



INSTITUTO POLITÉCNICO NACIONAL

---

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

LABORATORIO DE REDES Y CIENCIA DE DATOS

Understanding Google's AlphaGo

TESIS

QUE PARA OBTENER EL GRADO DE  
MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA

Ing. Mario Eduardo Aburto Gutiérrez

DIRECTORES DE TESIS:

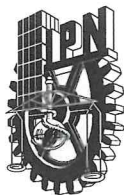
Dr. Ricardo Menchaca Méndez

Dr. Rolando Menchaca Méndez



CIUDAD DE MÉXICO

DICIEMBRE DE 2016



# INSTITUTO POLITÉCNICO NACIONAL

## SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

### ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 12:00 horas del día 12 del mes de diciembre de 2016 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

#### *Centro de Investigación en Computación*

para examinar la tesis titulada:

**"Understanding Google's AlphaGo"**

Presentada por el alumno:

**ABURTO**

Apellido paterno

**GUTIÉRREZ**

Apellido materno

**MARIO EDUARDO**

Nombre(s)

Con registro:

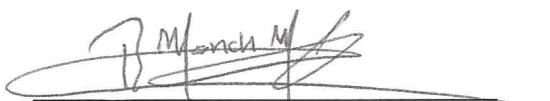
B	1	4	0	4	1	0
---	---	---	---	---	---	---

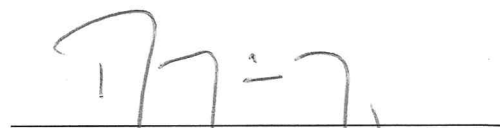
aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron **APROBAR LA TESIS**, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

#### LA COMISIÓN REVISORA

Directores de Tesis

  
Dr. Ricardo Menchaca Méndez

  
Dr. Rolando Menchaca Méndez

  
Dr. Grigori Sidorov

  
Dr. Marco Antonio Moreno Armendáriz

  
Dr. Francisco Hiram Calvo Castro

  
Dr. Mario Eduardo Rivero Ángeles

PRESIDENTE DEL COLEGIO DE PROFESORES

  
  
Dr. Marco Antonio Ramírez Salinas  
DIRECCION



**INSTITUTO POLITÉCNICO NACIONAL**  
**SECRETARÍA DE INVESTIGACIÓN Y POSGRADO**

*CARTA CESIÓN DE DERECHOS*

En la Ciudad de México el día 15 del mes de diciembre del año 2016, el que suscribe Mario Eduardo Aburto Gutiérrez alumno del Programa de Maestría en Ciencias de la Computación con número de registro B140410, adscrito al Centro de Investigación en Computación, manifiesta que es autor intelectual del presente trabajo de Tesis bajo la dirección de Dr. Ricardo Menchaca Méndez y Dr. Rolando Menchaca Méndez y cede los derechos del trabajo intitulado Understanding Google's AlphaGo, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección maburto00@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

---

Mario Eduardo Aburto Gutiérrez

## **Resumen**

Esta tesis describe los elementos que componen al programa AlphaGo, desarrollado por Google, y que hicieron que este programa tuviera un desempeño superior al humano en el juego de Go. Se presentan algunos detalles de los diferentes métodos usados en AlphaGo así como la implementación de algunos de ellos para ayudarnos a entenderlos mejor. Usando estas implementaciones se creó NdsGo, un programa que puede jugar Go en tableros de tamaño  $2 \times 2$ ,  $3 \times 3$ ,  $9 \times 9$  y  $19 \times 19$ . Para las primeras dos medidas de tablero, el programa logra jugar de manera perfecta, pero en  $9 \times 9$  y  $19 \times 19$  se necesita más trabajo para que este programa logre ser competitivo.

## **Abstract**

This thesis describes the elements used in Google's AlphaGo that made it achieve super-human performance in the game of Go. We present some theory on the different methods used in AlphaGo and implement some of them in order to understand them better. Using these implementations, we created NdsGo, a computer Go program able to play in  $2 \times 2$ ,  $3 \times 3$ ,  $9 \times 9$  and  $19 \times 19$  board sizes. For the first two sizes, the program plays perfect play, but on  $9 \times 9$  and  $19 \times 19$  more work needs to be done for this program to be competitive.

# Agradecimientos

Agradezco a mis asesores, Ricardo Menchaca Méndez y Rolando Menchaca Méndez, por su apoyo para la elaboración de esta tesis, así como por todo el apoyo recibido durante mis estudios de maestría.

También agradezco a los miembros de mi jurado y de mi comité tutorial por las valiosas observaciones que hicieron a este trabajo.

Quiero agradecer a mis amigos, que me acompañaron durante la maestría, pues sin ustedes este proceso no habría sido el mismo. Me llevo maravillosos recuerdos gracias a cada uno de ustedes.

Así mismo, agradezco a mis padres por el apoyo que me han brindado durante toda mi vida, pues a ellos les debo más que a nadie lo que tengo y lo que soy. Muchas gracias.

Por último, agradezco al Instituto Politécnico Nacional, al Centro de Investigación en Computación y al Consejo Nacional de Ciencia y Tecnología, por el apoyo para realizar mis estudios de maestría.

# Contents

<b>Resumen</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 The game of Go . . . . .	2
1.3 Objectives . . . . .	6
1.4 Limitations . . . . .	6
<b>2 Deep Learning</b>	<b>7</b>
2.1 Neural Networks, Backpropagation and Stochastic Gradient Descent . . . . .	7
2.2 Convolutional Neural Networks . . . . .	9
2.3 Regularization and Learning Rate Decay . . . . .	10
<b>3 Reinforcement Learning</b>	<b>12</b>
3.1 Markov Decision Processes and Bellman equations . . . . .	13
3.2 Monte Carlo prediction and control . . . . .	15
3.3 Value functions using function approximators . . . . .	16
3.4 Policy Gradient Reinforcement Learning . . . . .	17
3.5 Monte Carlo Tree Search . . . . .	18
<b>4 Related Work</b>	<b>20</b>
4.1 Introduction to Computer Go . . . . .	20
4.2 Traditional techniques in Computer Go . . . . .	22
4.3 Convolutional Neural Networks in Computer Go . . . . .	23
4.4 Monte Carlo Tree Search in Computer Go . . . . .	25
4.5 AlphaGo: Deep Learning and Reinforcement Learning in Computer Go . . . . .	26

<b>5</b>	<b>Proposed Solution</b>	<b>30</b>
5.1	Implementation of Board representation and the Go Text Protocol . . . . .	30
5.2	Implementation using Monte Carlo Self-Play for $2 \times 2$ and $3 \times 3$ boards . .	32
5.3	Datasets and data formatting . . . . .	33
5.4	Convolutional Neural Networks for move prediction using Supervised Learning . . . . .	35
5.5	What is left to reach AlphaGo performance? . . . . .	37
<b>6</b>	<b>Experimental Results</b>	<b>38</b>
6.1	Results for the Monte Carlo Self Play Implementation on $2 \times 2$ and $3 \times 3$ boards . . . . .	38
6.2	Results for the Convolutional Neural Network implementation on the $9 \times 9$ board . . . . .	41
6.3	Results for the Convolutional Neural Network implementation on the $19 \times 19$ board . . . . .	42
<b>7</b>	<b>Conclusions and Future Work</b>	<b>44</b>
	<b>References</b>	<b>46</b>
<b>A</b>	<b>Source Code</b>	<b>49</b>



# List of Figures

1.1	Stones in atari (a) and captured stones (b)	3
1.2	A suicide move is illegal	3
1.3	The ko rule. White has to play somewhere else before playing in the ko position.	4
1.4	Two eyes. Stones are said to be alive.	4
1.5	False eyes. Stones with only one (real) eye are said to be dead	5
1.6	Go rank system	6
3.1	The Reinforcement Learning problem. Image taken from [24]	12
4.1	AlphaGo architecture. Taken from [21].	27
4.2	AlphaGo MCTS. Taken from [21].	29
5.1	CNN architecture with 2 convolutional layers	35
5.2	CNN architecture with 3 convolutional layers	36
5.3	CNN architecture with 13 convolutional layers	36
6.1	$Q(s,a)$ values for $2 \times 2$ board learned after 100, 10,000 and 1,000,000 games of self-play.	38
6.2	$Q(s,a)$ values for $3 \times 3$ board learned after 100, 10,000 and 1,000,000 games of self-play.	39
6.3	GnuGo (B) vs NdsGo (W) on the $2 \times 2$ and $3 \times 3$ boards.	39
6.4	Precision on the test set for the GoGoD $9 \times 9$ dataset.	41
6.5	NdsGo (B) vs GnuGo (W) on $9 \times 9$ without komi.	42
6.6	Precision on the test set for the KGS 2016 $19 \times 19$ dataset.	43
6.7	NdsGo (B) vs GnuGo (W) on $19 \times 19$ without komi.	43

# List of Tables

4.1	Prizes offered by the Ing Foundation. Taken from [8]. US\$1 was equal to NT\$25 at the time of publication (2013). . . . .	20
4.2	Prizes for beating a top professional Go player. Taken from [8]. . . . .	21
5.1	Engine commands for the GTP protocol. . . . .	31
5.2	KGS dataset as of October 2016 . . . . .	33
5.3	Datasets used . . . . .	34
6.1	Memory to store the $Q(s,a)$ for the Monte Carlo Self Play implementation .	40
A.1	Description of the most important source code files . . . . .	49

# Chapter 1

## Introduction

Early this year, one of Artificial Intelligence grand challenges was solved: a computer program beat a professional player in the game of Go. The program, called AlphaGo, was developed by DeepMind, a company whose goal is “to solve intelligence”. This was a huge achievement, since many researchers on Computer Go thought there was still a long time for this to happen. However, using some of the most recent techniques of Reinforcement Learning and Deep Learning, AlphaGo did what was thought as impossible.

In this work, we present the techniques used by AlphaGo, and also, the implementation of some of them in our computer Go program NdsGo. The purpose of this is to understand the elements that compose AlphaGo, and how the combination of all of them permitted super human performance.

### 1.1 Motivation

Board games have interested researchers since long ago. There has been a lot of work to create computer programs that are able to play well on different games such as Othello, Checkers and Chess. Many of the techniques developed to play well on these games have been translated to other problems. This is possible because games resemble the real world, but in a more simple and controlled environment.

During a long time, chess was the most challenging game in artificial intelligence. After the world champion was beaten by DeepBlue in 1997, many researchers looked for another challenging problem. This problem was the game of Go. Nevertheless, Computer Go research has been around for a long time, having the first work in 1970 [29]. But it was not until the 1990s that increased interest to this game began.

During a long time, Go was thought to be too difficult to be solved. For example,

Muller [17] predicted in 2002 that at least 50 years would pass before a computer reached the level of a professional Go player. In 2015, the computer program AlphaGo became the first program to beat a professional player in an even game. This game was against the European Go Champion. In 2016, Google arranged a match against Lee Sedol, one of the best current players, which has more titles than any other players in the world.

Although AlphaGo beat Lee Sedol and now has super-human performance, the problem of Go is not solved yet and it is still useful to try new techniques. Since the victory achieved by AlphaGo, many other academic and commercial programs have implemented some ideas from AlphaGo. Also, interest in the game of Go and research in computer Go has increased as a result of AlphaGo's winning. In the years to come, we may see those same ideas applied to other areas as well. These are some of the reasons of why it is important to understand these concepts very well and to have a framework to work with them.

## 1.2 The game of Go

The game of Go is a game that has been played for at least 2000 years. It is thought that it was created around 2500-4000 years ago. This game is a game of strategy that originated in China, and was played mainly by nobles. A legend says that a Emperor invented the game to teach his son about discipline [20]. The game became very popular in Japan after its introduction, where it developed a lot of strategy and became really competitive.

Despite its great complexity, the game is actually very easy to learn. The game consists of two players who take turns to put stones on the intersections formed by the lines of the board. The player with the black stones plays first. The objective of the game is to surround intersections; these surrounded intersections are called territory. Each surrounded intersection counts as one point. Also, the players can capture opponent stones which are called prisoners; each prisoner counts as one point. To finish the game, both players must pass their turns. The player with more points at the end wins the game.

