SAPIENZA
UNIVERSITÀ DI ROMA

Deep learning for Othello

OLIVAW: Mastering Othello with neither humans nor a penny

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Computer Science

Candidate

Antonio Norelli

ID number 1612487

Thesis Advisor

Prof. Alessandro Panconesi

Academic Year 2017/2018

Thesis defended on 14 January 2019
in front of a Board of Examiners composed by:

Prof. Marina Moscarini (chairman)
Prof. Chiara Petrioli
Prof. Alessandro Panconesi
Prof. Maria De Marsico
Prof. Emanuele Gabrielli
Prof. Enrico Tronci
Prof. Gianni Franceschini

Deep learning for Othello OLIVAW: Mastering Othello with neither humans nor a penny
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: noranta4@gmail.com

Ad Anna, e alla nostra comune passione per Olivaw

Contents

Chapter 1

Introduction

Learning is one of the greatest mysteries of nature, we often do not realize how prodigious it is. After all we are used to its most powerful manifestation, as humans. Think of a child learning to recognize animals, after a few examples he is able to distill the abstract concept of a lion from countless combinations of its visual representations, whether it is in front, on the side or in the dark. He brings together completely different configurations of the micro electric signals on his retina.

At a fundamental level, learning is an information processing phenomenon and as such it is investigated by computer science.

This investigation led to the birth of a prolific research field devoted to the replication of learning on computers - machine learning. In recent years a disruptive innovation revolutionized the field, the ability to train deep neural networks. This was a breakthrough in the machines learning power, with immediate applications that dramatically improved the state of the art in many domains of science, from computer vision to genomics.

A milestone in this revolution was the victory of AlphaGo against the Go world champion Lee Sedol. While previous methods in Go achieved only an amateur level, AlphaGo through deep neural networks was capable to reach a profound Go knowledge by training on masters games.

But an even bigger milestone was AlphaGo Zero one year later. In the space of few days, provided only with the rules of the game, and so starting *tabula rasa*, AlphaGo Zero was able to rediscover much of the Go knowledge accumulated by humankind over thousands of years, as well as new strategies, beating its previous version 100-0.

Nevertheless the greatest applications of deep neural networks are the prerogative of a few entities, often private, because of the huge resources needed, both from a data and hardware point of view. For instance the hardware cost of the AlphaGo Zero system has been estimated to be around $25 million. It is no coincidence that the result reached in Go by AlphaGo Zero (Google) was replicated only by ELF (Facebook).

This brings us to the goal of this thesis, to replicate the AlphaGo Zero result on a game simpler than Go but still with an interesting complexity, using only free resources available to anyone with a laptop and an internet connection.

To this end I designed Olivaw (Othello Learning In Very Anthropic Way), a software inspired by the AlphaGo Zero paper to achieve a superhuman level of play in Othello within a time frame of few days and without any knowledge except from

the rules of the game. OLIVAW has been completely executed on Colaboratory, a free cloud computing service.

Othello is one of the most well-known mind sports games in the world. There are professional players competing in official tournaments organized by world and national federations. Italy has a good tradition in Othello, it is one of the top national team after Japan. In the last ten years Italy won a world championship (Japan eight) in 2008 in Oslo with Michele Borassi, a second place as team also in Oslo and a second place in 2010 in Rome newly with Michele Borassi.

After a month-long training, OLIVAW defeated the former Italian Othello champion Alessandro Di Mattei[1].

A challenge with Michele Borassi is scheduled on 19 January 2019 in Rome.

OLIVAW has also defeated other Othello programs as Zebra or Saio at superhuman levels and analyzing orders of magnitude less positions to choose each move.

The thesis is organized as follows. In Chapter 2 it is discussed the usefulness of studying games in Artificial Intelligence research, as well as the choice of Othello for this work.

Chapter 3 summarizes the elements of a traditional game engine, with a focus on Othello. The core innovations of the AlphaGo approach are then presented - Deep Neural Networks in partnership with a Monte Carlo Tree Search.

In Chapter 4 the AlphaGo Zero algorithm is presented in detail, highlighting the differences with OLIVAW. In the last section the full algorithm is shown in pseudocode.

Chapter 5 presents the details on the training phase, with the many insights gained from the self-play games. At the beginning the hardware configuration used is briefly discussed, while the last section reports relevant games between OLIVAW and other engines.

The last Chapter 6 presents the games from the two matches against Alessandro Di Mattei, the Italian Othello champion.

The Appendix contains all the source code.

[1] Italian champion in 2016, 2017 and 2019. 10th at the World Othello championship in 2010.

Chapter 2

Games and Artificial Intelligence

When talking about a computer system, we refer to it as AI-based if it solves a problem that in our opinion requires *intelligence*. Among these problems, games may appear at the bottom of the list of priorities, yet the development of playing programs has always been at the heart of AI research. This chapter discusses the usefulness of studying games in the context of Artificial Intelligence research.

2.1 Why games?

In 1949, at the dawn of the computer age, talking about *computer routines* and putting *program* only between inverted commas, Claude Shannon presents on the *Philosophical Magazine* the prototype of a Chess playing machine in one of the first *de facto* Artificial Intelligence papers. [1]. (The term *Artificial Intelligence* will be coined only later in 1956 by John McCarthy)

Shannon's first concern was to clarify that a success in creating a strong Chess program can lead to progress in problems of greater significance. Among the presented applications there were machines capable of translating from one language to another or machines capable of orchestrating a melody. According to Shannon, chess was an ideal starting point because of the modeling simplicity and the recognized *intelligence* required for skillful play.

