UNIVERSIDAD DE INVESTIGACIÓN DE TECNOLOGÍA EXPERIMENTAL YACHAY

Escuela de Ciencias Matemáticas y Computacionales

TÍTULO: REINFORCEMENT LEARNING NEURAL AGENTS IN CLEVER GAME PLAYING

Trabajo de integración curricular presentado como requisito para la obtención del título de Ingeniero en Tecnologías de la Información

Autor:

Cárdenas López Kevin Fabricio

Tutor:

Ph.D. Chang Tortolero Oscar Guillermo

Urcuquí, abril 2022

UNIVERSIDAD YACHAY TECH

Urcuquí, 10 de junio de 2022

SECRETARÍA GENERAL
(Vicerrectorado Académico/Cancillería)
ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES
CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN
ACTA DE DEFENSA No. UITEY-ITE-2022-00007-AD

En la ciudad de San Miguel de Urcuquí, Provincia de Imbabura, a los 10 días del mes de junio de 2022, a las 10:00 horas, en el Aula S_CAN de la Universidad de Investigación de Tecnología Experimental Yachay y ante el Tribunal Calificador, integrado por los docentes:

Presidente Tribunal de Defensa	Dr. FONSECA DELGADO, RIGOBERTO SALOMON , Ph.D.
Miembro No Tutor	Dr. CAMACHO , FRANKLIN , Ph.D.
Tutor	Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.

Se presenta el(la) señor(ita) estudiante **CARDENAS LOPEZ, KEVIN FABRICIO**, con cédula de identidad No. **1754319943**, de la **ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES**, de la Carrera de **TECNOLOGÍAS DE LA INFORMACIÓN**, aprobada por el Consejo de Educación Superior (CES), mediante Resolución **RPC-SO-43-No.496-2014**, con el objeto de rendir la sustentación de su trabajo de titulación denominado: **REINFORCEMENT LEARNING NEURAL AGENTS IN CLEVER GAME PLAYING**, previa a la obtención del título de **INGENIERO/A EN TECNOLOGÍAS DE LA INFORMACIÓN**.

El citado trabajo de titulación, fue debidamente aprobado por el(los) docente(s):

Tutor	Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.

Y recibió las observaciones de los otros miembros del Tribunal Calificador, las mismas que han sido incorporadas por el(la) estudiante.

Previamente cumplidos los requisitos legales y reglamentarios, el trabajo de titulación fue sustentado por el(la) estudiante y examinado por los miembros del Tribunal Calificador. Escuchada la sustentación del trabajo de titulación, que integró la exposición de el(la) estudiante sobre el contenido de la misma y las preguntas formuladas por los miembros del Tribunal, se califica la sustentación del trabajo de titulación con las siguientes calificaciones:

Tipo	**Docente**	**Calificación**
Tutor	Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.	10,0
Presidente Tribunal De Defensa	Dr. FONSECA DELGADO, RIGOBERTO SALOMON , Ph.D.	10,0
Miembro Tribunal De Defensa	Dr. CAMACHO , FRANKLIN , Ph.D.	10,0

Lo que da un promedio de: **10 (Diez punto Cero)**, sobre 10 (diez), equivalente a: **APROBADO**

Para constancia de lo actuado, firman los miembros del Tribunal Calificador, el/la estudiante y el/la secretario ad-hoc.

[illegible]

CARDENAS LOPEZ, KEVIN FABRICIO
Estudiante

[illegible]

Dr. FONSECA DELGADO, RIGOBERTO SALOMON , Ph.D.
Presidente Tribunal de Defensa

Oscar Chang

Dr. CHANG TORTOLERO, OSCAR GUILLERMO , Ph.D.
Tutor

[illegible]

Dr. CAMACHO , FRANKLIN , Ph.D.
Miembro No Tutor

[illegible]
MEDINA BRITO, DAYSY MARGARITA
Secretario Ad-hoc

Autoría

Yo, **KEVIN FABRICIO CARDENAS LOPEZ**, con cédula de identidad 1754319943, declaro que las ideas, juicios, valoraciones, interpretaciones, consultas bibliográficas, definiciones y conceptualizaciones expuestas en el presente trabajo; así cómo, los procedimientos y herramientas utilizadas en la investigación, son de absoluta responsabilidad de el/la autor/a del trabajo de integración curricular. Así mismo, me acojo a los reglamentos internos de la Universidad de Investigación de Tecnología Experimental Yachay.

Urcuquí, marzo 2022.

Firmado electrónicamente por:
KEVIN FABRICIO CARDENAS LOPEZ

Kevin Fabricio Cárdenas López

CI: 1754319943

Autorización de publicación

Dedication

"This thesis project is dedicated to my family and friends, especially to my mother Mónica, who has been my guide and unconditional support during all these years."

Acknowledgment

I want to thank my family for always guiding my steps and being my motivation to keep going. I thank my grandparents Alfredo and Fabiola, my uncles Yolita, Yoli, Darwin, and Freddy, and my cousins Daniela, Gaby, and Bryan for watching over me since I began my university education. I thank my mother, Mónica, the person I admire the most, for her infinite love and unconditional support during all these years of study; She has been my source of inspiration to not give up on my dreams and to whom I owe many of my achievements, including this one. Also, to my sister Tania who brings joy to my soul and for whom I try to be an excellent example to follow. To my girlfriend and best friend Johanna, who has always been by my side and has encouraged me to be a better person every day.

A special thanks to my friends Mabe, Rafa, Erick, and Mateo, with whom I shared most of my university life and who are my second family. I have shared many beautiful moments with them that I will always keep in my heart.

Finally, I thank Yachay Tech University for opening its doors for me, being my second home, and providing high-quality education. To all the professors who were part of my academic studies, especially my tutor, Oscar Chang, for guiding me and giving me all his support and knowledge in developing this project.

Resumen

Todas las culturas humanas a lo largo de la historia han jugado juegos y desde el comienzo del desarrollo de la Inteligencia Artificial ha existido un gran interés en los juegos como plataforma de investigación. Juegos como Backgammon, Ajedrez, Checkers, Go, Othello y Tic-Tac-Toe son ampliamente utilizados para estudiar la capacidad de aprendizaje de las máquinas y desarrollar algoritmos de aprendizaje por parte de los grandes concursantes del mundo digital como Google, Facebook, Windows, etc. Este proyecto pretende mejorar los programas, métodos y resultados obtenidos en el trabajo Self-taught Neural Agents in Clever Game Playing, que utiliza agentes inteligentes y aprendizaje por refuerzo en redes neuronales. La característica central de este proyecto es un agente inteligente capaz de percibir su entorno a través de cuadros de video, y responder o actuar en su entorno de manera racional, es decir, correctamente y con la tendencia a maximizar una recompensa o resultado esperado, mediante el desarrollo de una capacidad de anticipación. La línea de investigación de este proyecto es la teoría de juegos y sus posibles aplicaciones en otros campos. Finalmente, el rendimiento se comparará con los métodos del artículo para probar su precisión y aplicabilidad.

Palabras Clave:
aprendizaje de refuerzo profundo, agentes, redes neuronales artificiales.

Abstract

All human cultures throughout history have played games and from the beginning of Artificial Intelligence development, there has existed a great interest in games as a research platform. Games such as Backgammon, Chess, Checkers, Go, Othello and Tic-Tac-Toe are widely used for studying the learning ability of machines and developing learning algorithms by the great contestants in the digital world such as Google, Facebook, Windows etc. This project aims to improve the programs, methods and results obtained in the paper Self-taught Neural Agents in Clever Game Playing, which uses intelligent agents and reinforced learning in neural networks. The central feature in this project is an intelligent agent capable of perceiving its environment through video frames, and responding or acting in its environment rationally, that is, correctly and with the tendency to maximize an expected reward or result, by developing a look ahead capacity. The research track of this project is game theory and its possible applications in other fields. Finally, the performance will be compared with the paper's methods to test its accuracy an applicability.

Keywords:
deep reinforcement learning, agents, artificial neural networks.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Artificial intelligence over the last few years has made great strides with reinforcement learning (RL) in the last century and with the advent of deep learning (DL) in the 1990s, especially the advance of convolutional networks in the field of artificial vision. Both approaches have led to the adoption of neural networks in RL, which has allowed a significant advance in human-level agents and autonomous systems [5]. This new technology called deep reinforcement learning has proved very successful in mastering human-level control policies in various tasks, such as object recognition with visual attention and control of high-dimensional robots. In particular, Deep QNetworks (DQN) has been shown to be effective for playing Atari 2600 games and most recently defeating world-class Go players [6].

Since this new approach combining RL and DP has been quite successful in games and computer vision, efforts have been made to develop and implement agents that can think and act autonomously in the real world. In this work, we have implemented a neural agent that learns to play tic-tac-toe, an ancient game that has been played for many human generations, and historical records of its existence have been found in ancient Egypt [7]. The neural agent is developed through an architecture formed by convolutional networks and a fully connected sigmoidal neural network. In addition, it is based on an approximation of the Bellman equation where MAX is not searched for in the Q domain but in the R domain. Finally, the results obtained are compared with a previous work called Self-taught Neural Agents in Clever Game Playing.

1.2 Problem Statement

Learning from video frames is an important issue in artificial intelligence because it opens the way to many practical applications in real life, such as self-driven vehicles, robotics, military, surveillance, computer-aided medicine, and others. Putting deep reinforcement learning into practice, a new field of great interest for researchers today has motivated the development of this study and work, where computer vision has allowed computers to obtain understanding through images and videos. Moreover, the games, specifically the tic-tac-toe, where reliable rules and enough complexity are combined to require strategies to achieve victory.
Moreover, the evolution of Deep Reinforcement Learning requires an environment that allows us to evaluate new strategies. The development of these environments imposes new challenges such as studying the behavior of the agents to find optimal policies, adjusting parameters of the visual processing components, implementing low-level algorithms to reduce the execution time associated with high-level languages.

1.3 Objectives

1.3.1 General Objective

To develop a deep reinforcement learning (DRL) environment where a computer software learns to play high level tic-tac-toe by watching video images of a physical board observed through a web cam.

1.3.2 Specific Objectives

- To study the agent behavior and improve its ability to learn optimal policies that give origin to intelligent game strategies.
- To tune the many parameters of the used visual processing and its deep neural network so that a successful deep control system is obtained.

- To debug and stabilize the required software written in C++ so that it can be used in future DRL research.

1.4 Contributions

This work proposes a deep reinforcement learning system where an agent rapidly learns to play high-level tic-tac-toe. So, it develops the software structure that makes possible to prove a valid, fast approximation of the Bellman equation, where the agent searches for reward in the R-matrix and not in the Q-matrix as it is normally done. Fast reinforcement learning is an important issue in rapid moving environments such as car driving or missile guiding.

1.5 Document Organization

This work is structured with 6 main sections. These are Introduction, Theoretical Framework, Related Works, Methodology, Results and Discussion, Conclusions and Future Work.

1. Section 1, presents a short introduction to this work, the problem statement and all the objectives, contributions and the document organization.
2. Section 2, establishes all the concepts to understand this work.
3. Section 3, presents a brief summary about all the related works.
4. Section 4, presents the methodology used in this work.
5. Section 5, discuss the results obtained.
6. Section 6, summarizes the best of this work and gives some ideas about looking forward in this work.

Chapter 2

Theoretical Framework

This chapter presents all the artificial intelligence concepts used to develop this work. These concepts are explained so that they can be understood in an easily way, starting from the basic concept of neural networks to neural agents.

2.1 Artificial Neural Networks

The human brain is a complex system capable of thinking, remembering, and solving problems. Throughout history, attempts have been made to emulate brain functions with a computer model. Generally, these have involved simulating a network of neurons, commonly called Artificial Neural Networks [8]. An Artificial Neural Network (ANN) is a mathematical model that tries to mimic the structure and functionalities of biological neural networks. The basic building block or basic unit of every artificial neural network is an artificial neuron which is a simple mathematical model (function) [1]. This model has three simple rules: multiplication, sum, and activation. The flow of information within an artificial neuron occurs as follows; each input is weighted, which means that each input value is multiplied by the individual weight. Then, the sum function is performed in the middle section of the artificial neuron, where all the weighted inputs and biases are added [1]. Finally, at the output of the artificial neuron, the sum of the previously weighted inputs and biases passes through the activation function, also called the transfer function, as shown in Figure 2.1. Next, the components of a neural network and the elements required for its training are described, such as loss function, transfer function, learning rate and epochs. Also, the different paradigms that exist in artificial neural networks, the backpropagation algorithm, the concepts of reinforcement learning, Markov decision process, dynamic programming, Bellman equation, Q-learning, convolutional networks and deep reinforcement learning are described.

