaluation
• vs hardcoded "scripted" bot
• vs previous similar bots (used to compute Trueskill)
• vs self (for humans to watch and analyze)

**Figure 2.14:** OpenAI Rapid. Source: [19]

PPO was used by the same company that created it (OpenAI) to defeat the Dota 2 world champions on April 13th, 2019 [20]. This esports game is very complex and presents lots of challenges to AI.

To achieve this result the authors created a distributed system called Rapid [19] that decoupled the agents threads that collect data in different environments, from the optimizer process. As illustrated in Fig. 2.14 they used thousands of remote workers that collect data using a recent copy of the model parameters, and a GPUs cluster that runs the Adam [4] optimizer on the network weights. This system also has workers evaluating the trained model against reference agents.

### 2.2.4 Model-Based Reinforcement Learning

In some domains, the model of the environment is known by the agent, so it is possible to use techniques that look ahead into the future in some controlled way. This was the main approach in historical applications of AI to games, like in 1997, to defeat the chess champion, Garry Kasparov, as they used alpha-beta pruning that tries to decrease the number of nodes that are considered by the Min-Max algorithm in its search tree. This mark was achieved by IBM's DeepBlue [21] that contained 8000 handcrafted chess features, largely hand-tuned weights by human experts, and special-purpose chess processors that searched 200 million positions per second. In this section will be discussed an approach that combines search with both value approximation and policy gradients.

The game of Go, created 3000 years ago, is the oldest game in human history, with currently 40 million active players around the world. It was considered a grand challenge to AI due to the large size of state space, with more than $10^{170}$ legal positions. This makes it difficult to use a simple tree-search based approach like it was done in chess, as the branching factor is enormous. Despite of this problem, and opposite to the general opinion of Go community, in 2016 DeepMind's AlphaGo [22] was able to defeat Lee Sedol, the 18 times world champion of Go at the time.

*AlphaGo 2016*

AlphaGo used two CNNs to represent knowledge about the board state, as CNNs are able to build up abstract features of particular regions of the board in each convolutional layer.

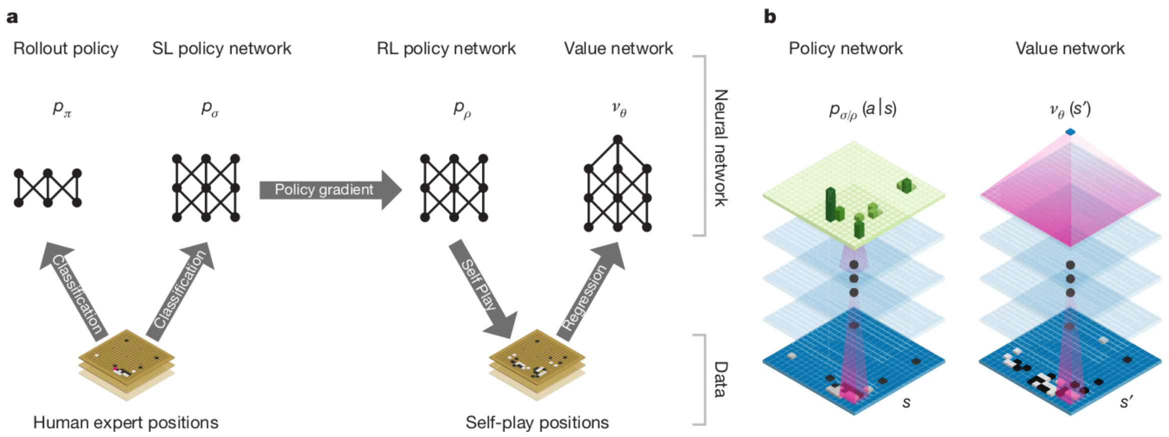


**Figure 2.15:** Training pipeline and the two CNNs. Source: [22]

The first CNN is the policy network that recommends the moves to play by a probability distribution over the legal actions (Fig. 2.15 b). The policy network was trained to imitate human experts by supervised learning techniques with a dataset containing the move chosen for each game state (Fig. 2.15 a). The second CNN is the value network that predicts the winner of the game given the board state (Fig. 2.15 b). The trained policy network is improved by applying policy gradient methods during complete simulated games of self-play, for now without any search, and those results were also used to train the value network by feeding the board positions and the corresponding winner of the game (Fig. 2.15 a).

The main idea is that now it is possible to use these networks to make the search space tractable. The policy network can be used to reduce the breath of the search tree by only considering a few moves recommended by the policy network. The value network can be used to reduce the depth of the search by truncating the search at any state, using the evaluation given by the value network to replace the remaining sub-tree.

Thus, it was possible to build a search tree, considering the most important paths, using an algorithm based on Monte Carlo Tree Search (MCTS) [23] that uses both networks in its heuristic.

*AlphaGo Zero 2017*

Later, DeepMind's team created a version of AlphaGo that excludes all human knowledge and uses a single residual CNN.

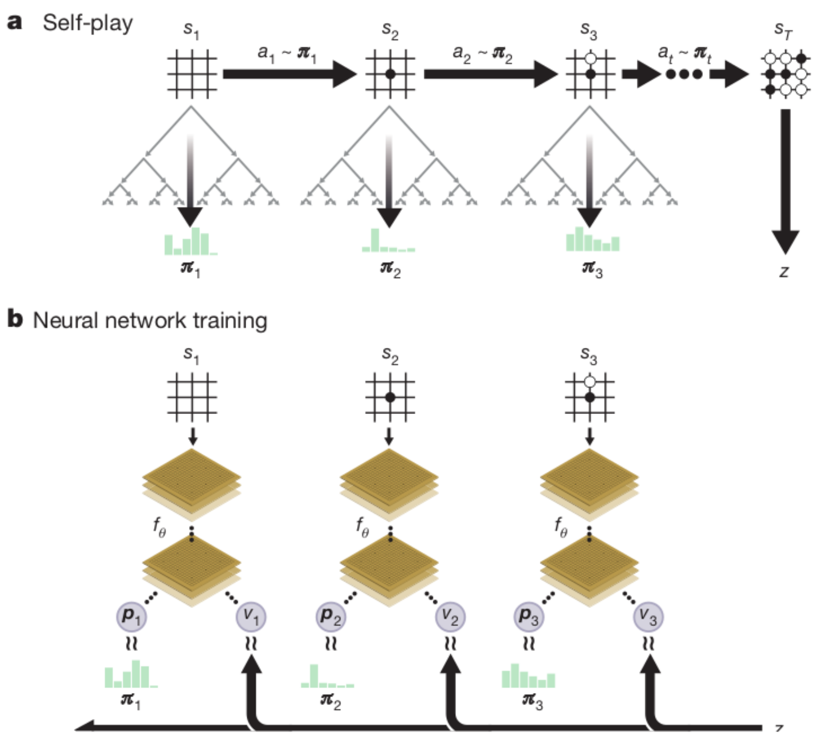


**Figure 2.16:** Self-play reinforcement learning in AlphaGo Zero. Source: [24]

The main idea of AlphaGo Zero [24] is that it generates high-quality data from playing against itself, because an MCTS is executed on every move of self-play using the CNN in its heuristic, as illustrated in Fig. 2.16 (a). Then, the CNN is trained to predict the actions chosen by the MCTS as well as to predict the winner of the game, on all board positions experienced during self-play, as shown in Fig. 2.16 (b). Therefore, in the next complete game of self-play, the improved CNN is used in the MCTS heuristic, and this process repeats. The idea is that each time this procedure iterates, a better player emerges and thus better quality data is generated for the CNN to learn.

Starting from zero knowledge and running for 3 days, using a single machine with 4 Tensor Processing Units (TPUs), this version was able to outperform the previous AlphaGo version that defeated Lee Sedol. An interesting fact is that AlphaGo Zero was able to discover known opening patterns, and as training proceeds even discovered new ones that humans didn't know, all starting with random weights.

*AlphaZero 2018*

In order to demonstrate that AlphaGo Zero can be a general algorithm, DeepMind's team created AlphaZero [25] and applied it to three different games. Beyond Go, they tested the algorithm with chess and shogi. Although chess valid board states are a lot less compared to Go, it is an interesting game because it was the most studied domain in the history of AI.

Shogi is called the Japanese chess, and is computationally harder, because it has a larger board and action space, as the captured pieces are then controlled by the opponent. Both chess and shogi state-of-the-art AIs, Stockfish and Elmo, respectively, are based on alpha-beta search and use hand-crafted features, unlike AlphaZero that does not use any human knowledge.

Among other differences that were made to accommodate various games, AlphaZero does not exploit the board symmetries to generate eight times as much data. This more general approach defeated the previous version, winning 61% of the games, and it can reach the level of the first AlphaGo version that defeated Lee Sedol, in only 30 hours, but using 5,000 first-generation TPUs to generate self-play games, and 16 second-generation TPUs to train the CNNs. In 4 hours it outperformed the previous world champion in chess, Stockfish. And in Shogi it took around 2 hours to reach the level of the world champion Elmo.

AlphaZero has shown that despite evaluating three orders of magnitude fewer positions than the opposing AIs and really evaluate them the right way, it is possible to get better performance and scalability over search time.

## 2.3 Tools

This section presents the tools used in this dissertation for the development of the DRL agents. Section 2.3.1 presents TensorFlow, section 2.3.2 describes Stable Baselines, and section 2.3.3 presents the Gym toolkit.

### 2.3.1 TensorFlow

Deep learning frameworks help to scale the computation of the algorithms as they facilitate the use of parallel processing in GPUs. They also can compute gradients automatically, allowing researchers to focus on the high level logic.

TensorFlow [26] is a deep learning framework developed by Google. In TensorFlow, numerical computations are expressed as a graph, in which the nodes are operations, and the edges are tensors that flow between nodes. Tensors can be seen as n-dimensional arrays. Operations can have any number of inputs and outputs and can be of three different types.

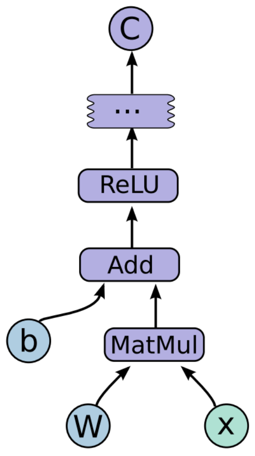


**Figure 2.17:** Hidden layer computation graph in TensorFlow. Source: [26]

The Fig. 2.17 represents the graph to compute the following hidden layer output:

$$h = ReLU(Wx + b) \tag{2.25}$$

where $x$ is the input, $W$ is a matrix of weights for all the neurons in this layer, $b$ is an array of biases for all the neurons, and ReLU is the activation function.

The different types of nodes are represented with different colors. The $b$ and $W$ are *variables*, $x$ is a *placeholder*, and the other nodes are *mathematical operations*.

*Variables* are nodes that retain their current value over multiple executions, and output it. They can be saved to disk, during and after training, in order to backup the network parameters. Also, the gradient updates will be applied by default to all *variables* in the graph.

*Placeholders* are nodes whose values are added into the computation during training. They are used to receive the inputs and the labels. They are defined by a data type and a shape.

Lastly, the *mathematical operations* nodes can compute operations such as addition, matrix multiplication, and ReLU.

TensorFlow supports many optimizers to compute the gradient of a loss function that should be represented in a graph node. The optimizer is also an operation node that, when evaluated, will calculate and apply the gradients to all the *variables* in the graph.

The graph is only a representation of the computation. Then, it can be deployed with a *session*. A *session* will bind the graph to an execution context as a CPU or GPU. Google also developed an integrated circuit called TPU to optimize tensor computations. The graph representation helps the framework to calculate the gradient, because every node has an attached gradient operation with respect to the *variables*.

A *session* receives the nodes we want to compute as well as a feed dictionary to fill in the *placeholders*. After the execution it can output the result of the nodes' evaluation.

### 2.3.2 Stable Baselines

Stable Baselines [27] is a framework that implements DRL algorithms using TensorFlow. It is a fork of OpenAI Baselines [28] with improved implementations, additional algorithms, and more readable code.

All DQN extensions (DDQN, PER, DNA), as well as the PPO algorithm, are implemented in Stable Baselines. For all the algorithms it is possible to use a MLP or CNN architecture depending on the input state format. The input format is defined using a Gym environment, as presented in the next section.

### 2.3.3 Gym

Gym [29] is a toolkit for RL that provides several already implemented environments, like the Atari games [1] and the simple CartPole [2], for testing RL algorithms. It also defines an interface for implementing custom environments. Stable Baselines algorithms are defined to

[1] `https://gym.openai.com/envs/#atari`

[2] `https://gym.openai.com/envs/CartPole-v1/`

be used in environments that follow the Gym interface. Gym environments are defined by an *observation space*, an *action space*, and a *step function*.

The *observation space* defines the input format in which the network will receive the state of the environment.

Since we are only going to deal with environments that have discrete *action spaces*, the *action space* will define the number of possible actions to be taken and, consequently, the number of neurons in the network output layer.

The *step function* is responsible for receiving the agent's action and executing it in the environment. Then, this function returns the next state of the environment, the agent's reward, a flag to indicate whether the episode is over, and possibly more additional information.

Some environment parameters, such as the *observation space* and the rewards system (defined in the *step function*), can and should be modified by the programmer, in order to understand which ones are capable of generating the best agents. This is what will be presented in the next chapter where DRL agents are developed and tested in custom Gym environments.

CHAPTER 3

# Development of DRL Agents for Computer Games

As mentioned in the introduction to this dissertation, computer games are used in the AI course at the University of Aveiro as scenarios to test the AI skills acquired by the students. They normally use pathfinding algorithms and custom heuristics to build an agent that will be competing with the other colleagues' agents for the final project. Each year the project game is different. The more recent ones were the Bomberman, Pac-Man, and the Snake. These games are developed by the teaching staff and can have custom rules.

In this chapter, we test DQN and PPO algorithms in one of these course games. But, as these games can be complex and may need a lot of computation power to train the agents, we decided to start with a simpler game like Tic-Tac-Toe.

In section 3.1 the various stages of the creation of a DRL agent that is able to play Tic-Tac-Toe against different skilled opponents are described.

After the success of the agents in Tic-Tac-Toe, the Pac-Man game was chosen to test the DRL algorithms. The different approaches in the implementation of the Pac-Man agents are described in section 3.2.

## 3.1 Tic-Tac-Toe DRL Agents

Tic-Tac-Toe is a two-player game where one player plays with crosses and the other with naughts in a $3 \times 3$ grid. Each turn, a player plays in a non-occupied cell. The game ends when one of the players is able to form a line (horizontal, vertical, or diagonal). This game is usually played by children, since it is not difficult to find an optimal strategy that always leads to a draw if played correctly by both players, thus turning the game not so interesting after learning the optimal strategy.

### 3.1.1 Problem specification

It is possible to implement an algorithm that plays this game perfectly, meaning it will never lose. But, we wanted to see if a DRL agent could learn this strategy by itself.

As two players are required to play Tic-Tac-Toe, some other agent will have to play against the training agent. That other agent will be designated as the environment agent since it plays the environment turn.

The learning process of the RL algorithms depends on the states that are experienced during training. In the case of a two-player game, the environment agent will influence the states that are experienced, so it will have a high impact on the learning process, possibly causing the DRL agent to overfit for the states visited against the environment agent.

Another project [30] has done experiments with DRL in Tic-Tac-Toe, but the DRL agent is evaluated only against the environment agent, so it is not possible to conclude whether the DRL agent will be optimal against other agents. Also, in that project, the agents are trained to play only as first player (meaning the one that plays in the first turn) or only as second player, which is quite limiting since the agent is unable to play all possible games.

Therefore, our goal will be to develop a DRL agent that is also optimal against an agent other than the environment agent that it was trained against. Also, the DRL agent should learn to play as both first and second player, so the first player will always be chosen at random during training.

### 3.1.2 Reference Agents

Two reference agents have been implemented to play against the DRL agent during the training and evaluation phases. These reference agents will be part of the Gym environment as they play in the environment turn.

*The Random Agent*

The Random agent simply chooses one of the valid plays at random. This agent should be easy to defeat, so we also implemented the perfect agent, described in the next section.

*The Min-Max Agent*

The perfect agent was implemented using the Min-Max algorithm. It finds an optimal action for a player but it can only be implemented when the model of the environment is known, as it needs to simulate future state transitions.

It starts by simulating all possible future states, then, when all final states are reached, it backpropagates the scores to the current state, selecting the action that is most valuable to each of the players. The player whose algorithm is trying to find the optimal action is called the maximizer and the other is the minimizer.