As pointed out by Van den Herik [2] games such as Chess, Go or Othello possess the property of creating a micro world, where real world experience and common sense knowledge are not relevant. Instead, a small set of rules determines all possible states and a clear objective makes results simply measurable. Moreover, games are designed by humans for humans, so mastering games requires all the range of skills that we would expect from our artificial counterpart, including intelligence.

In the end, citing Allis [3], when investigating sufficiently complex games with the goal of outperforming human beings, success is likely to yield new AI techniques as their products, while failure presents a better understanding of problems and obstacles encountered.

2.1.1 Abstract board games

Chess, along with other abstract board games such as Checkers, Othello, Shogi or Go, presents several convenient properties:

- **The game state is fully observable**, all the information needed to play a move is available to the players. This simplifies the representation of a game state.
- **It is two-player and zero-sum**, any gain of one player is a loss for the other. This simplifies the measurement of the goodness of a game state.
- **It is deterministic, static and discrete**, there is no chance in the outcome of an action, the game state changes only after an action and both game state and action sets are finite. This simplifies greatly the simulation of the evolution of the game.

This ease of modeling allows the use of convenient data structures like trees, which are the starting point for the construction of effective algorithms to master games (see Section 3.1).

2.2 From Chess to Go and beyond

On one hand we can consider Shannon's first article on chess machines ahead of its time, with many relevant ideas on the practical construction of such algorithms proved correct only decades later. On the other, in the 50s Shannon was not alone in his vision about the importance of investigating games. Arthur Samuel, Alan Turing, Allan Newell, Herbert Simon and others demonstrated early interest in the field.

In addition to a theoretical interest, these efforts were motivated by seeking public attention for computers in general, which were not as mainstream as today. A machine capable of skillful playing in games like Chess or Checkers, people reckoned, would have impressed the average person.

However this attention turned out to be harmful, for few realized the computational limits of the time to produce such programs, and the early days of Artificial Intelligence research were plagued by over optimistic predictions [4].

Interest in investigating games began to decline, partly because of the poor results, partly because the most promising approach to master Chess and Checkers turned out to be very close to brute force, in the sense of calculating as many future positions as possible when deciding for a move. Performance began to be measured as positions calculated per second.

After decades of slow progress, results came with Moore's law in 90s, in 1994 Chinook defeated the world Checkers champion [5], in 1997 Deep Blue won the rematch against the world Chess champion Garry Kasparov [1] [6] and few months later Michael Buro's Logistello defeated the world Othello Champion Takeshi Murakami [7].

Although these successes brought new ideas into AI research, like Logistello patterns (see Section 3.2 and Figure 3.2), they derived largely by the mere progress in computer performance that permitted the analysis of billions of positions for a single move. Kasparov said about Deep Blue:

> *I can smell that the decisions that it's making are intelligent because I would come to the same conclusion by using my intuition.*
> *But if I use* 90% *of my intuition and positional judgement and* 10% *of calculation, and Deep Blue uses* 95% *of computation and* 5% *of built in*

[1] one of the best chess players of all time, ranked number 1 in the world from 1986 until his retirement in 2005

chess knowledge, and the result matches four times out of five, maybe we should talk about some sort of artificial intelligence.

Was Deep Blue a demonstration of intelligence? This question loomed large in many comments on the match. Surely that victory led to progress, if not in AI, in the metaphysical aspect of determining what an AI is. This, too, was predicted by Shannon in 1949:

Chess is generally considered to require "thinking" for skilful play; a solution of this problem will force us either to admit the possibility of a mechanized thinking or to further restrict our concept of "thinking".

When the field of computer games seemed destined to move away from AI, Go, the oldest among the board games, brought it back into the limelight.

The huge branching factor of Go makes it not approachable by brutal tree search. Since the reachable depth is very low, such programs can play only at an amateur level. A successful computer Go player can only be based on a strong Go knowledge, that provides pattern recognition for bad and good shapes and positional judgement. It is therefore not a case that the decisive turning point came from Computer Vision with a technology that was revolutionizing the field, Deep Neural Networks (see Section 3.3.1).

Using Deep Neural Networks, the DeepMind team achieved first astonishing results in many Atari Games [8] and then built AlphaGo, the program that learning from masters games defeated Lee Sedol, one of the best Go players in history, in a 5-games match in 2016 [9]. .

One year later they announced the creation of AlphaGo Zero, another Go program based on DNNs that defeated its previous version learning completely from scratch, knowing only the rules of the game, achieving a long-standing goal in Artificial Intelligence.

70 years after the article of Shannon, a chessboard stands on the cover of the December 7, 2018 issue of Science, introducing AlphaZero, the new strongest player entity in the world of Chess, Go and Shogi at the same time [10], demonstrating that the AlphaGo Zero methods work for many games. Including Othello with "normal" computing resources, as it will be shown in this thesis.

As pointed out by Murray Campbell[2] in an editorial introducing the main article [11], this result closes the multidecade chapter of board games in AI research. Nevertheless a new era has already begun, multiplayer games with a partially observable environment and very large state and action spaces such as StarCraft II has been already proposed as new challenges [12].

2.3 Why Othello for OLIVAW

Othello is an abstract strategy board game for two players. It is a perfect information game, meaning that players have no information hidden from each other and there is no element of chance in the game mechanics.

[2] One of the creators of Deep Blue

2.3.1 Rules of the game

Othello is played on a 8×8 board of a uniform color and with a set of disks white on one side and black on the other. Before starting 4 disks are placed in the center of the board in the shape of a X. Then, players move alternately (black begins) by placing a new disk in an empty square in order to imprison one or more opponent's disks between the played disk and those of its own color already on the board. At this point the imprisoned pieces are flipped and become owned by the player who moved. It is possible to capture disks horizontally, vertically, and diagonally, and with each move, disks can be captured in one or more directions. Capture always occurs in a straight line. (see Figure 2.1).