Now we will explain how to capture stones. We say that two stones are connected, if they are horizontally or vertically adjacent to each other. Many connected stones form a group of stones. Liberties are the intersections adjacent to the stone or group of stones that are empty. We capture a stone by reducing the number of liberties of that stone to zero; the same applies to groups of stones. In other words, if we surround a group of enemy stones, we capture them, so they are removed from the board (and stored as prisoners for later counting). We say that a group of stones is in atari if it has only one liberty left; that is, in the next move, we can capture that stone. In Figure 1.1 we can see some examples of

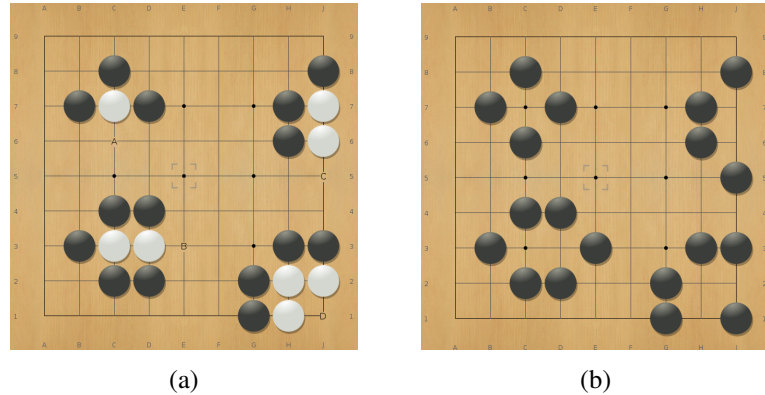


Figure 1.1: Stones in atari (a) and captured stones (b)

groups that are in atari, and how the board will be after capturing them.

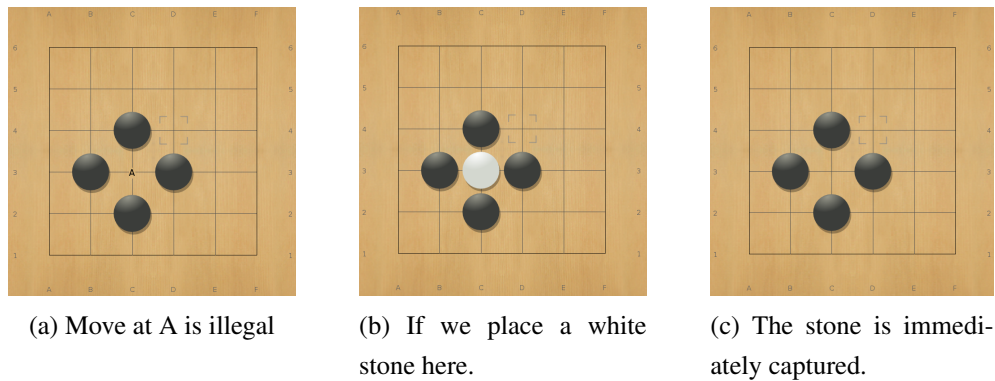


Figure 1.2: A suicide move is illegal

There are some moves that are illegal. The first one is trivial: we cannot place a stone in an occupied position. The second one is: suicide moves are forbidden. Suicide moves are moves that by playing them makes one of our groups to be completely surrounded, so the stones are removed. This rule prevents the player from giving free points to the other player.

The rule of ko is the last one. This rule says the following: in our turn, it is forbidden to make a move that makes the entire board configuration equal to the last board configuration. This rule is designed to prevent infinite loops (See Figure 1.3). So in order to place a stone at the forbidden position, we must play in another place first, and then if we want, we could place a stone there.

That are all the rules, now we will describe some of the implications that come after beginning to play the game. We will define what we mean when we say a group is "dead"

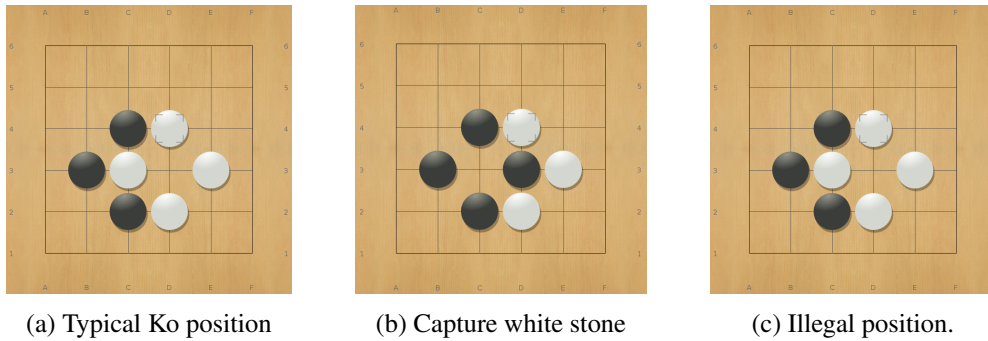


Figure 1.3: The ko rule. White has to play somewhere else before playing in the ko position.

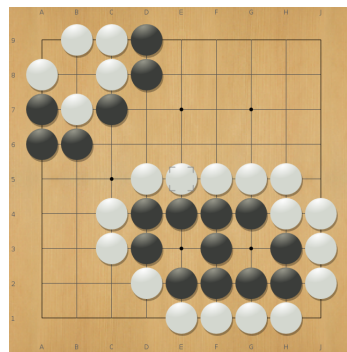


Figure 1.4: Two eyes. Stones are said to be alive.

and when we say a group is "alive". When there is no way for a group to survive, we say that the group is dead. However, there is a way to survive even if the group is surrounded; we have to make two eyes. An eye is an empty intersection that is surrounded by our stones; the opposing player can play here only if by playing he captures our stones, because if he cannot capture then it is an illegal position. Then, for a surrounded group to survive, the group must have two eyes, that is, two suicide positions (See the examples in Figure 1.4. The opposing player can never capture the group, because he cannot place two stones in one turn, so as to reduce the number of liberties of the group to zero. At the end of the game, all dead stones are removed from the board and counted as prisoners.

There is a concept called false eyes. We have a false eye when the intersection is not really a suicide move, since the player can capture some stones. The problem here is that we have two groups instead of only one group of stones, so that is why we have two eyes. We can see examples of false eyes in Figure 1.5.

There are many rulesets to play the game of Go, and they vary from country to country, or from club to club. The most recognized scoring rules are the Japanese rules and the Chinese rules. Although difference in score is minimal, the scoring rules are always specified

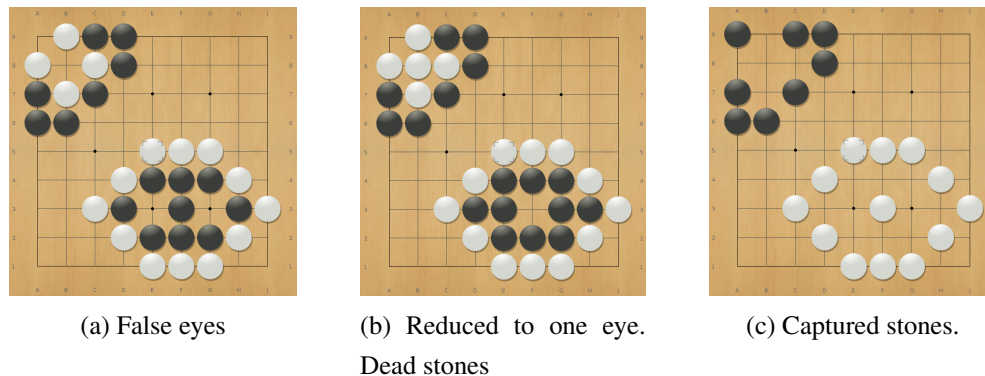


Figure 1.5: False eyes. Stones with only one (real) eye are said to be dead

in tournaments to avoid problems. In computer Go there is a popular scoring ruleset called Tromp-Taylor scoring.

The Chinese scoring is very simple. In these rules, points come from territory and number of stones in the board; captures are not counted. So, once both players pass, the procedure to count is to fill white's territory with white stones, and black's territory with black stones.

The last counting rule is Tromp-Taylor scoring. This scoring was developed by Tromp and Taylor in an attempt to formalize the rules of Go. They described the rules of Go in a formal way, and that included scoring. This scoring assumes that all the stones in the board are alive, so we only have to count white territory plus white stones in the board and black territory plus black stones in the board. There can be neutral points, that are not counted.

The difference in score is minimal, although for computers it is easier to count Chinese scoring.

Additionally to the scoring used, most of the time there is an initial quantity of points given to the white player to compensate of him playing second. This quantity is called komi and is generally 4.5, 5.5, 6.5 or 7.5; the quantity varies from tournament to tournament and from club to club.

In the game of Go, the level of players is defined as in Figure 1.6. The *kyu* levels are for beginners, and the *dan* levels are for the masters. However, we have two categories for the *dan* levels, those for amateur players and those for professional players. As we can see in the figure, the maximum level for amateur players is 7 *dan*. Nevertheless, this rule varies among go clubs and associations, with some of them having also 8 and 9 *dan* for amateur players. This is the case in the Kiseido Go Server (KGS), where the amateur *dan* levels go from 1 to 9.

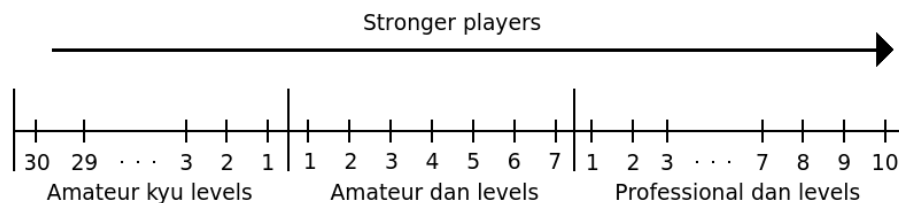


Figure 1.6: Go rank system

## 1.3 Objectives

The main goal of this thesis is:

- To understand the elements that made possible for AlphaGo to beat one of the best professional Go players.

Additionally, we have the following objectives:

- To implement a framework to experiment with computer Go.
- To implement a computer Go player for  $2 \times 2$  and  $3 \times 3$  boards using the most basic techniques of Reinforcement Learning to give intuition about how this works.
- To implement a computer Go player for  $9 \times 9$  and  $19 \times 19$  by creating a move predictor using Supervised Learning and Convolutional Neural Networks.

## 1.4 Limitations

This section is especially important since the work presented here is very limited compared with the work done by the team that developed AlphaGo. Some of the limitations are:

- Hardware: We use a regular PC with a single GPU (GTX 960) for the CNN. Google used 50 GPUs in the training.
- Time: The time for training the Convolutional Neural Networks reported in the AlphaGo paper is in the order of weeks. We trained at most for some hours.

That said, this work is intended to provide the reader with an explanation of the main techniques used, and present some experiments using those techniques, but it is not our intention to replicate exactly what AlphaGo did.



# Chapter 2

## Deep Learning

Deep Learning is a topic that has gained a lot of popularity amongst computer scientists since the last decade. Deep learning is a set of tools and techniques that make learning faster for deep neural networks. This boost in speed has let the computers reach new levels of performance on many difficult areas like Computer Vision, Speech Recognition and, recently, in Computer Go. This of course are just some examples, since every day a new application is found.

In this chapter we will give a brief introduction to neural networks and backpropagation; then, how to extend this knowledge to convolutional neural networks, which are one kind of Deep Neural Networks; and finally, we will introduce some techniques to improve convolutional neural networks that were used in this work. The formulas and the notation used in this chapter come mainly from [18]

### 2.1 Neural Networks, Backpropagation and Stochastic Gradient Descent

Neural Networks have had many ups and downs along their history. The beginning of this field was in 1943 when the first model of an artificial neural network was developed by McCulloch and Pitts. Many years later, in 1959, Widrow and Hoff developed ADALINE, the first artificial neural network that was applied to a real world problem: predicting the next bit in a streaming of bits from a phone line.

Artificial Neural Networks are used in many tasks; here, we will describe its use in the classification task. In this task, we want the computer to learn to classify objects in classes. For this, we need a dataset with examples, with which our network will learn to correctly classify the different objects. This is a very abstract definition, so the applications are very

diverse. For example, we can apply neural networks to classify images. In this thesis, we will use "classification" to predict the next move for our computer Go program. The move can be seen as a class.

The most basic unit of neural networks are the neurons themselves. These neurons are mathematical models of a neuron. The neuron has some number of parameters, the weights and the bias. Each input to the neuron has an associated weight. The network sums all the weighted inputs and the bias, and then send it to an activation function. The most basic activation function is the step function. However, this function has the disadvantage of not being differentiable at zero. For that reason, many times, the sigmoid function is used instead. However, we can use other activation functions; some examples are the tanh function, the sigmoid function and the rectifier function.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Gradient Descent is a common optimization method that can be used to learn parameters, but it depends on having an efficient way to calculate the gradient. The backpropagation algorithm is simply a way to compute the gradient with respect to the parameters of the network. The way to do this is first to define the cost that we will minimize. The key idea for this algorithm is the chain rule from calculus. The idea is to get the partial derivatives of parameters from earlier layers as a function of the partial derivatives of the weights in later layers. The following equations define the backpropagation algorithm.