The Fig. 3.1 illustrates the selection of the optimal action, where the maximizer is the naught player. The scores, from -1 to 1, are in the perspective of the maximizer, and have the following meanings:

- 1 : Naught player wins

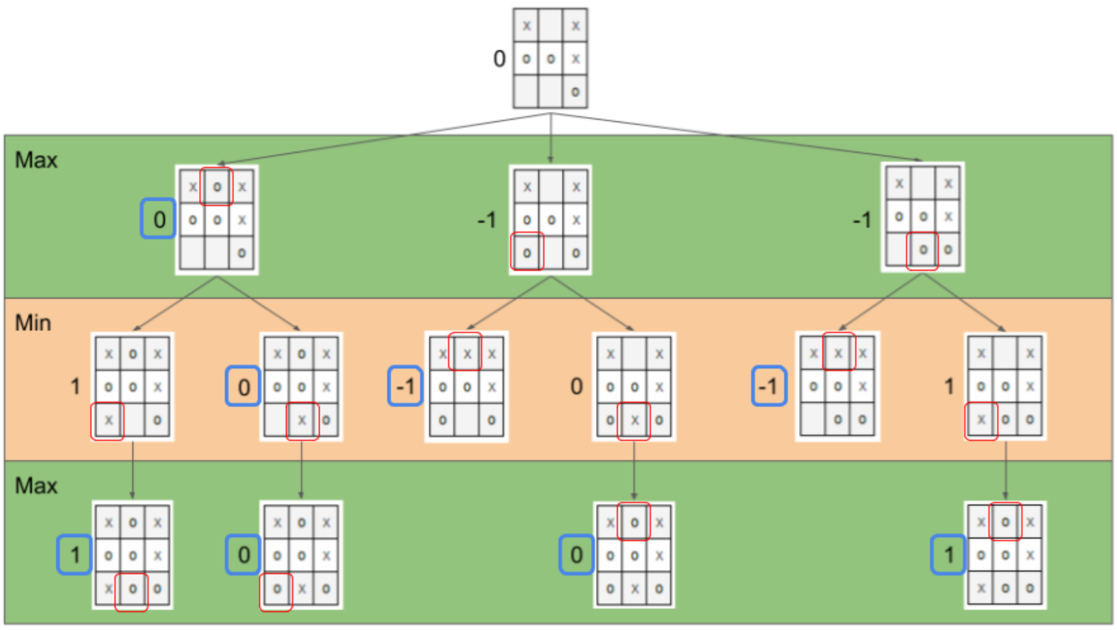


**Figure 3.1:** Min-Max algorithm searching the optimal action. In green are the maximizer turns, and in orange is the minimizer turn. Also, the scores of the selected actions are in blue, and all the actions taken are in red. Source: Adapted from [30]

- 0 : Draw
- -1 : Cross player wins

Therefore, when it is on the maximizer's turn, the algorithm chooses the action that obtained the maximum score. And, when it is on the minimizer's turn, the algorithm chooses the action that obtained the minimum score, thus assuming that the minimizer will play its best action, which in turn is the action that will minimize the score of the maximizer. If several possible plays are found with the same score, one of them is chosen at random.

### 3.1.3 Initial Setup

A customized Gym environment was created to test the DQN algorithm, using the implementation of the Stable Baselines library. Several parameters of the Gym environment and some algorithm hyperparameters have been modified to improve the agents' learning. The initial parameters will be explained in this section.

*Environment Agent*

In the beginning, the Random agent was used to play the environment turn.

*Observation Space*

The first representation used for the state of the environment was an array with the 9 cells of the board, where each cell can have a number from 0 to 2, with the following meanings:

- 0 : Empty cell
- 1 : Cross cell (X) - Agent

- 2 : Naught cell (O) - Environment

Therefore, the network will have the MLPs format, with an input layer of 9 neurons. This representation will be referred as Raw.

Note the agent will always play with crosses, regardless of whether it is the first player or the second player, as it must learn to play as both.

*Action Space*

There are 9 cells where the agent can play, so the network output must have 9 neurons. Numbers from 0 to 8 are used to represent each of the 9 actions.

A layer of unexpected complexity appeared due to the set of valid actions decreasing throughout the episode. That is, the first player starts with 9 valid actions, but the second player can no longer play in the cell occupied on the first turn and therefore only has 8 valid actions.

It would be possible to simply ignore the invalid actions and, therefore, in DQN, the experience buffer would only contain valid actions and the algorithm would possibly learn to play correctly. But, the Stable Baselines library automatically chooses actions during training and so this possibility could not be tried.

Therefore, a new possible episode outcome was introduced. In addition to the agent's victory, loss, or draw, the game also ends with an invalid play. This leads to the agent being forced to learn that it cannot play in already occupied cells.

*Rewards*

To obey the Gym interface, at each step of the environment the agent must receive a reward. In this case, it does not make sense to give a reward until the end of the episode, so it is given a reward of 0 in all the intermediate steps of the game. With that in consideration, the rewards are given as follows:

- 2 : Agent wins
- 1 : Draw
- 0 : Intermediate step
- -1 : Environment wins
- -2 : Agent plays an invalid action

*Hyperparameters*

To begin with, DQN was tested in its most basic implementation, without any extension (DDQN, PER, DNA). The hyperparameters chosen to start were the default of the Stable Baselines library, excluding the network architecture and the discount factor $\gamma$.

The default architecture has two hidden layers of 64 neurons, but as the problem seemed to be simple, we started with a network with two hidden layers of 16 neurons.

The default $\gamma$ is 0.99, but the reward is obtained only at the end of the game, and all intermediate rewards are 0. So, a $\gamma$ less than 1 would cause the agent to receive a greater reward the sooner the game is over, because the more steps pass, the more the final reward will be discounted. As in this game there is no advantage to ending the game sooner, a $\gamma$ equal to 1 was used.

Also, the duration of the training in Stable Baselines is measured in the number of steps, but as in this environment the reward is given only at the end of each episode, we decided to always train for a certain number of episodes. Various numbers of training episodes were attempted, and 40,000 appeared to be sufficient for agents to converge on all variations of the parameters.

### 3.1.4 Evaluation Metrics

The DRL agent is evaluated against Random and Min-Max reference agents. It is also tested against itself, which should result in 100% draws, if the agent has already converged on something close to the optimum.

During the training of an agent, it is evaluated 10 times. Each evaluation consists of 5000 test episodes against the Random agent and 5000 against the Min-Max agent. Against itself it is only tested in 500 episodes, just to double-check if the agent has converged, and this result is not used in the score metric that will be presented below.

Since an agent being the first player or the second player has a big impact on the outcome of the game, as demonstrated in the next section, the test episodes against Random and Min-Max are played half as first player and the other half as second player. That is, in 2500 test episodes the DRL agent plays first and in the other 2500 plays second. For each training situation, the percentages of wins, draws, losses, and invalids are obtained, from the perspective of the evaluated agent. Table 3.1 shows the results of an example evaluation where the agent under evaluation is in bold.

| N Episodes | First Player | Second Player | Wins | Draws | Losses | Invalids |
|---|---|---|---|---|---|---|
| 2500 | **Agent** | Random | 89.96 | 7.64 | 2.2 | 0.2 |
| 2500 | Random | **Agent** | 78.08 | 18.92 | 2.04 | 0.96 |
| 2500 | **Agent** | Min-Max | 0.0 | 90.72 | 7.92 | 1.36 |
| 2500 | Min-Max | **Agent** | 0.0 | 90.88 | 9.12 | 0.0 |
| 500 | **Agent** | **Agent** | 0.0 | 100.0 | 0.0 | 0.0 |

**Table 3.1:** Example evaluation results.

As there are many results generated by a single evaluation, a score metric was created to summarize the results of an evaluation. For each test situation (each line), against Random and Min-Max agents, the results are converted to a partial score given by:

$$Partial_{score} = c_{win} \times Wins + c_{draw} \times Draws + c_{loss} \times Losses + c_{invalid} \times Invalids \quad (3.1)$$

where the coefficients $c$ depend on the opponent and are described in Table 3.2. Note that the best possible result against the Min-Max agent is a draw, so it was decided that this result should be worth as much as a victory against the Random agent, in order to maintain the scale range between -200 and 100 for all partial scores.

| Opponent | $c_{win}$ | $c_{draw}$ | $c_{loss}$ | $c_{invalid}$ |
|---|---|---|---|---|
| Random | 1 | 0.5 | -1 | -2 |
| Min-Max |  | 1 | -1 | -2 |

**Table 3.2:** Partial score coefficients.

Then, the score metric is given by the average of the partial scores, and can also be in the range of -200 to 100. For the results described in the example evaluation in Table 3.1, the score obtained is calculated in Table 3.3.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| **Agent** | Random | 89.96 | 7.64 | 2.2 | 0.2 | 91.18 |  |
| Random | **Agent** | 78.08 | 18.92 | 2.04 | 0.96 | 83.58 | 84.15 |
| **Agent** | Min-Max | 0.0 | 90.72 | 7.92 | 1.36 | 80.08 |  |
| Min-Max | **Agent** | 0.0 | 90.88 | 9.12 | 0.0 | 81.76 |  |

**Table 3.3:** Score calculation for an example evaluation.

Of the 10 evaluations that are made during training, the one with the best score is chosen to represent the training, and the agent's network parameters are saved.

Also, as the agent's learning depends on the states it is witnessing during training, there is a possibility that the results will vary too much between training sessions, even maintaining the parameters of the environment, as the data distributions of states and rewards are constantly changing during training. Therefore, for each set of parameters of the environment, 5 repetitions of the same training are performed and an average is made taking into account the best scores of each repetition.

*Baseline Scores*

In order for the agent's performance to be compared with the performance of the reference agents, the Random agent and the Min-Max agent were subjected to the same evaluation. Table 3.4 shows the results of the Random agent and Table 3.5 shows the results of the Min-Max agent, where the agent under evaluation is in bold.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| **Random** | Random | 58.55 | 12.54 | 28.91 | 0.00 | 35.92 |  |
| Random | **Random** | 28.87 | 12.77 | 58.36 | 0.00 | $-23.11$ | -34.10 |
| **Random** | Min-Max | 0.00 | 22.20 | 77.80 | 0.00 | $-55.60$ |  |
| Min-Max | **Random** | 0.00 | 3.19 | 96.81 | 0.00 | $-93.62$ |  |

**Table 3.4:** Random agent evaluation results.

From the results of the Random agent playing against itself, it is clear that being the first player or the second player makes a lot of difference in the outcome of the game. Also, the Random agent, as first player, draws 22% of the games against Min-Max, and as second player, only draws 3%. Therefore, the DRL agent is expected to achieve much higher draw percentages against Min-Max compared to the Random agent, and consequently should also obtain a superior score.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| **Min-Max** | Random | 96.81 | 3.19 | 0.00 | 0.00 | 98.41 | |
| Random | **Min-Max** | 77.80 | 22.20 | 0.00 | 0.00 | 88.90 | 96.83 |
| **Min-Max** | Min-Max | 0.00 | 100.00 | 0.00 | 0.00 | 100.00 | |
| Min-Max | **Min-Max** | 0.00 | 100.00 | 0.00 | 0.00 | 100.00 | |

**Table 3.5:** Min-Max agent evaluation results.

Symmetrically, the Min-Max agent, as first player, wins 97% of the games against the Random agent, and as second player, only wins 78% of the games. It is possible that the DRL agent will be able to have more percentages of victories against the Random agent, and perhaps achieve an even higher score than the Min-Max agent.

### 3.1.5 Learning vs Random

As mentioned in section 3.1.3, the initial agent used to play in the environment turn was the Random agent and the initial input was the 9-position raw array.

With this configuration, the average score of the 5 trained DQN agents was -47.17, which is even worse than the Random agent score, since these agents can make invalid plays.

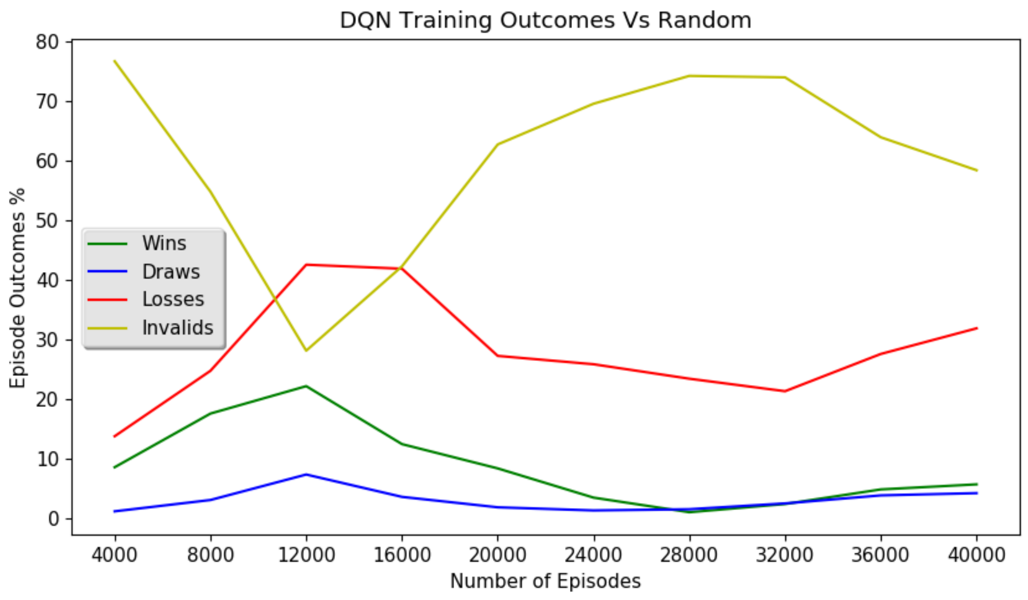


**Figure 3.2:** Training outcomes percentages of a DQN agent with Raw input.

Figure 3.2 shows the outcomes percentages during training of one of the trained DQN agents. It seems that this agent has not managed to learn that it cannot play in already occupied cells. This can happen because the input, although categorical, is represented with numbers from 0 to 2 which can lead the network to think that there is an order relationship between these numbers. In fact, there is no relation of order or hierarchy as these numbers represent only 3 different states for each of the 9 cells. For example, a cell with a cross is no better than an empty cell, and vice versa.

Therefore, in the next section the input will be modified to better represent the game state.

*One-Hot Encoded Input*

To represent categorical values, it is common to use a technique called One-Hot Encoding. It allows to distinguish all the categories using a vector of the size of the number of categories, where all the values are 0, except for the represented category that is 1.

| | | |
|---|---|---|
| O | X | O |
| | O | |
| | X | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 0 | 2 | 0 | 0 | 1 | 0 |

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

**Figure 3.3:** One-Hot Encoding example. The game board is at the top, the Raw input is in the middle, and the corresponding One-Hot Encoded input is in the bottom.

In this case, each of the 9 cells in the game board can have 3 different states. Therefore, an array of 27 cells can be used, in which the first 9 cells represent the empty cells, the second 9 represent the cells with a cross, and the last 9 represent the cells with a naught. An example of this input format is in Figure 3.3.

The improvements in the score are notable with this input. The average score of the 5 trained agents was 21.2, much higher compared to the previous average score of -47.17.

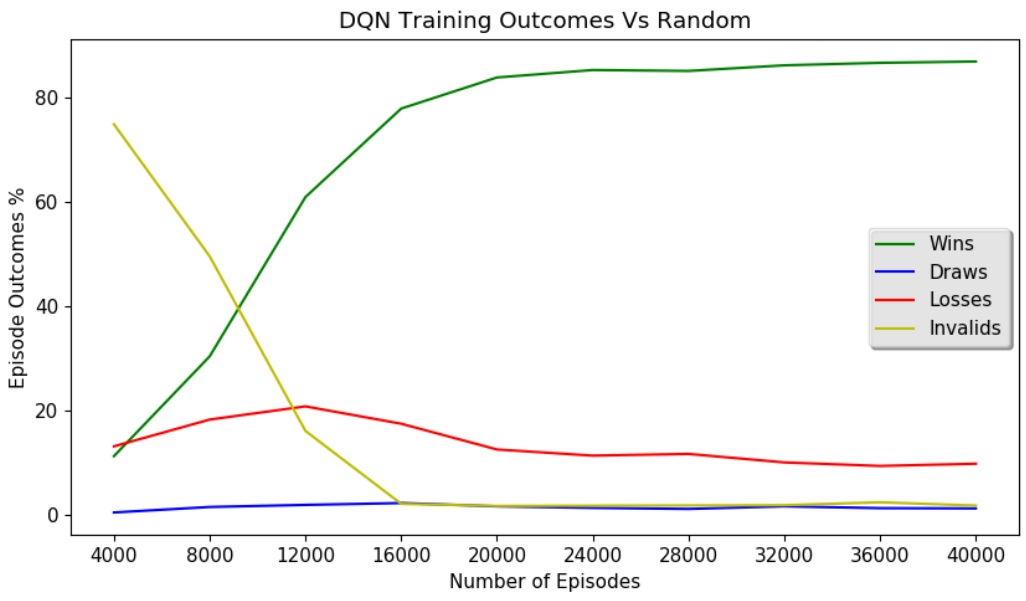


**Figure 3.4:** Training outcomes percentages of a DQN agent with One-Hot Encoded input.

The agent whose score is closest to the average was chosen, and observing its results in Figure 3.4, it appears that it successfully learned not to make invalid plays, most likely because it is now possible to make a better distinction between the states of the environment. Also, during training, the agent wins consistently against the Random player, although it still has some percentages of losses. Note that at the end of the training there are still small percentages of invalid plays due to the exploration factor inherent in the DQN algorithm.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| DQN | Random | 95.72 | 2.16 | 2.12 | 0.0 | 94.68 | |
| Random | DQN | 79.32 | 5.44 | 15.24 | 0.0 | 66.8 | 22.81 |
| DQN | Min-Max | 0.0 | 48.96 | 51.04 | 0.0 | −2.08 | |
| Min-Max | DQN | 0.0 | 15.92 | 84.08 | 0.0 | −68.16 | |

**Table 3.6:** Evaluation results of a DQN agent with One-Hot Encoded input.

Table 3.6 shows its best evaluation, where it can be seen that, as expected, this DQN agent learned not to make invalid plays, and has a very high percentage of victories against the Random player, despite still losing many times mainly when playing as a second player. Against Min-Max, it is normal for the agent to lose most games since the states visited against Min-Max should rarely be experienced during training against Random.