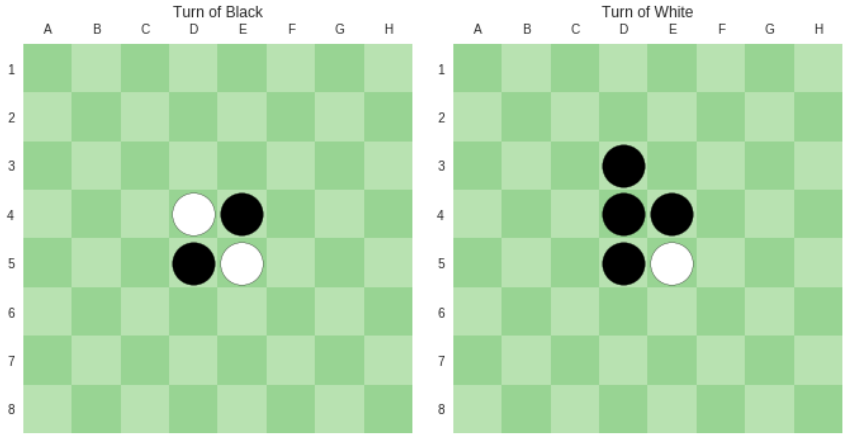


Figure 2.1. The starting position and the resulting position from one of the four possible moves of Black. Black moves D3 capturing the white disk in D4 trapped by the new disk in D3 and the one in D5.

Only moves capturing at least one disk are allowed, in the absence of moves the player skips the turn. It is not possible to pass the turn if there is at least one valid move. When none of the players can move or when (more frequently) the board is full, the player with more disks on the board wins.

2.3.2 A suitable choice

Many aspects make Othello a good choice for my purposes:

- It is a perfect information game, necessary condition to use the presented learning algorithm.
- The compact representation of a game state and the limited number of moves reduce the computational cost, making it compatible with the available hardware resources. See Section 5.1 for more details on the hardware.
- It is an interesting game. It has not yet been solved in its classic 8×8 variant and despite the simple rules it is very profound, presenting many strategies at any level of play. Therefore the final performance of the learning algorithm is significant and even the intermediate performances during the training are interesting.
- It is a well known game, played all over the world. There are professional players competing in official tournaments organized by world and national

federations, as well as strong programs that at full strength do not grant more than a draw. This is fundamental to evaluate the strength of Olivaw.

2.4 Concerning humans... Why games?

After answering why we study games in AI, let us ask ourselves *why we play?* The answer is not trivial, playing a game is a fundamental and complex phenomenon.

The study of the strategies used by rational players involved many Nobel Prizes and has become a prolific research field in economics and mathematics - game theory, founded by John von Neumann with the demonstration of the minimax theorem.

Even some animals play, spending time and energy on something not strictly related to surviving or breeding. Playing is fun, but from the point of view of natural selection fun is costly, what is the rest of the story?

Despite a complete answer to this mystery does not yet exist, observations show that play-deprived individuals suffer in real world situations. (see [13] for an experiment with rats and other references involving primates and humans).

Games are laboratories to tackle real world problems, it is not a coincidence that young individuals play more. If we want a computer program to do the same, why not use this system refined over the ages by nature?

Chapter 3

How an Othello engine works

This chapter discusses more in detail the methods used by a computer game program, taking Othello as example. It is presented the standard algorithm used by traditional programs and then the innovations of the AlphaGo approach are introduced.

3.1 Games as trees

Any sequential perfect information game can be fully represented in a convenient data structure, a tree.

Chosen a game of this class, such as Tic-Tac-Toe, Connect 4, Checkers, Othello, Chess or Go, set the starting position as root of the tree. Every possible move from the starting position leads to a new position, represent those as a branch from the root. Then for every new position repeat recursively the process and attach all the branches representing next positions to the starting branch.

If the game finishes in finitely many moves, as Othello and the classic ones cited above, this recursion has an end. Every ramification ends in a leaf, a branch without children, representing a final position that can not be further expanded, e.g. a position with no legal moves for both players in Othello or a checkmate position in chess.

This construction grasps all the complexity of the game, representing every possible position. Given such a tree it is easy to find a perfect strategy of play. Considering one player, it is possible to label each leaf position as a win or a defeat (or a draw, if the game admits it). Having those labels, even branch positions that conduct directly to leaves can be labeled as a win or a defeat, backpropagating the best option for the player of that turn, and so on until the root. This method is called minimax algorithm, a simple example is shown in Figure 3.1.

The problem with this approach is that it can take an impractical amount of time and memory. Games with many possible moves from each position result in trees with a high branching factor, this makes the tree size intractable even after few moves from the root.

For these games we have to give up the search for the perfect strategy and focus our efforts on finding a good approximation.

The quality of this approximation depends on two key elements:

- **The tree search algorithm**. Since we can explore only a limited number of branches, it is crucial to focus the energies on the more significant ones. i.e. favoring the expansion of promising moves.