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b_j^l} = d_j^l$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} d_j^l$$

Gradient Descent is an optimization method; that is, if we want to maximize or minimize a function, we can use gradient descent to do it. Since the gradient of a function is a vector that tells us the direction of maximum growth, we can use this vector to move around the function surface and find a maximum or the minimum, using the negative of the gradient. This algorithm is guaranteed to find the global optimum (the values of the

parameters that optimize the function) as long as the function is convex; so, if we do not have a convex function, gradient descent only guarantees local optimums.

$$w' = w - \frac{\eta}{N} \sum_j \frac{\partial C}{\partial w_k}$$

$$b' = b - \frac{\eta}{N} \sum_j \frac{\partial C}{\partial b_k}$$

Generally in deep learning we use a variation of Gradient Descent called Stochastic Gradient Descent. The change is that instead of calculating the gradient for all the dataset of size  $N$  and then applying the updates on the parameters, we calculate the gradient for a mini-batch of the data of size  $m$ , such that  $m < N$ . The gradient of the mini-batch is only an estimate of the true gradient, but in practice it is generally good enough, and it makes training faster. The size of the mini-batch becomes an hyperparameter of the network.

$$w' = w - \frac{\eta}{m} \sum_j \frac{\partial C}{\partial w_k}$$

$$b' = b - \frac{\eta}{m} \sum_j \frac{\partial C}{\partial b_k}$$

There are many ways in which we can select the cost (or objective) function. Some common choices are the least squares cost function and the cross-entropy cost function:

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)]$$

$$C = \frac{1}{2n} \sum_x \sum_j (y_j - a_j)^2$$

There are advantages and disadvantages to each one. For each application we have to analyse which one is better. Sometimes, this is done simply by testing different cost functions and seeing which one gives better results. This is a recurrent problem in Neural Networks and Deep Learning, that there is still not enough theory to understand them, so for now we have mainly a trial and error approach.

## 2.2 Convolutional Neural Networks

Deep Learning is a subfield of machine learning that focuses on computational models for hierarchical data. It can be seen as a set of techniques for learning in very deep neural networks [18]. One of these techniques that has had a lot of impact in many applications

is the use of convolutional neural networks. Convolutional Neural Networks (CNN) can be seen as Multi-layer neural networks with the following characteristics:

- Sparse connectivity. The neurons are only connected with a subset of the outputs from the previous layer.
- Shared weights. The weights from each layer are shared among the neurons, so each neuron in the layer has the same weights.
- Automatic feature detection. The convolutional layer permits the feature selection

Convolutional Neural Networks are commonly used for data that has two or three dimensions. A great example of this is the classification of images using CNNs. The idea is to take advantage of spatial data characteristics to reduce the number of parameters that we need to learn; for a single neuron, we will only input some neighbourhood of the data, not the entire image. The same can happen with later layers.

It is common to have a fully-connected layer at the end of the network. Some researchers interpret this as using the CNNs as automatic feature detectors. Then, using the output from the convolutional layers, we can then pass it to a fully connected layer. This is just an interpretation, because we do not know very much about neural networks in general; more theoretical work has to be done. However they have been successful in various domains.

The operation of convolution is defined by:

$$\sigma \left( b + \sum_l \sum_m w_{l,m} a_{j+l,k+m} \right)$$

Notice that this equation is different from traditional convolution in that here we invert the indices. We could use the original form, but this is easier to work with visually, since the learned filters are generally interpreted later and it is better to have them in this position.

## 2.3 Regularization and Learning Rate Decay

There is a big problem that often arises in neural networks: overfitting. Overfitting refers to a neural network that is too specialized in reproducing the training set, but does not generalize well. This is a problem because we do not want to learn to predict on the training set but on new information.

There are many techniques to address the problem of overfitting and they are called regularization techniques. One of the most common is the L2 regularization. The idea is to

try to get the weights of the network not too large, so we add the L2 squared norm of the vector of all the weights to the cost function.

$$C = C_0 + \frac{\lambda}{2d} \sum_w w^2$$

The parameter  $\lambda$  becomes a hyperparameter of the network which controls how much we penalize large weights; if  $\lambda$  is large, then the parameters will be small; but if  $\lambda$  is small, then the parameters can increase.

There are many other ways to solve this problem. A popular one is early stopping. To do early stopping we need to evaluate on a validation set. The error on the validation set must always decrease. When we see that the validation error increases, we can stop the training. There are many variations on this technique, but the idea is the same.

Another problem is the trade-off of speed versus precision that the learning rate has. If we select a learning rate that is too small, we will have better accuracy, but the training will take a lot of time. If on the other side we choose a large learning rate, we could have faster training, but we have the risk of making the model diverge, or simply not have enough precision.

To solve this problem we can use a learning rate schedule. This schedule is very dependant to the application and we need to experiment a lot to come with a good schedule. One way to do this is to decrease the learning rate a fixed percent every  $n$  epochs, or every  $n$  mini-batch steps.

# Chapter 3

## Reinforcement Learning

Reinforcement Learning is an area of Machine Learning. It was born from a combination of various disciplines in science; however, the main foundations come from research in Psychology, Dynamic Programming and Temporal Difference methods. There are many examples of successful applications using Reinforcement Learning. One of the earliest applications was the Tic-Tac-Toe player called MENACE [16], which learned how to play Tic-Tac-Toe just by playing against itself.

The area of reinforcement learning solves problems which generally have these characteristics:

- Closed-loop problems. The actions taken by the system affect future inputs.
- Not having direct instructions for actions. The actions taken should be learned from experience.
- Not knowing when the consequences of actions appear. This includes reward signals and next states seen.

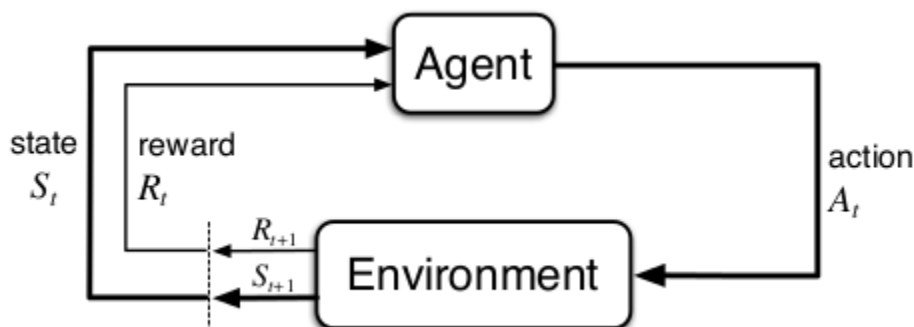


Figure 3.1: The Reinforcement Learning problem. Image taken from [24]

In Reinforcement Learning we have two objects that interact: the agent and the environment (See Figure 3.1). We can think of the agent as the program that will learn; the environment is everything outside the program. Given a state in which the agent is, the agent gives the environment an action and in return it gets a reward signal and the next state. The objective is to maximize the sum of the reward signals over the long run. In Reinforcement Learning we have four main elements that describe the problem:

- Policy  $\pi(a|s)$
- Reward signal  $r(s, a)$
- Value function  $v_\pi(s)$  or  $q_\pi(s, a)$
- Model of the environment  $p(s'|s, a)$

Many techniques use only some of them, but the policy and the reward signal are always necessary to be able to solve the problem.

Reinforcement Learning has had a lot of success in games. One of the first successful applications of RL was TD-gammon. This program achieved a master level of play, just a little below the top players. The program learned to play backgammon by many iterations of self-play. Another successful application of RL was made on the game of Go, where it was responsible for the increase in level of Computer Go programs from amateur *kyu* levels to amateur *dan* levels. In this chapter we will explain some techniques that are relevant to this thesis.

In this chapter we describe some of the basics of Reinforcement Learning and some of the techniques used in AlphaGo to achieve super human performance. This chapter is mainly based on the book Reinforcement Learning by Barto and Sutton [24]. For more detail the reader can see the book.

## 3.1 Markov Decision Processes and Bellman equations

The reinforcement learning problem can be defined as the way to maximize the expected value of the total rewards over episodes. The objective is to find  $G_t$ , called the return. The return for state  $s$  at time step  $t$  is the sum of all the rewards obtained after that time step.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$

Some times we want the return to be "discounted", that is, that future rewards count less if they are more distant in the future. This is especially important once we take the

expected value, because for us it can be more important to have immediate rewards than future rewards. So, to include the discount we have the equation:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

Once we have this, we have to make an assumption in order to make the Reinforcement Learning methods work: all the transitions from state to state must follow the Markov property:

$$Pr(S_{t+1} = s', R_{t+1} = r | S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t) = Pr(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

Basically, what this property says is that the next state is completely determined by the current state and action. So in this case, the previous history of states, actions and rewards are not needed to know what the next state will be (or the probability of having some next state). In practice, methods often work even in non-Markovian environments, but they can work not so well.

The reinforcement learning problem can be completely specified by a Markov Decision Process. The following equations define the Markov Decision Processes. It is important to note that some methods only need the reward in order to work. We say that methods that do not need these equations to work are model-free methods.

$$r(s, a) = E[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r | s, a)$$

$$p(s' | s, a) = Pr(S_{t+1} = s' | S_t = s, A_t = a) = \sum_{r \in R} p(s', r | s, a)$$

$$r(s, a, s') = E[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{r \in R} r p(s', r | s, a)}{p(s' | s, a)}$$

Now we define the concept of state value function and action value function. These functions give us the expected return given the current state, or the current state and action, for a given policy  $\pi$ . By learning these functions for each state, we can see the current rewards that we can obtain given the current policy. The idea of many methods is to use these values to improve the policy to get better returns.

$$v_{\pi}(s) = \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) \{r + \gamma v_{\pi}(s')\}$$



$$q_{\pi}(s, a) = r(s, a) + \gamma \sum_{s'} p(s'|s, a) \sum_{a'} \pi(a'|s') q(s', a')$$

The optimality bellman equations are the equations that define the expected return given the current state, or state and action, for the optimal policy  $\pi^*$ .

$$v_*(s) = \max_{a \in A(s)} \sum_{s', r} p(s', r|s, a) [r + \gamma v_*(s')]$$

$$q_*(s, a) = \sum_{s', r} p(s', r|s, a) [r + \gamma \max_{a'} q_*(s', a')]$$

These equations form a system of equations so, if we know the model of the environment we could solve the system using traditional linear algebra techniques. However, there is a more efficient solution called Dynamic Programming (DP). But even Dynamic programming fails for problems where we have a very large state or action spaces, or continuous state or action spaces. RL methods try to solve the Dynamic Programming problem for these cases, where DP is not feasible.

## 3.2 Monte Carlo prediction and control

The value function gives an idea of how good it is to be in a specific state. The state value function of state  $s$  is defined as the expected value of the return given that we are at state  $s$ . The action value function for state  $s$  and action  $a$  is defined as the expected return given that we are at state  $s$  and we take action  $a$ .

$$v_{\pi}(s) = E[G_t | S_t = s]$$

$$q_{\pi}(s, a) = E[G_t | S_t = s, A_t = a]$$

Next is the unifying formula for both episodic and continuous tasks for the return.

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

For this thesis, we will use approximate value functions, through the use of function approximation using neural networks. This is a topic that we will explore in the next section. Although these techniques are studied by supervised learning, we give here some of the theory concerning its use in Reinforcement Learning.

In order to improve a policy we will use a process called Generalized Policy Iteration (GPI). For this we have two steps: evaluation and improvement. An alternative is to use a method called value iteration. However, for the purposes of this thesis, GPI is enough. The problem with the described method is that we need to store the value function for every state, so if the state space is very large then this is not a feasible solution. That is why we use approximate value functions using artificial neural networks.

### 3.3 Value functions using function approximators

The method described before assumes we store the value for each state-action pair. There is a problem with that approach: if the set of states or the set of actions are too large we could not even store the value functions. But storage is not the only problem: even if the value function fits in memory, the learning will be very slow to converge since Monte Carlo convergence is based on visiting each state-action pair infinitely many times. Lastly, we want to generalize better, that is, if we have similar state-action pairs that are "close" to each other, we can think it is reasonable to have similar values to also learn from experience of only one of them. To address all of these problems we use function approximation for the value functions. This section is based on Chapter 9 from [24].

So, in order to overcome the problems mentioned before, we need to use a function approximator. The most common function approximator used in Reinforcement Learning is the linear function approximator.

$$v(s, \theta) = \theta^\top \phi(s)$$

where the function  $\phi(s)$  returns a vector of features for state  $s$  and the vector  $\theta$  is the vector of parameters that define of function approximator and that we will need to learn. The linear approximator can give good results, but depending on the application, we could use other approximators, like artificial neural networks.