*Larger network*

At this point, a network with more neurons was tried to see if this way it was possible to increase the score. Instead of using layers of 16 neurons, we started using a network with two hidden layers with 32 neurons each.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| DQN | Random | 97.76 | 1.2 | 1.04 | 0.0 | 97.32 | |
| Random | DQN | 87.12 | 3.84 | 9.04 | 0.0 | 80.0 | 46.13 |
| DQN | Min-Max | 0.0 | 74.44 | 25.56 | 0.0 | 48.88 | |
| Min-Max | DQN | 0.0 | 29.16 | 70.84 | 0.0 | −41.68 | |

**Table 3.7:** Evaluation results of a DQN agent with two hidden layers of 32 neurons.

The increase in the average score was notable, going from 21.2 to 43.5. Table 3.7 shows the best evaluation of the agent whose score is closest to the average. It is possible to verify that the results against the Random agent as second player have improved, but in general, there are still losses against the Random agent. It would be expected that there were practically no losses against the Random agent since the training is against this agent, but if we compare the percentages of wins against the Random agent with the same percentages obtained by the Min-Max agent in Table 3.5, we can see that this DQN agent obtains higher percentages of wins than the Min-Max agent itself. It seems that the losses come from the fact that the agent becomes greedy and risks winning since it is playing against the Random agent. Figure 3.5 illustrates an example where it seems worth to play greedy since, in the next turn, the Random player has an 80% chance of playing in a cell where it does not win, allowing the DQN agent to win.

| | | |
|---|---|---|
| X | O | |
| *a* | O | |
| | *b* | |

**Figure 3.5:** Gready move example: *a* represents the gready move, and *b* represents the safe move.

Also, looking at table 3.7 it is possible to verify that the results against the Min-Max agent improved, but the agent continues to have many losses, as expected since its training was against the Random agent.

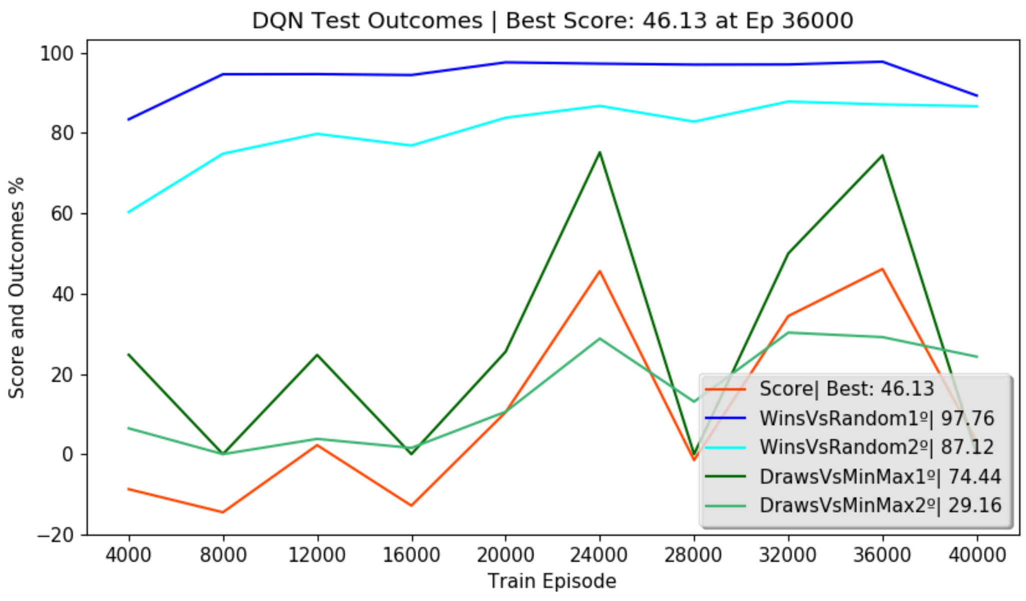


**Figure 3.6:** Evaluation outcomes percentages of a DQN agent with two hidden layers of 32 neurons.

Figure 3.6 shows the score and some of the main results obtained in each of the 10 evaluations made during training. It is possible to observe that the knowledge acquired to play against the Min-Max agent is easily forgotten with the progression of the training, since the DQN agent is training against the Random agent, and ends up rarely experiencing the states visited in games against the Min-Max agent.

*D3QN with PER*

Before moving on to training directly against Min-Max, the use of the state-of-the-art DQN implementation was tried, i.e Dueling Double Deep Q-Network (D3QN) with Prioritized Experience Replay (PER). The average score improved considerably, from 43.5 to 76.6. This agent will be designated by D3QN.

Looking at the evolution of the score of one of the trained agents shown in Figure 3.7, it appears that the D3QN agent is able to maintain the results against the Min-Max agent, instead of getting worse with the training progress as it happened in the previous version.

This is possibly due to the PER which makes the rarest and most unexpected experiences more likely to be chosen for the agent's network to train. In this way, the transitions that

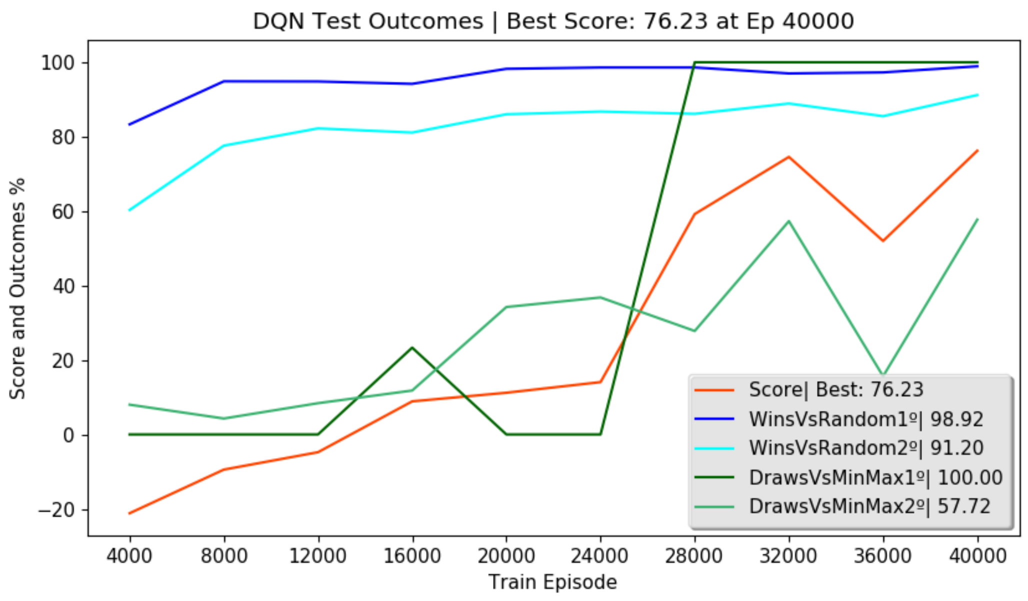


**Figure 3.7:** Evaluation outcomes percentages of a D3QN agent.

help the agent to play against Min-Max that rarely happen against the Random agent, are no longer forgotten with the training progress, as they continue to be sampled from the replay buffer.

The agent achieved 100% draws against the Min-Max agent, as first player. However, it was not able to generalize the learned strategies from the training against the Random agent, for the games against the Min-Max agent as second player, achieving only 57.72% draws and 42.28% losses, as shown in Table 3.8.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 98.92 | 0.84 | 0.24 | 0.0 | 99.1 | |
| Random | D3QN | 91.2 | 5.32 | 3.48 | 0.0 | 90.38 | 76.23 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 57.72 | 42.28 | 0.0 | 15.44 | |

**Table 3.8:** Evaluation results of a D3QN agent.

Table 3.8 also shows that the games against the Random agent have also improved, but the losses remain, probably due to the agent's greedy behavior, as explained in the previous section.

### 3.1.6 Learning vs Min-Max

The agents were able to learn a lot when training against the Random player, but it seems to be difficult for them to achieve 100% draws in the games against the Min-Max agent. So, we tried training with the Min-Max agent playing the environment turn.

The average score increased from 76.6 to 80.6, but looking at figure 3.8 that shows the evolution of the score of one of the trained agents, we can see that the problem of the trained agents overfitting to the games against the environment agent remains. This time, the trained

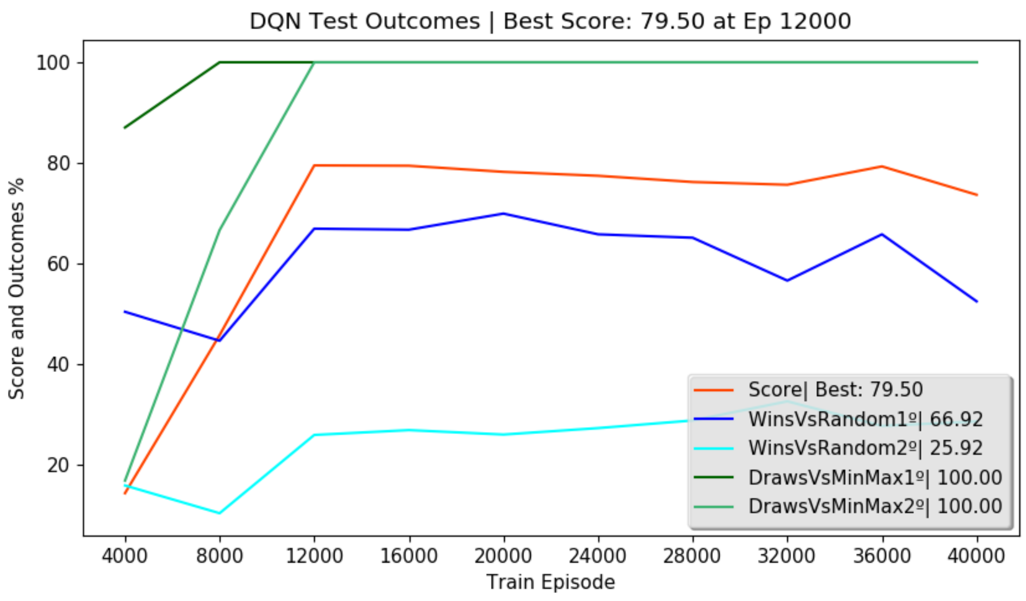


**Figure 3.8:** Evaluation outcomes percentages of a D3QN agent during training vs Min-Max.

agent quickly learns to play against Min-Max, that is the agent of the training environment but is not able to generalize to play against the Random agent. This is most likely due to the set of states visited during training against the Min-Max agent that is scarcely intercepted by the set of boards visited in the evaluations against the Random agent. In fact, all the states the agent experiences during training end up leading to a draw, at best, therefore never getting to win during training.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 66.92 | 28.76 | 4.32 | 0.0 | 76.98 | |
| Random | D3QN | 25.92 | 59.44 | 14.64 | 0.0 | 41.0 | 79.5 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |

**Table 3.9:** Evaluation results of a D3QN agent learning vs Min-Max agent.

Table 3.9 presents the results of the best evaluation of the same agent in Figure 3.8. We can see that in fact, this agent has difficulties in winning against the Random agent, and as a second player it ends up drawing most of the time.

*2D Input*

An input in 2D format was tried because it is more representative of the state of the environment and maybe in this way the agent could better generalize the strategies learned for states not yet visited.

The 2D representation of the game state was divided into 3 channels, one for empty cells, one for cells with a cross, and the other for cells with a naught, as illustrated in Fig. 3.9.

Therefore, a convolutional layer with 64 neurons was added after the ANN's input, and the two fully connected hidden layers of 32 neurons were maintained. The convolutional layer

|   |   |   |
|---|---|---|
| O | X | O |
|   | O |   |
|   | X |   |

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 0 | 1 |

|   |   |   |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |

|   |   |   |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

**Figure 3.9:** 2D input example. The game board is at the top and the corresponding 2D input is in the bottom.

has a 3 × 3 filter with stride 1, and padding "SAME". Other combinations of filter size and padding have been tried, but this one has had the best results. In TensorFlow, the "SAME" padding means that the size of the dimensions of the input image that do not correspond to the channels are preserved in the output, by adding zeros in the input image if necessary. In this case, zeros are added all around the input image and the output of the convolutional layer is a 3 × 3 × 64 image.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 75.08 | 20.04 | 4.88 | 0.0 | 80.22 | |
| Random | D3QN | 35.64 | 59.56 | 4.80 | 0.0 | 60.62 | 85.21 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |

**Table 3.10:** Evaluation results of a D3QN agent learning vs Min-Max agent with 2D input.

The average score increased from 80.66 to 85.26. Table 3.10 contains the results of one of the tested agents, and we can see that the percentages of victories against the Random agent increased in general, as well as the losses as second player decreased considerably.

*Learning vs Min-Max with randomness*

It seems that playing against just one agent during training, the D3QN agent is not able to obtain a score at the level obtained by the Min-Max agent in Table 3.5. Therefore, a mixture of the two reference agents was used during the training phase, that is, the Min-Max agent with a randomness parameter, which defines its percentage of random moves.

Training against the Min-Max agent with 20% randomness, the average score increased from 85.26 to 98.48, which is higher than the score obtained by the Min-Max agent itself, which was 96.83. Experiments were also carried out with 50% randomness but did not have as good results.

In Figure 3.10 we can see the evolution of the score of the best trained agent, whose score was 98.61. It appears that the agent quickly mastered all the represented test situations.

Also, from Table 3.11 we see that this agent seems to have optimal results since it draws 100% of the time against the Min-Max agent and never loses against the Random agent.

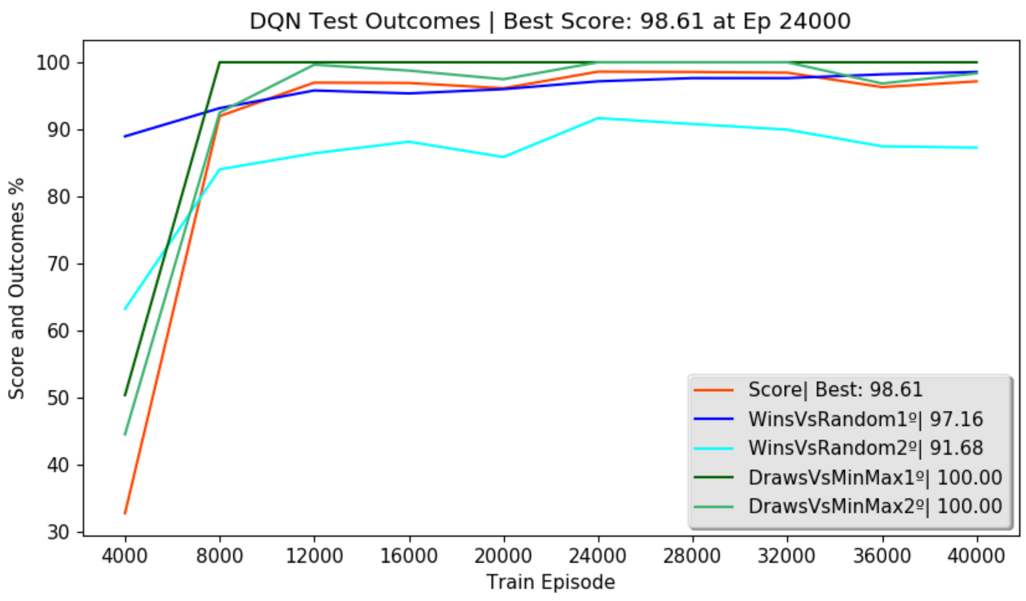


**Figure 3.10:** Evaluation outcomes percentages of a D3QN agent during training vs Min-Max with 20% randomness.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 97.16 | 2.84 | 0.0 | 0.0 | 98.58 | |
| Random | D3QN | 91.68 | 8.32 | 0.0 | 0.0 | 95.84 | 98.61 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |

**Table 3.11:** Evaluation results of the best D3QN agent learning vs Min-Max with 20% randomness.

Therefore, it seems that this agent was able to learn a strategy against the Random agent in which it does not need to be greedy to maximize its rewards.

Note that the percentages of victories against the Random agent are higher than those obtained by the Min-Max agent, therefore this agent could be considered even better than the Min-Max agent. This is due to the Min-Max agent assuming that its opponent is also a perfect player and so it does not exploit its weaknesses. For example, in the first turn, the Min-Max agent simulates all possible future states and concludes that the best result is a draw for all the possible plays, so it chooses a random one since they all lead to the same outcome. The D3QN agent, on the other hand, takes advantage of the fact that it is playing against a random player to achieve the maximum possible victories.

It was possible for the agent to reach this level of score, because it is able to explore a wide range of states during training, which includes the states visited in the games against both test agents. As we can see in Figure 3.11, the agent experiences not only draws during training as it did in the training against the pure Min-Max agent but also experiences victories.