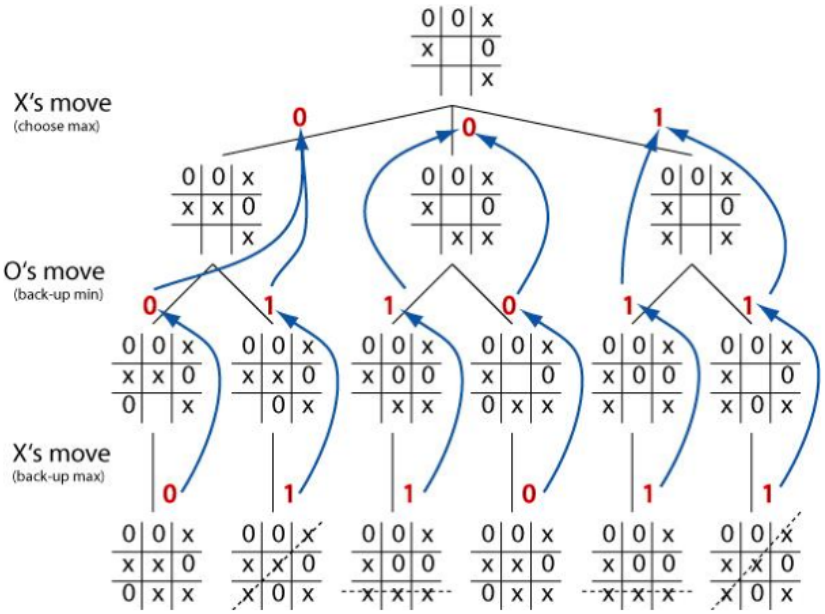


Figure 3.1. A final portion of the Tic-Tac-Toe tree. Using the minimax algorithm the position on the top can be labeled as a win for X. *Taken from [14]*

- **The evaluation function**. Unless we are in the final stages of a game, most of the times our tree search must stop before reaching a final position. How do we label these pseudo-leaves to backpropagate their values? We need a good horacle callable on the fly to judge if a position is promising or not.

3.2 Anatomy of a traditional computer Othello program

In this section the components of a typical computer Othello program are described more in detail, focusing specifically on Saio [15], one of the strongest Othello engines in the world.

Concerning the construction of the evaluation function, each position s is described by a sparse vector of handcrafted features $\phi(s)$. First programs encoded as feature concepts derived from human Othello knowledge, like the maximization of mobility or the possession of a corner. A step forward was made with the introduction of *patterns* as features by the program Logistello [7], meaning with pattern a configuration of a specific portion of the board selected by hand, see Figure 3.2.

A weight w_i is assigned to each feature ϕ_i through a manual or an automatic tuning which, in the case of Logistello and Saio, consists of a logistic regression on a dataset derived from human games played by masters.

So, the evaluation function is a linear combination of the features $v(s, w) = \sum_i \phi_i(s)\, w_i$.

In Saio a further level of complexity is introduced using 60 different evaluation functions, one for each possible number of empty spaces on the board. The tuning of all weights required millions of games.

Having this horacle, the final evaluation of a position s is computed through a limited search on the tree, that uses Alpha-beta pruning to safely cut any branch that is provably dominated by another variant. Alpha-Beta search algorithm is an improvement on naïve minimax search, in the best case scenario drops the cost from

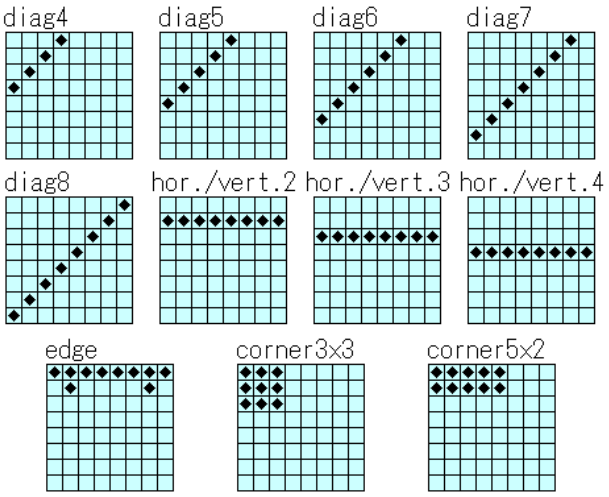


Figure 3.2. Patterns introduced by Logistello. Every possible configuration of each pattern is used as a feature. Those patterns are close to the idea of filters in modern convolutional neural networks.

$O(b^d)$ to $O(\sqrt{b^d})$, where b is the branching factor and d is the search depth. See figure 3.3 for an overview of the Alpha-Beta algorithm.

Using these techniques on a standard machine with few seconds of computing time, Saio is capable to build a tree with a depth of 14 to evaluate a move. To go deeper maintaining a computational time of seconds, additional lossy cuts based on principal variation search and probabilistic considerations are used, reaching a depth of 26.

When the end of the game is close enough, the horacle is not necessary anymore and the program plays a perfect final expanding the tree until the end.

On top of that, modern programs use a carefully tuned library of positions to select moves in the early game, but also in midgame and endgame for principal variations. Saio's library is the largest among Othello engines and counts more than 180 millions of positions collected in more than 15 years, with 240.443 complete draw lines from the start position to the end of the game. i.e. Saio can play an entire game without making any calculation.

None of the described techniques are used by Olivaw, that derives its play policy completely from scratch and analyzes orders of magnitude less positions to reach a superhuman level of play.

3.3 AlphaGo revolution

With the exception of Go, the approach described in the last section was the state of the art in perfect information game programs until 2016. Any famous game engine was based roughly on these ideas, from Logistello to Saio in Othello, from Deep Blue to Stockfish in Chess, or Elmo in Shogi.

In the first AlphaGo paper published in 2016 [9], Silver et al. proposed the usage of a deep neural network trained on a database based on expert moves as evaluation function paired with a Monte Carlo tree search algorithm.

AlphaGo demonstrated a superhuman level of play in Go beating Lee Sedol, one of the strongest players in history. This result was a milestone in AI research.