To learn the parameters of the function approximator we need to define an error measure to see how much the output from the approximator and the value function we want to approximate differ. In this case we will choose the Mean Squared Value Error (MSVE):

$$MSVE(\theta) = \sum_{s \in S} d(s) [v_\pi(s) - \hat{v}]^2$$

where  $d(s)$  is a distribution that weights the importance of the error on each state. We call error to the squared difference between  $v(s, \theta)$  and  $v_\pi(s)$  (the value we are trying to

approximate). This "error" function is often called the objective function, because this is the function we will try to minimize.

The previous definition has two problems: we did not say how to choose  $d(s)$ , and we do not know the real value function  $v_\pi(s)$ . We will address the first problem first. One simple way to choose  $d(s)$  is by the number of times we visit each state during the simulations or episodes of experience. The second problem can be solved by using an estimator instead of the real value function. For example, for Monte Carlo simulations we can use the return  $G_t$  as an estimator.

Finally, to learn the parameters, we need to minimize the MSVE given before. To do this we can use any optimization method. One of the most common optimization methods is Stochastic Gradient Descent, which was explained before. It is important to note that in this case the training examples will be constructed from experience, so the input vector will be the state  $S_t$  and the output of the function will be  $v_\pi(S_t)$  (actually, an estimator of it). Then the update of the parameters will follow the formula:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{1}{2} \alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, \theta_t)]^2 \\ &= \theta_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, \theta_t)] \nabla \hat{v}(S_t, \theta_t)\end{aligned}$$

where  $\alpha$  is the learning rate. It is important to see that the gradient in the last equation is only over  $\hat{v}(S_t, \theta_t)$ . This is because the gradient is over the parameters  $\theta$  and they are only present in  $\hat{v}(S_t, \theta_t)$ . In general, for any function  $f(\theta)$  we have:

$$\nabla f(\theta) = \left( \frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, \dots, \frac{\partial f(\theta)}{\partial \theta_n} \right)^\top$$

The formulas presented before are very general, so from them we can define different algorithms. The most simple algorithm is Monte Carlo prediction. In Monte Carlo prediction with function approximation we run an episode and get a return. Then for each time-step  $t$  of that episode, we update the parameters using the formula given before, but with  $G_t$  instead of  $v_\pi(S_t)$ .

### 3.4 Policy Gradient Reinforcement Learning

Policy Gradient methods are another way to solve the Reinforcement Learning Problem. However, instead of learning the state value function or the action value function, we learn the policy directly. There are many variations on these methods, but the algorithm we will

use is called the REINFORCE algorithm. This algorithm assumes we have a differentiable function approximator defining our policy. This section is based on Chapter 13 from [24].

In Policy Gradient methods we have a parametrized policy. We need to define a performance measure  $\eta(\theta_t)$  that defines how well the policy is doing. Then, we need to learn the weights that maximize this performance measure. To do this, we use optimizations methods like gradient ascent. Note that now we say ascent instead of descent because in this case we follow the gradient and not the negative of the gradient, because we want to maximize instead of minimize. The update rule for the parameters will be:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla \eta(\theta_t)}$$

where the  $\widehat{\nabla \eta(\theta_t)}$  is an approximation of the performance measure that we selected. Depending on the election of  $\eta(\theta)$  we will have different algorithms. For the episodic case we generally use  $\eta(\theta) = v_{\pi_\theta}(s_0)$  and for the non episodic cases we use  $\eta(\theta) = r(\theta)$  where  $r(\theta)$  is the average reward rate.

The REINFORCE algorithm updates the parameters with the following rule.

$$\theta_{t+1} = \theta_t + \alpha \gamma' (G_t - b(S_t)) \frac{\nabla_{\theta} \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)}$$

where  $b(S_t)$  is called the baseline and is used to diminish variance, but it could be zero. One kind of baseline that we can use is an approximation of state value function or the action value function. Generally this is done by having an estimated value function using some kind of function approximator; this kind of method is called actor-critic [24]. We can use the log trick [22]  $\nabla \pi(A_t | S_t, \theta) = \pi(A_t | S_t, \theta) \nabla \log \pi(A_t | S_t, \theta)$  to simplify the last formula.

## 3.5 Monte Carlo Tree Search

Monte Carlo Search Tree (MCTS) has been very useful in the Game of Go. Thanks to this method, the computer Go program Fuego was the first program to defeat a 9 dan professional Go player in an even match on the  $9 \times 9$  board [7]. Previous to AlphaGo, all the strong playing programs implemented this technique, but for  $19 \times 19$  this was not enough. This section is based on section 8.7 from [24] and section 3.3 from [10].

Monte Carlo Tree Search is different from the previous described techniques because it focuses on planning more than on learning. In order to select an action, the agent has to make a search tree. The method is based on "random" simulations (like in Monte Carlo method) but in this case we do many simulations to update the nodes of the tree. We will

describe the most basic version of MCTS with a technique called Upper Confidence Tree (UCT).

Each node represent a state-action pair (s,a). Each node in the tree has two values: a count n that tells how many times the node has been visited during simulations and an action value Q(s,a) that is estimated by the simulations. At fist we only have the root of the tree. Then we apply these four stages iteratively:

1. Selection. From the current node, if that node has child nodes, we select next node (action) by selecting one according to a tree policy.
2. Expansion. When we reach a state-action pair that is not in the tree, and at the end of the episode, get the Q(s,a) of this state-action pair and add it to the tree.
3. Simulation. From the new added node, we simulate a game from that position using a rollout policy and get the total return of the episode.
4. Backup. We update the value of the child node, and update the values of the previous nodes, using this node's value.

The most simple tree policy is  $\epsilon$ -greedy. However, a technique called Upper Confidence Tree can be used to select actions within the tree. This method is based on the Upper Confidence Bound that works on the bandit problem [24]. So in this case, we will treat each action as a bandit, so we select the action that maximizes:

$$Z(s,a) = \frac{W(s,a)}{N(s,a)} + B(s,a)$$

$$B(s,a) = C \sqrt{\frac{\log N(s)}{N(s,a)}}$$

where  $B(s,a)$  is called the exploration bonus and  $\frac{W(s,a)}{N(s,a)}$  is the approximated action value function. We can see here that as we visit a state action pair more times, the exploration bonus goes to zero.

# Chapter 4

## Related Work

Computer Go research has been present since 1970 [29], but in this chapter we focus specially on the works done from 2000 to the present year. In this chapter, we will introduce a synthesis of the work done previous to AlphaGo, since AlphaGo, although proposing some innovations, is mainly a combination of multiple techniques.

### 4.1 Introduction to Computer Go

Research on Computer Go began in the 1970s, with the first Computer Go engine made in 1970 as part of the PhD thesis of Zobrist [29]. Some of the early progress on research was made by Wilcox and Reitman [17], who set the basis for many programs to come. The first Go program to play better than an absolute beginner was designed by them. This program illustrates the subsequent generation of Go programs that used abstract representations of the board, and reasoned about groups. They developed the theory of sector lines and dividing the board into zones, so as to reason about these zones.

Place	Prize
1st	NT\$200,000
2nd	NT\$40,000
3rd	NT\$20,000

Table 4.1: Prizes offered by the Ing Foundation. Taken from [8]. US\$1 was equal to NT\$25 at the time of publication (2013).

During the 1990s many tournaments were created for computer Go programs to play. One of the most important was made by the Ing Wei-Chi Educational Foundation. They

made yearly tournaments and gave prizes for the top programs. The prizes for these yearly tournament are shown in Table 4.1 and Table 4.2.

Handicap Moves	Games	Prize (NT\$)	Status
Even	4 of 7	40,000,000	-
First play	3 of 5	20,000,000	-
One move	2 of 3	10,000,000	-
Two moves	2 of 3	5,000,000	-
Three moves	2 of 3	2,000,000	-
Four moves	2 of 3	1,000,000	-
Five moves	2 of 3	850,000	-
Six moves	2 of 3	700,000	-
Seven moves	2 of 3	550,000	-
Eight moves	2 of 3	400,000	-
Ten moves	2 of 3	250,000	Won by Handtalk in 1997
Twelve moves	2 of 3	200,000	Won by Handtalk in 1995
Fourteen Moves	2 of 3	150,000	Won by Handtalk in 1995
Sixteen Moves	2 of 3	100,000	Won by Goliath in 1991

Table 4.2: Prizes for beating a top professional Go player. Taken from [8].

The tournament consisted of a pre-qualification where programs around the world could participate. To qualify for the tournament they had to be able to beat two low level programs. The tournament required 12 to 16 players so when less players were available, local computer Go programs were invited. Once the tournament was over, the computer-human handicap games began. For the handicap games, the winning program competed against three young professionals which rank was 5 *dan* or 6 *dan*, to try to beat at least two of them to get the prize. If the match was won, then another match followed with less handicap moves, until the program could not beat the human players. The purpose of the tournaments and prizes was to get a computer Go program able to beat a professional Go player by the year 2000. Although this was not achieved, the prizes motivated a lot of research in these years, as noted by Muller [17], both at universities and private companies.

During the 1990s and beginning of the 2000s the research was focused on solving Go in two levels of abstraction: the strategic level and the tactical level. Some work on combinatorial game theory was able to surpass professionals in very late game positions [2]. In this time, Temporal Difference learning, a technique from Reinforcement Learning, was known and tested, but with little success, leading to a weak player. In this period

the majority of programs used pattern recognition and dictionaries of common responses to these patterns to play, also making some local searches and few global ones. These techniques will be described in the next section.

## 4.2 Traditional techniques in Computer Go

We call these techniques traditional techniques because, as mentioned in the introduction, they share a lot in common with the first computer Go program. These techniques were the techniques that were used before the arrival of MCTS to computer Go. These techniques were very difficult to implement, and consisted of many modules that needed to work together to get a strong computer Go program. One of the noted difficulties is that these programs played generally very good at the local positions, but performed bad globally, at least, when playing against stronger players.

An example of this type of programs is GnuGo. GnuGo is an open-source software which uses many traditional techniques like pattern matching and databases of common patterns. The program has a strength of between 5 to 7 *kyu* and has become the standard benchmark program to beat. This is mainly because it represents the older programs, and is open-source and free to download, so this is a good advantage.

The computer Go programs from this period contained all or many of the following modules [17]:

- Patterns and Pattern Matching. The program uses a database and a pattern matching subsystem.
- Knowledge representation.
  - Go board.
  - Go rules, executing and undoing moves.
  - Blocks.
  - Connections, dividers and section lines.
  - Chains.
  - Surrounded areas, potential and safe territory.
  - Groups.
  - Global move generation.
  - Move evaluation.



- Game Tree search.
  - Single goal search.
  - Multi goal search.
  - Global level search.
- Subproblems solver.
  - Life and death.
  - Safety of stones and territory.
  - Semeai (capturing race).
  - Endgame.

One can see that these techniques required a lot of work. According to Muller [17], "most competitive programs required 5-10 person-years of effort, and contain[ed] 50-100 modules dealing with different aspects of the game". The code of GnuGo can give us some insight about this. That is why the MCTS revolution was so amazing, but we will talk about that in a later section.

## 4.3 Convolutional Neural Networks in Computer Go

Neural Networks were used since 2003 in Computer Go [27], but with very limited success. It was not until recently that Neural Networks, specifically Convolutional Neural Networks, had a huge success. This was done thanks to the recent advances on neural network research. Additionally, the combination with Reinforcement Learning gave performance even a bigger boost. Convolutional Neural Networks have been introduced to many research areas successfully, and Computer Go is not the exception. In this section we will present some of the work done in Computer Go. This section is based on the work done by Van Der Werf [27], Sutskever and Nair [23], Clark and Storkey [3] and Maddison et al.[15].

Early work using neural networks for move prediction was done by Van Der Werf [27] on 2003. His approach used a two layer fully connected neural network and consisted of four different techniques combined. First, he avoided "unnecessary" weight adjustments by defining an error function that went to zero when the prediction was accurate. Also, he used dimensionality reduction because of the large number of features. Then, preprocessing was applied to get the most relevant features. Lastly, a second round of training was done with

the newly selected features. The features were encoded as binary features which included: stones, edges, ko, liberties, liberties after move, captures after move, nearest stones and last move. This approach managed to get a 25% on predicting expert moves.

Later work by Sutskever and Nair in 2008 [23], showed an improvement in performance by using two layer convolutional neural networks for move prediction. They trained various convolutional neural networks and averaged over the predictions of all of them. The architecture of their neural networks were: in the first convolutional layer, they used 15 kernels of size  $9 \times 9$ . In the second convolutional layer, they used 15 kernels of size  $5 \times 5$ . Each convolutional layer was zero-padded in order to preserve the output size on each layer. The output layer only applied a softmax to the outputs in order to have a probability distribution over the different positions to make a move on the board. Their ensemble of networks achieved an accuracy of 36.9% and their best convolutional neural network achieved an accuracy of 34%.