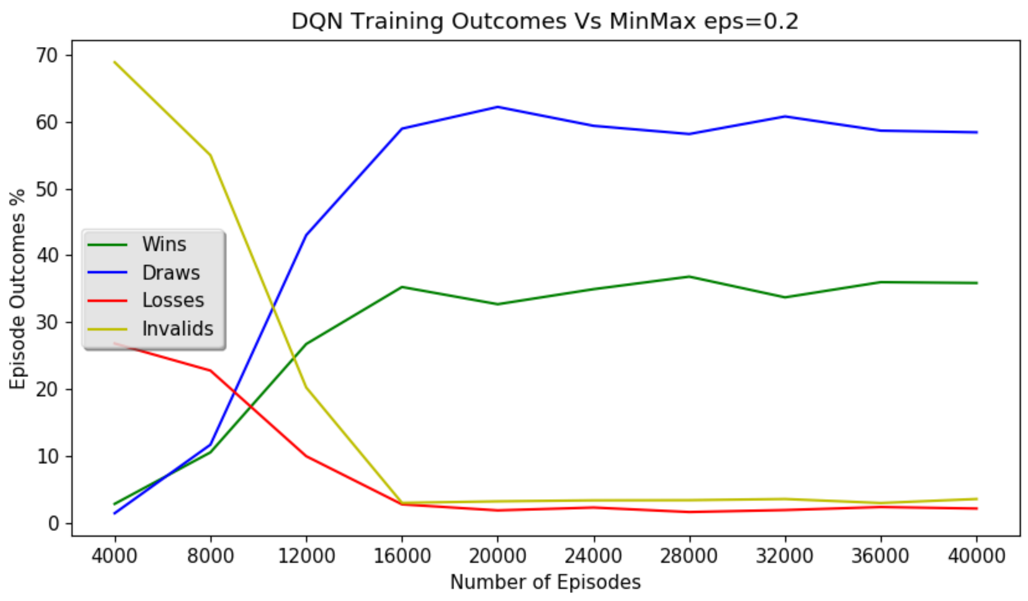


**Figure 3.11:** Training outcomes percentages of a D3QN agent learning vs Min-Max with 20% randomness.

### 3.1.7 Learning vs Self

We thought it might be interesting to train an agent against itself, to see if it could learn an optimal strategy without ever playing against the Min-Max agent during training. For this, after each of the 10 evaluations that are made during training, the environment agent is updated to the current version of the training agent. That is, the environment agent starts random, since its network is initialized with random parameters, and after each evaluation, the parameters of the environment agent network are updated.

The average of the 5 trained agents was 92.92, which is higher than the best average obtained learning against the Random agent or the pure Min-Max agent. The best of these agents had a score of 97.2 and its results are shown in Table 3.12.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 95.88 | 4.12 | 0.0 | 0.0 | 97.94 | |
| Random | D3QN | 82.24 | 17.6 | 0.16 | 0.0 | 90.88 | 97.2 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |

**Table 3.12:** Evaluation results of the best D3QN agent learning vs itself.

We can see that this agent was able to learn to play optimally against the Min-Max agent, even though it never played against it during training. But, as a second player, this agent is not able to have optimal results against the Random agent.

Analyzing Figure 3.12 which shows the percentages of outcomes during the agent training, we see that after a certain episode, most games end in a draw, just like in the training against the pure Min-Max agent. Therefore, next we will try to apply a percentage of randomness in

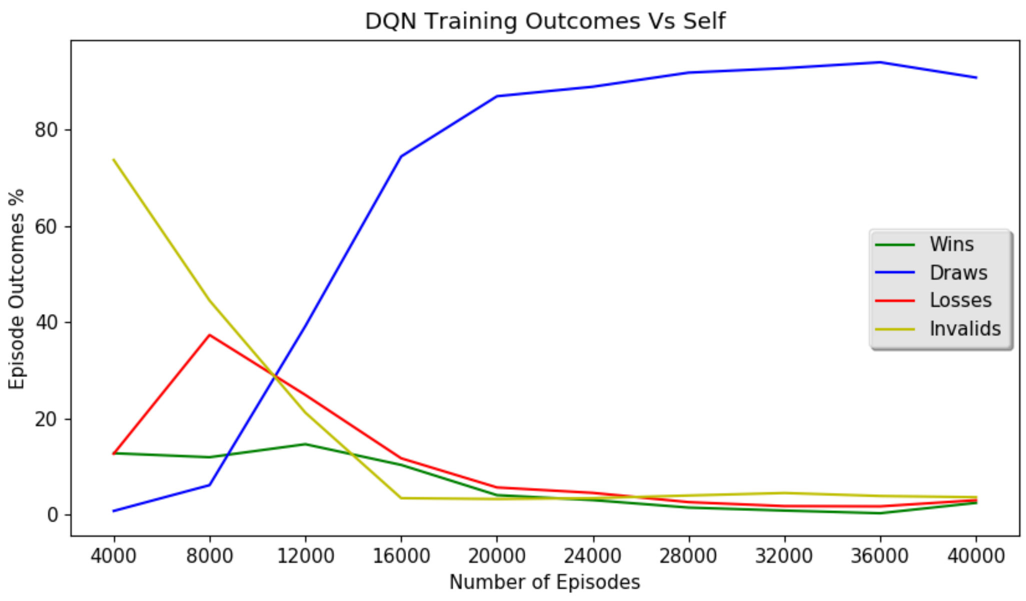


**Figure 3.12:** Training outcomes percentages of the best D3QN agent learning vs itself.

the training against itself, just as we did in the training against the pure Min-Max agent.

*Learning vs Self with randomness*

Applying the same 20% randomness percentage to the training against itself, the average score increased from 92.92 to 98.36.

| First Player | Second Player | Wins | Draws | Losses | Invalids | Partial Score | Score |
|---|---|---|---|---|---|---|---|
| D3QN | Random | 98.92 | 1.08 | 0.0 | 0.0 | 99.46 | |
| Random | D3QN | 90.28 | 9.72 | 0.0 | 0.0 | 95.14 | 98.65 |
| D3QN | Min-Max | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |
| Min-Max | D3QN | 0.0 | 100.0 | 0.0 | 0.0 | 100.0 | |

**Table 3.13:** Evaluation results of the best D3QN agent learning vs itself with 20% randomness.

The best trained agent obtained a score of 98.65, and its results are shown in Table 3.13. We can see that this agent is at the same level as the best agent of the training against Min-Max with randomness, described in section 3.1.6. So, this agent is also able to obtain perfect results against the Min-Max agent and at the same time obtain very high percentages of victories against the Random agent without ever losing, thus surpassing the performance of the Min-Max agent.

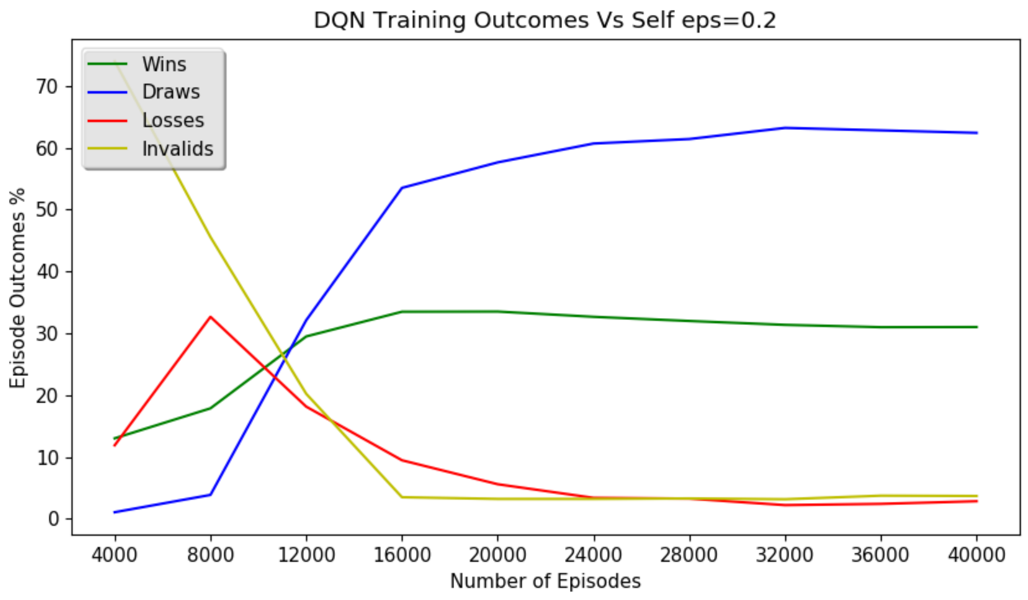


**Figure 3.13:** Training outcomes percentages of the best D3QN agent learning vs itself with 20% randomness.

In Figure 3.13 we can see that the outcomes during the training of this agent are also identical to the ones obtained by the best agent of the training against Min-Max with randomness. So, this agent now manages to explore more states that lead to victories during training, thus allowing it to obtain better results against the Random agent, compared to the previous version without randomness.

For all the improvements presented, Student's t-tests were carried out to check whether the differences in the mean scores were statistically significant. It was found that only in the transition to learning against the Min-Max agent, in section 3.1.6, the score improvement has no significance. But, in this case, the increase in the score is not relevant, what is relevant is the fact that the agent overfits the states of the new environment agent (Min-Max) and worsens its performance against the Random agent.

The code developed to run the described experiments can be found in a public repository on GitHub [1]. In the same repository are also the model parameters and the results of all the agents mentioned in this work, as well as the code that calculates the Student's t-tests.

In the other mentioned GitHub project [30], which also applies DRL algorithms in Tic-Tac-Toe, the author trains the agents only for one position (first player or second player), and also only obtains the results against the training environment agent. In comparison, we end up getting much more robust agents, in the sense that they can play as first and second player, and are evaluated against two distinct reference agents.

## 3.2 Pac-Man DRL Agents

Pac-Man is a very popular arcade game that was released in 1980. The player controls the Pac-Man that has to eat all the energies (points) of the maze, avoiding the four ghosts that chase it. When Pac-Man collides with a ghost, it loses a life and returns to its original position. By eating the big energies (boosts), the Pac-Man gains the power to eat the ghosts for some time, earning more points for that. When a ghost is eaten, it returns to its initial position. The game ends when Pac-Man eats all energies (victory), or when it loses all lives. The goal is to have the highest possible score.

### 3.2.1 Problem specification

As mentioned in the introduction to chapter 3, this game was used in the AI course at the University of Aveiro to test the algorithms developed by the students, being the game designed by the teaching staff. The source code of the project is public on a GitHub repository [2] and the visualization of the game is represented in Fig. 3.14. In this version, the general rules of the game are the same, but it contains customizable parameters and has some specificities such as the difficulty of the ghosts, the score criteria, among others.

[1] `https://github.com/mdaraujo/deep-rl-tictactoe`

[2] `https://github.com/dgomes/iia-ia-pacman`

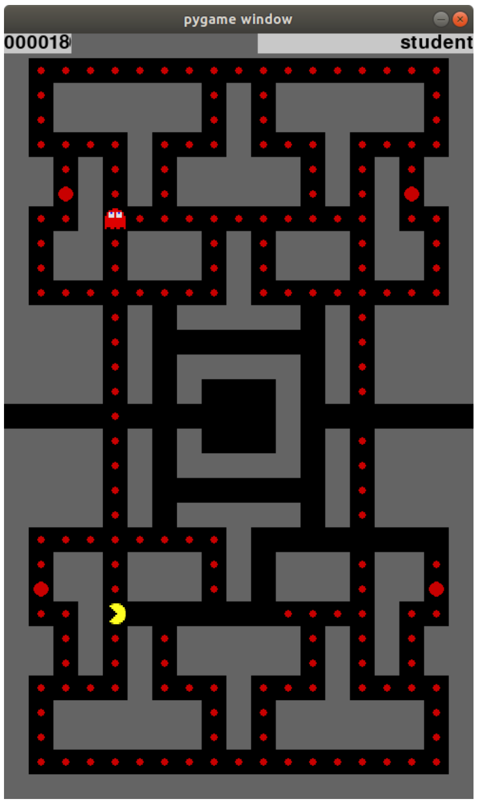


**Figure 3.14:** The Pac-Man game visualizer in Map 1.

*Parameters*

The following parameters are customizable:

- Number of ghosts: Can go from 0 to 4 (Default is 1)
- Level of the ghosts: The difficulty level of the ghosts can go from 0 to 3 (Default is 1)
- Map: The maze format can be passed as a BMP file (Default is Map 1, represented in Fig. 3.14)
- Number of lives (Default is 3)
- Step timeout: The game also ends when the number of steps is equal to the step timeout (Default is 3000)

The number of lives, as well as the step timeout, will be kept default during all experiments, but the remaining parameters will be modified.

*Ghosts*

Pac-Man and ghosts move through the maze at the same speed. Also, the duration of the boost power is equal to 30 steps. Under this effect, the ghosts flee from Pac-Man at half speed. When the ghosts are in this eatable state, they will be designated as zombies.

The functioning algorithm of ghosts depends on their level. At all levels, the ghosts keep a memory of their previous positions, in order to give priority to unvisited cells. Also, the ghosts have certain visibility that allows them to see the Pac-Man (in a straight line, with no walls in between) and start chasing it.

At level 0, ghosts have 2 cell visibility, and when in zombie mode, they flee in a random direction.

At level 1, ghosts have 4 cell visibility, so they are able to maintain chase even when Pac-Man changes direction. When in zombie mode, they flee in the opposite direction to Pac-Man.

At level 2, ghosts have 8 cell visibility. When in zombie mode, they flee in the opposite direction to Pac-Man. Also, they give priority to spreading, that is, moving away from other ghosts.

At level 3, the ghosts have visibility of only 6 cells but plan their actions taking into account the position of the Pac-Man (if it is visible) and the position of the other ghosts, thus managing to enclose the Pac-Man making it inevitable that it will lose a life.

*Score*

The agent's score is the sum of the points earned during a game. Points are given as follows:

- 1 point for each energy eaten
- 10 points for each boost eaten
- 50 points for each ghost eaten
- Bonus in case of victory equivalent to 1 point for each remaining 5 steps until reaching the step timeout

*Game State*

At all steps, the agent has access to the complete state of the game, which consists of:

- Step count
- Score
- Number of lives
- Pac-Man's position
- A list of information for each ghost, containing their position, a flag indicating if they are in zombie mode, and the remaining duration of the zombie mode
- A list of positions containing energies
- A list of positions containing boosts
- An object that represents the map and contains information about the walls and spawning positions of Pac-Man and ghosts

*Actions*

There are 4 possible actions:

- w: move up
- a: move left
- s: move down
- d: move right

Pac-Man moves by inertia. If it encounters an obstacle, it can stand in that position until an action with another direction is given. Also, the game runs at 10 frames per second, in a client-server paradigm, so if the agent (client) does not make an action in 100 milliseconds, Pac-Man will move by inertia.

*Evaluation*

The evaluation of the agents made in the first delivery of the AI course mentioned before consists of a set of games with different difficulties, described in Table 3.14.

| Map | Number of Ghosts | Ghosts Level | Number of Games |
|---|---|---|---|
| Map 1 | 0 | 0 | 1 |
| Map 1 | 1 | 1 | 10 |
| Map 1 | 2 | 2 | 10 |
| Map 1 | 4 | 0 | 10 |
| Map 1 | 4 | 1 | 11 |
| Map 1 | 4 | 2 | 9 |
| Map 2 | 1 | 1 | 10 |
| Map 2 | 2 | 1 | 12 |
| Map 2 | 4 | 0 | 12 |
| Map 2 | 4 | 1 | 10 |
| Map 2 | 4 | 2 | 10 |

**Table 3.14:** Games played in an agent's evaluation.

In total, an agent is evaluated on 105 games and their average score is calculated. In this dissertation, the objective will be to develop DRL agents that have the maximum possible score in this evaluation and to compare the performance of those agents with the students' agents.

### 3.2.2 Initial Setup

A custom Gym environment was created, using the source code of the AI course game, in order to use the Stable Baselines DRL algorithms (DQN [3] and PPO [4]).

The only modification to the source code of the course game, was the adaptation of the function that computed the game step asynchronously, in the client-server paradigm, for a synchronous function, maintaining both possibilities, and allowing the agent to train synchronously, in a Gym environment in which the step is executed right after its action. The agent model is saved throughout the training and can be used to run the game asynchronously in the original paradigm.

Several parameters of the Gym environment (including the game customizable parameters) and some algorithm hyperparameters have been modified to improve the agents' learning. The initial parameters will be explained in this section.

*Observation Space*

Since most of the information that composes the state of the game is positional, the way the DRL agent receives the information is through an image. Therefore, information about the position of the Pac-Man, the position of ghosts and zombies, the location of energies and boosts, and the position of the walls were initially encoded in a 2D image with one channel.

[3] `https://stable-baselines.readthedocs.io/en/master/modules/dqn.html`

[4] `https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html`

So, a number between 0 and 255 has been assigned to each of the elements as illustrated in Fig. 3.15. The elements are encoded in the image with the following numbers:

- 0: Ghost
- 42: Wall
- 84: Empty
- 126: Energy
- 168: Boost
- 210: Ghost Zombie
- 255: Pac-Man

The choice of the numbers did not seem to influence the agent's learning, but an order was maintained that suggests to the agent which elements it should or should not come into contact with.

With this input, the agent's network loses some relevant information about the state of the game, such as the step count, the number of lives of Pac-Man, and the score. A possible solution for this would be to use a custom network architecture, in which, after the convolutional layers, this information was injected into some neurons of a fully connected layer. But the Stable Baselines library does not allow defining a network that receives this customized input.