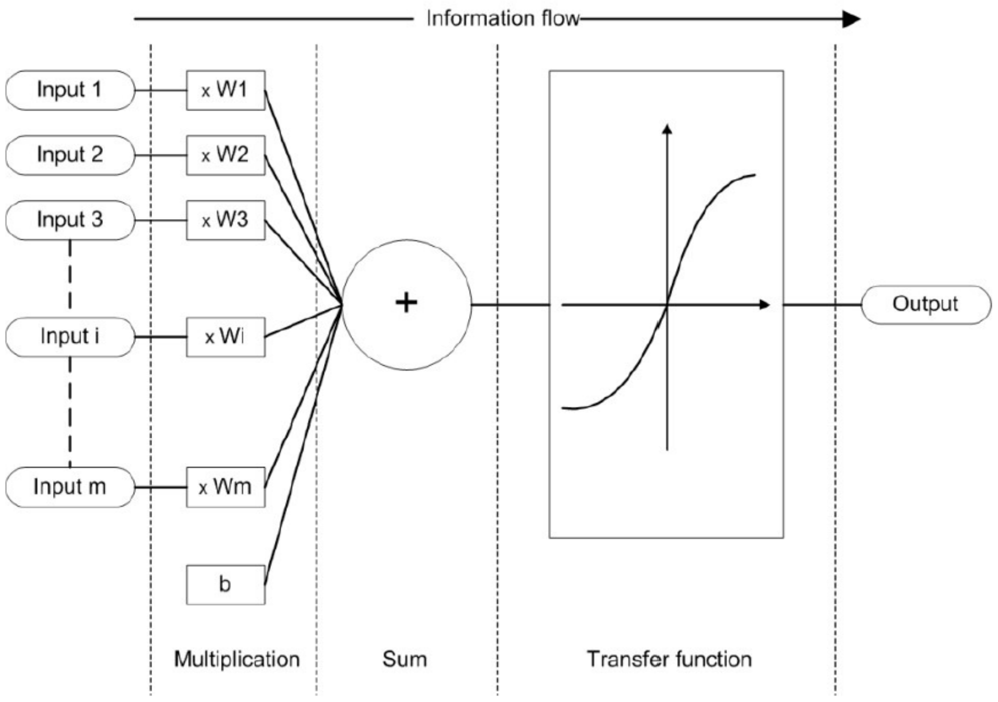


Figure 2.1: Working principle of an artificial neuron [1]

2.1.1 Artificial Neuron

The artificial neuron is the essential component of any artificial neural network, and it tries to replicate the structure and behavior of the natural neuron [9]. Its design and functionalities have been implemented thanks to the study of the biological neuron, the fundamental component of biological neural networks that are part of the brain, spinal cord, and peripheral ganglia [1]. The similarities in the design and functionalities of the biological neuron and the artificial neuron can be seen in Figure 2.2 and Figure 2.3. Figure 2.2 represents a biological neuron with its soma, dendrites, and axon, and Figure 2.3 represents an artificial neuron with its inputs, weights, transfer function, bias, and outputs.

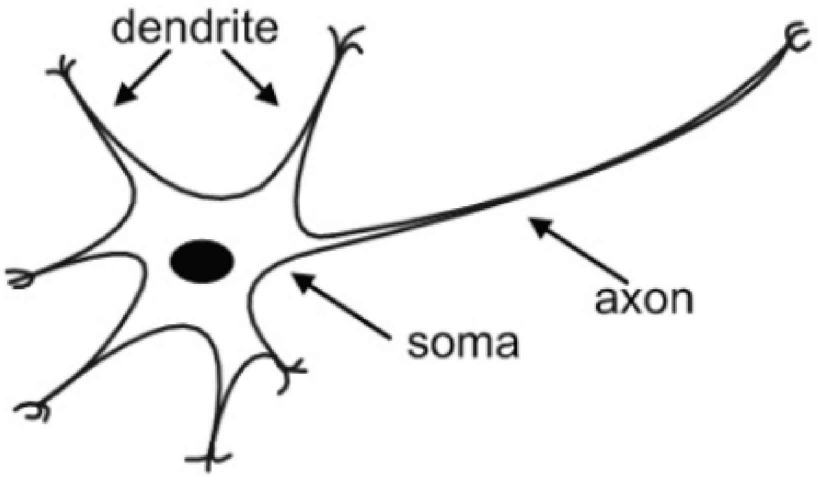


Figure 2.2: Biological Neuron [1]

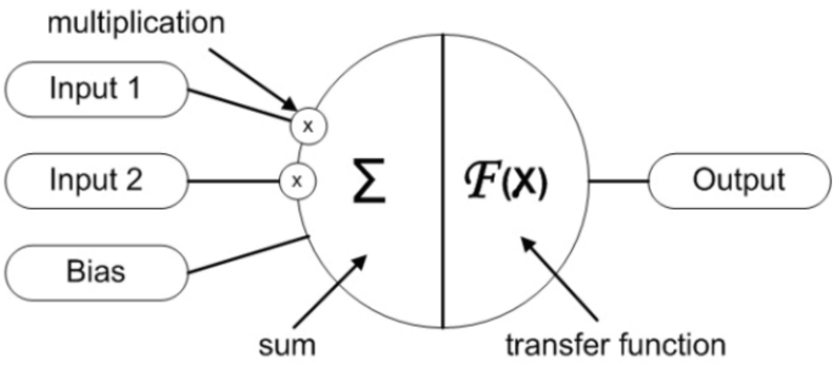


Figure 2.3: Artificial Neuron [1]

In the case of a biological neuron, information comes into the neuron via dendrite; soma processes the information and passes it on via axon [1]. On the other hand, in an artificial neuron, information enters through weighted inputs which are individually multiplied by a weight. It then adds the weighted inputs, bias and processes the sum with a transfer function. Finally, an output is produced. For more detail, an artificial neural network can be studied using the following mathematical model:

$$y(k) = F\left(\sum_{i=0}^{m} w_i(k) \cdot x_i(k) + b\right) \tag{2.1}$$

where:

- $x_i(k)$ is the input at discrete time k,

- $w_i(k)$ is weight value in discrete time,
- b is bias,
- F is a transfer function,
- $y(k)$ is output value in discrete time k.

2.1.2 Loss Function

The loss functions show how far the prediction is from the actual prediction. The machines learn to change/decrease the loss function by moving closer to the ground reality. There are many functions to find the loss based on predicted and actual values depending on the problem. Moreover, optimizers are used to minimize loss to make better predictions [10]. One of the most used loss functions is the mean square error (MSE) which is the average of the squared difference between the predicted values and the real values. It is defined as follows:

$$\mathrm{MSE}(x,\hat{x}) = \frac{\sum_{i=1}^{N} \|x_i - \hat{x}_i\|^2}{N} \tag{2.2}$$

Where x is the target, $\hat{x}$ is the obtained value and $N$is the number of samples.

2.1.3 Transfer Function

Transfer functions or also known as activation functions are those that are used in artificial neural networks to calculate the neuron's output. It receives the weighted sum and biases, which is used to decide if a neuron can be activated or not [11][12]. In addition, it is one of the most important variables in equation 2.1, and it defines the properties of artificial neurons and can be any mathematical function [1], there are different types, and each one is used according to the problem that needs to be solved. The definition of the most used activation functions is presented below.

Linear Function

The linear activation function is directly proportional to the input. For this reason, it is not of great benefit to use this activation function because the neural network will not be able to identify complex patterns from the data. Therefore, linear functions are ideal where interpretability is required and for simple tasks [13]. It is defined as follows:

$$f(x) = x \tag{2.3}$$

Binary Step Function

The binary step function is the most straightforward activation function and is generally used to create a binary classifier. For its mathematical representation, the value of the function changes abruptly when a threshold value θ is reached [14]. It is defined as follows:

$$f(x) = \begin{cases} 0 & \text{if } x \leq \theta \\ 1 & \text{if } x > \theta \end{cases} \tag{2.4}$$

Sigmoid Function

Also known as the logistic or squashing function, it is one of the most used activation functions since it is a nonlinear function, making it more effective than the linear and step activation function. The sigmoid function transforms the values into the range from 0 to 1 in an S-shape [13]. The sigmoid function is used in the output layers of the Deep Neural Networks presented in this work and is used for probability-based output [15]. It is defined as follows:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

Hyperbolic Tanget Function

The hyperbolic tangent function is centered at zero, and its range is between -1 and +1. So it makes it easy to model inputs with strongly negative, neutral, and strongly positive values [15]. It is defined as follows:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.6}$$

Rectified Linear Unit Function

The Rectified Linear Unit function has excellent performance and a simple structure, is more efficient than other functions because it helps deep neural networks realize sparse activation, and sparse activation is not only more in line with the mechanism of human brain activity, it has a good advantage at the mathematical level [16]. The ReLU activation function produces 0 as an output when $x \leq 0$, and then produces a linear with slope of 1 when $x > 0$. It is defined as follows:

$$\text{ReLU}(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.7}$$

There are some variations of this activation function to avoid the problem of dead neurons. There are cases where certain neurons do not participate and remain without activity during the backpropagation step in neural network training. Some of these variations are: Leaky ReLU function, Parametrized ReLU function, and Exponential Linear Unit [13].

Leaky ReLU Function

Leaky ReLU is a version of the ReLU function where for negative values of x, the values are defined as a minimal linear component of x instead of defining the value as zero [16]. It can be expressed mathematically as:

$$\text{Leaky-ReLU}(x) = \begin{cases} 0.001x & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.8}$$

Parametrized ReLU Function

It is also a variant of the ReLU function with a slight variation, just like Leaky ReLU. This function solves the problem of the ReLU gradient becoming zero for negative values of x by introducing a new parameter a. The value of a, when set to 0.01, behaves like a leaky ReLU function, but here a is also a trainable parameter. For optimal and faster convergence, the network learns the value of a [13].

$$\text{Parametrized-ReLU}(x) = \begin{cases} ax & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.9}$$

Exponential Linear Unit

The Exponential Linear Unit or ELU is also a variant of the Rectified Linear Unit. ELU introduces a parameter slope for negative values of x, for which it uses a logarithmic curve [13]. It can be expressed mathematically as:

$$\text{ELU}(x) = \begin{cases} a(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases} \tag{2.10}$$

2.1.4 Learning Rate

In artificial neural networks, the learning rate is a parameter that determines how much the weights can change during the training phase in response to the observed error. The value of this constant is usually in the range [0;1], and the choice of this learning rate can greatly affect generalization accuracy as well as training speed [17]. For example, if the learning rate is too large, the accuracy will be poor, and the training speed will be poor.

2.1.5 Epochs

An epoch indicates the number of passes of the entire training dataset the neural network has completed. It means if the training set is finite, training occurs by performing iterations for one cycle [17]. For example, If a neural network is trained to 1000 epochs, the learning algorithm moves through 1000 different models [18].

2.1.6 Artificial Neural Networks Paradigms

The three main learning paradigms or techniques are supervised, unsupervised, and reinforcement. Supervised is the most common training paradigm used today to develop neural network prediction and classification applications. In contrast, unsupervised learning is often used for clustering and segmentation in data mining to support decision-making in time optimization and adaptive control [19]. Figures 2.4, 2.5, and 2.6 show a general representation of each paradigm.

Supervised Learning

Supervised learning is based on using labeled data sets to train algorithms that accurately classify data or predict outcomes. As input data is fed into the model, the model adjusts its weights until the model has been fitted correctly [20]. In this way, the model is trained until it detects the relationship between the input data and the output labels, which allows it to generate accurate tagged results when presented with never-before-seen data. In other words, supervised learning employs a ”teacher” to assist in training the network by telling the network what the desired response to a given stimulus should be [21].

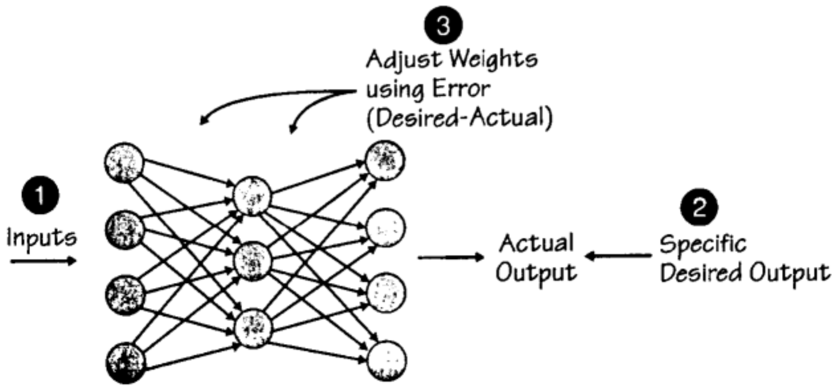


Figure 2.4: Supervised Learning paradigm [2]

Unsupervised Learning

Unsupervised learning is very similar to supervised learning, but with the difference that there is no ”teacher” in the learning process [21]. Unsupervised neural networks discover in the input data and autonomously: characteristics, regularities, correlations, and categories. Also, neural networks trained using unsupervised methods are considered self-organizing because they are not given any indication of what to expect or what the correct output should be. When presented with a set of input patterns, output processing units organize themselves, initially competing to identify patterns and then cooperatively adjusting their connection weights [2].