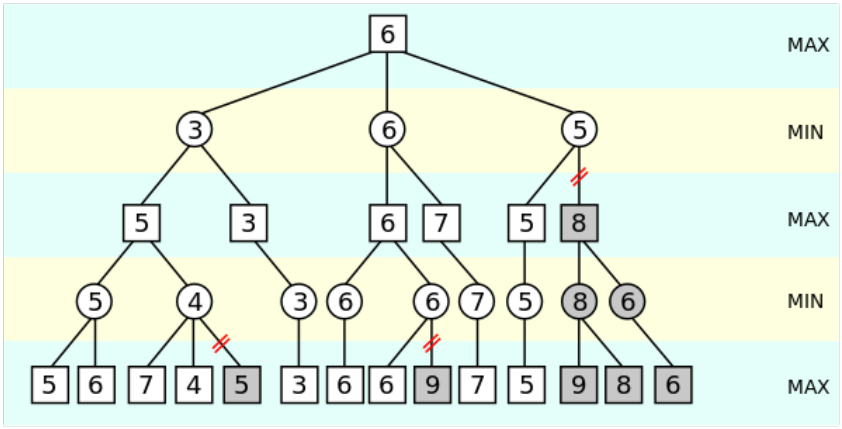


Figure 3.3. An illustration of alpha–beta pruning. The grayed-out subtrees need not be explored (when moves are evaluated from left to right), since we know the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The max and min levels represent the turn of the player and the adversary, respectively. *Taken from Alpha-Beta pruning entry on Wikipedia.*

Since Kasparov's defeat versus Deep Blue, Go had become the new benchmark in board games, the last goal to be overcome. The traditional approach has never gone beyond an amateur level in Go, because of the huge tree, consequence of the large branching factor.

One year later was released AlphaGo Zero, an even stronger program with the same elements of AlphaGo and a different learning algorithm that uses only self-play games, without any external knowledge different from the rules. This further result marked another great step forward in AI, a system that learns, tabula rasa, superhuman proficiency in a challenging domain.

These achievements inspired many subsequent works in AI research, starting with a whole new generation of game engines. The idea of Olivaw comes from here.

The key elements of the AlphaGo approach, deep neural networks and Monte Carlo tree search, are presented and discussed in this section, with a specific focus on their application to an Othello engine.

The learning algorithm of AlphaGo Zero is presented and discussed in the next chapter, highlighting the differences introduced in Olivaw.

3.3.1 Deep Neural Networks

AlphaGo uses a deep neural network f_θ with parameters θ as evaluation function. This section presents briefly what is a deep neural network and why it is a good choice for an Othello evaluation function.

Deep learning

Machine learning is the study of algorithms that allow computer systems to make predictions or decisions through models inferred from data, and so without being explicitly programmed to perform a task.

Such data-based approach is the standard of any modern AI system. The availability of data in the digital age makes possible the usage of machine learning algorithms in many areas, introducing new possibility of innovation and optimization.

In recent years a class of these algorithms has made a huge step forward in the recognition of complex patterns, and so in predicting power, improving dramatically the state of the art in computer vision, natural language processing, speech recognition and many other domains such as drug discovery or advertising. This breakthrough has led to the emergence of a new field of study, deep learning, focused specifically on this class of algorithms, deep neural networks (DNNs).

Today, the key problem in solving a detection or classification task is identifying useful features in raw data. The success of DNNs over conventional machine learning techniques derives from their ability to extract those complex features automatically, so without the help of slow, expensive and often scarce human experts of the specific domain, but using only a large amount of data.

Deep neural networks

A DNN is a collection of connected neurons organized in multiple sequential layers. Each neuron produces a non-linear response of a given input signal. *n input neurons* receive an external signal that is propagated through the network according to connection weights. The state of the network can be measured looking at the state of *m output neurons*. Mathematically speaking, a DNN is nothing that a function that maps a space of n dimensions to a space of m dimensions. (Figure 3.5 **a**)

For instance a DNN can map a 784-dimensional vector representing the image of an handwritten digit to a one-hot 10-dimensional vector representing numbers from 0 to 9 (Figure 3.4). This mapping depends on the connection weights, that are adjusted using labeled data during a *training* process. (Figure 3.5 **b, c, d**).

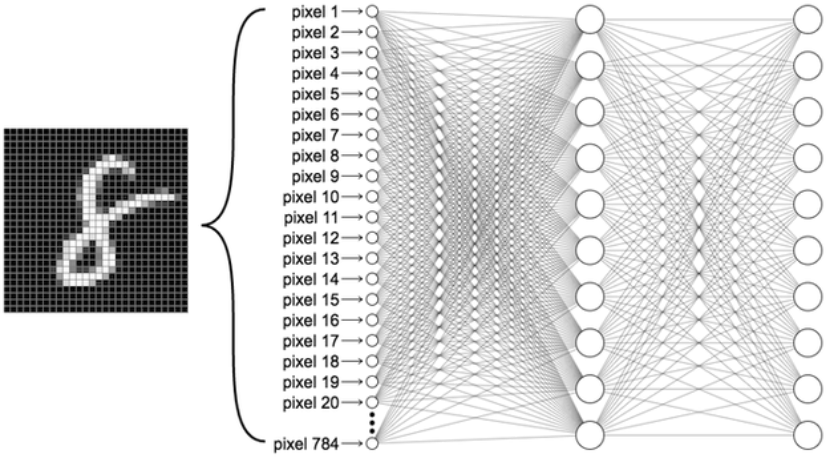


Figure 3.4. Architecture of a DNN with one hidden layer to recognize handwritten digits. The "8" image is taken from the MNIST database, that contains 70000 labeled 28×28 images.