Later we will try to encode more information in other channels of the image, so this input format will be referred to as Single-Channel.

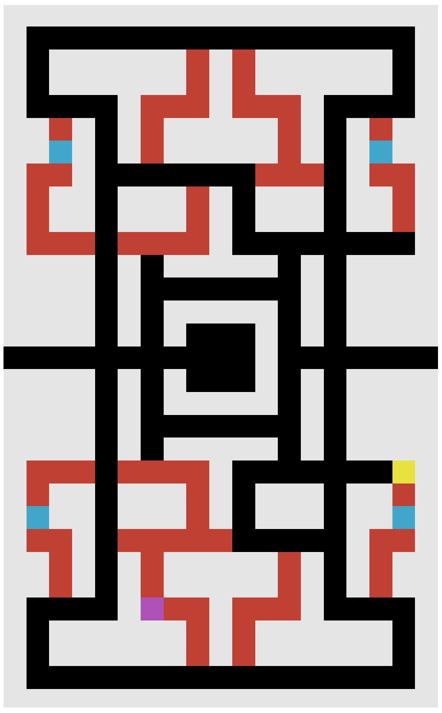


**Figure 3.15:** Single-Channel input representation. The colors represent the number of each element, as the image has only one channel. The Pac-Man's position is in yellow, the ghost is in purple, the energies are in red, the boosts are in blue, the empty cells are in black and the walls are in gray.

*Action Space*

As explained in the previous section, there are 4 possible actions that represent the 4 directions in which Pac-Man can move. Therefore, the agent's network output has 4 neurons, and numbers from 0 to 3 were used to represent each of the actions.

*Rewards*

Initially, the reward given to the agent was the score difference at every turn. This reward system represents exactly the score obtained by the agent.

*Game Parameters*

To start the experiments, the game's default parameters were used, that is, only a level 1 ghost in Map 1.

*Algorithm and Hyperparameters*

The first algorithm tested was DQN with all its upgrades(D3QN with PER). This algorithm will be designated as DQN. The hyperparameters used in the algorithm, were the default of the Stable Baselines library, except the CNN architecture.

By default, Stable Baselines uses the CNN architecture from the original DQN paper [8], but it was not compatible with the environment input image. The Map 1 has $19 \times 31$ pixels in width and height, respectively, and the filter sizes and strides used in the DQN original architecture do not fit for these dimensions without using padding.

Therefore, the original architecture of DQN has been slightly changed so that the dimensions of the filters, combined with the stride, fit perfectly in the input image and thus it is not necessary to use padding. The adopted architecture uses the same number of layers as in the original DQN and its details are shown in Fig. 3.16.

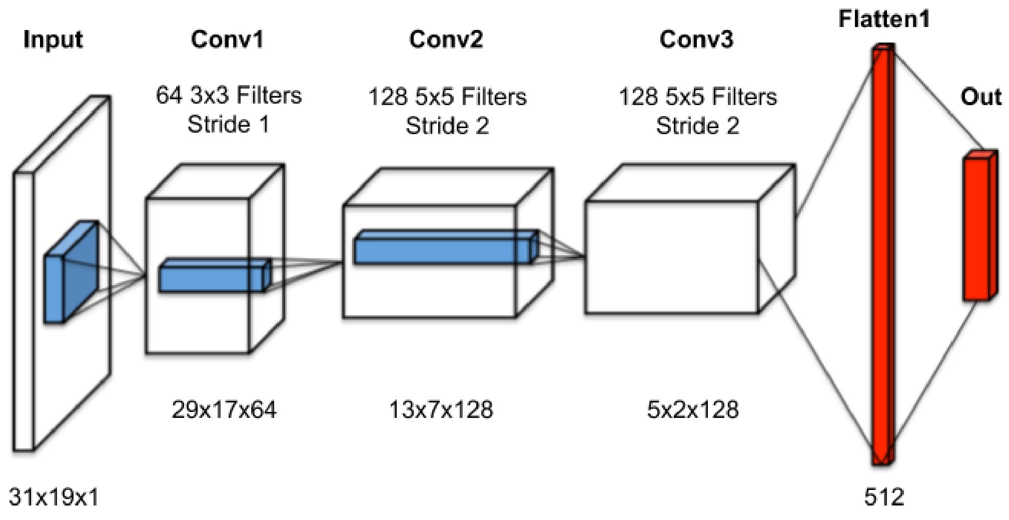


**Figure 3.16:** Pac-Man custom CNN architecture. The output dimensions of the convolutional layers are represented in the format $Height \times Width \times N\_Channels$. Source: Adapted from [9]

*Evaluation*

In the beginning, the evaluation of the DRL agents was different from the evaluation made in the AI course because the agent was trained to play only on the default game parameters. So, each evaluation consists of 30 games with the same parameters as in training, where the average, maximum, minimum, and standard deviation values of the score are calculated. During the training of an agent 10 evaluations are made.

*Server and agents' models*

The server used to run all experiments presented in this dissertation is composed by one NVIDIA Tesla K40c GPU.

Each agent created is assigned an incremental numeric identifier so that we are able to distinguish the agents more easily. The name of an agent is given by the name of the algorithm used and its identifier, for example, DQN_10.

In each training, two models of the agent network are saved. One corresponds to the model of the best evaluation and is used to make local tests and visualize the agent playing. The other is the model of the end of the training and can be used if we want to continue the training.

The behavior of some agents mentioned in this dissertation can be seen in a video [5], where the agents are identified by their name.

### 3.2.3 Using Deep Q-Network (DQN)

Training during 100k (thousand) and 250k steps was not enough for a DQN agent to obtain an average score greater than 105 and learn a consistent strategy. But during a training of 1M (million) steps, the DQN_3 agent managed to obtain an average score of 200 points, in the 30 test episodes of its best evaluation, and learned an interesting strategy that will be described bellow. The results of all the agent's evaluations during the training process are shown in Fig. 3.17. In total, the training took about 16 hours of processing time.

The evolution of the return obtained by the DQN_3 agent, in the training episodes, was also plotted. In Fig. 3.18 we can see the evolution of the agent's return during training, and in Fig. 3.19 we see the same results presented in another way, in which the points represent the average return obtained in the previous 100 training episodes.

Since the return is equivalent to the score, we can see that both during training and during evaluations, the maximum score obtained by the agent was around 300. Also, the total number of training episodes was 3962.

A maximum score of 300 means that the DQN_3 agent has never been able to eat all the energies, therefore never earning the final bonus reward. But, observing the agent playing it is possible to see that it learned an interesting strategy. At the beginning of the game, the DQN_3 agent brings the ghost to a boost in order to eat it and thus receive the 50 points. After that, the agent waits for the ghost to come, and takes it to another boost to eat it. But

[5] `https://www.youtube.com/watch?v=xDTF0w38WR0`

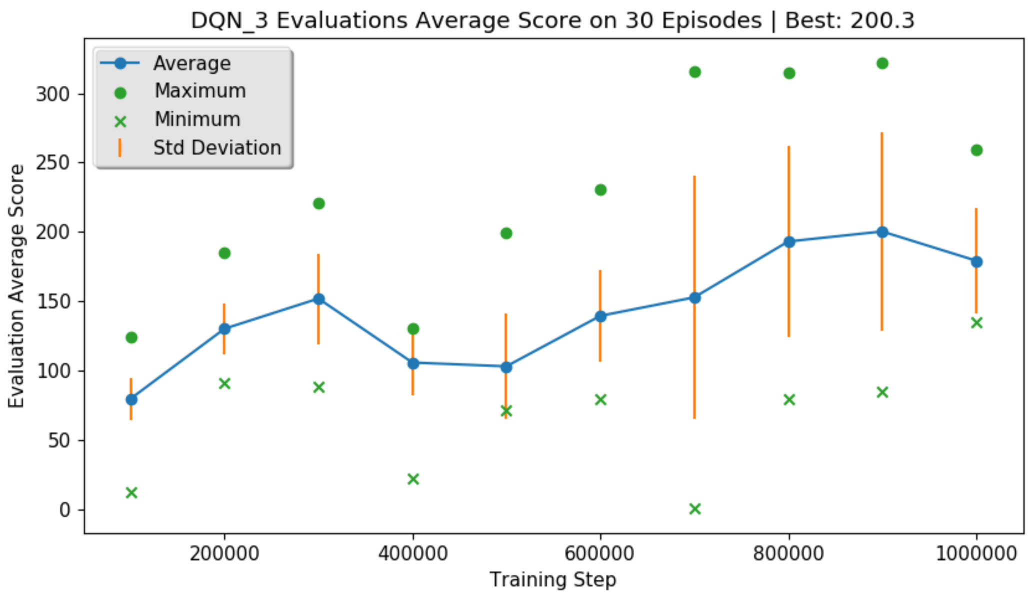


**Figure 3.17:** Evaluation results of the DQN_3 agent during a 1M steps training.

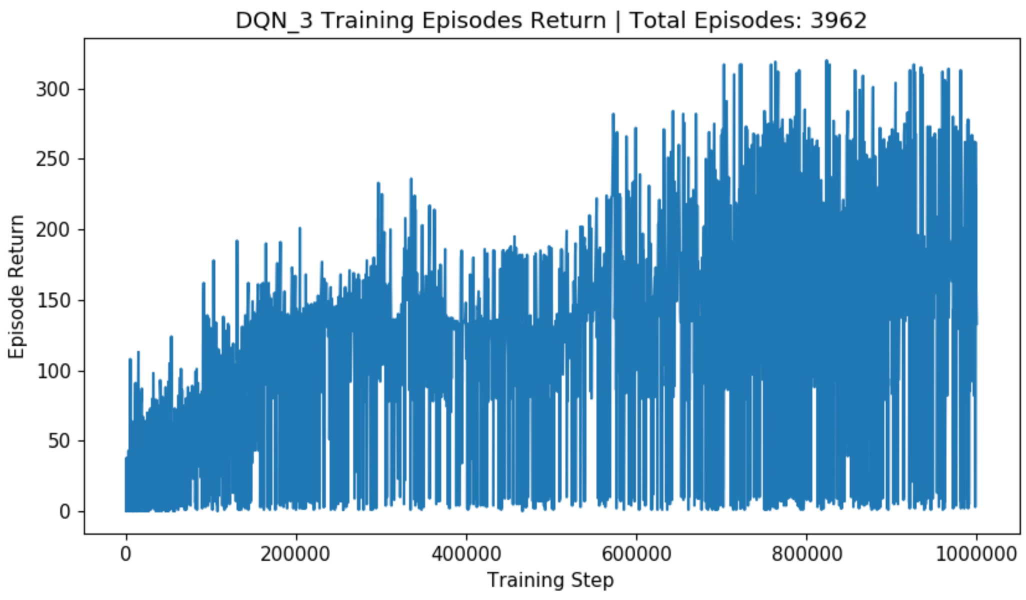


**Figure 3.18:** Return obtained by the DQN_3 agent during a 1M steps training.

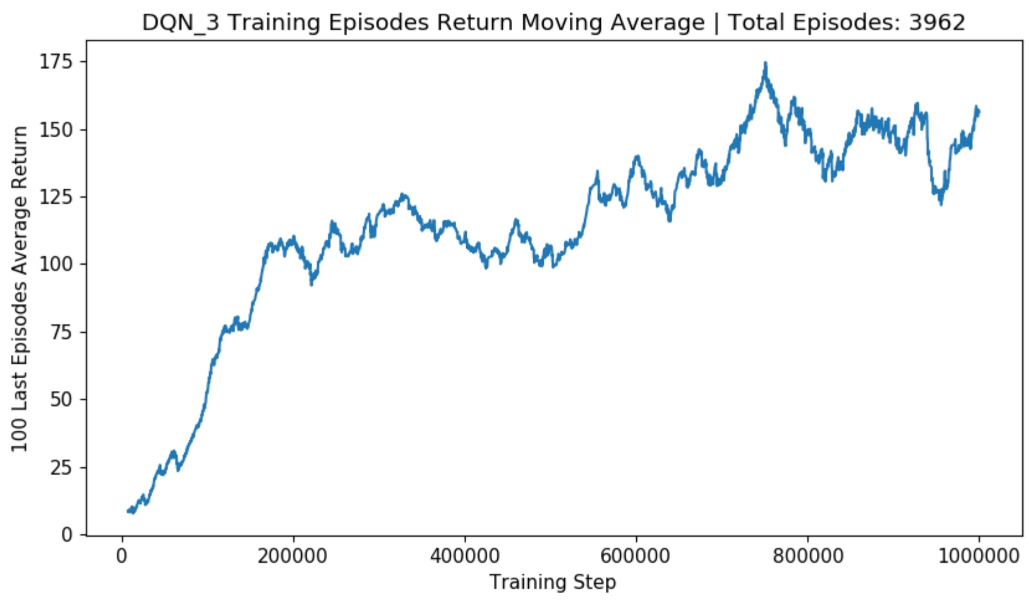


**Figure 3.19:** Moving average return obtained by the DQN_3 agent during a 1M steps training.

apart from this strategy the agent is lost in the map and does not move for long periods of steps. The behavior of this agent can be seen in the video [6].

As the DQN algorithm is deterministic, if the agent is not moving because it is going against a wall, the only thing that can make it change direction is the movement of the ghost that generates new states, but many times this agent remains still, even with the ghost's movement.

Perhaps the strategy of the DQN_3 agent makes sense since the reward given for eating a ghost is 50 times greater than the reward for eating an energy, and so it focused on eating ghosts. We could have changed the reward system to, for example, add more value to the energies, but as in the end what we want is to maximize the score of the game, we decided not to change the reward function already and try another algorithm first.

### 3.2.4 Using Proximal Policy Optimization (PPO)

As the DQN_3 agent was not able to obtain an average score greater than 300 during a training that used 16h of processing time, we decided to try the PPO algorithm to see if it would get a better score.

All PPO hyperparameters used were the default of the Stable Baselines library. The hidden layers of the value network and the policy network are shared and have the same configurations as those used for DQN and are illustrated in Fig. 3.16.

Training with PPO for the same number of steps (1M), the training consumed only 4.5 hours of processing time and the agent obtained an average score of 285. PPO trains faster for the same number of steps as it only updates the network parameters from time to time,

[6]https://www.youtube.com/watch?v=xDTFOw38WR0&t=3s

unlike DQN which updates the network in all steps. By default, in Stable Baselines, PPO updates the parameters every 128 steps.

Even performing much fewer optimization steps than DQN_3, the PPO_161 agent was able to obtain better results and even managed to win, obtaining the final bonus reward in some episodes during training, as we can see in Fig 3.20.

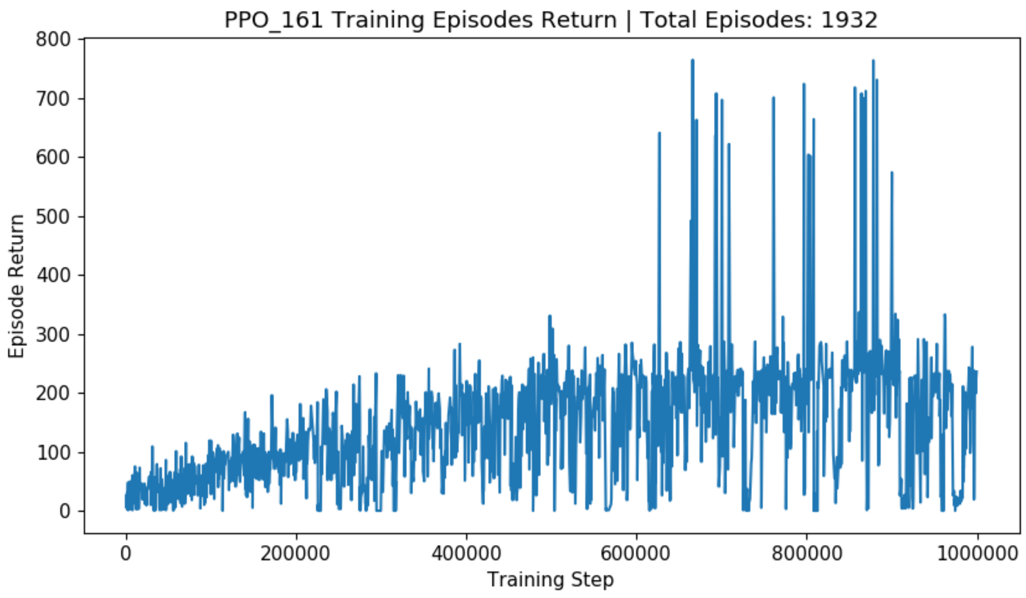


**Figure 3.20:** Return obtained by the PPO_161 agent during a 1M steps training.

The agent was also able to win during its last evaluations, as we can see in Fig. 3.21, by looking at the maximum scores.