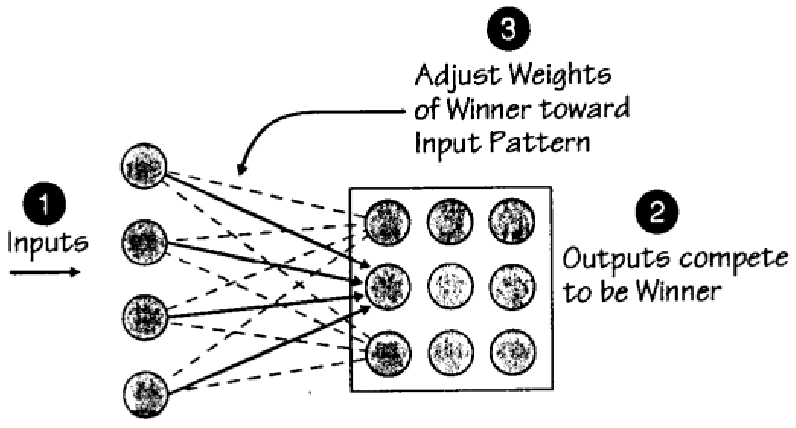

3
Adjust Weights of Winner toward Input Pattern
1
Inputs
2
Outputs compete to be Winner

Figure 2.5: Unsupervised Learning paradigm [2]

Reinforcement Learning

Reinforcement learning is inspired by behavioral psychology; it is a machine learning technique that establishes the parameters of an artificial neural network, where data is generally not provided but generated through interactions with the environment [19]. In addition, the neural network reinforcement learning approach allows for solving challenging temporal (time-dependent) problems [2]. It has been applied successfully to various problems, including robot control, telecommunications, and games such as chess and other sequential decision-making tasks [19]. In section 2.4, we will delve into the concepts of this machine learning technique.

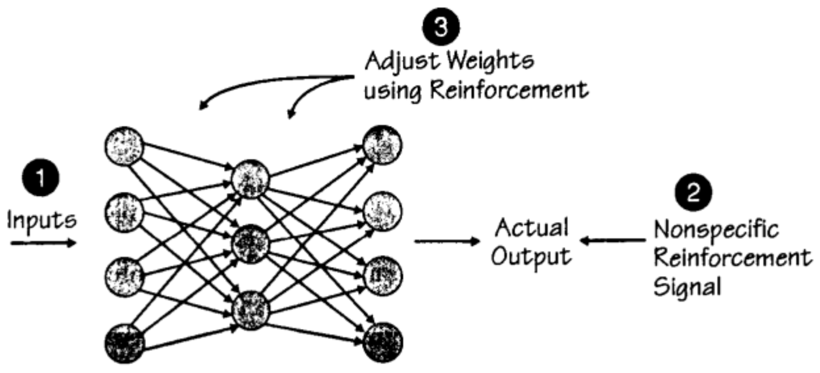

3
Adjust Weights using Reinforcement
1
Inputs
Actual Output
2
Nonspecific Reinforcement Signal

Figure 2.6: Reinforcement Learning paradigm [2]

2.2 Backpropagation Algorithm

The backpropagation algorithm has become the most popular method of training neural networks due to its underlying simplicity and relative power. It utilizes the loss function explained in subsection 2.1.2 and gradient descent to realize the modification to the connection weight of the network [22]. It contains two main phases, which are the forward and backward phases, the first one is required to compute the output values and the local derivatives at various nodes, and the second one is required to accumulate the products of these local values over all paths from the node to the output.

i. Forward phase: In this phase, the neural network is fed through the inputs; this results in a cascade of computations forward through the network layers, using the current weight values. Then, the derivative of the loss function with respect to the output is calculated. At this point, it is necessary to calculate the derivative of this loss for the weights in all layers of the backward phase [23].

ii. Backward phase: The main objective of this phase is to learn the gradient of the loss function for the different weights using the chain rule of the difference calculus [23]. These gradients are used to update the weights, and this process is from the output node to the input nodes.

2.2.1 Description of Backpropagation Algorithm in Mathematics

Let's define Z^L as the result of the weighted sum of the last layer which is expressed as

$$Z^L = W^L X + b^L \tag{2.11}$$

where:

- W^L are the weights of the layer L,
- X are the weight values,
- b^L is bias of the layer L

The weighted sum of the last layer L is passed through the activation function (a), so we have

$$a\left(Z^L\right)$$

The result of the activation of the neurons in the last layer makes up the result of the artificial neural network, which is evaluated by the cost function to determine the error, it is defined as

$$C\left(a\left(Z^L\right)\right)$$

Thus, we obtain a composition of functions and to calculate its derivative the chain rule is used, therefore we have the following expressions:

$$Z^L = W^L a^{L-1} + b^L \tag{2.12}$$

The derivative of the parameter w^L with respect to the cost function is:

$$\frac{\partial C}{\partial w^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} \tag{2.13}$$

The derivative of the parameter b^L with respect to the cost function is:

$$\frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L} \tag{2.14}$$

where:

- $\frac{\partial C}{\partial a^L}$ is the derivative of the activation function with respect to the cost function,
- $\frac{\partial a^L}{\partial z^L}$ is the derivative of the activation function with respect to the weighted sum z^L,
- $\frac{\partial z^L}{\partial w^L}$ and $\frac{\partial z^L}{\partial b^L}$ is the derivative of the weighted sum with respect to the parameters w^L and b^L, respectively.

The derivative of the activation function with respect to the cost function tells us how the cost of the artificial neural network varies when the output varies. For example, if our cost function is the mean square error

$$C\left(a_j^L\right) = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2 \tag{2.15}$$

The derivative of the function with respect to the output of the artificial neural network is

$$\frac{\partial C}{\partial a_j^L} = \left(a_j^L - y_j\right) \tag{2.16}$$

The derivative of the activation function with respect to z^L tells us how the output of the neurons varies when we vary the weighted sum. For example, if our activation function is a sigmoid function

$$a^L\left(z^L\right) = \frac{1}{1 + e^{-z^L}} \tag{2.17}$$

The derivative will be

$$\frac{\partial a^L}{\partial z^L} = a^L\left(z^L\right) \cdot \left(1 - a^L\left(z^L\right)\right) \tag{2.18}$$

The derivative of the weighted sum with respect to the bias term is 1, since it is independent, so its derivative is constant

$$\frac{\partial z^L}{\partial b^L} = 1 \tag{2.19}$$

The derivative of the weighted sum with respect to the term w^L is the output of the previous layer

$$\frac{\partial z^L}{\partial w^L} = a_i^{L-1} \tag{2.20}$$

In this way, the first two terms of the equation 2.13 can be defined as

$$\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \tag{2.21}$$

So, the equation 2.21 tells us how the error varies as a function of z^L, which is the value of the weighted sum, this is known as the error imputed to the neuron and is represented by the symbol δ^L.

Therefore, restructuring and simplifying equations 2.13 and 2.14 as a function of the error of the neurons in the L layer, we have that

$$\frac{\partial C}{\partial b^L} = \delta^L \cdot \frac{\partial z^L}{\partial b^L} \tag{2.22}$$

$$\frac{\partial C}{\partial w^L} = \delta^L \cdot \frac{\partial z^L}{\partial w^L} \tag{2.23}$$

Replacing equation 2.19 in 2.22 and equation 2.20 in 2.23 we have that

$$\frac{\partial C}{\partial b^L} = \delta^L \tag{2.24}$$

$$\frac{\partial C}{\partial w^L} = \delta^L a_i^{L-1} \tag{2.25}$$

where equation 2.24 indicates that the derivative of the cost function with respect to the bias term is equal to the error of the neurons and equation 2.25 indicates that the derivative of the cost function with respect to the term w^L is equal to the error of the neurons multiplied by the activation of the previous layer.

Now, to calculate the parameters of the previous layer $L-1$ we apply the chain rule to our new composition of functions

$$C\left(a^L\left(W^L a^{L-1}\left(W^{L-1}a^{L-2} + b^{L-1}\right) + b^L\right)\right)$$

so we have that

$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}} \tag{2.26}$$

$$\frac{\partial C}{\partial b^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial b^{L-1}} \tag{2.27}$$

In equations 2.26 and 2.27 the only term that we need to calculate is

$$\frac{\partial z^L}{\partial a^{L-1}} = W^L \tag{2.28}$$

which is the matrix of parameters that connects both layers, that is, the layer L and $L-1$

Finally, we can apply the same logic for the previous layers and it can be summarized as follows:

i. Computation of the error of the last layer

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \tag{2.29}$$

ii. Backpropagation of the error to the previous layer

$$\delta^{l-1} = W^l \delta^l \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} \tag{2.30}$$

iii. Calculation of the derivatives of the layer using the error

$$\frac{\partial C}{\partial b^{l-1}} = \delta^{l-1} \quad \frac{\partial C}{\partial w^{l-1}} = \delta^{l-1} a^{l-2} \tag{2.31}$$

2.3 Reinforcement Learning

Reinforcement learning (RL) refers to the problem of a learning agent interacting with its environment to achieve a goal [24]. It mainly combines two tasks; the first is to explore new situations because the agent does not receive examples or instructions of the desired behavior; this is done by trial and error. The second is to use that experience to make better decisions and get the most reward. In other words, the agent has to exploit what it already knows to get the reward, but it also has to explore in order to make better action selections in the future [25].

In addition to the agent and the environment, four main sub-elements of a reinforcement learning system can be identified: a policy, a reward signal, a value function, and, optionally, an environment model [25].

i. A **policy** is the core of a reinforcement learning agent as it determines how it behaves at any given time. Typically, the policy may be a simple function or lookup table, while in others, it may involve extensive computation, such as search process [25].

ii. A **reward signal** is a goal in a reinforcement learning problem; it defines the good and bad events for the agent. Therefore, the agent's sole objective is to maximize the total reward it receives in the long run [25].

iii. A **value function** depends on how the agent picks actions to perform.

iv. An **environment model**, which is an element of some reinforcement learning systems, is something that mimics the behavior of the environment; it allows inferences to be made about how the environment will behave. For example, given a state and an action, the model could predict the next resulting state and the next reward [24].

2.3.1 Exploration

In the exploration phase, the agent has to test actions to gather information and thus make better selections of actions in the future to obtain the highest reward.

2.3.2 Explotation

In the exploitation phase, the agent chooses actions that it has carried out in the past and that maximized its accumulated reward where it proved to be more efficient.

2.4 Markov Decision Process

The Markov Decision Process (MPD) is a mathematical model used to solve reinforcement learning problems, which is defined as follows [7] [3]:

i. $\mathcal{S}$ is a finite set of states, where $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}, s_t \in \mathcal{S}$ denotes the state at time t.

ii. $\mathcal{A}$ is a finite set of actions, where $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}, a_t \in \mathcal{A}$ denotes the action executed at time t.

iii. $\mathcal{P}$ is a transition function, where $P(s, a, s')$ specifies the probability of arriving at any state $s' \in \mathcal{S}$ after performing action a in state s.

iv. R is a reward function, where $R(s, a)$ is the reward of executing action a in state s, r_t denotes the reward function obtained at time t.

v. A discount factor $0 \leq \gamma \leq 1$, which makes the agent value immediate rewards more than later rewards.

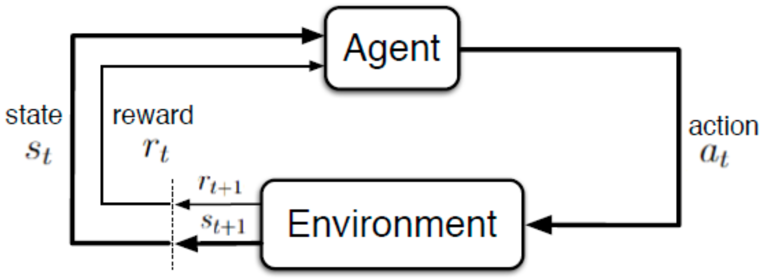


Figure 2.7: The agent-environment interaction in reinforcement learning [3]

2.5 Dynamic Programming

Dynamic programming (DP) like the divide-and-conquer method, is a technique that solves problems by combining the solutions to subproblems. It is usually used in optimization problems and dynamic programming algorithms can be developed following the next steps [26]:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Performing steps 1 through 3 forms the basis of a dynamic programming solution to a problem. If we only need the value of an optimal solution, and not the solution itself, then we can skip step 4. However, when we do step 4, we sometimes keep additional information during step 3 so that we can easily build an optimal solution. In this context, we can convert Bellman equations (section presented in 2.6) into update rules to improve the approximations of the desired value functions.

2.6 Bellman Equation

The Bellman equation expresses a relationship between the value of a state and the values of its successors states and is defined as follows [25]:

$$v_\pi(s)=\mathbb{E}_\pi\left[G_t \mid S_t=s\right]=\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \mid S_t=s\right] \tag{2.32}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step, the function v_π is called the state-value function for policy π.