A deep neural network as evaluation function

Having a suitable database of games, using a deep neural network f_θ as evaluation function provides some advantages:

- f_θ **can output a complex non-linear transformation of its input**. Such a function is much more expressive than the linear $v(s, w) = \sum_i \phi_i(s)\, w_i$ of traditional programs presented above, in other words it can grasp many

more details about the input position. Moreover f_θ is a sequence of non-linear combinations, making it capable of recognizing patterns at different abstraction levels.

This expressiveness is crucial when dealing with complex data like Othello positions, where there are no clear and simply identifiable features. The simple possession of a corner or the maximization of mobility are principles that lose early their absoluteness.

Moreover an Othello position often have many hot points, and the best move may derive from the correct identification of the most urgent situation rather than solving the simple local combination. e.g. when you have to give up a corner, it is important to give up the corner correctly, but the big question is, which corner should I give? This is why an evaluation function capable of recognizing patterns at different abstraction level is important. Note that this is impossible with a linear evaluation function, where any further stage is pointless. Suppose to define a new evaluation function as $\tilde{v}(s, w', w'') = \sum_j \sum_i \phi_i(s)\, w'_i\, w''_j$, then $\sum_j \sum_i \phi_i(s)\, w'_i\, w''_j = \sum_i \phi_i(s)\, \tilde{w}_i = v(s, \tilde{w})$ without any expressiveness gain.

- In addition to the goodness of the position, f_θ **can naturally output a multidimensional response representing the evaluation of every single possible move**. Such an output may seem redundant, but it provides several benefits.

  It seems redundant because an evaluation on every move can be simply obtained with a function valuating only the goodness of a position, it is sufficient to call that function on the positions resulting from every possible move. Nevertheless, this construction points out the first benefit, the calls to the neural network are reduced by a factor of b, where b is the branching factor.

  Another consideration comes from the architecture of the neural network. Since most of the trainable parameters are common to the two outputs, this choice increases the size of the training data to tune these parameters. Having more meaningful data is always a good news when training deep neural networks.

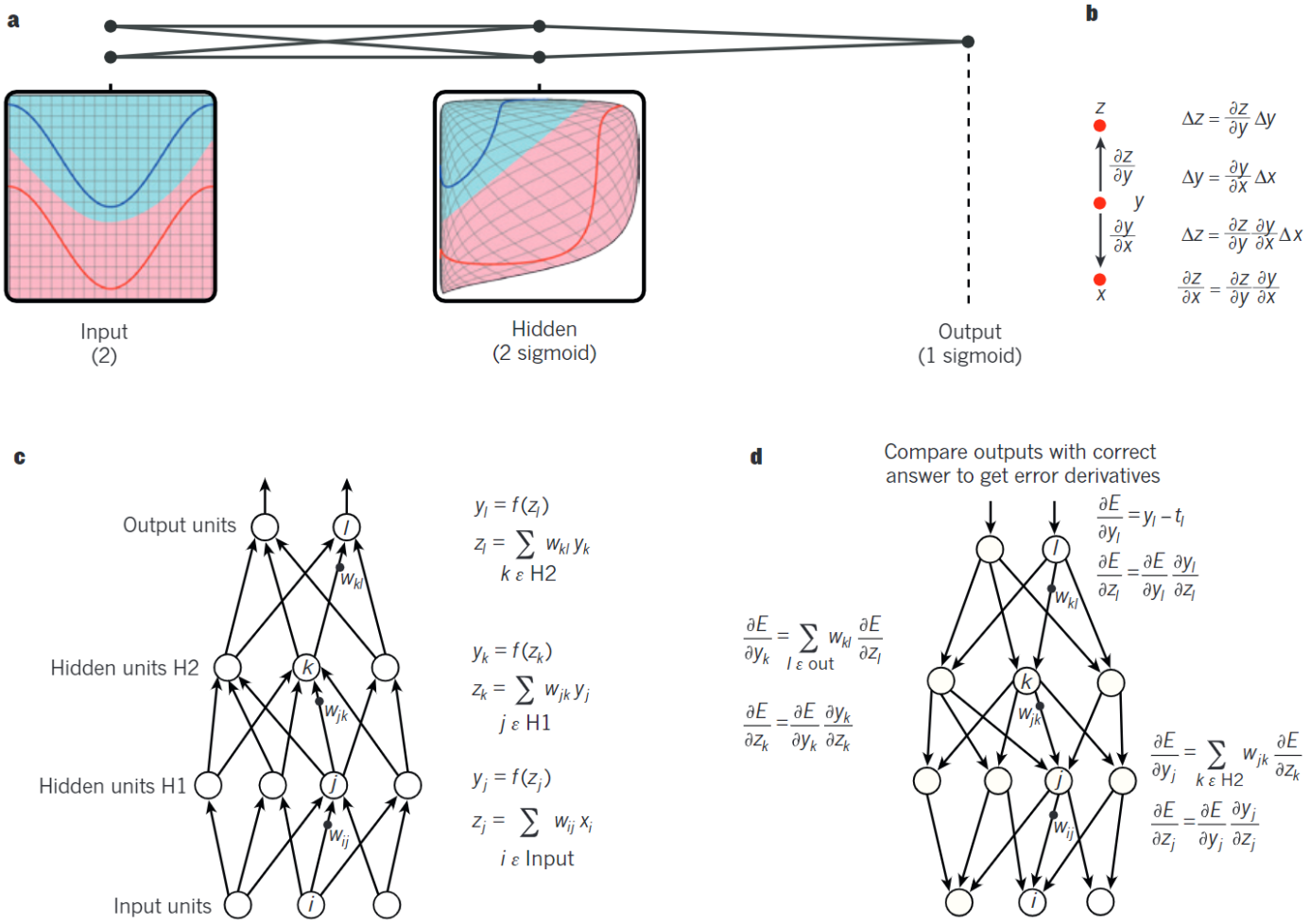


Figure 3.5. Deep Neural Network essentials

a, Consider the problem of recognizing two entangled classes of data, red and blue. Hidden layers of a DNN distort progressively the input space to make the classes linearly separable by the last layer. Visit playground.tensorflow.org to experience this distortion interactively.