It was not until 2014 that more progress was made using convolutional neural networks. The work done by Clark and Storkey [3] used deep convolutional neural networks to increase performance. They used some techniques to take into account symmetries in the weights to improve the performance of their networks. Their networks had 8 layers, including one final fully connected layer. As before, the convolutional layers were zero-padded. Although they experimented with many architectures, the best network used one convolutional layer with 64 filters of size  $7 \times 7$ , two layers with 64 filters of size  $5 \times 5$ , two layers with 48 filters of size  $5 \times 5$ , two layers with 32 filters of size  $5 \times 5$  and one fully connected layer. The results for this network were of 44% of accuracy on predicting moves made by experts on the KGS dataset. Also, they tested the network against GnuGo, and the network won 86% of the time.

The work published by Maddison et al., in 2015, achieved even better accuracies. They trained a large convolutional neural network consisting of 12 layers. They also used symmetries on the weights of the network to improve performance. The architecture of their network was: the first convolutional layer had filters of size  $5 \times 5$ ; the next layers had filters of size  $3 \times 3$ . Again, the convolutional layers were zero-padded to preserve the output size. Each layer had the same number of filters, ranging from 64 to 192. The accuracy achieved by their best network was of 55% for predicting expert moves. They also were able to defeat GnuGo 97% of the time, using this network directly to play.

## 4.4 Monte Carlo Tree Search in Computer Go

Monte Carlo Tree Search (MCTS) became very popular in Go after the Upper Confidence Tree (UCT) algorithm was proposed by Kocsis and Szepesvári in 2006 [13]. Gelly implemented the first program using MCTS UCT in 2006. The first computer Go program to beat a professional Go player on the  $9 \times 9$  board used this technique and some extensions [7]. Here we will explain this technique and some of its extensions. This section is based on [10].

Monte Carlo Tree Search uses Monte Carlo prediction on a game tree. The procedure is similar to minimax search, but the tree will grow vertically rather than horizontally, because it adds new nodes of states that were experienced during simulations. Also, here we do not have an evaluation function, so the values of the nodes are estimated from experience.

In MCTS, and in many other techniques of Reinforcement Learning, we always have the trade-off of exploitation vs exploration. We would like to simulate more over the nodes that gives us the best action values, that is, the ones that give greatest return. However, if we only do that, we give up on exploration, so maybe there are some very good nodes that we have not explored and that have greater values. A simple method to address this problem is the  $\epsilon$ -greedy algorithm, but we have the disadvantage that the agent will always have some probability of exploring, which does not diminish over time.

One of the techniques developed to balance this exploration-exploitation problem is the Upper Confidence Tree (UCT) algorithm. The idea is that at the beginning, we want to explore many nodes, but as time passes, we want to diminish that exploration. This algorithm is based on the Upper Confidence Bound algorithm for the bandits problem [24]. We will choose actions based on a score, instead of directly choosing from the estimated action value function.

$$Z(s, a) = \frac{W(s, a)}{N(s, a)} + B(s, a)$$

$$B(s, a) = C \sqrt{\frac{\log N(s)}{N(s, a)}}$$

where  $B(s, a)$  is called the exploration bonus,  $C$  controls the level of exploration,  $N(s, a)$  is the number of times this node was selected during simulations, and  $W(s, a)$  is the sum of returns for this node. We see that as  $N(s, a)$  goes to infinity, the exploration bonus goes to zero, so the score will only take into account the estimated action value  $\frac{W(s, a)}{N(s, a)}$ .

This algorithm is proved to converge given enough exploration. The idea behind this is that as time passes the best nodes will be selected, and their estimates will be more accurate. Although convergence could take a lot of time to occur, even with estimates we have a very good algorithm, since intuitively the nodes of the tree are chosen in a more intelligent way. Many variations of the UCT algorithm try to address the problem of slow convergence.

The main problem of MCTS is that we do not generalize, and each node estimates its value only considering the simulations that passed through that node. It would be good if we could improve our estimates by improving from the simulations in which similar nodes occur. There is an heuristic called all-moves-as-first (AMAF) treats every action the same, independently of the state in which it was played.

Based on AMAF, Silver and Gelly [9] developed the Rapid Action Value Estimate (RAVE) algorithm. The idea is that in the subtree from any node, the value of taking an action will be the same for any node in that subtree. The estimate resulting from this procedure will not be an accurate estimate as time passes, but RAVE helps in making faster updates.

$$\tilde{Z}(s, a) = (1 - \beta(s, a)) \frac{W(s, a)}{N(s, a)} + \beta(s, a) \frac{\tilde{W}(s, a)}{\tilde{N}(s, a)}$$

where  $\beta(s, a)$  controls how much the AMAF estimate is taken into account,  $\tilde{N}(s, a)$  is the AMAF count and  $\tilde{W}(s, a)$  is the AMAF total reward. The function  $\beta(s, a)$  goes to zero as  $N(s, a)$  goes to infinity.

There are other extensions that permitted further improvement using MCTS. However, for the purposes of this thesis this is enough. Later, when we describe the architecture of the MCTS part of AlphaGo we will see the similitude with the MCTS-RAVE algorithm.

## 4.5 AlphaGo: Deep Learning and Reinforcement Learning in Computer Go

The previous sections described some of the work done on Computer Go using Reinforcement Learning or Convolutional Neural Networks, but not together. These previous works gave the foundations for AlphaGo. AlphaGo is composed of four elements. In this section we will describe all the elements, given that we know a lot of the previous work, and see how this was innovative. The most important thing was the creativity to combine all of the techniques. This section is based on [21]. The notation is also taken from there.

The first element of AlphaGo is a move predictor implemented by a deep neural network. As we saw in previous sections, a move predictor is not necessarily a good player,

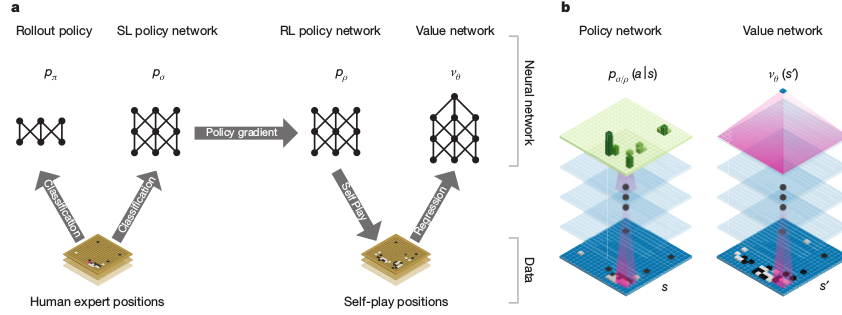


Figure 4.1: AlphaGo architecture. Taken from [21].

however it can give very good results alone. The neural network was trained using the KGS dataset [12], and the architecture consisted of 12 layers, with each layer implementing a convolution and a linear rectifier. The input to the network is a representation of the board through features like color of stone, liberties, ko, etc. They used 192 filters in each layer with size  $5 \times 5$  for the first layer, and size  $3 \times 3$  for the following layers. The algorithm used for training was Asynchronous Stochastic Gradient Descent using following hyper parameters: learning rate  $\alpha$  of 0.003 and mini batch size  $m$  of 16. The parameters  $\sigma$  were updated using the following formula:

$$\Delta\sigma = \frac{\alpha}{m} \sum_{k=1}^m \frac{\partial \log p_{\sigma}(a^k | s^k)}{\partial \sigma}$$

The second stage used the same network but applied reinforcement learning to it. As we saw before, we can improve a policy, in this case the move predictor, by policy gradient and games of self play. In this case, the old parameters  $\sigma$  were copied to parameters  $\rho$ . A pool of players was created, initializing this pool with the fixed parameters  $\sigma$  from the previous step. Each game was played between the player with parameters  $\rho$  and a randomly chosen player from the pool of players. Every 500 games of self play, the parameters  $\rho$  were backed up and stored in the pool of players. This was done in order to avoid overfitting. At the end of each episode the reward  $z_t = \pm r(S_{T^i})$  was used to update the parameters. The reward is +1 if the player won and -1 if it lost. The algorithm used for learning was the REINFORCE algorithm using  $v(s_t^i)$  to reduce the variance. The formula used to update the parameters of the network is:

$$\Delta\rho = \frac{\alpha}{n} \sum_{i=1}^n \sum_{t=1}^{T^i} \frac{\partial \log p_{\rho}(a_t^i | s_t^i)}{\partial \rho} (z_t^i - v(s_t^i))$$

The last part of training was the creating of a state value function. This could be seen as the most important element of AlphaGo and one of its main contributions. This state

value function gives use a measure of how good it is to be in a given state. It was thought that obtaining this "evaluation" function was impossible because of the complexity of the game of Go. However, the team that created AlphaGo made a good approximation of this evaluation function using the approximate state value function. We saw earlier that the state value function is given by the following formula:

$$v^p(s) = \mathbb{E}[z_t | s_t = s, a_{t...T} \sim p]$$

To approximate this value function we can use a neural network that just outputs a single value between -1 and 1. This can be done solving the problem of regression using supervised learning, using a deep neural network as before. The dataset to train the neural network was generated from the RL network since it was stronger than the SL network.

The dataset was generated by playing games of self play and obtaining the result. One of the problems with the algorithm in Chapter 3 for the approximate value functions is that the positions in each episode are highly correlated, so the assumptions of supervised learning do not hold. In order to have a dataset that holds the assumption of independence, for each game of self play they took a single training example  $(s, z)$  where  $s$  is the state of the board and  $z$  is the score at the end of the game. The algorithm used for learning was stochastic gradient descent, using the mean squared error (MSE) as the objective function to minimize. The formula used to update the parameters of the network is:

$$\Delta\theta = \frac{\alpha}{m} \sum_{k=1}^m (z^k - v_{\theta}(s^k)) \frac{\partial v_{\theta}(s^k)}{\partial \theta}$$

where  $z^k$  is the score, and  $s^k$  is the state of the board, for the  $k$ -th training example,  $\alpha$  is the training rate, and  $m$  is the size of the mini-batch.

The dataset used contained 30 million examples. At the end of training, the training error was 0.226 and the test error was 0.234, indicating that there was no overfitting. This value network was tested against Monte Carlo evaluation and gave similar accuracy but with 15,000 times less computation.

The only element left to explain of the AlphaGo program is searching. For this, the AlphaGo team developed a new version of the MCTS algorithm. This algorithm was called Asynchronous policy and value MCTS algorithm (APV-MCTS). In this algorithm, each node represents an state  $s$  and from each node we have edges  $(s, a)$  for all the possible actions from that state. The edges store the following variables:  $P(s, a)$ ,  $N_v(s, a)$ ,  $N_r(s, a)$ ,  $W_v(s, a)$ ,  $W_r(s, a)$ , and  $Q(s, a)$ , where  $P(s, a)$  is called the prior probability initialized on the output of the move predictor with parameters  $\sigma$ ,  $N_v$  and  $N_r$  are counters of the number of times the node was visited during simulations and  $W_v$  and  $W_r$  are the total rewards obtained

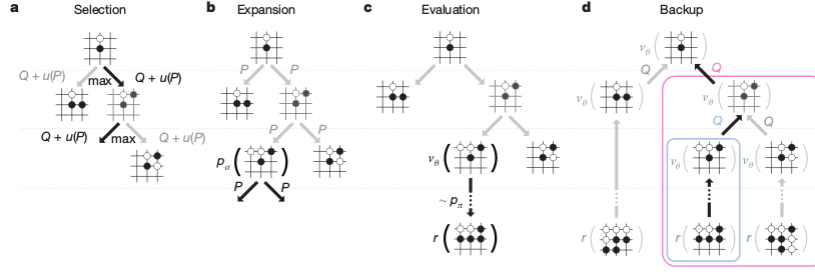


Figure 4.2: AlphaGo MCTS. Taken from [21].

during simulations for this node. The sub-index  $v$  indicates that the estimated value of the node  $\frac{W_v}{N_v}$  uses the value network, and the sub-index  $r$  denotes that the estimated value of the node  $\frac{W_r}{N_r}$  uses the roll-out network. We can combine both estimates using parameter  $\lambda$  and the formula:

$$Q(s, a) = (1 - \lambda) \frac{W_v(s, a)}{N_v(s, a)} + \lambda \frac{W_r(s, a)}{N_r(s, a)}$$

During searching, multiple simulations are executed in parallel on separate search threads, and also, the evaluation using the state value function is done in parallel. The APV-MCTS algorithm proceeds in the four stages outlined in 4.2.