In the same way, the action-value function defined as $q_\pi(s,a)$ for policy π define the value of taking action a in state s under a policy π. This function is defined as:

$$q_\pi(s,a)=\mathbb{E}_\pi\left[G_t \mid S_t=s, A_t=a\right]=\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1} \mid S_t=s, A_t=a\right] \tag{2.33}$$

So, the general equation used in reinforcement learning problems is the following:

$$Q(s,a)=r+\gamma\max_{a'}Q\left(s',a'\right) \tag{2.34}$$

2.7 Q-Learning

Q-learning was implemented in 1989; it is a form of model-free reinforcement learning, and it can also be viewed as a method of asynchronous dynamic programming (DP) [27]. The basic version of Q-learning keeps a lookup table of values $Q(s,a)$(Equation 2.36) with one entry for every state-action pair. Also, is one of the reinforcement learning techniques which not require a model of the environment to learn to execute complex tasks [27]. It works by successively improving its evaluations of the quality of particular actions at particular states. Therefore, the goal of Q-Learning is to learn a set of rules or travel chart that tells an agent what action to take under what circumstances. $Q\left(s_t,a_t\right)$ means the value of taking action a_t in a state s_t. The equation 2.34 is the basis of the Q-learning algorithm, the value $Q\left(s_t,a_t\right)$ of a current state and action can be decomposed into to the immediate reward r plus the discounted maximum future expected reward after the transition to a next state s_{t+1}. This is known as the Bellman equation and can be written as follows [28]:

$$Q\left(s_t, a_t\right)=r+\gamma \max_a Q\left(s_{t+1}, a_{t+1}\right) \tag{2.35}$$

where γ is the discount factor. The value $Q\left(s_t, a_t\right)$ is computed by the agent and then use the following equation 2.35 to update its own estimate of $Q^*\left(s_t, a_t\right)$. The equation is defined by

$$Q^*\left(s_t, a_t\right)=Q\left(s_t, a_t\right)+\alpha\left[r+\lambda \max_a Q\left(s_{t+1}, a_{t+1}\right)-Q\left(s_t, a_t\right)\right] \tag{2.36}$$

where α is the learning rate. The $\max_a Q\left(s_{t+1}, a_{t+1}\right)$ gives the maximum value for all actions in the next state. Q-learning is an off-policy algorithm since it updates the Q-values without making any assumptions about the actual policy being followed [28].

2.8 Convolutional Neural Networks

Convolutional Neural Networks (CNN or ConvNets) are among the most popular categories of neural networks, especially for high-dimensional data, such as images or videos [4]. They are a network architecture for deep learning that learns directly from data without manually extracting features. Furthermore, these networks are particularly useful for finding patterns in images to recognize objects, faces, and scenes directly from pixels. They also effectively classify non-image data, such as audio data, time series, and signals [29]. Also, these networks can have tens or hundreds of layers that learn to detect different features of an image, from here the terminology of deep networks. Filters are applied to each training image with different resolutions, and the output of each convolved image is used as input for the next layer. Filters can range from elementary features, such as brightness and edges, to more complex ones, such as features that uniquely define the object [29].

2.8.1 Convolutional Layer

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. A convolution takes groups of neighboring pixels from the input image, a tiny image and operates mathematically with a small matrix called a kernel or filter.

(a)–(i) show the computations performed at each step, as the filter is slid onto the input

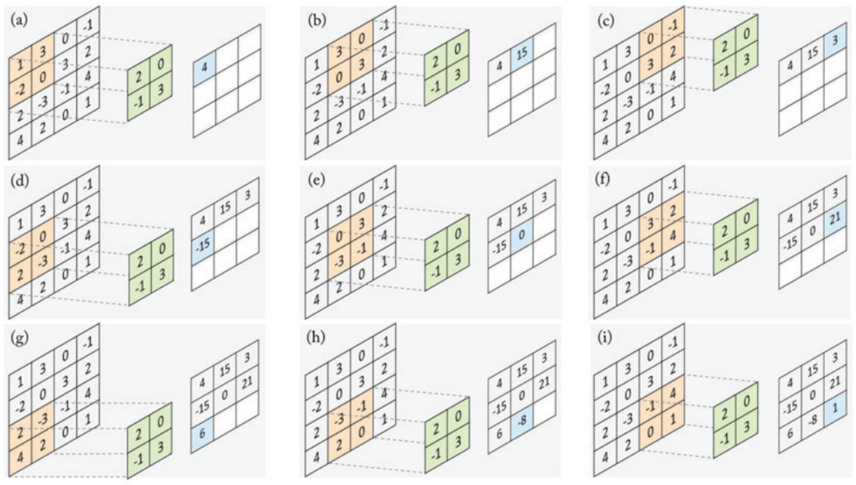


Figure 2.8: The operation of a convolution layer [4]

feature map to compute the corresponding value in the output feature map [4].

2.8.2 Kernel

The kernel is the number of pixels processed together as a tiny matrix. As in traditional neural networks, the kernel values or weights have an initial value at the beginning of the training. Later, during the process, these weights are updated until they approach the optimal conditions to make good predictions.

2.8.3 Stride

The stride is a component of convolutional neural networks or neural networks tuned for the compression of images and video data. Stride is a neural network's filter parameter that modifies the amount of movement over the image or video. If a neural network's stride is set to 1, the filter will move one pixel at a time.

2.8.4 Padding

Padding is the number of pixels added to an image when it is being processed, allowing for more accurate analysis. This padding adds extra space around the image, which helps the kernel improve performance.

2.8.5 Pooling Layer

A pooling layer operates on blocks of the input feature map and combines the feature activation. This combination operation is defined by a pooling function such as the average or the max function. Similar to the convolution layer, we need to specify the size of the pooled region and the stride [4].

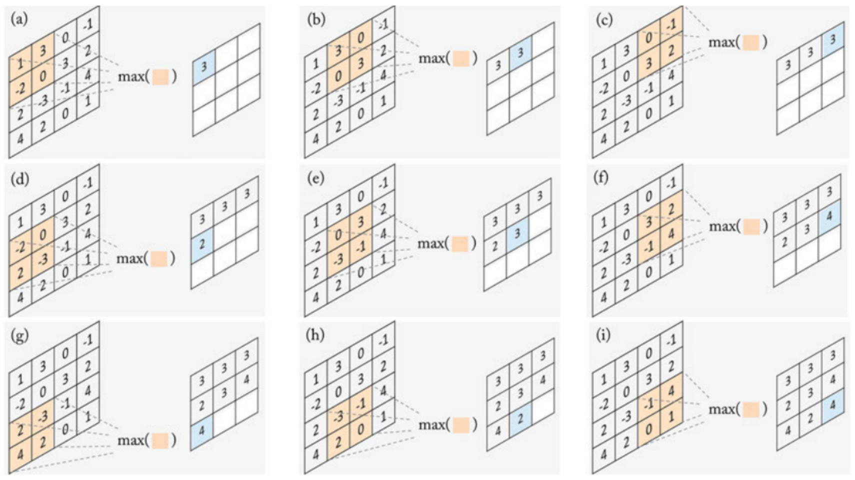


Figure 2.9: The operation of max-pooling layer when the size of the pooling region is 2x2 and the stride is 1 [4]

2.9 Deep Reinforcement Learning

Deep reinforcement learning (DRL) is one of the fields with remarkable growth both in industry and in research. It represents a step towards constructing autonomous systems with a higher level of understanding of the visual world. DRL combines deep networks with reinforcement learning and allows the techniques of machine learning to be extended to previously intractable problems, such as learning to play video games directly from pixels [30].

One of the goals of DRL is to create systems that are capable of learning to adapt to the real world. For this reason, several investigations and previous works in DRL have been based on the extension of the previous work in RL to high-dimensional problems. This involves bringing together the learning of low-dimensional feature representations and neural networks' powerful function approximation properties. For example, convolutional neural networks (CNNs) can be used as components of RL agents, allowing them to learn directly from visual inputs. DRL is generally based on training deep neural networks to approximate the optimal policy and the optimal value functions [30].

2.10 Summary of Concepts

This chapter began by introducing the concept of artificial neural networks, their basic structure, the artificial neuron, and how it can be implemented mathematically to have the same functionalities as a biological neuron. The elements required for training are the loss function which shows how far the prediction is from the actual. The transfer function which is used to decide if a neuron can be activated or not, its different types such as linear function, binary step function, sigmoid function, hyperbolic tangent function, rectified linear unit function, and its different variations. In addition, learning rate and epochs were explained which are essential during training. Then, the different paradigms that exist in artificial neural networks such as supervised, unsupervised, and reinforcement learning were explained. The backpropagation algorithm is the most popular method for training neural networks. The concepts of reinforcement learning in detail since it is a fundamental topic in the development of this thesis. Finally, the concepts of the Markov decision process, dynamic programming, Bellman Equation, Q-learning, and convolutional neural networks were presented. All these concepts are related to the methodology (Chapter 4) to create a neural agent that learns and finds the best policies to play three in a row.

Chapter 3

Related Work

In this chapter, some deep reinforcement learning works were reviewed, including the methods used to obtain a neural agent that learns to play a specific game.

3.1 Chang, Zhinin-Vera & Quinga

In this work, Chang, Zhinin-Vera, and Quinga [7] proposed a Tic-Tac-Toe learning environment based on a self-motivated neural agent that learns the game situations and then uses the knowledge in real-world tournaments, where it mimics a Markov model. Their work aimed to develop self-taught agents who take on a future vision of the game, that is, a brilliantly anticipated sequence of moves, as the authors call it, an "I already won" or IAW+4 game vision. They implemented the model of the self-motivated neural agent as a chain of 9 sigmoidal neurons that operate in real-time and inhibit each other with small common negative weights. All neurons are equally excited by a repetitive ramp K. On each repetition, the agent burns "dark energy," fires, and declares a single winner, which is used to pick a tile on the board. The agent will continue to give valid moves even if it is disconnected from the (external) advisory neurons.

Furthermore, for the learning networks, they first took the state of each tile, which is represented by three neural signals 010, 001, and 100, representing filled "empty" tiles, "O" and "X," respectively. The resulting 27 signals are a sparse encoding representation of the state of the board and are fed into a network with 27 inputs, 67 hidden, and 9 output neurons. During training, the agent explores future moves based on the Bellman equation, which involves the first three terms of the equation, and thus memorizes game patterns that ensure winning situations. After training, the most exciting output neuron will indicate the optimal policy predicted by the network, which is what should be filled or, in other words, what move should be made. Many of these subnets are used and trained as indexable advisors, which advise the agent to make smart moves.

Finally, during the operation phase, the neural agent receives advice from the trained networks and recognizes and executes IAW+4 game situations with high security. Therefore, the authors have succeeded in showing that the self-motivated neural network can be used as a free-running random agent that explores all possible game situations and is trained with backpropagation to memorize good game sequences by using advisory indexable subnets. Also, the reinforcement learning method used supports a successful future search for maximum rewards. This work could be improved by introducing deep reinforcement learning in the model and this is what this thesis proposes.

3.2 Chang & Zhinin-Vera

Chang and Zhinin-Vera [31] presented an inspired robot capable of learning high-level tic-tac-toe game policies on its own and then using that knowledge to compete advantageously with humans. The robot has been designed to execute the physical actions resulting from the logical decisions of the self-taught neural agent. The robot was built with a robotic arm, a machine vision system, and a self-motivated neural agent.

The agent is based on the architecture implemented in the related work of section 3.1, where the Bellman equation with three terms has been proposed to search for future rewards. The mechanical design of the robot uses three axes: shoulder, elbow, finger, and servo motors, power supply, Arduino board, and connections from the computer to the servo motor. Regarding artificial vision, this has been implemented with a camera that captures color images of the real world. These images are processed using OpenCV and sent to a convolutional network specialized in recognizing Xs, Os, and empty spaces.

As a result, the authors have managed to implement a robot where it observes the game board and uses its robotic arm to perform its movements. These actions carried out by the robot are thanks to the neural agent that has learned to play the game of tic-tac-toe in an extraordinary way and to the artificial vision system to look at the board in the real world. This work is related to 3.1 and has been taken into account since the built robot can be used to test the deep reinforcement learning software that will be implemented in this thesis.

3.3 Gatti & Embrechts

Gatti and Embrechts [32] presented an agent that learns to play tic-tac-toe; first, they built the environment, which consists of encoding the board's structure and the rules of the game. The environment consisted of a 3×3 matrix with each element corresponding to a location on the tic-tac-toe board. The game's rules defined the legal actions that each player could take; players could only place pieces in open locations on the board. In addition, it was verified if any of the players had won to end the game and give the corresponding reward to the agent. The agent represented by O, received a reward of 1 if he had won the game, -1 if he had lost game, or 0 if the game ended in a draw.

The agent was represented by a 3-layer neural network with 11 input nodes, 40 hidden nodes, and 1 output node. They used nodes in the input and hidden layers, with a constant input of 1. The nodes in the hidden layer used the hyperbolic tangent transfer function and used a linear transfer function at the output node. On the other hand, the learning rates α for each layer were established individually, with $\alpha = 0.01$ for the input hidden layer weights and $\alpha = 0.007$ for the output hidden layer weights. The network weights were updated through an iterative method to update them after each training episode.

The agent was trained and evaluated to know the performance it can perform. The agent was trained with a game strategy of its own. Greedy late-game moves were allowed during training games, including greedy wins and greedy bans. Later, during evaluation games, the agent was not allowed to make greedy moves at the end of the game but had to rely only on his learned knowledge. The agent's gaming performance was evaluated over 500 games against a random opponent so that all of this player's moves were randomly selected. Furthermore, the agent represented by O always made the first move in both the training and evaluation games.