b, The chain rule of derivatives is the basis of the training process. It describes how changes on chains of dependent variables are composed. If $z \to y \to x$ where $\to$ stays for *is function of*, then a small chenge Δx on x affects z by a $\Delta z = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x$.

c, The equations used for computing the output $\hat{y}$ of a neural network. Those are used during the training process to compute the error $E(\hat{y}, y)$ respect to the ground truth y and then to make predictions once the training is complete. The response of each neuron is the result of a non-linear function $f()$ applied on a weighted sum of its input $z_j = \sum_{i \in input} w_{ij} x_i$. Common non-linear functions are ReLU $f(z) = max(0, z)$, hyperbolic tangent $f(z) = \tanh(z)$ or sygmoid $f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$.

d, The equations used for computing the partial derivatives of E at each layer of the neural network using the chain rule. The goal is to minimize E adjusting the weights w_{ij}. Those equations are necessary to obtain the decreasing direction of E respect to each weight w_{ij}, given by $-\frac{\partial E}{\partial w_{ij}}$. Once the decreasing direction of each weight is known, the weights are updated using $w_{ij} = w_{ij} - \alpha\frac{\partial E}{\partial w_{ij}}$. This is a step of *gradient descent*, the optimization method used to minimize the error E. The algorithm used to compute all the partial derivatives is called *backpropagation*, since derivatives are computed from the last to the first layer. α is the learning rate, a parameter controlling the step length.

Image taken from [16].

3.3.2 Monte Carlo Tree Search

AlphaGo uses a Monte Carlo Tree Search (MCTS) as tree search algorithm. This section presents a brief explanation of the algorithm and why it is a good partner of the f_θ evaluation function.

The search algorithm

In a nutshell, a MCTS performs an asymmetric search (and therefore a lossy search, differently from Alpha-Beta) using random extractions.

The search tree is built repeating the following cycle composed of four steps:

1. **Selection**, a leaf node L of the tree is selected. Starting from the root, recursively a child is selected until a leaf node is reached. There are many policies to choose a child s, generally it is chosen the node that maximizes a potential $U(s) \propto \frac{v(s)}{1+n_{visits}(s)}$ that balances exploration and exploitation.
2. **Expansion**, if L is not a terminal node, i.e. it does not end the game, L's children are added to the tree
3. **Simulation**, L is evaluated. The value can come directly from an evaluation function $v(S)$ or it can be the result of a playout, a random simulation from L until a result is reached.
4. **Backpropagation**, the potential of all L's ancestors is updated considering the new value of L.

An illustration of the cycle is showed in Figure 3.6.

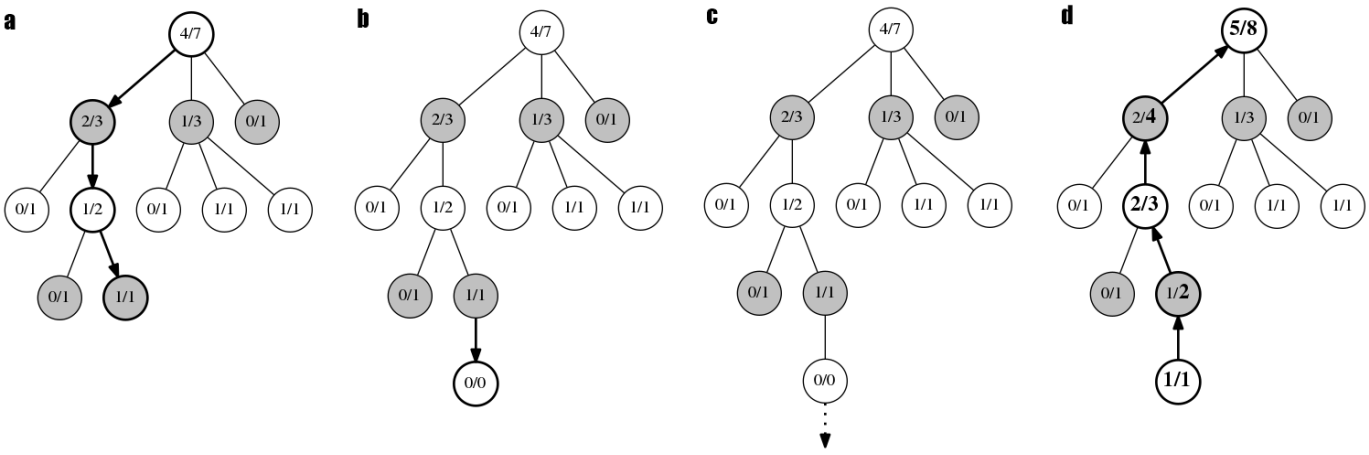


Figure 3.6. Steps of Monte Carlo tree search. Each node is tagged with its potential.
a, Selection. **b**, Expansion. **c**, Simulation. **d**, Backpropagation.