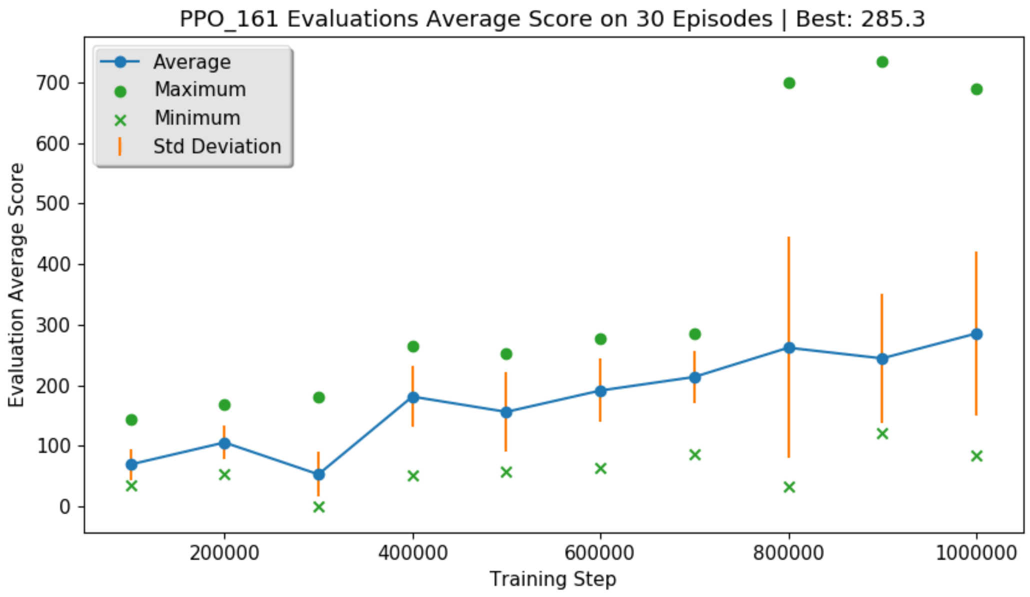


**Figure 3.21:** Evaluation results obtained by the PPO_161 agent during a 1M steps training.

The PPO_161 agent strategy gives priority to eating the energies, instead of eating the ghost. Still, this agent is not able to win very often. Its behavior can be seen in the video [7].

*More training steps*

By increasing the number of training steps to 2.5M, the training took around 11.5 hours of processing time, and the PPO_173 agent was able to obtain an average score of around 680 points. The results of the agent's evaluations are shown in Fig. 3.22.

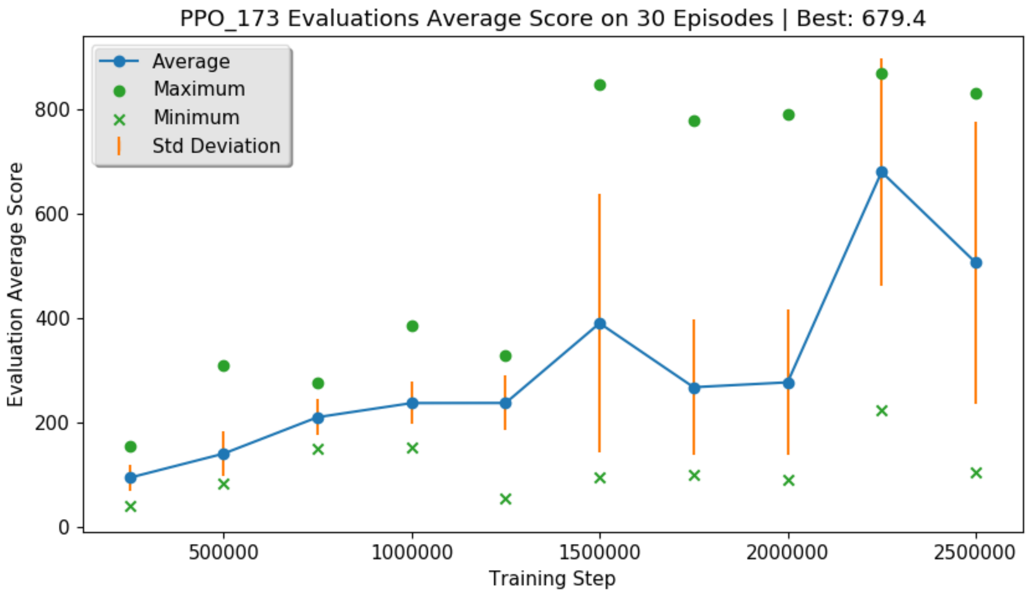


**Figure 3.22:** Evaluation results obtained by the PPO_173 agent during a 2.5M steps training.

The PPO_173 agent is able to achieve victories much more often than the previous agent, thus getting a higher average score. But, observing this agent playing in an environment with 2 ghosts of level 3, it appears that this agent has many difficulties in escaping from the ghosts, being unable to reach the final bonus reward. This is expected since the agent never trained in an environment with these characteristics. The behavior of this agent can be seen in the video [8].

### 3.2.5 Using Multi-Channel Input

Although the Single-Channel input is working very well, we decided to try another version of the input that separates the different elements of information into their own channels. In this way, the agent should be able to better distinguish the different elements.

The state of the game was then represented in an image with 6 channels. One channel for the walls, another for empty cells, another for energies and boosts, another for ghosts, another for zombies, and another for Pac-Man. In most channels, the pixels that have content related to the channel element have the value 255, and the remaining pixels have the value 0.

[7]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=32s`
[8]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=73s`

In the energies and boosts channel, the energies appear with the value 64 and the boosts with the value 255 so that the network can distinguish the elements.

This input should also be better because it can represent more information. For example, if a ghost is in the same position as an energy, the Single-Channel input can only represent the ghost, because ghosts were given priority in the Single-Channel input. But in this case, it is not necessary to give priority to any element, since they can all be represented simultaneously, even if they are in the same position.

*Increased game difficulty*

As the agent was already able to win often, the most difficult game parameters were used to see if the agents would be able to keep the average score. For the agent to be able to play against 4 ghosts, the number of ghosts for each training game was randomly chosen between 1 and 4. As the most difficult level of ghosts is level 3, we decided to use it, because we did not know yet that the students' agents' evaluation was made with ghosts at the maximum level of 2, therefore the level of ghosts will be adjusted later.

Also, the discount factor $\gamma$ used was the default of Stable Baselines which is 0.99. Other experiments were done with other values for $\gamma$, initially 0.999 and 0.95, and then for 0.8 and 0.9, and a $\gamma$ of 0.95 had the best results. Therefore, several experiments were tried for $\gamma$ around 0.95.

Discount factors from 0.92 to 0.98 with increments of 0.01, for both Single-Channel and Multi-Channel inputs, were tried in several training experiments of 2.5M steps. The agent with better results was PPO_193, with a $\gamma$ of 0.92 and Multi-Channel input. This agent trained for 13h and its evaluation results are shown in Fig. 3.23.

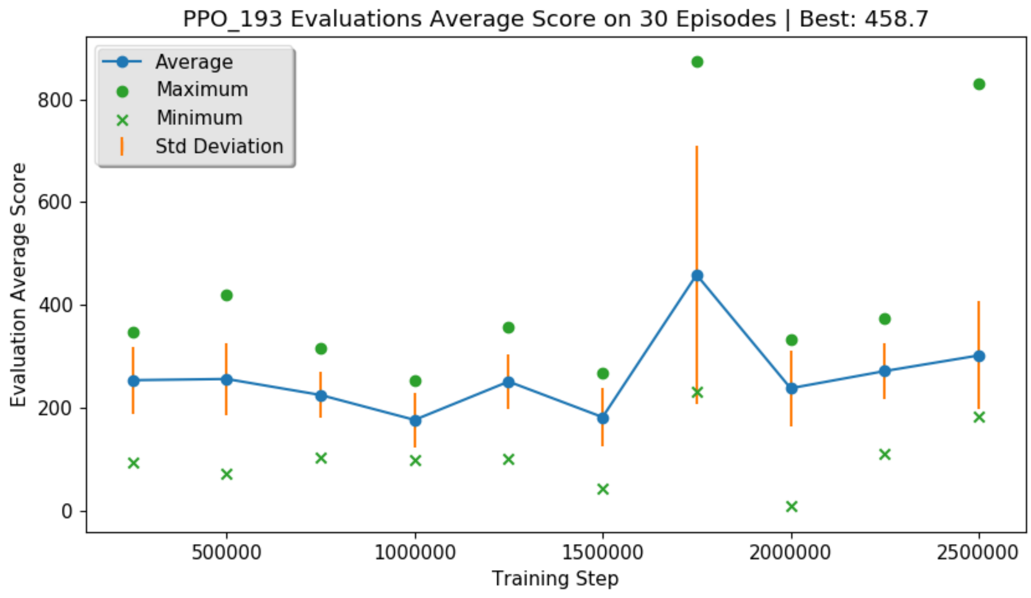


**Figure 3.23:** Evaluation results obtained by the PPO_193 agent during a 2.5M steps training.

The PPO_193 agent can win against 2 ghosts of level 3. But, against 4 ghosts of level 3, it is not able to eat all the energies. The behavior of this agent can be seen in the video [9].

*Centered Input on Pac-Man*

In an attempt to better represent the agent's current position, an experiment was carried out where the input was transformed so that the Pac-Man position was always the central cell of the map. Therefore, the Multi-Channel input has only 5 channels, since the Pac-Man channel is no longer needed because its position is always the center of the image. A representation of this new input is shown in Fig. 3.24.

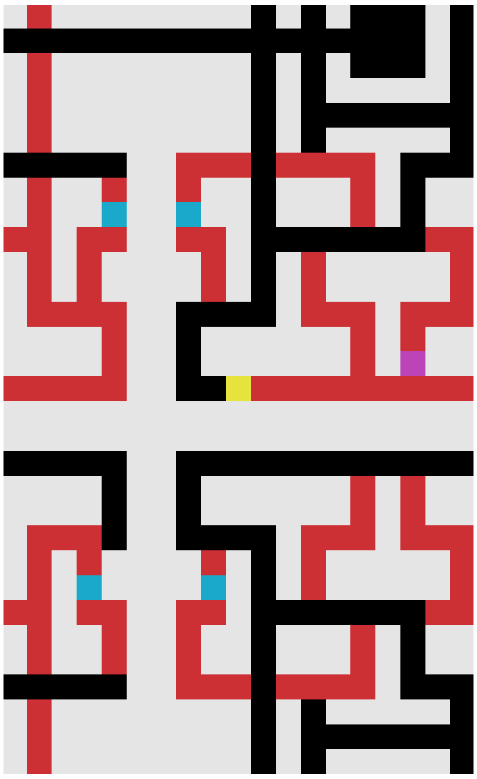


**Figure 3.24:** Centered Input on Pac-Man. The colors represent the different elements in the map. The Pac-Man's position is in yellow, the ghost is in purple, the energies are in red, the boosts are in blue, the empty cells are in black and the walls are in gray.

The agent's evaluation has also changed. The number of test episodes went from 30 to 40, with the number of ghosts starting at 1 and every 10 episodes increasing by 1. Thus, the agent is tested exactly 10 times for each number of ghosts, keeping the ghost level fixed at 3.

Using the centered input and training only for 1M steps, the PPO_473 agent was able to maintain higher average scores during the training evaluations, compared to the previous agent, achieving the best average score of 478. Also, from figure 3.25 we can see that since the 300K training steps the agent is able to win and then reached maximum scores above 1000 points. The previous agents never achieved a maximum score of 900, and the total processing time of this agent was only 6h.

Observing PPO_473 agent playing against 4 ghosts of level 3 we can see that it manages to get much higher scores than the previous agent, eating the ghosts many times. Even so,

[9]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=138s`

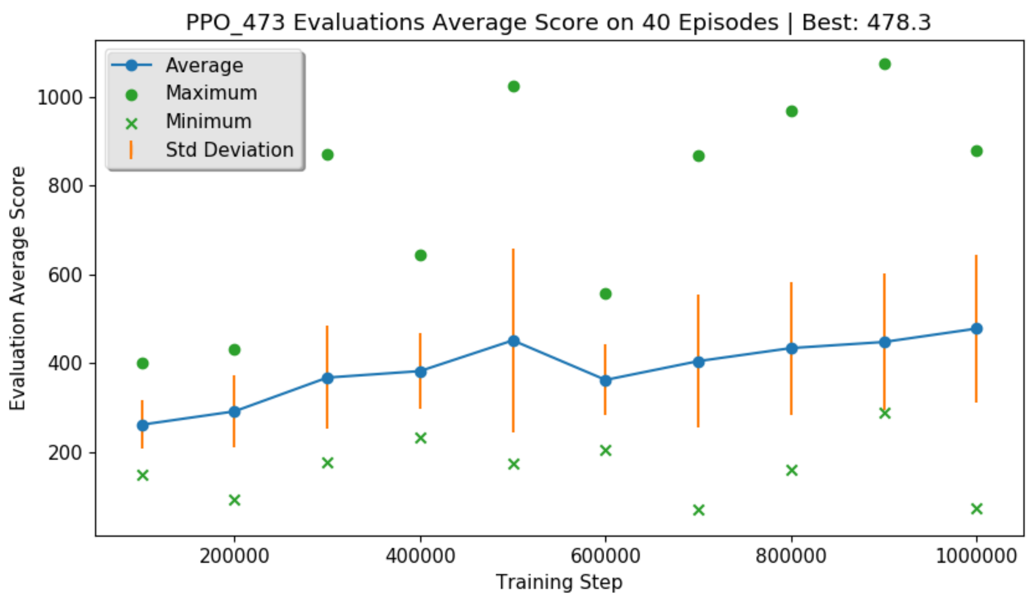


**Figure 3.25:** Evaluation results obtained by the PPO_473 agent during a 1M steps training.

this agent is not able to win the game, due to the fact that the ghosts' difficulty is very high. Its behavior can be seen in the video [10].

By this time, we have verified that the level of ghosts used in the AI course evaluations never reaches level 3, and therefore the next agents will be trained and evaluated against level 1 ghosts.

*More training steps*

In addition to reducing the difficulty of the ghosts to level 1, the number of training episodes has been increased to 10M.

Also, the number of games per evaluation has been increased from 40 to 80 in order to have even more accurate results. Therefore, each evaluation now has 20 episodes for each of the 4 possible numbers of ghosts (1, 2, 3, and 4).

The PPO_894 agent trained for 29h to complete the 10M training. In fact, it was only 9.5M steps because the training was interrupted as the agent maintains an almost constant average score during the evaluations, as shown in Fig. 3.26. Despite the long training, the average score does not evolve much, with the agent achieving the best average score of 568.

Observing the PPO_894 agent playing against 4 ghosts of level 1, we can see that this agent focuses on eating the ghosts, but cannot win the game. This agent even goes to the spawning position of the ghosts only to eat two of them and, obviously, lose a life right after. Therefore, we are going to change the reward system so that the agent is penalized for losing lives, and maybe it avoids these risky moves, achieving higher scores. The behavior of this agent can be seen in the video [11].

[10]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=201s`
[11]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=233s`

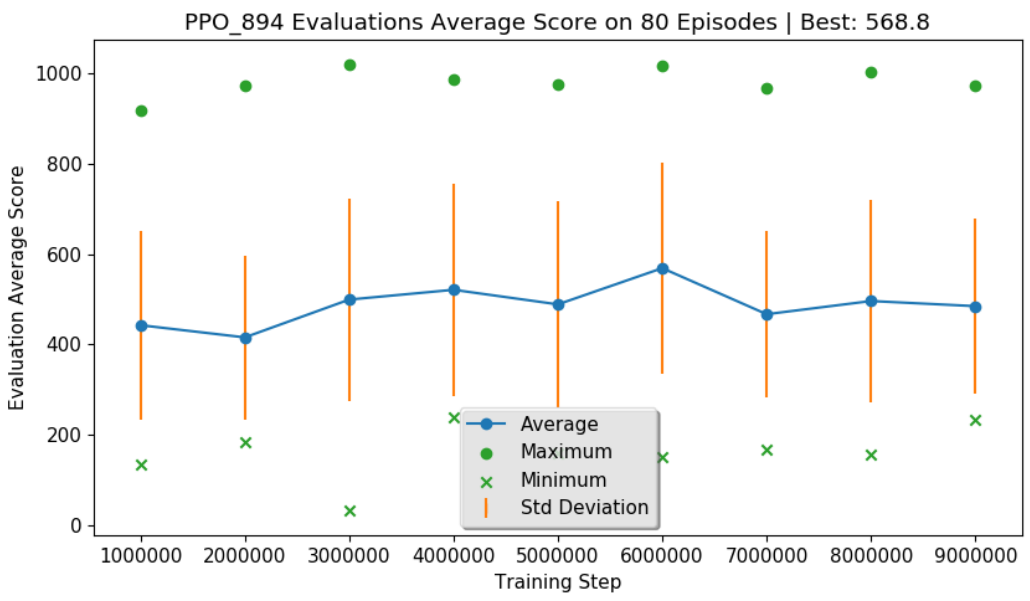


**Figure 3.26:** Evaluation results obtained by the PPO_894 agent during a 9.5M steps training.

*Penalizing the agent for losing lives*

The reward system has been changed so that the agent is penalized for losing lives. Therefore, when the agent collides with a ghost and loses a life, it receives a negative reward of 50 points. Also, we add a new input channel that contains a representation of the agent's lives, that is, when the agent has 3 lives, this channel is full of values at 255, when the agent has 2 lives this channel has only 2/3 of cells at 255, and when the agent has only 1 life, 1/3 of the cells is at 255. This way the agent can make decisions according to its number of lives.