The results obtained by the authors were that the agent was able to learn to play the game of tic-tac-toe against a random opponent at an acceptable level with relatively little training. In addition, the game was learned by the agent in about 500 games to win almost 95% of the evaluation games. This work presents very good results, however, it is tied to the Q matrix. For the development of this thesis, it is proposed to change the Q matrix by a neural network called Q-net to give the model more flexibility.

Chapter 4

Methodology

This chapter presents the entire process that has been followed to meet the objectives of the project. It describes the implementation of a neural agent which learns to play tic-tac-toe.

4.1 Phases of Problem Solving

- The first step is to implement the environment and rules on which the tic-tac-toe game is based.
- The second step is to implement the deep neural network architecture defining the number of convolutional layers, filters, max pool layer, inputs (input layer), the number of hidden neurons (hidden layer), and the number of output neurons (output layer).
- The third step is to implement the fully connected sigmoidal artificial network that substitutes the Q-matrix, it will be called the Q-net.
- The next step is to implement the architecture's graphical model to visualize the agent's behavior during the deep neural network training process.
- The last step is to use the deep network training during the exploiter stage to analyze the agent's behavior, its moves, its convergence to a stable and efficient solution and its capacity to generate look ahead moves and tested in a real robot already built.

4.1.1 Description of the Problem

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to big advances in machine vision [17]. These methods use a variety of neural network models, including convolutional, perceptron, multilayer, and recurrent neural networks. These methods already have different applications to a wide range of problems, such as robotics, where control policies for robots can now be learned directly from real-world camera inputs [30]. This is the kind of problem treated in this work, where a neural agent has to learn to play high-level tic-tac-toe by looking at a real-world board through a webcam. The first problem is to reduce the dimension of the real-world board image that the webcam perceives and converts it to a low-dimensional, compressed form of neural information. The second problem is creating a reinforcement learning agent that uses this compressed information and a fully connected network to learn an optimal policy to win the game. In order to be optimal, this policy must be able to recognize the implicit sequential logic of the game and use this information in its decision-making structure, right from the beginning of the process, i.e., from the first moves or aperture up to winning game situations. This is capacity to link present states with future states is the main objective of reinforcement learning. Finally the capacities of the agent will be tested in a real robot used in a previous works [31].

4.1.2 Analysis of the Problem

To solve the first problem and reduce the dimension of the image of the real tic-tac-toe board, a software developed with OpenCV, that has been developed and used in previous computer vision works, was studied an adapted to this thesis. To solve the second problem and implement the neural agent, several software modules and functions have been developed, tested an debugged using the C++ language. They constitute the main contribution of this thesis and are explained in detail in section 4.1.4.

4.1.3 Implementation

This phase consists of planning and executing the coding of the proposed model. We decided to use the C++ programming language because it is a high-level language used for writing applications when performance and proper use of resources are essential and used in resource-intensive applications, AI in games, and robot locomotion. Also extensive libraries in neural networks and agents has been developed through the years by the adviser of this thesis and his students. The compiler used for the proposed model is Borland C ++ 5.5, which is a C and C++ IDE (integrated development environment), including our neural libraries. In addition OpenCV libraries, previously used and tested in real-time computer vision projects in Yachay, where used for training purposes. Below are the modules and functions utilized by the neural agent. Some of these modules where developed from scratch for the purposes of this thesis. Other where adapted or refined from existing libraries.

- ***CheckGameWin.h***
  Module that checks if there is a winner, O or X.
  - ***check_game_winner*** is the function implemented in this module where it checks if O or X has won vertically, horizontally or diagonally.
- ***NeuralLibmmt.h***
  Module that contains the structure of the network such as its hidden layer and output layer, its hyperparameters and the calculation functions of the neural network.
  - ***random_weight*** is the function that generates random weights.
  - ***initialize_weights*** is the function that initializes the weights of the hidden layer and the output layer.
  - ***correct_weights*** is the function that calculates the error for the hidden layer and the output layer. In addition, it corrects the weights for the hidden layer and the output layer.
  - ***sigmoide*** is the activation function used for network training.
  - ***calculate_hidden_layer*** is the function to calculate the values of the hidden layer.
  - ***calculate_output_layer*** is the function to calculate the values of the output layer.

- ***backpropagation*** is the function that contains the function *calculate_hidden_layer* and *calculate_output_layer*.
- ***inject_noise_weights*** is the function that injects noise into the weights of the hidden layer and the output layer.

- ***OLearns.h***

  Module developed from scratch containing the following functions:

  - ***look_winning_neuron*** is the function that indicates the best move when the agent has to play.
  - ***X_fill_square*** is the function that performs the moves of random player X.
  - ***O_plays*** is the function that makes the moves of agent O.
  - ***human_plays*** is the function that allows to play a person against the agent with a defined layout of keys.
  - ***feed_forward*** is the function that combines backpropagation and correct the weights.

- ***PlotNetItem.h***

  Module that graphs the parameters of the network

  - ***plot_inputs*** is the function that plots the neurons of the input layer.
  - ***plot_hidden_outputs*** is the function that plots the neurons of the hidden layer.
  - ***plot_outputs*** is the function that plots the neurons of the output layer.
  - ***plot_targets*** is the function that plots the targets, that is, the output neuron that shows the best move.
  - ***plot_hidden_weights*** is the function that plots the hidden weights.
  - ***plot_board*** is the function that plots the board.
  - ***plot_guide_board*** is the function that shows the keys with which a human can play against the agent.
  - ***plot_board_map*** is the function that shows the number of squares on the board in this case from 0 to 8.

  - ***plot_game_graphics*** is the function that contains all the previously mentioned functions to observe the game environment and the agent.

- ***PrintConsole.h***

  Module that shows the value of the output and ordinate neurons.

  - ***print_ordered_neurons*** is the function that returns the neurons ordered from largest to smallest.

- ***LoadWeights.h***

  Module that saves and loads the weights of the neural network.

  - ***save_weights*** is the function that saves the network weights in a file with a .dat extension.
  - ***load_weights*** is the function that loads the weights of the network which have been generated by the function *save_weights* in a file with extension .dat.

- ***AgentPlayer.cpp***

  It is the main program that contains all the modules described above to be able to set up the environment and that the agent can play.

4.1.4 Testing

For the agent tests, it will first be trained with a random player, in this case X. After its training, the agent will play against the same random player and then with a human player, all this will be done to verify if it can generate residual knowledge of the future and thus have a look ahead capacity and do good moves right from the beginning of the game (aperture).

4.2 Model Proposal

In the present work, the architecture to develop the neural agent that we propose is based on a deep neural network that replaces the Q matrix that is usually used in reinforcement learning. This architecture is based on an approximation of the Bellman equation. This section explains in detail how the model we propose is established.

4.2.1 Environment and Deep Neural Network Architecture

The neural agent has to learn to play tic-tac-toe by looking at video frames of a real-world board with an image dimension of 720x480. As mentioned before the used software was mostly developed for previous papers an used OpenCV to obtain a 100x100 pixels version. For the agent to recognize this reduced tic-tac-toe board of 100x100, nine convolutional networks with identical structures are implemented, where each network analyzes one square of 30x30 pixels on the board. This 30x30 dimension is the square where the 'O', X' or empty will be located and is a portion of the total 100x100 board image. Thus each convolutional network processes the image of a board square through a stride of one, which is multiplied by four filters of dimension 3x3; horizontal, vertical, and two diagonals, one diagonal to the left and one diagonal to the right. Afterward, a feature map is generated for each filter which is represented by a 27x27 matrix. Then the sum of the four feature maps is performed, and we obtain the main feature map, which is processed with the max-pooling operation with a window of 3x3 and stride of 3 obtaining a 9x9 matrix. Finally, after obtaining all these feature maps, we can pass them as independent inputs to our fully connected network. This process describes how these convolutional networks help in pre-processing the video images and reduce their dimension to identify if each square has X, O or empty in terms of three output neurons, as represented in Figure 4.1. These three outputs across the nine board frames produce the 27 inputs that constitute the game state in sparse code. This code is delivered to a fully connected sigmoidal network called the Q-net that is trained to behave as a Q-matrix and provide an optimal policy to decide what move to make on the board.

4.2.2 Convolutional Neural Network Settings

The hyper parameters in each convolutional neural network (one for each board square) are tuned as follows:

- The input layer has 900 input neurons (image of 30X30)
- The filters have a dimension of 3x3
- The stride is set to 1

- The feature matrix has a dimension of 27x27
- The max pool window size is 3x3
- max pool matrix has a dimension of 9x9
- The max pool stride is set to 3
- For the training of the convolutional network, a sigmoidal neural network has been used with the following configuration
  - The input layer has 81 input neurons
  - The hidden layer has 18 hidden neurons
  - The output layer has 3 output neurons

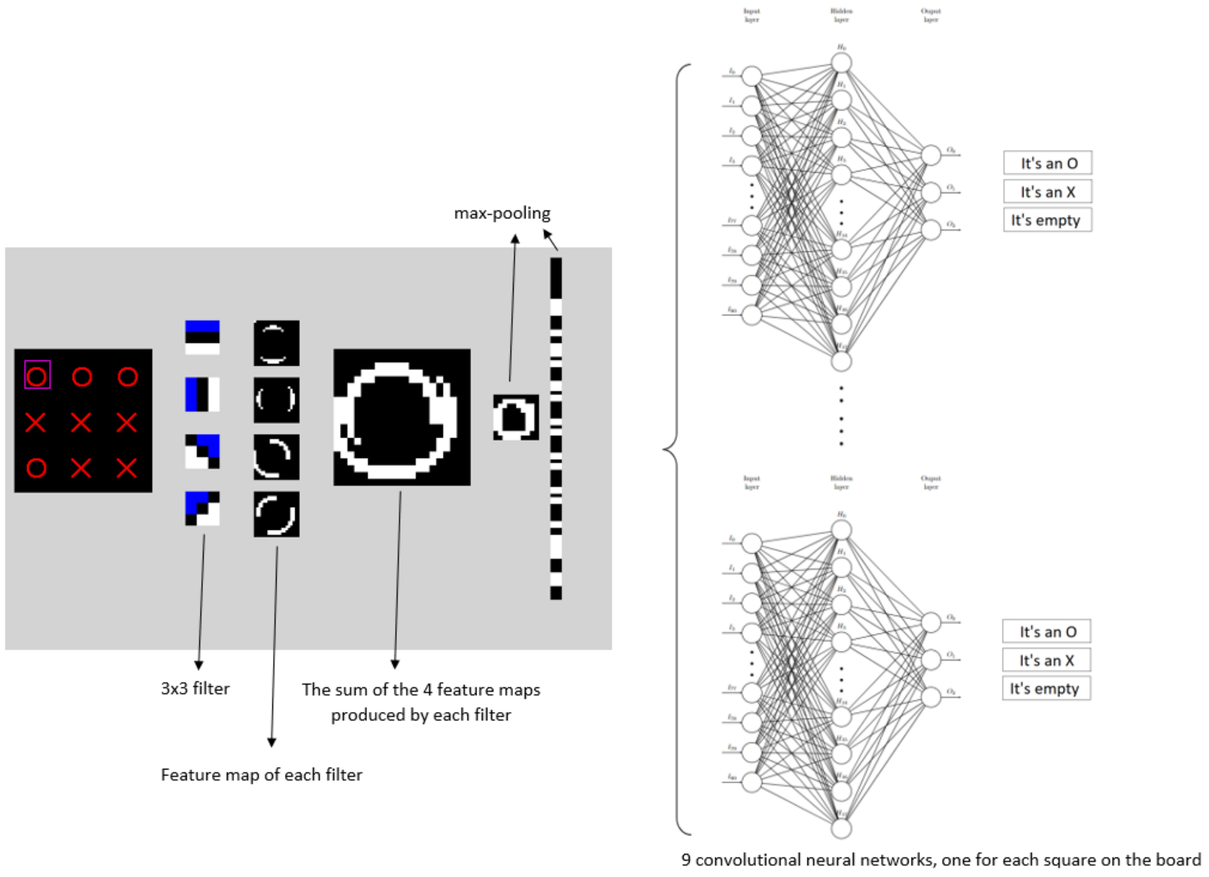

max-pooling
3x3 filter
The sum of the 4 feature maps produced by each filter
Feature map of each filter
It's an O
It's an X
It's empty
It's an O
It's an X
It's empty
9 convolutional neural networks, one for each square on the board

Figure 4.1: Convolutional Neural Network

4.2.3 Fully connected sigmoidal Q-Neural Network Settings

To establish the configuration of this neural network, several previous experiments were carried out where the hyperparameters, the hidden layers, number of hidden neurons as well as the activation function were manually varied. Initially, a configuration with a ReLU activation function with one and two hidden layers was tested. These experiments had no relevance in terms of results since ReLU network turns out to be quite unstable. Finally, the experiment with the best results was a sigmoidal network with a hidden layer which is explained in detail below.