# Chapter 5

## Proposed Solution

In the previous chapters we gave the theory needed to understand the elements of AlphaGo, and how those elements worked separately. Now we will describe the development of our own computer Go playing program called NdsGo. This program is meant to help to understand the concepts better by implementing them. The program was designed to play in  $2 \times 2$ ,  $3 \times 3$ ,  $9 \times 9$  and  $19 \times 19$  boards. We used Python to develop this program along with some libraries like Numpy, which we was used for the management of arrays, and Tensorflow, used for the implementation of different architectures of neural networks.

The program is composed of many modules that have different functions. In general, we can describe the program as having these parts:

- Implementation of board representation.
- Implementation of Go Text Protocol for playing against other programs.
- Data processor which converts a set of files in SGF format to a binary file for CNN training.
- Implementation of a Player for  $2 \times 2$  and  $3 \times 3$  based on Monte Carlo self-play.
- Implementation of a player for  $9 \times 9$  and  $19 \times 19$  using CNNs to predict moves.

In this chapter we describe the details of each of these elements.

### 5.1 Implementation of Board representation and the Go Text Protocol

A computer Go player must have an internal board representation in order to be able to know how to play. For this task we based our board implementation in the one described by



Muller in [17]. The board is stored in a flat array, and we transform from two-dimensional coordinates to one-dimensional coordinates. The basic features that the board representation should have are the ability to manage capturing stones, and the ability to detect and prevent illegal moves like playing in a ko position, playing in a non empty space, or playing a suicidal move.

Another important feature for a computer Go program is the implementation of the Go Text Protocol. This protocol allows the programs to communicate in order to play against each other. This is an important feature for our program, because it permits to compare our program against other programs.

The Go Text Protocol (GTP) is a very simple communication protocol which involves two parts, the controller and the engine [11]. The specification says that it can be implemented. There are many use cases in the specification of the protocol; however, we only implemented the "engine" part of the protocol. The implemented commands are shown in table 5.1

Command	Description
name	Returns the name of the Go playing program.
version	Returns the version of the Go playing program.
known_command	Command to ask if a command is implemented.
list_commands	List all the implemented commands.
quit	Finish the connection.
boardsize	Define a new boardsize.
clear_board	Clear the board and number of captures to begin a new game.
komi	Defines the komi given to the white player.
play	Play a stone of the indicated color in the indicated position
genmove	Generate a move for the indicated color.
undo	Undo the previous move.
showboard	Tells the program to print the board.

Table 5.1: Engine commands for the GTP protocol.

These commands made it possible for the program to play against other programs using GoGui [6] and against human players using the Kiseido Go Server <sup>1</sup>. See the description of these tests in the experimental results chapter.

<sup>1</sup>W. Schubert, KGS Go Server. [Online]. Available: <https://www.gokgs.com/>

## 5.2 Implementation using Monte Carlo Self-Play for $2 \times 2$ and $3 \times 3$ boards

The techniques that we reviewed in Chapter 3 work for MDPs, where we have an agent and an environment. But in games we want to learn the policy for "two agents"; that is, the two players. The techniques work by taking into account some details. Instead of using MDPs we will work on Markov Games.

This section describes the implementation of a computer player which learns the action value function  $Q(s,a)$  by Monte Carlo Reinforcement Learning. However, there is a difference with the Monte Carlo technique described before. In the case of games, we have two players but we will only learn a single value function. So we need to define what we mean by the value function. We consider zero sum games, that is:

$$R^{P1} + R^{P2} = 0$$

The formula says that the rewards for both players at each time step have the same absolute value, but opposite sign. The new value function takes into account both players, and this function is the one we will learn. The policy that we want to learn is  $\pi = \langle \pi^1, \pi^2 \rangle$ . So taking this into account, the value function is the same:

$$q_\pi = \mathbb{E}[G_t | S_t = s, A_t = a]$$

For this module, we want to learn by Monte Carlo control the estimated action value functions. For this the program plays  $n$  games of self play to train. For each state-action pair it stores an action value estimate  $Q(s,a)$  and a count of how many times that value has been seen (and updated),  $N(s,a)$ . After each game, it updates both  $Q$  and  $N$  according to the following formulas:

$$N(s,a) = N(s,a) + 1$$

$$Q(s,a) = Q(s,a) + \frac{1}{N(s,a)} [G_t - Q(s,a)]$$

We used the  $\varepsilon$ -greedy policy to allow exploration. Also we experimented with Optimistic Initialization for exploration.

## 5.3 Datasets and data formatting

For the supervised learning training of our networks we used two datasets: one comes from the Kiseido Go Server dataset<sup>2</sup> and the other from the Games of Go on Disk<sup>3</sup>. The first dataset was used for training the  $19 \times 19$  player; the second one was used for training the  $9 \times 9$  player. Some details about the KGS dataset are shown in table 5.2. We used only the games from 2016, mainly because of storage issues.

KGS dataset	no of games
2001	2,298
2002	3,646
2003	7,582
2004	12,106
2005	13,941
2006	10,388
2007	11,644
2008	14,002
2009	18,837
2010	17,536
2011	19,099
2012	13,665
2013	13,783
2014	13,029
2015	8,133
2016	19,442
Total	199,131

Table 5.2: KGS dataset as of October 2016

We trained neural networks for predicting moves in  $19 \times 19$  and  $9 \times 9$  boards. The dataset used for training the  $19 \times 19$  CNN were obtained from the KGS dataset. This dataset contains games of from 2001 to 2016 played on the KGS Server. This dataset also has the characteristic of only containing games in which one player is 7 dan or both players are 6 dan. Remember from the introduction that although this rank is in the amateur level,

---

<sup>2</sup>U. Görtz, KGS Game Records, 2016. [Online]. Available: <https://u-go.net/gamerecords/>

<sup>3</sup>T. M. Hall and J. Fairbairn, GoGoD - Games of Go on Download. [Online]. Available: <http://gogodonline.co.uk/>

Dataset	Board size	games	train examples	test examples
KGS 2016	19x19	19,442	2,776,980	385,694
GoGoD 9x9 subset	9x9	146	4,608	643

Table 5.3: Datasets used

7 dan is the maximum possible rank, so the level of play is very good. The total number of games in this dataset is 190,450. Since in average we have 150 moves per game, we have approximately 28 millions of training examples. Later we will talk about how we divided this dataset for train and test.

The format of the data is converted to a more convenient format to feed the deep neural networks that we created. The original format is called SGF, and is easy to understand since it describes each move made by the players during a match. However, for the neural network we need to represent each state of the game by the complete configuration of the Go board. After converting the SGF files, we divided the datasets into a train set and a test set. The details are shown in table 5.3

For the format, we encoded the board configuration using four planes of size  $19 \times 19$  or  $9 \times 9$ , depending on the dataset. These four planes are composed of 1's and 0's that describe the features of the board. We selected the most basic board features to begin experimentation, but adding more features will surely improve performance, as was seen in the works of Sutskever [23], Clark and Storkey [3] and Maddison [15]. Then, we have four channels, described here:

1. First plane. Presence of player's stones. A 1 in the position if a player stone is present and 0 otherwise.
2. Second plane. Presence of opponent's stones. A 1 in the position if an opponent stone is present and 0 otherwise.
3. Third plane. Presence of empty spaces. A 1 in the position if the position is empty and 0 otherwise.
4. Fourth plane. All ones. This feature helps the network to distinguish the boundaries of the board, since in the network we use zero-padding in each convolutional layer.

Since the information is only 1's and 0's, after creating these planes we stored them in binary form. The "label" for each of these binary arrays was the next move to play, encoded as a number between 0 and 360 (or 0 and 80 for the  $9 \times 9$  board). For this number we used two bytes to encode it in binary form. We stored each example as a fixed length

record were the first two bytes contained the label, and the following  $361 \times 361 \times 4$  (or  $81 \times 81 \times 4$ ) bytes corresponded to the board configuration.

After creating records for the entire SGF dataset, we then divided these records into training and test sets. We then shuffled each set because during training since we will estimate the gradient from mini batches, we want those mini batches to be as diverse as possible, in order to be good representatives of the entire dataset. Also it is important to note that we first divided the data and then shuffled it; this is important, because we want the test data to be as much independent from the training data as possible. In the way we did it, the examples of the test data are from different games than those on the training data. This has the intention of being able to measure if the neural networks generalize well.

## 5.4 Convolutional Neural Networks for move prediction using Supervised Learning

The first stage of training used in AlphaGo was Supervised Learning using Convolutional Neural Networks to predict expert moves. We did the same, experimenting with three different architectures. Also we experimented with the  $19 \times 19$  dataset and the  $9 \times 9$  dataset, so the architectures take this into account. In the diagrams shown below, for  $9 \times 9$ , the details are the same, just changing the size to 9 instead of 19. All the architectures used the same hyperparameters: fixed learning rate of 0.1, batch size of 128 and number of filters on each convolutional layer equal to 32, with the exception of the last layer which only has one filter.

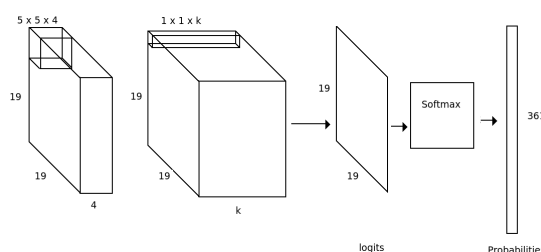


Figure 5.1: CNN architecture with 2 convolutional layers

The first architecture consisted of a network with 2 convolutional layers. This network was inspired in the work done by [23]. The first convolutional layer has 32 filters of size  $5 \times 5 \times 4$ . After the convolution, we add a bias and apply a rectifier function. The second and last layer has only one filter of size  $1 \times 1 \times 32$ ; This outputs a feature map of size  $19 \times 19$  to which we add a bias for each position and apply a linear rectifier. The outputs

are "numerical preferences" for each position on the board. Using these outputs from the convolutional layer we pass them to a softmax layer, which outputs probabilities for each position of the board.

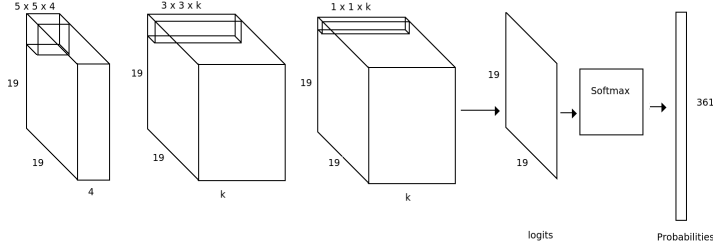


Figure 5.2: CNN architecture with 3 convolutional layers

The second architecture consisted of a network with 3 convolutional layers. The first convolutional layer has 32 filters of size  $5 \times 5 \times 4$ . After the convolution, we add a bias and apply a rectifier function. The second layer has 32 filters of size  $3 \times 3 \times 32$ ; we also add a bias and apply a rectifier function here. The third and last layer has only one filter of size  $1 \times 1 \times 32$ . As before, these outputs are passed to a softmax layer, which outputs probabilities for each position of the board.

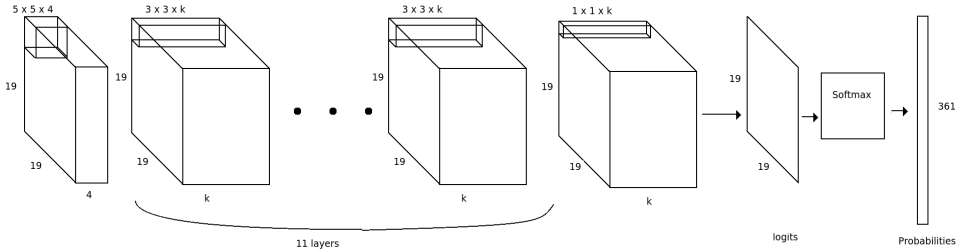


Figure 5.3: CNN architecture with 13 convolutional layers

The third architecture consisted of a network with 13 convolutional layers. This is the same as the one used by AlphaGo, but we have different hyper parameters. The difference in hyper parameter is mainly because we do not use the same algorithm; AlphaGo used Asynchronous Stochastic Gradient Descent and we used Stochastic Gradient Descent. The first convolutional layer has 32 filters of size  $5 \times 5 \times 4$ . After the convolution, we add a bias and apply a rectifier function. The layers 2 to 12 have 32 filters of size  $3 \times 3 \times 32$ ; we also add a bias and apply a rectifier function in each one of them. The last layer has only one filter of size  $1 \times 1 \times 32$ . As before, the outputs from the last convolutional layer are passed to a softmax layer, which outputs probabilities for each position of the board.