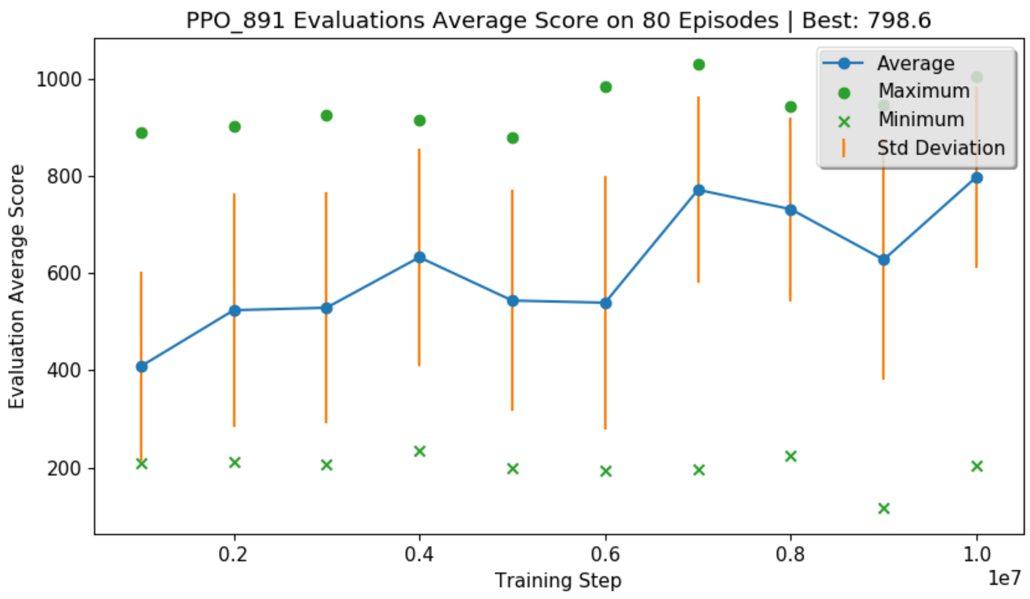


**Figure 3.27:** Evaluation results obtained by the PPO_891 agent during a 10M steps training.

Also training for 10M steps and taking about 29 hours of training, the PPO_891 agent was able to obtain average scores much higher than the agent previously presented, managing to reach an average score of 798. The results of the agent evaluations are shown in Fig. 3.27.

Observing the PPO_891 agent playing against 4 ghosts of level 1, we can see that this agent is more careful, loses fewer lives, and still manages to win the game. Although this agent has not trained against level 2 ghosts, it still manages to win. Its behavior can be seen in the video [12].

### 3.2.6 Using Multiple Maps

The PPO_891 agent is able to win in games with different characteristics in Map 1, so the next step was to develop an agent capable of playing both in Map 1 and Map 2 in order to obtain a good score in the evaluation made in the AI course.

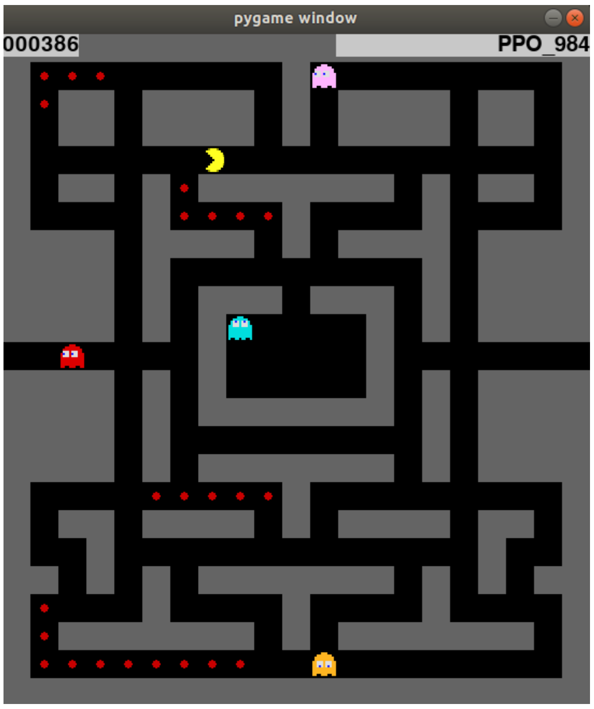


**Figure 3.28:** The Pac-Man game visualizer in Map 2.

The maps have different sizes. As already mentioned, Map 1 has $19 \times 31$ pixels. But, Map 2, represented in Fig. 3.28, is wider and shorter, and has $21 \times 24$ pixels, in width and height, respectively. In order to be able to deal with the two maps, the agent's network input has the maximum size of the two maps in each dimension, that is, the network input will have the size $21 \times 31$.

Two different input formats were developed for the original input to be transformed into the image of larger dimensions that will be fed to the network. One method replicates the limits of the map and the other loops the map information as illustrated in Fig. 3.29.

Also, the reward system has been slightly modified to encourage the agent not to go against the walls, reducing the agent's reward by 0.5 on each step it stands still.

[12]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=269s`

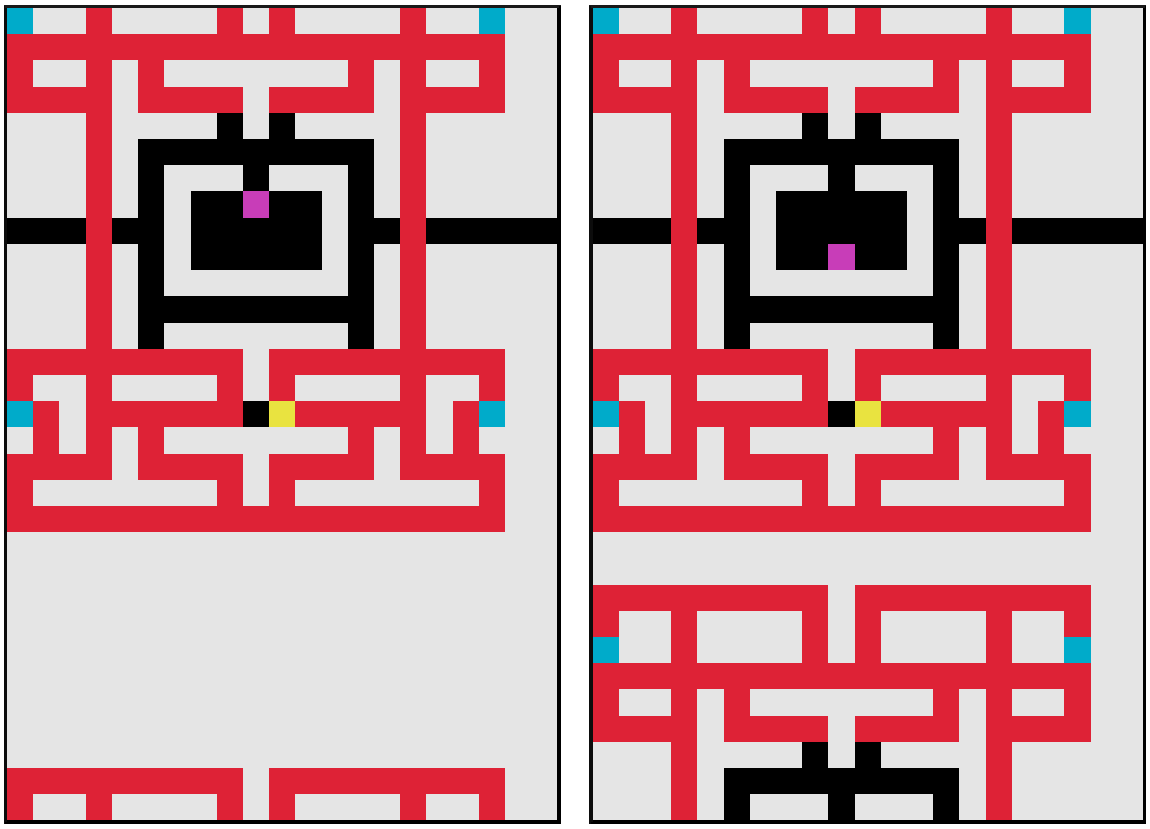


**Figure 3.29:** Representation of the different input formats in Map 2. On the left is the replicated limits input format. On the right is the looped input format. Note that the input is centered on the Pac-Man position.

The parameters used for the training games are the same as those used in the evaluation of the AI course agents, which were listed in Table 3.14. Only the first row configuration (which has zero ghosts) is not used since the agent is just playing one game on this configuration during the evaluation, and it should be able to generalize to play against zero ghosts. Therefore, for each training game, one of the 10 possible configurations is chosen at random.

Also, the evaluation used is not yet equal to the AI course evaluation, but it is similar. For each of the 10 possible game configurations played during training, 10 games are played during an evaluation, making a total of 100 games. Later, the exact evaluation done in the course will be used.

Training for 20M training steps, using the replicated limits input format, the PPO_998 agent reached an average score of 780 in its best evaluation. The training took about 69 hours of processing time, and the results of all evaluations are shown in figure 3.30.

Using the looped input format, and training for the same number of steps, the PPO_982 agent had the same performance, achieving an average score of 784. The training consumed about 73 hours of processing time, and the results of all evaluations are shown in Fig. 3.31.

The score evolution plots for both agents PPO_982 and PPO_998 seemed to be increasing at the end, so their training was continued for another 5M steps. Also, the evaluations that will be made to the new agents are now exactly the same as the evaluations made in the AI

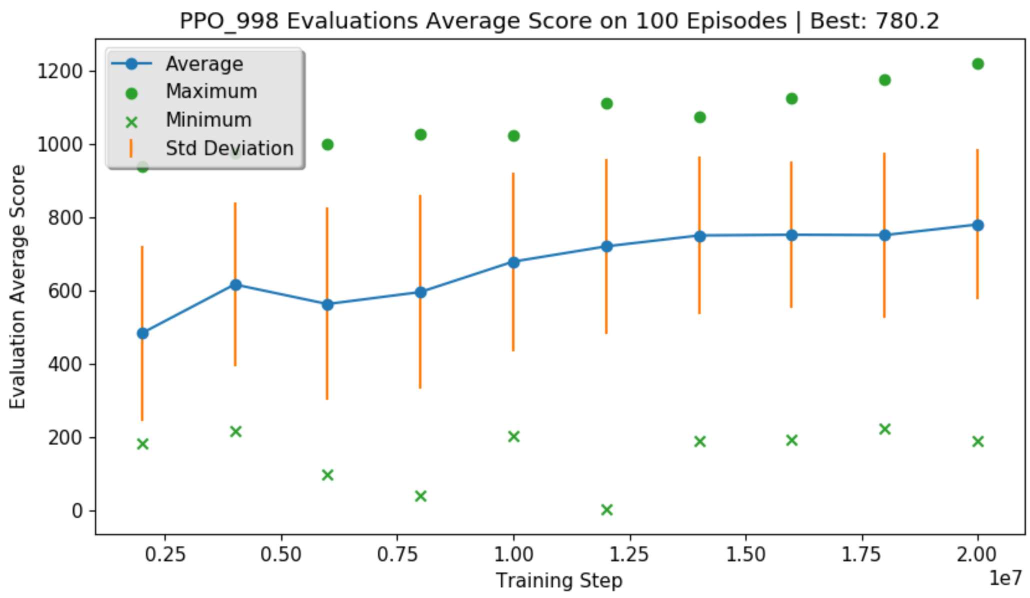


**Figure 3.30:** Evaluation results obtained by the PPO_998 agent during a 20M steps training using the replicated limits input format.

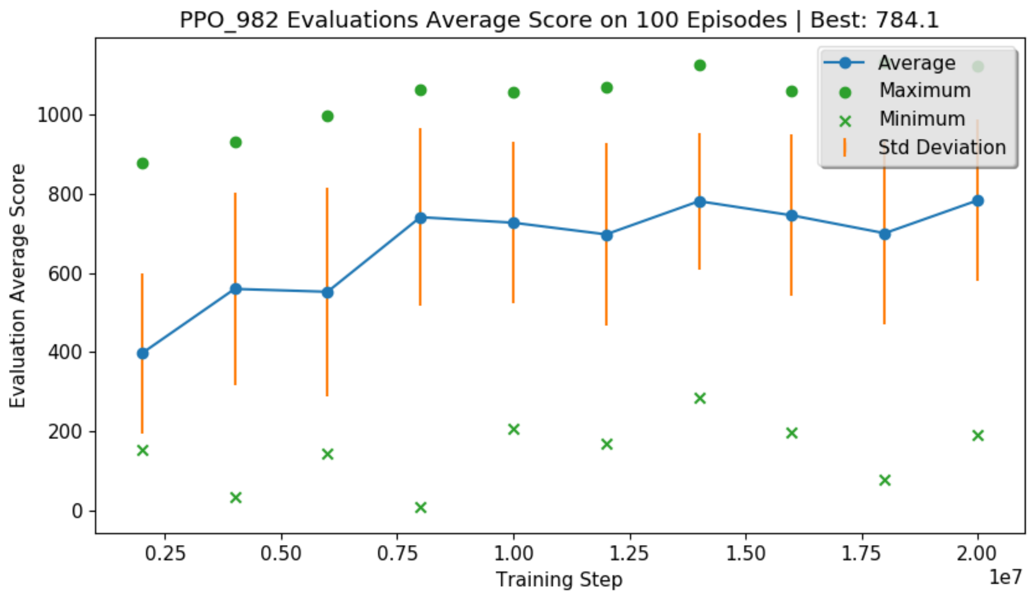


**Figure 3.31:** Evaluation results obtained by the PPO_982 agent during a 20M steps training using the looped input format.

course, described in section 3.2.1, so that we can compare the results with those obtained by the students' agents.

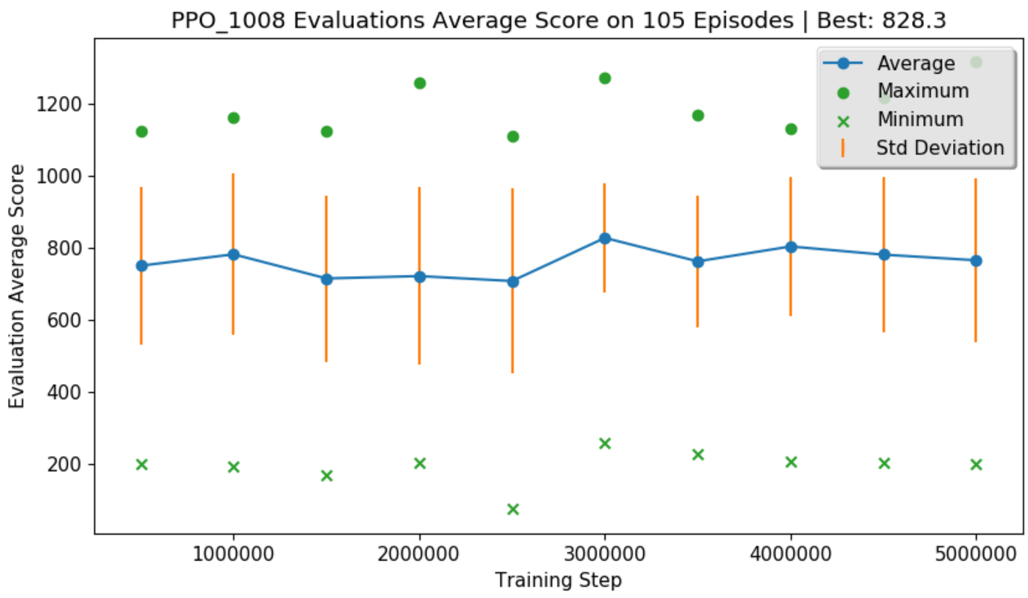


**Figure 3.32:** Evaluation results obtained by the PPO_1008 agent during a 5M steps training that continues the 20M steps training of the PPO_982 agent using the looped input format.

Continuing the training of the PPO_982 agent, with the looped input format, the PPO_1008 agent was able to reach an average score of 828 in its best evaluation. This agent trained for another 22 hours to complete the 5M steps, doing a total training of 25M steps in 95 hours. The complete results of its evaluations are shown in Fig. 3.32.

| Map | Number of Ghosts | Ghosts Level | Number of Wins | Average Score |
|---|---|---|---|---|
| Map 1 | 0 | 0 | 100 | 788.0 |
| Map 1 | 1 | 1 | 99 | 808.8 |
| Map 1 | 2 | 2 | 96 | 941.2 |
| Map 1 | 4 | 0 | 96 | 892.0 |
| Map 1 | 4 | 1 | 88 | 898.9 |
| Map 1 | 4 | 2 | 65 | 936.2 |
| Map 2 | 1 | 1 | 100 | 775.4 |
| Map 2 | 2 | 1 | 100 | 791.8 |
| Map 2 | 4 | 0 | 98 | 767.2 |
| Map 2 | 4 | 1 | 86 | 762.7 |
| Map 2 | 4 | 2 | 71 | 668.3 |

**Table 3.15:** Results obtained by the PPO_1008 agent in tests of 100 episodes for each of the configurations used in the AI course evaluation.

Of the 105 games played in the best evaluation, the PPO_1008 agent won in 96 games. In fact, as shown in Table 3.16, the agent is able to win frequently in all the game configurations used in the AI course evaluation. Note that the agent was able to generalize to play against zero ghosts in Map 1, without ever playing in that configuration during training.