- The input layer has 27 input neurons for sparse board state representation
- The hidden layer has 41 hidden neurons with gain set to 0.5
- The output layer has 9 output neurons with gain of 1.5
- Eta has 0.25; learning coefficient.

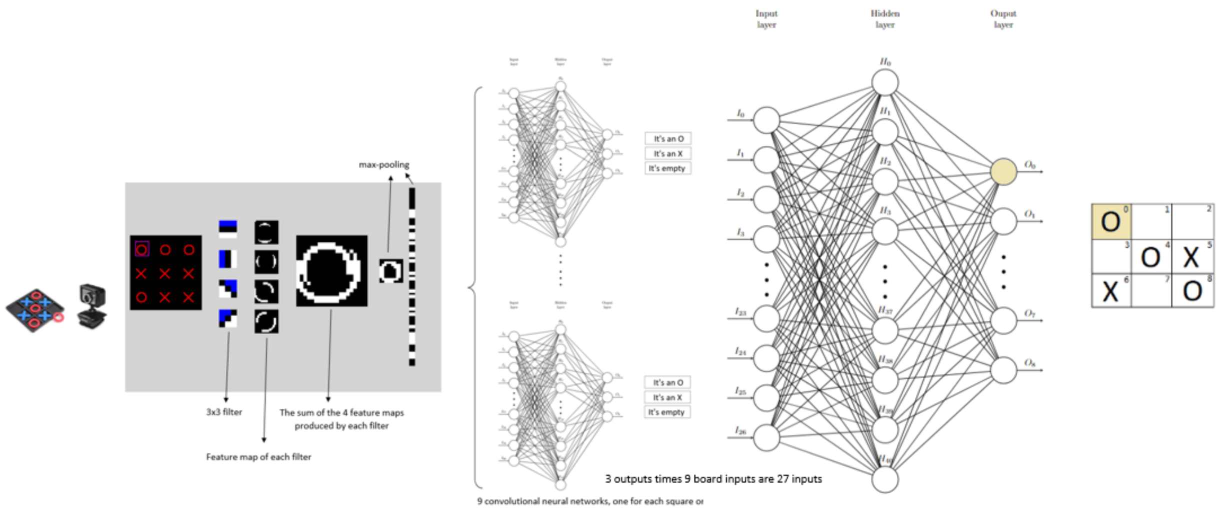


Figure 4.2: Neural agent deep network architecture

4.2.4 Bellman Equation Aproximation

Since we are doing a particular type of search for max, we are bringing back the max in a spatial neural ambient where the max is not a single real number but a neural vector. Therefore, the approximation of the Bellman equation that we are using is stated as follows:

-

$$Q(s,a) = r + \gamma \max_{a'} R\left(s', a'\right) \tag{4.1}$$

Notice that equation 4.1 is an approximation and does not search for MAX in the Q matrix but rather in the R matrix, becoming a fast solution since the neural agent does not search for a gradient connection with early plays or apertures in the memory Q. Instead, it creates a deep search for future events that lead to a maximal reward in the R-matrix. The expected result is that this search for the future produces a useful “residual future information” in the neural network that is substituting the Q-matrix.

4.2.5 Training and Testing

Below is the pseudo-code of the implemented code for training the neural agent

Algorithm 1 shows the pseudo-code of how a game between the agent and a random player has been implemented. This algorithm is used for the agent exploration phase, when agent must make its move, it first explores the future and then makes its move.

Algorithm 1: Agent O plays vs X random player

```
while non-trained do
    if player-selector then
        explore future;
        O plays;
    end
    if !player-selector then
        X plays;
    end
end
```

Algorithm 2 shows the pseudocode of how the exploration of the agent’s future has been implemented. As a first step all targets in the network are set to zero. The agent O explores all the squares, looks for the empty ones and places an O in it. Then it checks to see if with this placed O it wins. If this is the case, the target of that position is set equal to 1, equivalent to getting the maximum immediate reward. In addition, when performing the feed forward if a specific flag (b_flag) is active, one backpropagation cycle is performed. This loop is performed several times (about 2000 cycles) until the error in the network decreases below a chosen value and the net is considered to be trained

Algorithm 2: Agent O explores future

```
1  for 9 times do
2      set all targets to zero;
3      if board is empty then
4          put O on an empty space on the board;
5          check game winner;
6          if O wins then
7              delete added O from board;
8              target pointer points to the winning square;
9              set the target with the winning square position to 1;
10             fill inputs;
11             feed forward;
12             if exploring do backpropagation;
13         end
14     end
15 end
```

After the exploration phase, the exploitation phase is carried out where the previously trained agent plays against the same random player with whom it was trained and can also be faced with a human player.

Algorithm 3 shows the pseudocode When agent O plays by exploiting the knowledge obtained during the exploration phase. First O does the first move. Then, after X plays, the agent checks the Q-net output and search for the most excited neuron and takes this pointer as its decision. If the game is won the record of the games won by O is increased.

Algorithm 3: Agent O exploit knowledge

```
1  O plays random;
2  while game not ends do
3      X plays random;
4      feed forward the net;
5      look for most excited neuron;
6      used pointer to place O move;
7      if O wins then
8          record O wins;
9      end
10 end
```

4.2.6 Agent Graphical Interface

The figure 4.3 shows the graphical interface of the agent with all its parts.

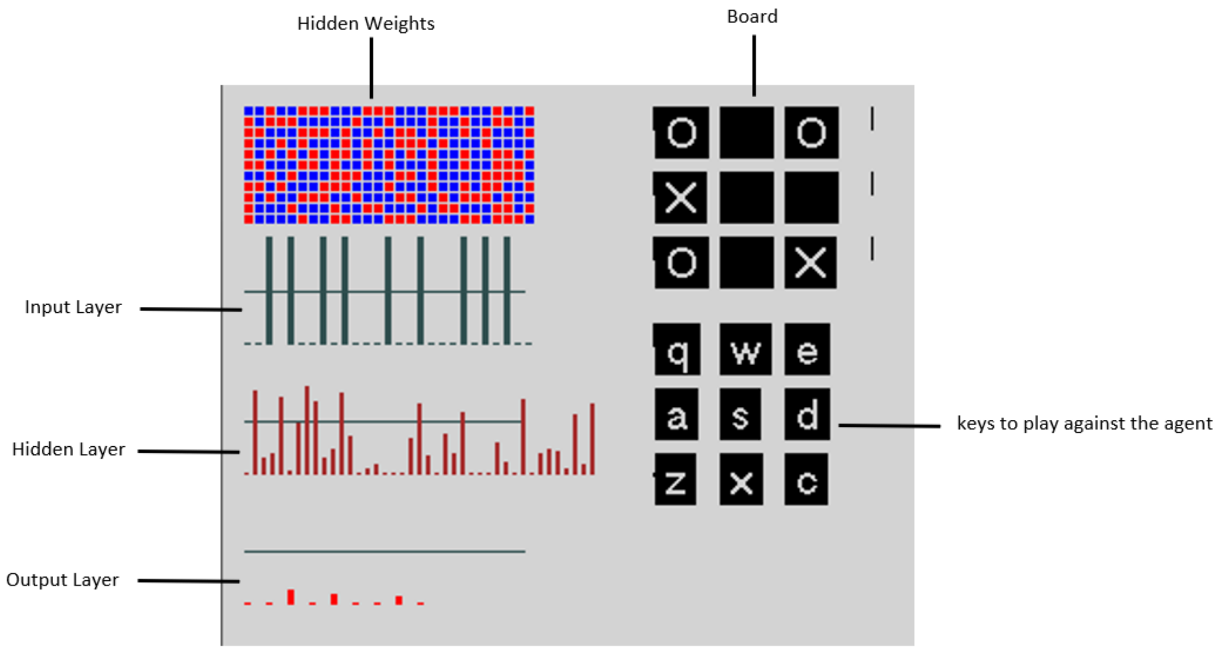


Figure 4.3: Agent Graphical Interface

Chapter 5

Results and Discussion

This chapter presents all the experimental and final results obtained by the neural agent we developed. The game process of the agent will be shown, from its first moves to the strategy it uses to win the game against random or human players.

5.1 Agent Moves

The neural agent learned to play the game of tic-tac-toe in a creative way and employs a good strategy from the first plays in order to win. In figure 5.1, when the agent starts playing, it makes its move in one of the corners, which gives excellent chances of winning if the opponent makes a wrong move.

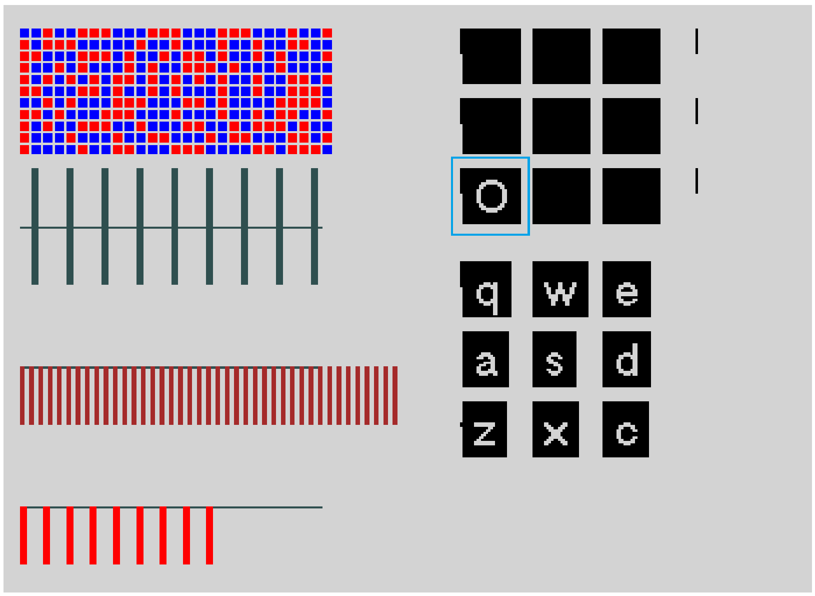
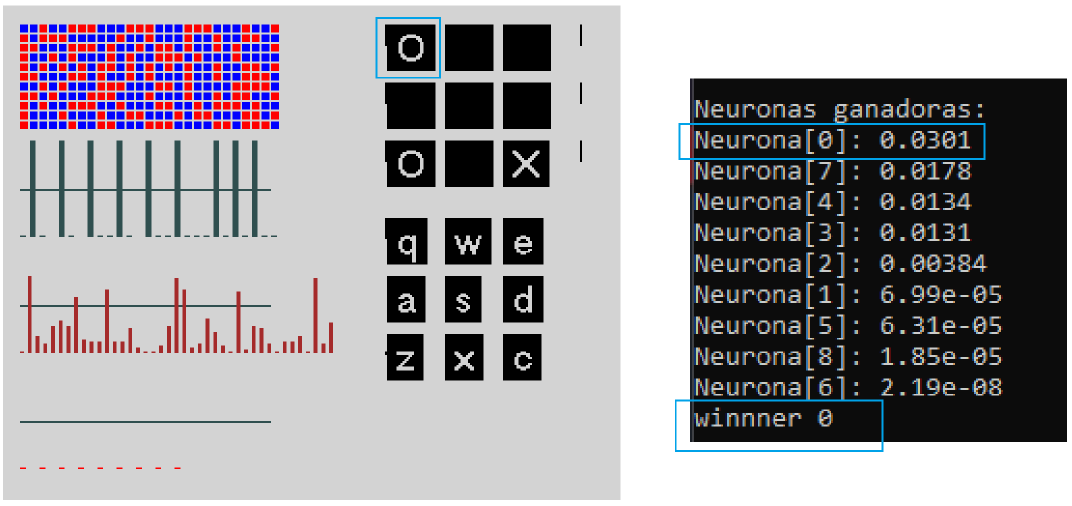


Figure 5.1: The first move made by the agent, which is in one of the corners of the board

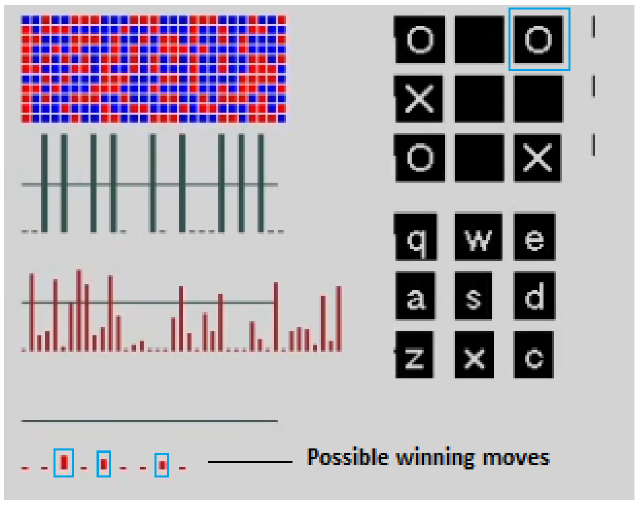
After the opponent's first move, in figure 5.2, the agent shows its second move to be made, the agent shows the best move as the "winning move," and the output neurons are sorted according to their value. This value is the probability of making the said winning move, and since each neuron is associated with a quadrant of the board, the move made by the agent is for the square of the first row and the first column. This move denotes a look ahead capacity and opens a "double rail" situation where the agent assures a future victory.