The training was done using Stochastic Gradient Descent, using a mini batch in each step. We trained for 10,000 steps for each of the architectures and each of the datasets. The results are reported in the next chapter.

## **5.5 What is left to reach AlphaGo performance?**

In order to complete the study on the different elements of AlphaGo, we would need to implement three more elements:

- A Convolutional Neural Network for move prediction using Policy Gradient
- A Convolutional Neural Network to approximate the state value function.
- A Monte Carlo Tree Search implementation.

The last step would be to make an unified architecture similar to that of AlphaGo. For these implementations, we would use a centralized system, because we only want to understand the concepts, not make a competitive program. But another option could be to try to implement a distributed version, similar to the one used in the distributed version of AlphaGo.

# Chapter 6

## Experimental Results

In this section we present some experiments and results from the NdsGo program. We used GoGui [6] to compete against GnuGo. Also we give some intuition about how the learning process was made, and a description of the difficulties found on each implementation.

### 6.1 Results for the Monte Carlo Self Play Implementation on $2 \times 2$ and $3 \times 3$ boards

For the  $2 \times 2$  board, we training the player for 1,000,000 games of self-play. For each episode, we saved the values of all the 810 possible state-value pairs in order to plot them later and see how the values changed over time. In Figure 6.1 we can see the evolution of these values as more games are played for training. The three graphs in Figure 6.1 show the values, but the first one only for the first 100 episodes, the second one for 10,000 episodes, and the last one for 1,000,000 episodes.

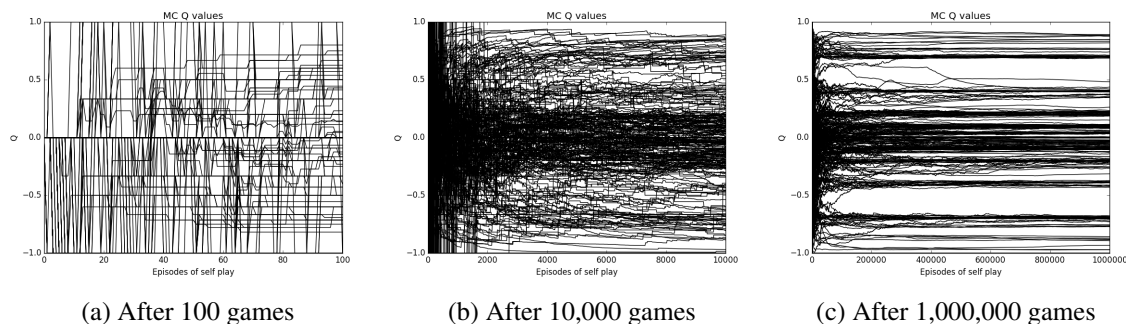


Figure 6.1: Q(s,a) values for  $2 \times 2$  board learned after 100, 10,000 and 1,000,000 games of self-play.



From these graphs we can see how the values start to converge after many episodes have been played. However, this happens only because the state space is too small, so exploration is enough. Also, we noted that during training there were a lot of long episodes because we added the passing move as one of the possible actions, so we did not finish the game until the program passed two consecutive times.

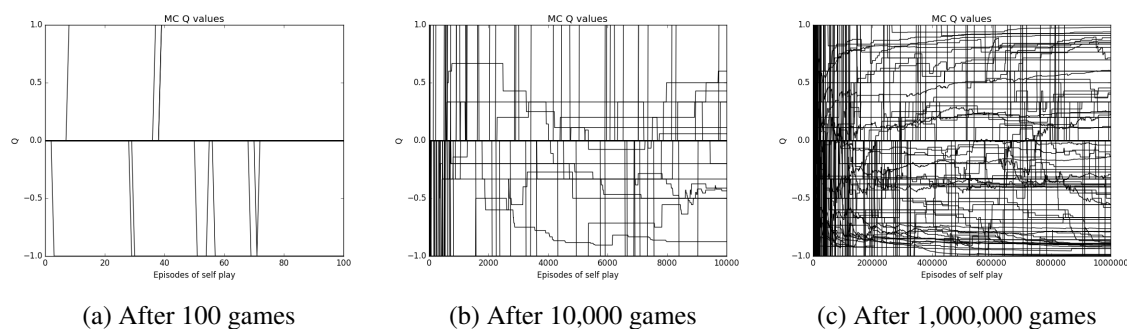


Figure 6.2:  $Q(s,a)$  values for  $3 \times 3$  board learned after 100, 10,000 and 1,000,000 games of self-play.

In Figure 6.2 we can see the evolution of the values for state action pairs, but this time for the  $3 \times 3$  board. Because of the limitations in memory, we only followed the evolution of 1,000 of the state action pairs during training; in total there were 393,660 different values. These state action pairs were selected randomly, but we can see from the graphs that convergence is not so clear as in the  $2 \times 2$  case. There are two factors for this, the first one is that we are not plotting all the values, so maybe the most promising values are not along these plotted values. The second reason is that since we have a larger state space, we would need more iterations of self-play in order to learn.

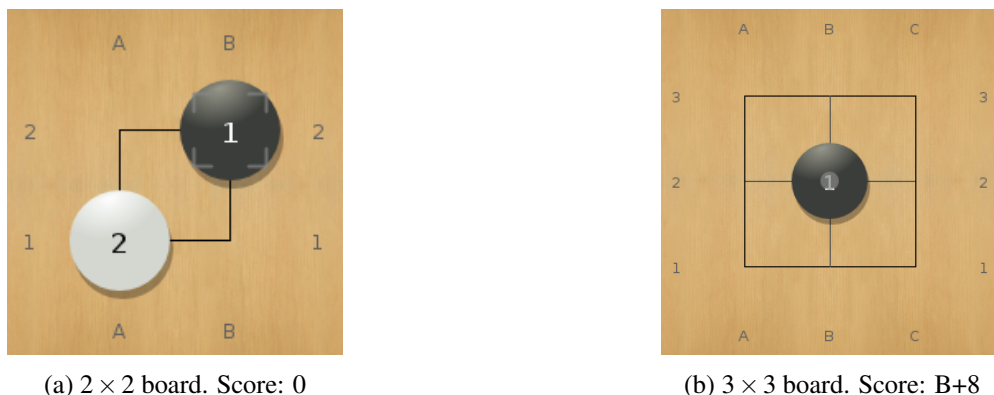


Figure 6.3: GnuGo (B) vs NdsGo (W) on the  $2 \times 2$  and  $3 \times 3$  boards.

We tested the program against GnuGo on  $2 \times 2$  and  $3 \times 3$  (6.3). In  $2 \times 2$  board the program ties against GnuGo on all its games (assuming a komi of 0), whether playing as Black or as White. This board is very simple and one can easily see that there is no other way to play to get more points. The problem here is that the only way to get points here is to capture stones, but when we try to capture stones we end up in an infinite loop where neither player gets more captures than the other one.

On the  $3 \times 3$  board the player playing black always wins if he plays in the center. Again, this is because of the limited size of the board that does not let us surround territory. The only way to gain points is to capture opponent stones, but here the first one to take the center has the advantage. So once one of the players takes the center, we can say that the game is over. It is important to note that the program learned to play in the center, although when playing as white, it tries to keep playing.

The main problem here is that we included the pass move among the possible actions. This turned out to be a bad approach. Although the first moves are perfect, then the program tries to keep playing, so we have infinite games. This is because it is difficult to learn when to pass, a problem that we will have later again with the neural network based implementation.

To solve this issue against other programs we made a simple trick: we pass when the other player passes. This is done assuming the other player is honest, and knows better than us when the game is over. This trick is applied directly on the GTP implementation. This must be changed for later versions of the program.

Board size	Byte size
2	6.4 KB
3	3 MB
4	10 GB
5	320 TB

Table 6.1: Memory to store the  $Q(s,a)$  for the Monte Carlo Self Play implementation

We limited this implementation to  $2 \times 2$  and  $3 \times 3$  because of the memory needed. We can see in Table 6.1 how the memory requirements increase exponentially for this kind of implementation. As we stated in chapter 3, this kind of reinforcement learning is very inefficient and almost exclusively used to understand the concepts.

## 6.2 Results for the Convolutional Neural Network implementation on the $9 \times 9$ board

For the  $9 \times 9$  board implementation, we used the GoGoD subset that contained  $9 \times 9$  games played by professionals. This dataset contains only 146 games, so it is very limited compared to the  $19 \times 19$  dataset.

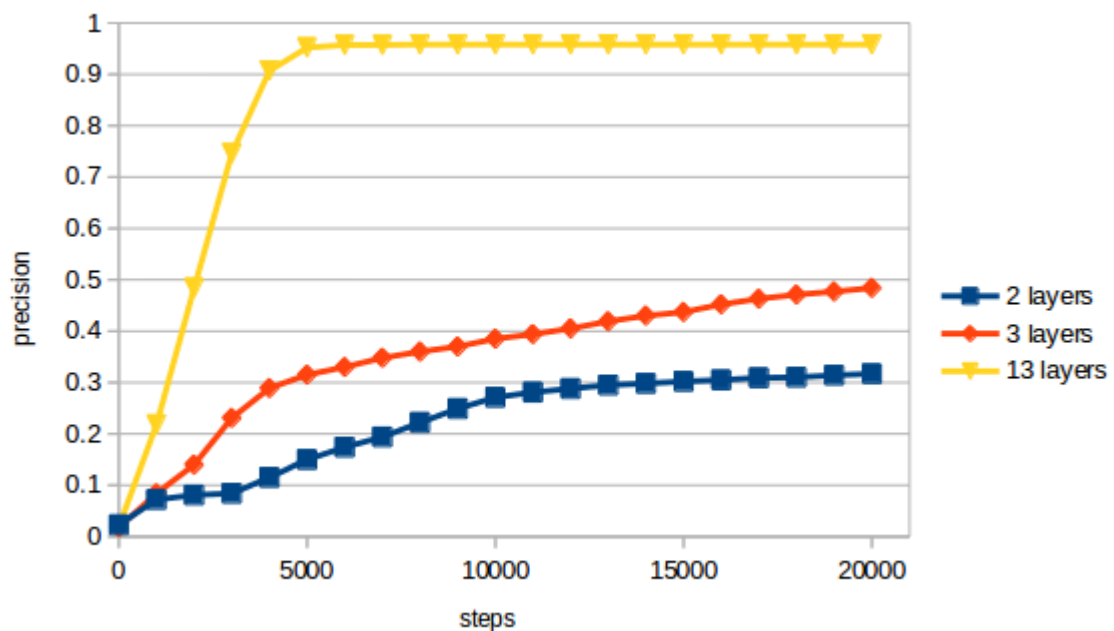
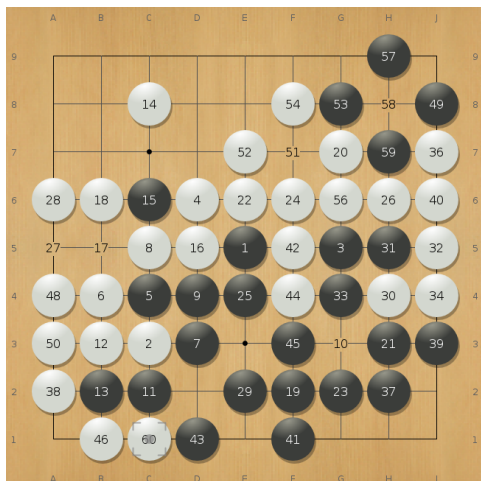


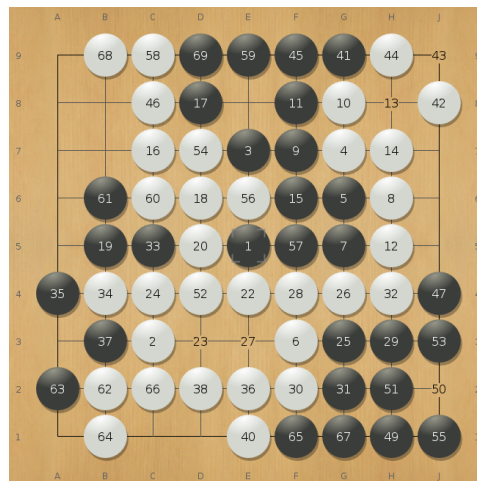
Figure 6.4: Precision on the test set for the GoGoD  $9 \times 9$  dataset.

In Figure 6.4 we can see the precision of classification on the test set. We trained for each of the architectures for 20,000 steps. We can see that the precision increases, so we do not reach overfitting yet. The regularization used helps in doing this.

We tested our program on the  $9 \times 9$  board against GnuGo. Figure 6.5 shows two games played against GnuGo. In both games, our program loses, however, in the first one, the program is able to build two eyes. We used the learned parameters for the architecture of 13 layers.



(a) Game 1. Score: W+26.