Even so, the PPO_1008 agent was unable to reach the level of some agents developed by the students. The agent at the top of the ranking table achieved an average score of 999. With an average score of 828, the PPO_1008 agent's ranking would be 7 out of 56, achieving a grade of 18.5 out of 20. This agent's performance is not bad, but in order to achieve higher scores, it would have to improve some strategies, because winning is not enough to reach an average score of 999.

For example, in Map 2 this agent eats the four boosts at the beginning of the game, often not taking advantage of their powers and not eating the ghosts to increase the score. In Map 1, it easily attracts the ghosts to the boosts, even though the fourth boost is not often utilized to eat the ghosts. This is the main reason why the average scores obtained in Map 2 games are significantly lower than in Map 1 games as shown in Table 3.16. The behavior of this agent in various training configurations can be seen in the video [13].

The code developed to build all the DRL agents is public in a GitHub repository [14]. The same repository includes also the model parameters of the agents mentioned in the dissertation as well as all the results generated.

*Testing in unknown maps*

Other maps were built to test whether the DRL agents are able to apply the acquired knowledge to unknown maps. The tests were done with the PPO_1008 agent.

Map 3 is identical to Map 1, but as the paths used by the agent to move upwards or downwards do not exist, the agent stops at the turning point for those paths, as shown in Fig. 3.33. Those paths were widely used by the agent, if only they existed and the others at the top and bottom of the map did not, the agent would be able to move correctly.

However, in the case of Map 6, which is also a smaller portion of Map 1, represented in Fig. 3.34, those paths do not exist, but the agent is able to win 100% of the games. This is probably due to the energies being only on the agent's right side.

Map 7 is completely different from the maps the agent trained on. In this map, the agent has a tendency to go to the right and get stuck in the square, as illustrated in Fig. 3.35. But besides that, the agent is able to move around and win some games.

In Map 8, represented in Fig. 3.36, which is identical to Map 7, but has that square covered, the agent's performance improves, managing to win in most games.

The results of the PPO_1008 agent in 20 test episodes against 1 ghost of level 1 in the mentioned maps are shown in Table 3.16.

The agent seems to be able to use the acquired knowledge to play in unknown maps. Although not perfectly, the agent is able to move around to eat the energies and escape from the ghost. However, it is possible that in some maps the agent will not be able to have a good performance, as in Map 3 where the agent gets paralyzed.

[13]`https://www.youtube.com/watch?v=xDTF0w38WR0&t=368s`

[14]`https://github.com/mdaraujo/deep-rl-pacman`

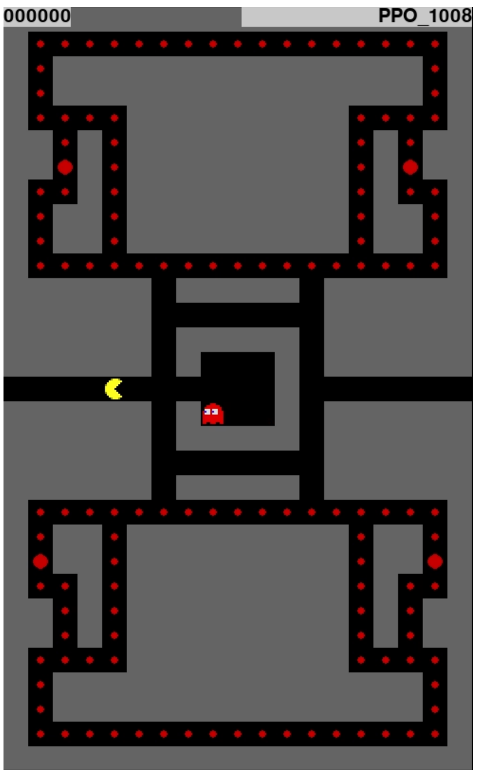


**Figure 3.33:** Map 3.

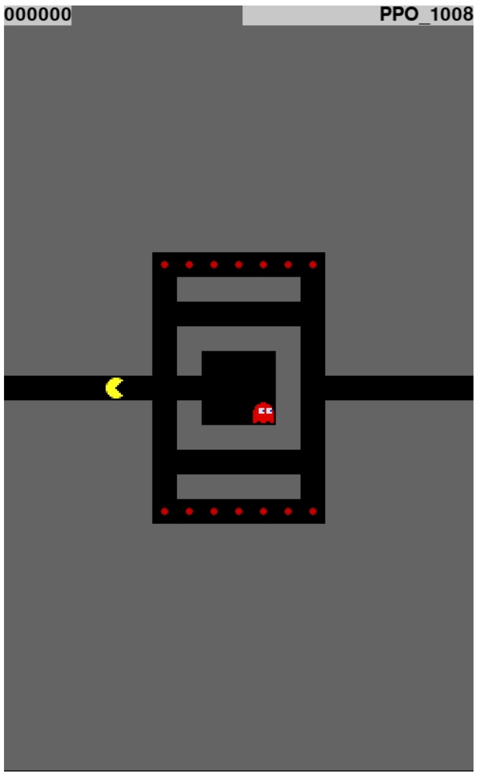


**Figure 3.34:** Map 6.

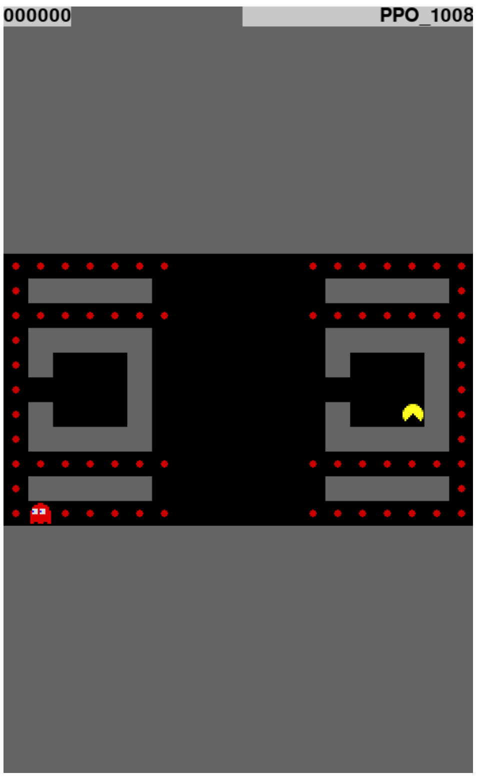


**Figure 3.35:** Map 7.

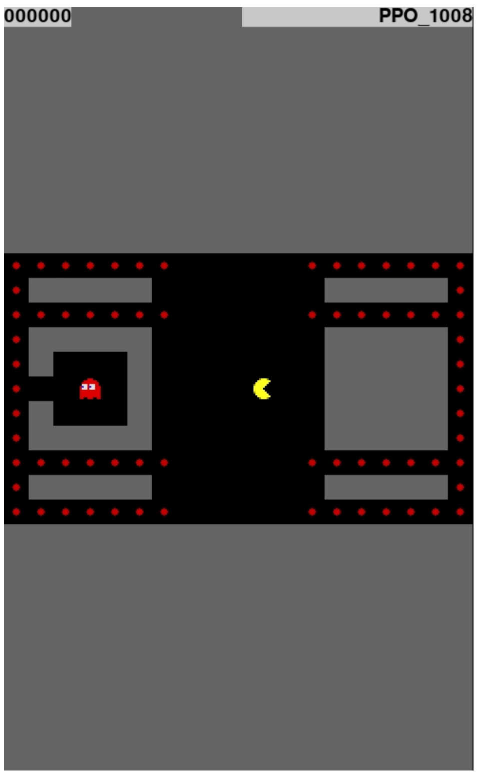


**Figure 3.36:** Map 8.

| Map | Number of Wins | Average Score |
|---|---|---|
| Map 3 | 0 | 0.0 |
| Map 6 | 20 | 602.3 |
| Map 7 | 8 | 281.1 |
| Map 8 | 15 | 484.1 |

**Table 3.16:** Results obtained by the PPO_1008 agent in tests of 20 episodes for each of the unknown maps against 1 ghost of level 1.

### 3.2.7 Failed attempts

The agents presented in the previous sections are the ones that emerged from successful variations of the Gym environment parameters, but throughout the process of developing agents for this Pac-Man environment, many other attempts have been unsuccessful.

*Symmetric Negative Rewards*

When the agent wins, the score increases by the bonus of 1 point for each remaining 5 steps until the timeout. Therefore, in order for the agent to understand that it should finish the game as soon as possible, a reward system was tried that at each step the agent's reward was reduced by 1/5, that is, equivalent to the bonus the agent would receive for each remaining step in case of victory. Then, if the agent wins, it does not receive the final bonus, but if it loses the game, the bonus it could receive in case of victory is deducted from the final reward.

In this way, the reward obtained by the agent is always the score obtained minus 600, because, as the steps' timeout is 3000, the maximum bonus that would be possible to receive in case of victory would be 600.

So, the reward can remain proportional to the score and still make the agent understand the need to finish the game faster, without having to win the game various times to understand it. This approach turned out to have much worse results, probably because it gives very negative feedback, making the agent unable to recover from such low returns.

*Other Reward Systems*

Other reward systems were attempted but ended up having worse results.

For example, we tried to give more value to the agent's last life, since the agent does not lose score points for losing any life, so only the last one affects the score because the game ends. One attempt was to give -50 reward on the first life lost, -100 on the second, and -150 on the third. The maximum average score of the trained agent was 607 in a 10M steps training. Other agents, such as PPO_982, exceeded the average score of 700 in fewer training steps.

Another attempt was inspired by the rewards of another Pac-Man environment that was used in a thesis [31] that also applies DRL. When the agent dies it receives a -500 reward, and when the agent eats both energies and boosts it receives a reward of 3. The maximum average score of the trained agent was only 366 during 7M steps of training.

CHAPTER 4

# Conclusion

Most of the algorithms presented in chapter 1 were able to learn human strategies without using any examples of human behavior, such as the opening patterns in Chess and Go, as well as some behavior patterns in Dota 2 and in the Atari environments. In Chess and Go, they even discarded some known openings and started to prefer different variations. This demonstrates that these algorithms have the ability to discover new knowledge and find unknown solutions to a wide range of problems, as they are becoming increasingly generic, and potentially applicable to domains beyond those in which they were tested. However, it takes a lot of computational power for these algorithms to work in complex environments, as they start from scratch and need a large number of steps to start learning good strategies. For example, the state-of-the-art agent on Go needed 13 days to complete a training of 700,000 steps, using 5,000 first-generation TPUs and 16 second-generation TPUs.

Tic-Tac-Toe was a good game to start the development of DRL agents since it is a simple game and does not require a lot of computing power to train the agents. But, the challenge of training an agent that could be good in the evaluation against both Random and Min-Max agents turned out to be more complicated than it initially seemed. Of the three approaches for the learning environment agent (Random, Min-Max, or Self), the Self turned out to be the approach with the best results, without using randomness in the environment agent. But using randomness, Min-Max and Self are both excellent and the trained agent managed to get a higher score than the Min-Max agent itself. This means that a DRL agent that trained against itself was able to surpass the performance of a perfect agent, in the sense that it is able to take advantage of the Random opponent weaknesses, therefore, obtaining more victories.

Developing DRL agents for the Pac-Man game was more difficult since this is a more complex game and long training sessions were needed. The agent that obtained the best results took almost 4 days of training. This agent was not able to surpass the performance of all the agents developed by the AI course students, but still, the scores obtained are really good and the agent managed to learn all the expected behaviors such as, eating all the energies, running away from ghosts, as well as eating the ghosts.

Also, it should be noted that the developed DRL Pac-Man agent is able to play on two maps with different dimensions and this obviously affects its performance. Other DRL experiments on Pac-Man [31] [32] only train the agent for a single map. The same happened in Tic-Tac-Toe where we ended up also having a more generic agent capable of handling more game configurations, in comparison with other experiments that apply DRL in Tic-Tac-Toe [30], where the agents train only for one position (first player or second player), and are evaluated against the training environment agent. Our Tic-Tac-Toe agents are able to play as first and second player, and are evaluated against two distinct reference agents.

In general, DRL agents with good performances were trained for both games, and interesting strategies were learned.

## 4.1 Future Work

More advanced DNNs architectures have been recently used in RL. For example, the agents developed by OpenAI that play Dota 2 [19] used Long short-term memory (LSTM) [33]. These networks have memory cells that allow them to store information about previous states. This seems to have great advantages in DRL, since the agent can take past information into account when making decisions, allowing it to yield long-term planning. Perhaps it will be possible to obtain better results in the Pac-Man game using this type of architecture.

It would also be interesting to experiment with these DRL techniques in other games of the AI course to understand what would be the differences in the environments that would be favorable or not for the agents' learning.

# Bibliography

[1] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity”, *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943, ISSN: 00074985. DOI: 10.1007/BF02478259. [Online]. Available: https://link.springer.com/article/10.1007/BF02478259.

[2] M. A. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com.

[3] N. Qian, “On the momentum term in gradient descent learning algorithms”, *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999, ISSN: 08936080. DOI: 10.1016/S0893-6080(98)00116-6.

[4] D. P. Kingma and J. L. Ba, “Adam: A method for stochastic optimization”, in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, International Conference on Learning Representations, ICLR, Dec. 2015. arXiv: 1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980v9.

[5] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks”, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: JMLR Workshop and Conference Proceedings, Apr. 2011, pp. 315–323. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.html.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning : an introduction*, 2nd ed. MIT Press, 2018, p. 526, ISBN: 9780262039246. [Online]. Available: https://mitpress.mit.edu/books/reinforcement-learning-second-edition.

[7] C. J. C. H. Watkins and P. Dayan, “Q-learning”, *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992, ISSN: 0885-6125. DOI: 10.1007/BF00992698. [Online]. Available: http://link.springer.com/10.1007/BF00992698.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning”, *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 0028-0836. DOI: 10.1038/nature14236. [Online]. Available: http://www.nature.com/articles/nature14236.

[9] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Frcitas, “Dueling Network Architectures for Deep Reinforcement Learning”, in *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, Nov. 2016, pp. 2939–2947, ISBN: 9781510829008. arXiv: 1511.06581. [Online]. Available: http://arxiv.org/abs/1511.06581.

[10] M. Riedmiller, “Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method”, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3720 LNAI, 2005, pp. 317–328, ISBN: 3540292438. DOI: 10.1007/11564096_32.

[11] H. Hasselt, “Double q-learning”, in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc., 2010, pp. 2613–2621. [Online]. Available: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[12] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning”, AAAI’16, pp. 2094–2100, 2016. arXiv: 1509.06461. [Online]. Available: http://arxiv.org/abs/1509.06461.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016. arXiv: 1511.05952. [Online]. Available: http://arxiv.org/abs/1511.05952.

[14] R. J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning", Tech. Rep., 1992, pp. 229–256. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BF00992696.pdf.

[15] V. Mnih, A. P. Badia, L. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", in *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, Feb. 2016, pp. 2850–2869, ISBN: 9781510829008. arXiv: 1602.01783. [Online]. Available: http://arxiv.org/abs/1602.01783.

[16] Y. Wu, E. Mansimov, S. Liao, A. Radford, and J. Schulman. (Aug. 2017). OpenAI Baselines: ACKTR & A2C, [Online]. Available: https://openai.com/blog/baselines-acktr-a2c/ (visited on 03/30/2020).

[17] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization", in *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, Feb. 2015, pp. 1889–1897, ISBN: 9781510810587. arXiv: 1502.05477. [Online]. Available: http://arxiv.org/abs/1502.05477.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms", *arXiv*, Jul. 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[19] OpenAI, *Openai five*, https://blog.openai.com/openai-five/, 2018.

[20] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning", *arXiv*, Dec. 2019. arXiv: 1912.06680. [Online]. Available: http://arxiv.org/abs/1912.06680.

[21] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep Blue", *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002, ISSN: 00043702. DOI: 10.1016/S0004-3702(01)00129-1.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 14764687. DOI: 10.1038/nature16961. [Online]. Available: http://www.nature.com/articles/nature16961.

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. DOI: 10.1109/TCIAIG.2012.2186810.

[24] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge", *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, ISSN: 14764687. DOI: 10.1038/nature24270. [Online]. Available: http://www.nature.com/articles/nature24270.

[25] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, ISSN: 10959203. DOI: 10.1126/science.aar6404. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/30523106.

[26] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine

Learning on Heterogeneous Distributed Systems”, Mar. 2016. arXiv: 1603.04467. [Online]. Available: http://arxiv.org/abs/1603.04467.

[27] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Stable baselines*, https://github.com/hill-a/stable-baselines, 2018.

[28] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, *Openai baselines*, https://github.com/openai/baselines, 2017.

[29] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540.

[30] C. Friedrich, *A tale about trying to train a machine to play tic tac toe through reinforcement learning*, https://github.com/fcarsten/tic-tac-toe. (visited on 02/26/2020).

[31] T. van der Ouderaa, *Deep reinforcement learning in pac-man*, Bachelor Thesis, University of Amsterdam, 2016. [Online]. Available: https://esc.fnwi.uva.nl/thesis/centraal/files/f323981448.pdf.

[32] A. Gnanasekaran, J. F. Faba, and J. An, “Reinforcement learning in pacman”, 2017. [Online]. Available: http://cs229.stanford.edu/proj2017/final-reports/5241109.pdf.

[33] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with LSTM”, *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000, ISSN: 08997667. DOI: 10.1162/089976600300015015.