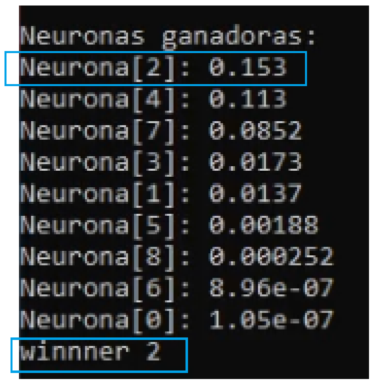


```
Neuronas ganadoras:
Neurona[0]: 0.0301
Neurona[7]: 0.0178
Neurona[4]: 0.0134
Neurona[3]: 0.0131
Neurona[2]: 0.00384
Neurona[1]: 6.99e-05
Neurona[5]: 6.31e-05
Neurona[8]: 1.85e-05
Neurona[6]: 2.19e-08
winnner 0
```

Figure 5.2: The second move made by the agent gives the possibility of achieving a double rail in the game

Likewise, after the opponent's second move, the agent chooses its third "winning move." However, the Q-network also shows other possible actions that it can take. Figure 5.3 shows the agent's behavior; in this case, given that the opponent has made a regular move, the agent seizes the opportunity and uses the best strategy, which is to open two possible wins patterns and ensure victory in the game.

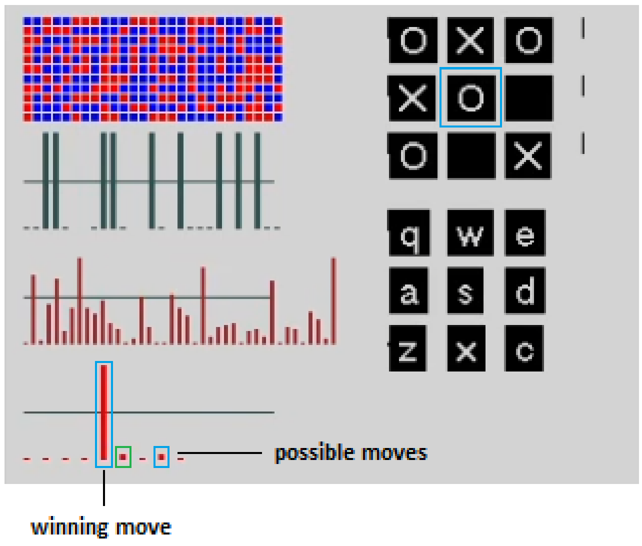


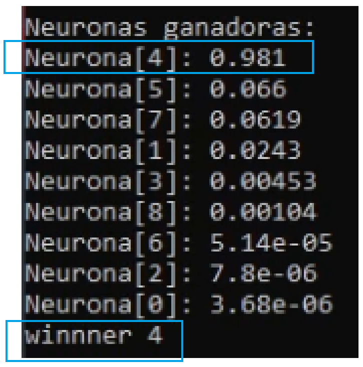


```
Neuronas ganadoras:
Neurona[2]: 0.153
Neurona[4]: 0.113
Neurona[7]: 0.0852
Neurona[3]: 0.0173
Neurona[1]: 0.0137
Neurona[5]: 0.00188
Neurona[8]: 0.000252
Neurona[6]: 8.96e-07
Neurona[0]: 1.05e-07
winnner 2
```

Figure 5.3: The third move made by the agent gives the security of achieving a double rail in the game and winning it.

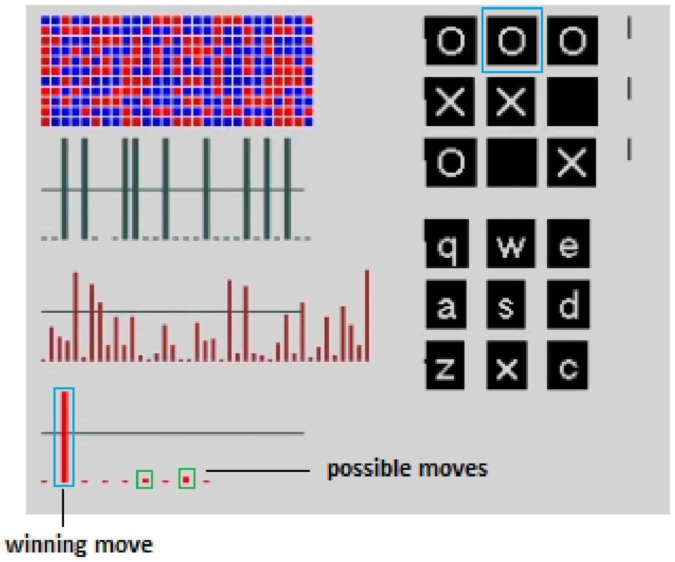
Finally, Figure 5.4 and Figure 5.5 shows the move made by the agent to win the game despite having other possible actions. The agent is very clear about the winning move it must take, even if the rival tries to avoid one of the two possible victories that it produced with the strategy previously used.





```
Neuronas ganadoras:
Neurona[4]: 0.981
Neurona[5]: 0.066
Neurona[7]: 0.0619
Neurona[1]: 0.0243
Neurona[3]: 0.00453
Neurona[8]: 0.00104
Neurona[6]: 5.14e-05
Neurona[2]: 7.8e-06
Neurona[0]: 3.68e-06
winnner 4
```

Figure 5.4: The fourth move made by the agent gives first victory in the game



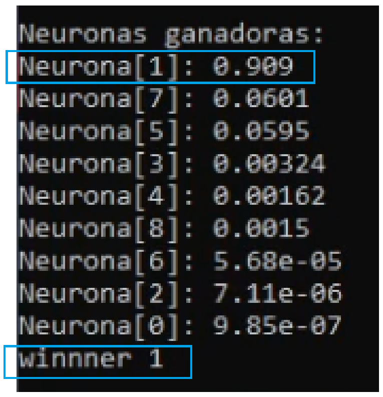
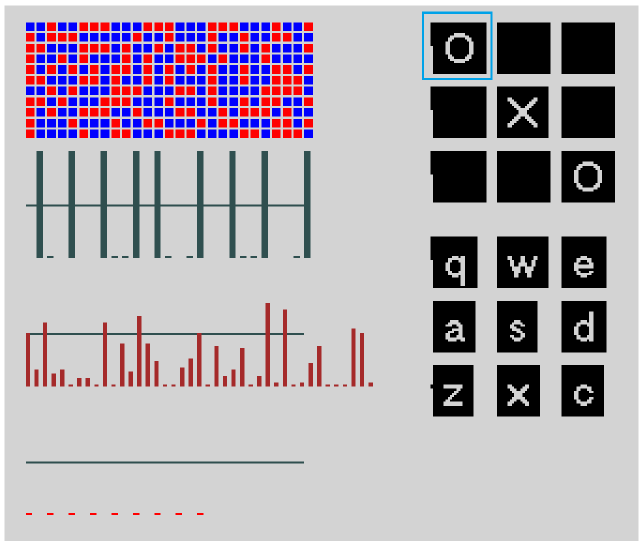

```
Neuronas ganadoras:
Neurona[1]: 0.909
Neurona[7]: 0.0601
Neurona[5]: 0.0595
Neurona[3]: 0.00324
Neurona[4]: 0.00162
Neurona[8]: 0.0015
Neurona[6]: 5.68e-05
Neurona[2]: 7.11e-06
Neurona[0]: 9.85e-07
winnner 1
```

Figure 5.5: The fourth move made by the agent gives second victory in the game

On the other hand, if the opponent is more experienced in the game (human player) and makes his move in the middle square of the second row, the agent tries to force the double rail by making its move in the top corner, in this example, the first box from the left. as shown in Figure 5.6.



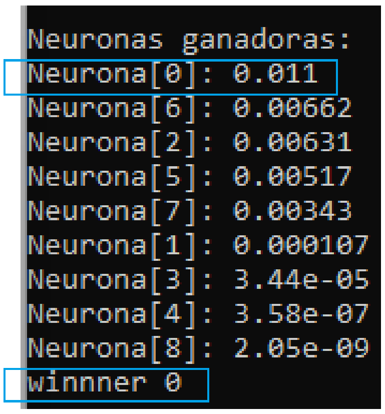
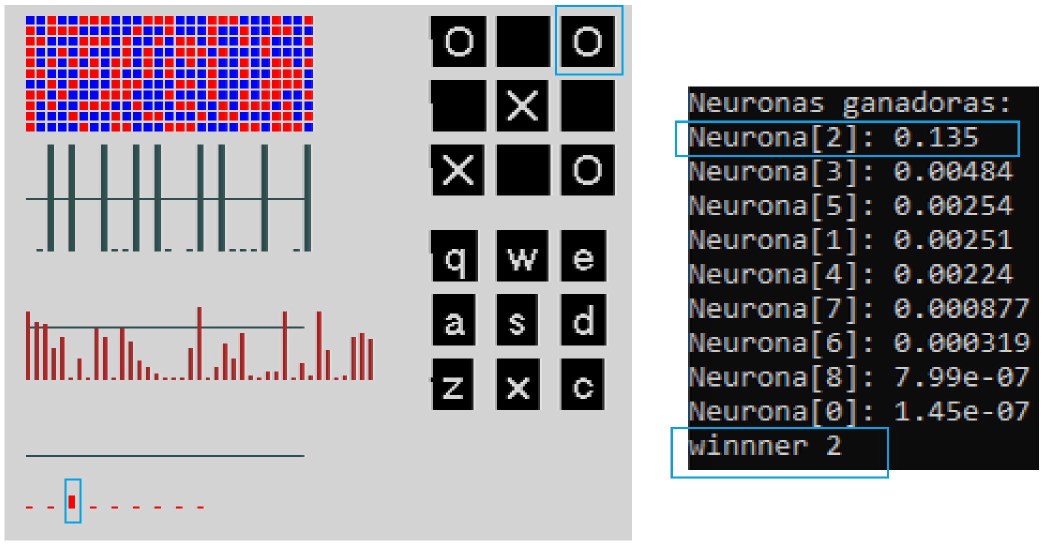

```
Neuronas ganadoras:
Neurona[0]: 0.011
Neurona[6]: 0.00662
Neurona[2]: 0.00631
Neurona[5]: 0.00517
Neurona[7]: 0.00343
Neurona[1]: 0.000107
Neurona[3]: 3.44e-05
Neurona[4]: 3.58e-07
Neurona[8]: 2.05e-09
winnner 0
```

Figure 5.6: The second move of the agent to force the double rail

If the opponent makes the mistake of making his move in one of the corners, the game ends with a double rail by the agent where it wins as shown in Figure 5.7.



```
Neuronas ganadoras:
Neurona[2]: 0.135
Neurona[3]: 0.00484
Neurona[5]: 0.00254
Neurona[1]: 0.00251
Neurona[4]: 0.00224
Neurona[7]: 0.000877
Neurona[6]: 0.000319
Neurona[8]: 7.99e-07
Neurona[0]: 1.45e-07
winnner 2
```

Figure 5.7: The agent's third move where it gets double rail

But in the event that the opponent makes its move on the edge of the second column, the game ends in a draw as shown in Figure 5.8.

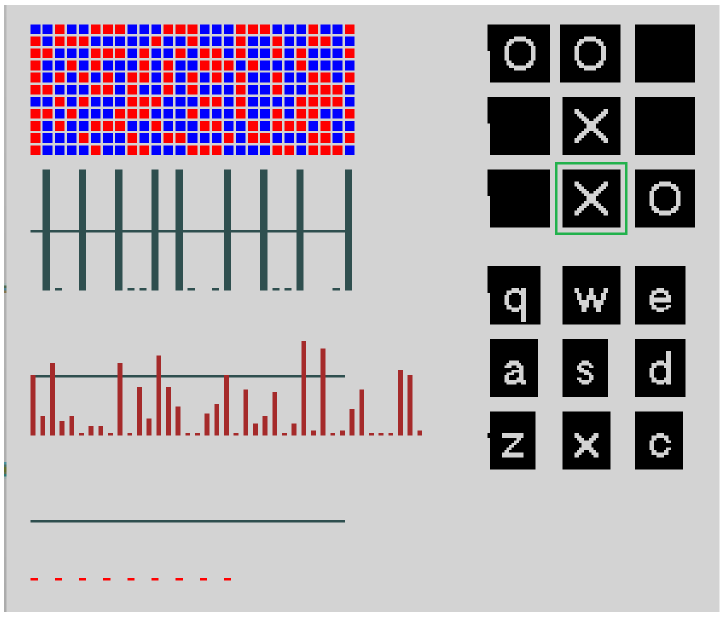


Figure 5.8: Position where the game ended in a draw

Although the agent explores the future with the Bellman equation approximation presented in Chapter 4, a gradient between early moves and final winning moves is indeed created. This gradient is not absolute and sometimes fails in pointing the optimal path during the first moves. However, despite this, we can say that it plays with a residual knowledge of the future that spontaneously appears during the Q-net training. Further mathematical analysis is required to explain this phenomena, which for the best of our knowledge, has not been previously described. Additionally, we connect the agent's output to the a previously constructed robot with Arduino interface to play on the actual board as shown in Figure 5.9. and again the agent shows its look ahead capacity.