(b) Game 2. Score: W+82.

Figure 6.5: NdsGo (B) vs GnuGo (W) on  $9 \times 9$  without komi.

### 6.3 Results for the Convolutional Neural Network implementation on the $19 \times 19$ board

As stated before, we used only the games from 2016 from the KGS dataset to train for the  $19 \times 19$  board. We trained for 20,000 steps in each case. We did training for each one of the three proposed architectures described earlier. The hyperparameters are the same for both sizes, as described earlier.

The results for the different architecture are shown in 6.6. We can see that the networks learn very fast at the beginning and they seem to be no improvement. This is not totally right, since they keep improving but more slowly.

We tested our program on the  $19 \times 19$  board against GnuGo. Figure 6.7 shows two games played against GnuGo. In the first game our program won by a small margin, but in the second game it loses terribly. We used the learned parameters for the architecture of 13 layers.

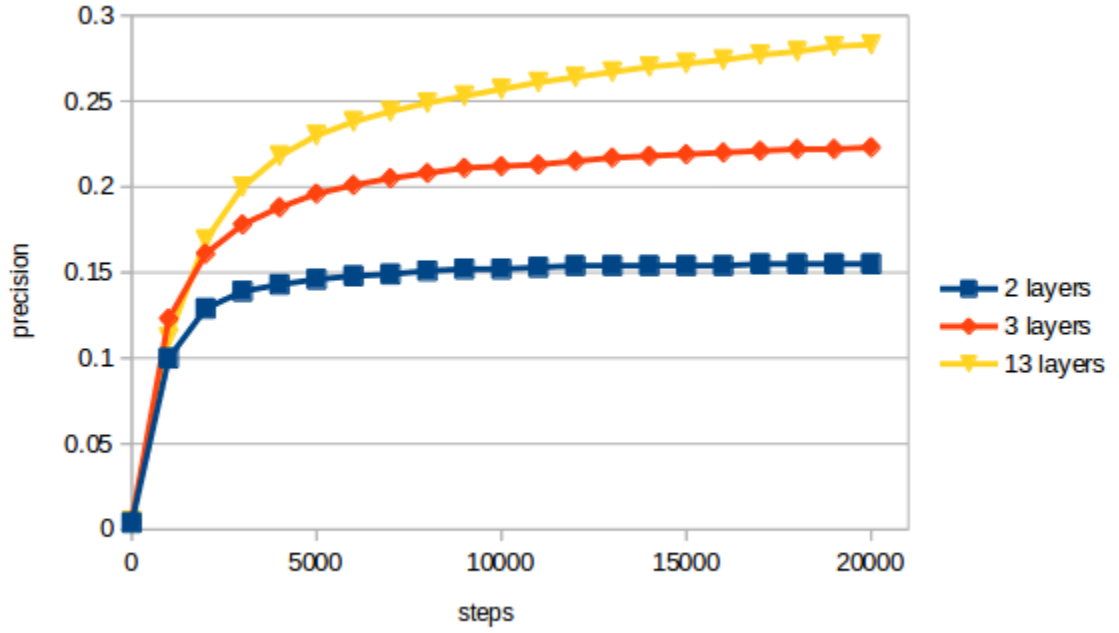
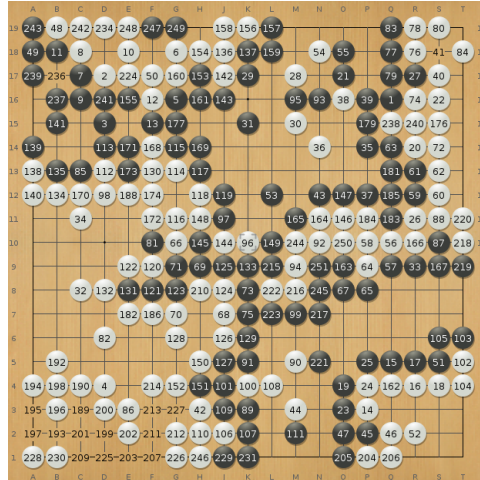
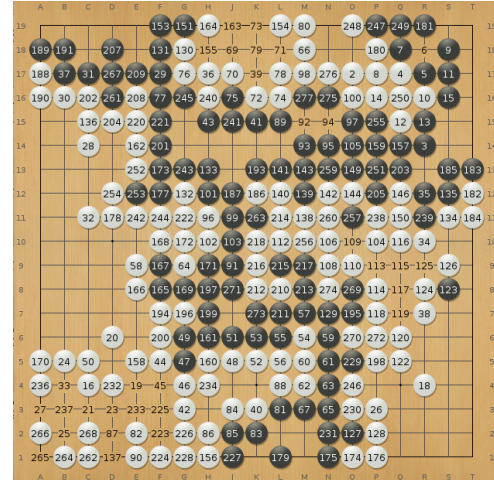


Figure 6.6: Precision on the test set for the KGS 2016  $19 \times 19$  dataset.



(a) Game 1. Final score: B+2



(b) Game 2. Final score: W+96

Figure 6.7: NdsGo (B) vs GnuGo (W) on  $19 \times 19$  without komi.

# Chapter 7

## Conclusions and Future Work

We presented the main elements used in AlphaGo and the previous work in which this program was based. We also created a computer Go program, that is not too strong, but permitted experimentation with Reinforcement Learning and Deep Learning. Another contribution of this work is that now we have a platform to experiment with; we can later use this platform to continue implementing the elements that we did not implement, like Policy Gradient and Monte Carlo Tree Search, or maybe try new ideas.

Some of the future work that can be done is:

- Implementation of Policy Gradient Reinforcement Learning on the CNNs in order to have a better player instead of a good move predictor.
- Implement Monte Carlo Tree Search and extensions.
- Test NdsGo in the Computer Go Server (CGOS) to compare against other programs.
- Test NdsGo in the KGS Go Server, to test the strength of the program against human players.

Although AlphaGo achieved something great by defeating a professional go player, computer go research will continue. It is true that there were a lot of innovative ideas that allowed AlphaGo to beat a professional, but the amount of computational resources used to do this was also influential. So, one of the next steps is to try to get those ideas working for smaller systems. This thesis was a small attempt to do so.

One of the difficulties during the implementation of the computer Go program was that although there is a lot of papers on computer Go, some very important information was difficult to find. For example, although all the programs need a board representation, we only found one paper describing how to implement it [17]. Another example is how to

tell the program to stop playing, that is, to tell it to pass when the game is over. This is a difficult task, because even for people it is often difficult to tell when the game is finished and when one should pass. However, every program needs at least a heuristic to do this. We were not able to find a description of this type of heuristic in the literature, which is crucial for a competitive computer Go program.

One of the things that made it easier to develop this computer Go program was the information contained in the Computer Go Mailing List <sup>1</sup>. We recommend to look at this list to find some discussions and answers to questions on computer Go research. The community there is very diverse, having developers, scientists and Go players which interchange messages. It is a good idea to subscribe to this mailing list to get the latest news from different researchers around the world. This mailing list is active as of today.

It seems impossible to get an AlphaGo-like program developed by only one person. The hardware available for students is also important since the experiments take longer depending on the equipment used. However, as seen here, we can develop some of the ideas and start experimenting with those simple ideas. In future work it is our hope that other students can develop from what we created.

---

<sup>1</sup><http://www.computer-go.org/>

# References

- [1] P. Baudiš and J. L. Gailly. PACHI: State of the art open source Go program. In: *Advances in Computer Games* (2012), pp. 24–38. DOI: 10.1007/978-3-642-31866-5\_3.
- [2] B. Bouzy and T. Cazenave. Computer Go: An AI oriented survey. In: *Artificial Intelligence* 132.1 (2001), pp. 39–103. DOI: 10.1016/S0004-3702(01)00127-8.
- [3] C. Clark and A. Storkey. Teaching Deep Convolutional Neural Networks to Play Go. In: *Machine Learning* 8.3 (2014), p. 9. URL: <http://arxiv.org/abs/1412.3409>.
- [4] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In: *Computers and games* 4630 (2007), pp. 72–83. DOI: 10.1007/978-3-540-75538-8\_7.
- [5] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In: *Nips* (2012), pp. 1–11. DOI: 10.1109/ICDAR.2011.95.
- [6] M. Enzenberger. GoGui - graphical user interface to programs that play the board game Go. URL: <http://gogui.sourceforge.net/>.
- [7] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. FUEGO - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 259–270. DOI: 10.1109/TCIAIG.2010.2083662.
- [8] D. Fortland. World Computer Go Championships. 2013. URL: <http://www.smart-games.com/worldcompgo.html> (visited on 11/02/2016).
- [9] S. Gelly and D. Silver. Achieving Master Level Play in 9 x 9 Computer Go. In: *Proceedings of AAAI* 1 (2008), pp. 1537–1540. URL: <http://dl.acm.org/citation.cfm?id=1620270.1620327>.



- [10] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, and O. Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. In: *Commun. ACM* 55.3 (2012), pp. 106–113. DOI: 10.1145/2093548.2093574.
- [11] S. Gelly and D. Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1876. DOI: 10.1016/j.artint.2011.03.007.
- [12] U. Görtz. KGS Game Records. 2016. URL: <https://u-go.net/gamerecords/>.
- [13] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In: *Proceedings of ECML (2006)*, pp. 282–203. DOI: 10.1007/11871842\_29.
- [14] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In: *Proceedings of the International Conference on Machine Learning 1* (), pp. 157–163. DOI: 10.1.1.135.717.
- [15] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver. Move Evaluation in Go Using Deep Convolutional Neural Networks. In: 2.3 (2015), pp. 1–8.
- [16] D. Michie. Experiments on the mechanization of game-learning. Part I . Characterization of the model and its parameters. In: *The Computer Journal* (1963), pp. 232–236.
- [17] M Müller. Computer Go. In: *Artificial Intelligence* 134.1-2 (2002), pp. 145–179. DOI: 10.1016/S0004-3702(01)00121-7.
- [18] M. A. Nielsen. *Neural Networks and Deep Learning*. 2015.
- [19] A. Rimmel, O. Teytaud, C. S. Lee, S. J. Yen, M. H. Wang, and S. R. Tsai. Current frontiers in computer go. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 229–238. DOI: 10.1109/TCIAIG.2010.2098876.
- [20] P. Shotwell. A Brief History of GO. 2014. URL: <http://www.usgo.org/brief-history-go> (visited on 10/29/2016).
- [21] 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. In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.
- [22] M. Sugiyama. *Statistical Reinforcement Learning*. Chapman & Hall CRC, 2015.

- [23] I. Sutskever and V. Nair. Mimicking go experts with convolutional neural networks. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5164 LNCS.PART 2 (2008), pp. 101–110. DOI: 10.1007/978-3-540-87559-8\_11.
- [24] R. S. Sutton and A. G. Barto. Reinforcement learning : an introduction. 2016. DOI: 10.1109/TNN.1998.712192.
- [25] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: Advances in Neural Information Processing Systems 12 (1999), pp. 1057–1063. DOI: 10.1.1.37.9714.
- [26] Y. Tian and Y. Zhu. Better Computer Go Player with Neural Network and Long-term Prediction. In: Arxiv 2015 (2015), pp. 1–10. URL: <http://arxiv.org/abs/1511.06410>.
- [27] E. van der Werf, J. Uiterwijk, E. Postma, and J. van den Herik. Local move prediction in Go. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2883.2883 (2003), pp. 393 –412.
- [28] R. J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. In: Machine Learning 8 (1992), pp. 229–256.
- [29] A. L. Zobrist. Feature Extraction and Representation for Pattern Recognition and the Game of Go. In: (1970), p. 152.

# Appendix A

## Source Code

The source code of the program developed for this thesis can be found at:

<https://github.com/maburto00/ndsgo>.

The version of the program at the time of submission of this thesis includes the following elements:

- Dataset generator that generates datasets for neural network training with the required format using SGF files.
- Computer go player based on basic Reinforcement Learning techniques for 2x2 and 3x3 boards.
- Computer go player based on Supervised Learning and Deep Learning using convolutional neural networks trained on the KGS dataset (19x19) and the GoGoD dataset (9x9).
- Implementation of the Go Text Protocol (GTP) for playing against other go programs.

File	Description
board.py	Implementation of the board.
lookup_players.py	Implementation of the 2x2 and 3x3 player using RL techniques.
param_players.py	Implementation of the 9x9 and 19x19 player using SL techniques.
gtp_engine.py	Implementation of the Go Text Protocol.
cnn_models	Directory containing the code for the neural network model.
data_processing	Directory containing the code for creating the datasets.

Table A.1: Description of the most important source code files

In table A.1 we can see the description of the most important files and directories of the program. There are other files in the project but they are just utilities for this core files.