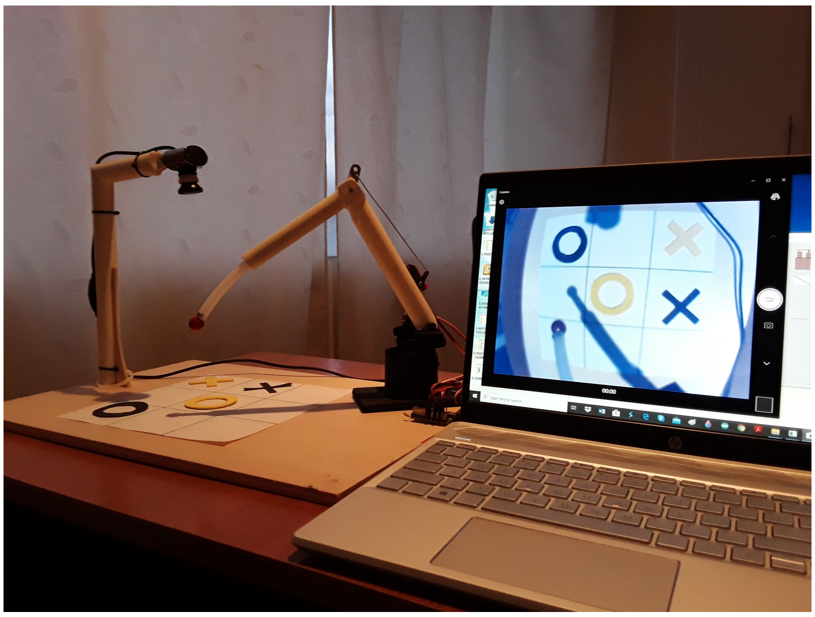


Figure 5.9: The found agent driving a robot playing tic-tac-toe

5.2 Agent Results

Figure 5.10 shows the agent's performance playing tic-tac-toe concerning the percentage of wins and losses against an opponent with random actions. As a result, it can be verified that our agent has been trained in approximately 22500 games, which shows a fairly fast training with the implementation in which it has been developed.

With respect to a human player, the game ends in a draw if the correct moves are made as shown in the position in Figure 5.8.

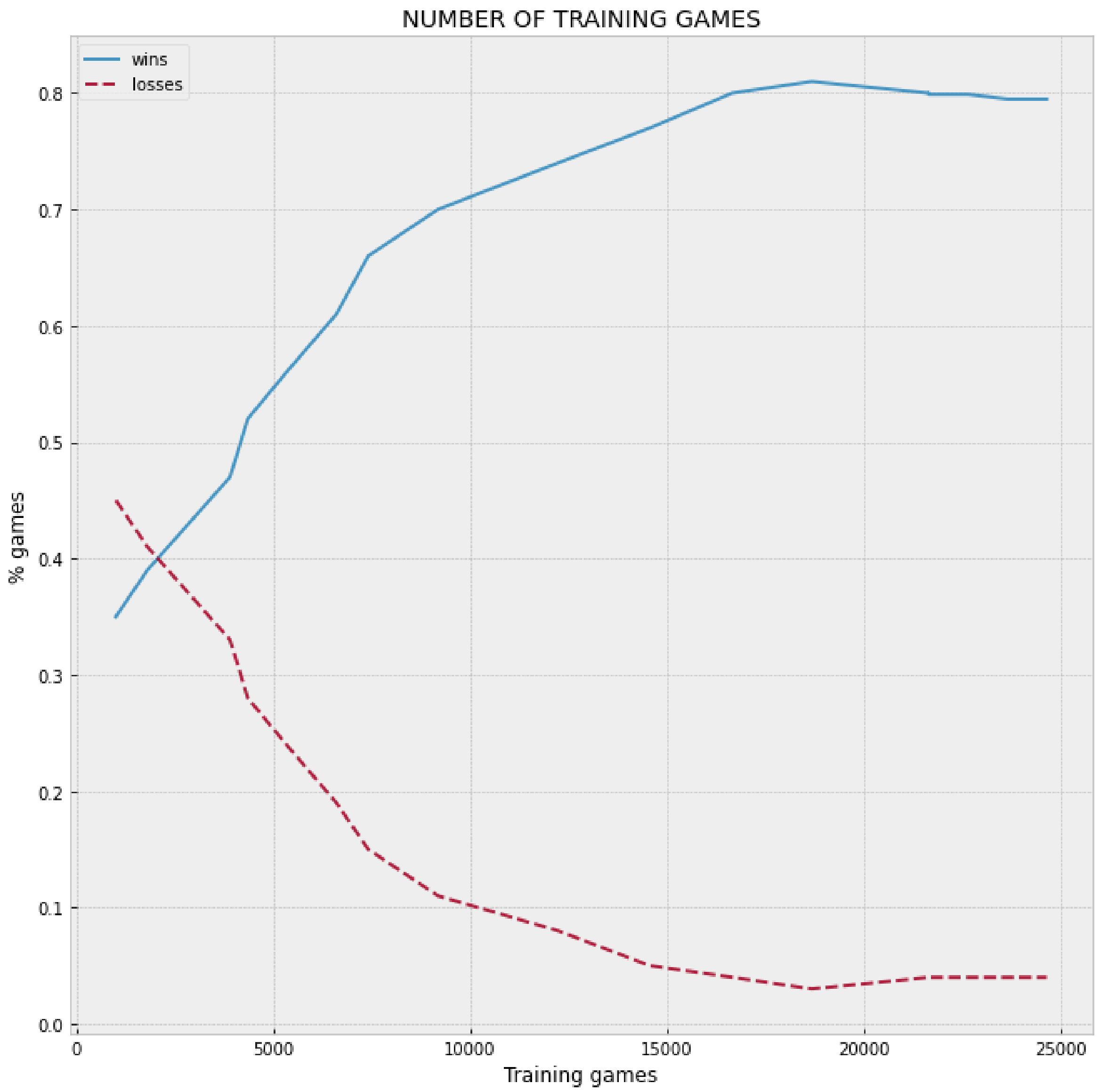


Figure 5.10: Performance of the agente playing Tic-Tac-Toe

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We can conclude that this thesis presents a deep reinforcement learning system where an agent learns to play high-level tic-tac-toe by looking at frames of video that are captured with a previously built webcam system that watches a physical board. The captured image is processed with OpenCV until a canny border image is obtained. This image is delivered to a convolutional neural network that produces a primary classification of images into O's, X's, and empty squares. Subsequently, this classified composed image is delivered to a fully connected network that is trained to behave as a Q-matrix and produce a policy that optimizes the control of the game execution.

The main contribution of this work is the development of computer software that makes possible to prove a valid, fast approximation of the Bellman equation, where the agent searches for reward in the R-matrix and not in the Q-matrix as it is normally done. The results of this work shows that the proposed method produces future residual information that spontaneously appears during the Q-net training and can be successfully used in the optimal decision taken of sequential process, represented by a tic-tac-toe game. In other words, the policy or strategy that the neural agent finds creates a gradient between the first moves (aperture) and future moves that lead to a maximal reward or game-winning. Further mathematical analysis is required to explain this not previously described phenomena, to the best of our knowledge.

The found solution makes it possible for an agent to rapidly learn in an unsupervised way to play a sequential game by watching video frames. In addition, its training is fast and does not consume many computational resources. However, at the end of the training stage, the degree of information that the agent has acquired is not absolute, and on some occasions, it fails in its decision taken. In principle this situation could be improve by incorporating further neural networks training techniques like dropout, etc. Due to its efficiency, the found fast solution opens the way to practical applications in real life, such as self-driven vehicles, robotics, military, surveillance, computer-aided medicine, and others.

6.2 Future Work

Since the work carried out presents a deep reinforcement learning system, as future work, the Bellman equation approximation can be implemented in other games and fields that need rapid decision making. The found programmed result shows that the proposed fast solution indeed generates look ahead capacities. This phenomenon requires to be mathematically proved.

Bibliography

[1] K. Suzuki, *Artificial Neural Networks - Methodological Advances and Biomedical Applications.* Rijeka: IntechOpen, Apr 2011. [Online]. Available: https://doi.org/10.5772/644

[2] J. P. Bigus, *Data Mining with Neural Networks: Solving Business Problems from Application Development to Decision Support.* McGraw-Hill Inc.,US, 1996, ch. Neural Network Models and Architechtures, pp. 63–64.

[3] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," 2018. [Online]. Available: http://arxiv.org/abs/1811.12560

[4] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, *A Guide to Convolutional NeuralNetworks for Computer Vision.* Morgan Claypool, 2018, ch. Convolutional Neural Network, pp. 43–45.

[5] Y. Fenjiro and H. Benbrahim, "Deep reinforcement learning overview of the state of the art," *Journal of Automation, Mobile Robotics and Intelligent Systems*, vol. 12, pp. 20–39, 12 2018.

[6] G. Lample and D. S. Chaplot, "Playing FPS games with deep reinforcement learning," *CoRR*, vol. abs/1609.05521, 2016. [Online]. Available: http://arxiv.org/abs/1609.05521

[7] O. Chang, L. Zhinin-Vera, and F. Quinga, "Self-taught Neural Agents in Clever Game Playing." Proceedings of the Future Technologies Conference, 2020.

[8] R. E. Uhrig, "Introduction to Artificial Neural Networks," pp. 33–37, Nov 1995.

[9] H. Kukreja, B. N, S. C. S, and K. S, "An Introduction to Artificial Neural Network," pp. 27–30, 2016.

[10] S. C. Nerella, "Loss Functions in Neural Networks," July 2021. [Online]. Available: https://becominghuman.ai/loss-functions-in-neural-networks-ec6482a15e97

[11] C. E. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning," 2018.

[12] M. Malik, "Basics of neural networks," Apr 2018. [Online]. Available: https://becominghuman.ai/basics-of-neural-network-bef2ba97d2cf

[13] S. Sharma and A. Athaiya, "Activation Functions in Neural Networks," *International Journal of Engineering Applied Sciences and Technology*, vol. 4, pp. 310–316, Apr 2020.

[14] K. Debes, A. Koenig, and H.-M. Gross, "Transfer functions in artificial neural networks - a simulation-based tutorial," *Brains Minds Media*, 2005.

[15] S. Bhardwaj, "Neural Networks and Activation Function," Apr 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/04/neural-networks-and-activation-function/

[16] E. N. Sanchez, J. D. Rios, A. Y. Alanis, N. Arana-Daniel, and C. Lopez-Franco, "Appendix a - artificial neural networks," in *Neural Networks Modeling and Control.* Academic Press, 2020, pp. 117–124. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128170786000167

[17] D. Wilson and T. Martinez, "The need for small learning rates on large problems," in *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*, vol. 1, July 2001, pp. 115–119 vol.1.

[18] J. G. Carney and P. Cunningham, "The epoch interpretation of learning," 1998.

[19] A. Krenker, J. Bester, and A. Kos, *Introduction to the Artificial Neural Networks*, Apr 2011. [Online]. Available: https://www.intechopen.com/chapters/14881

[20] I. C. Education, “What is supervised learning?” https://www.ibm.com/cloud/learn/supervised-learning, Agust 2020.

[21] K. L. Priddy and P. E. Keller, “Supervised training methods,” in *Artificial Neural Networks: An Introduction*, 2005, pp. 13–14.

[22] J. Li, J.-h. Cheng, J.-y. Shi, and F. Huang, “Brief introduction of back propagation (bp) neural network algorithm and its improvement,” in *Advances in Computer Science and Information Engineering*, D. Jin and S. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 553–558.

[23] C. C. Aggarwal, *Neural Networks and Deep Learning.* Cham: Springer, 2018.

[24] P. Winder, *Reinforcement Learning Industrial Applications of Intelligent Agents.* O’Reilly Media, Inc, November 2020.

[25] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, 2018.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001. [Online]. Available: http://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262032937%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0262032937

[27] C. J. Watkins and P. Dayan, “Q-Learning,” *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[28] M.-J. Li, Y.-J. H. An-Hong Li, and S.-I. Chu, “Implementation of deep reinforcement learning,” in *Proceedings of the 2019 2Nd International Conference on Information Science and Systems*, 2019, pp. 232–236. [Online]. Available: https://doi.org/10.1145/3322645.3322693

[29] Matlab, “CNN design and training with matlab.” [Online]. Available: https://es.mathworks.com/discovery/convolutional-neural-network-matlab.html

[30] K. Arulkumaran, M. Deisenroth, M. Brundage, and A. Bharath, "A brief survey of deep reinforcement learning," *IEEE Signal Processing Magazine*, vol. 34, 08 2017.

[31] O. Chang and L. Zhinin-Vera, "A wise up visual robot driven by a self-taught neural agent," 10 2020.

[32] C. J. Gatti and M. J. Embrechts, *Reinforcement Learning with Neural Networks: Tricks of the Trade.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 275–310. [Online]. Available: https://doi.org/10.1007/978-3-642-28696-4_11