MCTS vs Alpha-Beta with f_θ

Alpha-Beta is the standard algorithm for traditional programs like Saio for Othello or Stockfish for chess. It performs a complete symmetric search in the tree, safely pruning branches of variants proven not optimal. If the branching factor is

sufficiently small, the tree search can go deep enough to reach a very high level of play, even with a linear evaluation function.

f_θ is a stronger evaluation function based on a DNN, able to perform non-linear evaluations on multiple levels of abstraction. However, despite each heuristic-based search algorithm benefits from a better one, there are two drawbacks of using f_θ with an Alpha-Beta search:

- A single call of f_θ is more expensive than a standard linear evaluation function. At the same computational cost, a full search with only Alpha-Beta pruning leads to a less deep tree. The trade-off between depth and evaluation power is not always for the second one, e.g. during the final phase of a game.
- f_θ's power in the evaluation of patterns derives from the large multi-layered architecture of the neural network, which is able to encode a highly complex non-linear model. The more complex the model, the more data are needed to train it. Unseen situations may, in fact, introduce large generalization errors in the evaluation, due to an overfitting in the unexplored region of data. Alpha-Beta computes an explicit minimax, and may propagate such errors directly to the root of the corresponding subtree. See Figure 3.7.

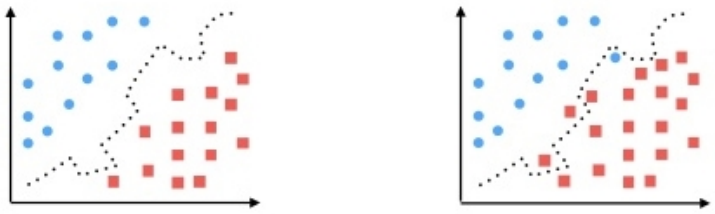


Figure 3.7. Example of a generalization error in a classification task. A complex model is trained on the data showed in the first image, the lack of data in the central region leads to many generalization errors when the model is tested on those points in the second image.

Monte Carlo Tree Search is less prone to these problems. It takes more advantage from a better evaluation function that conducts the asymmetric search in depth into the best variants. Concerning the generalization errors, MCTS averages over the node evaluations within a subtree instead of backpropagating the value of the best leaf. In the evaluation of a large subtree, this means averaging many errors, which tend to compensate, instead of using only the largest one.

Chapter 4

AlphaGo Zero algorithm

Olivaw is an Othello engine based on the methods presented in the AlphaGo Zero [17] paper, with some variations to optimize the learning time, partly inspired by a Connect-4 implementation of the AZ algorithm [18].

This chapter presents the AlphaGo Zero algorithm in detail, highlighting the differences introduced in Olivaw.

4.1 The intuition

The ultimate goal is to build an agent that takes in input a position and outputs a move, so we start by discussing the representation of a position and a move.

- A move can be unequivocally represented by a one hot encoded vector sized as the action space of the game, that is for instance $19 \cdot 19 + 1 = 362$ for Go or $8 \cdot 8 + 1 = 65$ for Othello. We will call a move a.
- The representation of a position is less trivial. We can not simply encode the position of the board, since it does not contain all necessary information to make a move. Whose turn is it? In chess, has the king been moved and so castling is not possible? In Go, has a K.O. just been played? We should encode all these informations, i.e. we want a *markovian* representation of a game state. The three-dimensional tensor used to represent a game state in AlphaGo Zero is showed in Figure 4.1. We will call a game state s.

AlphaGo Zero uses a deep neural network f_θ with parameters θ. This DNN takes as input a game state s and outputs both move probabilities and a value, $f_\theta(s) = (\mathbf{p}, v)$.

Given a certain game state s, $\mathbf{p}$ represents the judgement of the neural network on each move normalized to sum 1, we can interpret it as a probability distribution, $p_a = Pr(a|s)$. The value $v \in [-1, 1]$ represents the evaluation of the position according to the neural network, $+1$ a certain victory, -1 a certain defeat. The architecture and other details on the neural network are discussed in the section **??**.

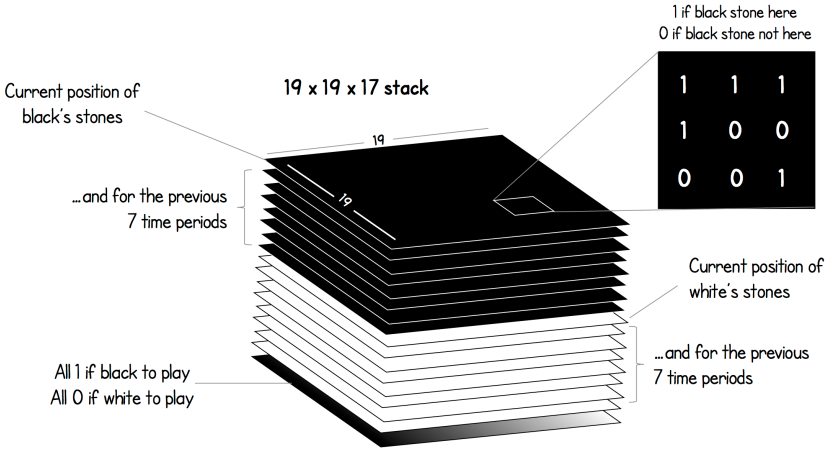


Figure 4.1. Three-dimensional tensor used to represent a game state in AlphaGo Zero. Past positions of the board are necessary to keep track of K.O.s. This is the input of the neural network f_θ. *Image taken from [19]*

Note that the single neural network is already a full playing agent, given a game state s we can select as move the one corresponding to the highest entry in $\mathbf{p}$. Its strength depends on the level of play in the games used as training data, but how can we improve its strength without using external knowledge?

Suppose that the level of play of the pure neural network agent A_0 is L, we need a generator of games at a level $L' > L$ to retrain f_θ and obtain a stronger f'_θ. In other words we need another agent A, stronger than the shallow neural network but that uses no more information than the neural network and the game rules.

Here comes into play the Monte Carlo tree search. Given a state s, A performs a MCTS search guided by the neural network f_θ. This search outputs probabilities $\boldsymbol{\pi}$ of playing each move. The moves selected by $\boldsymbol{\pi}$ typically are much stronger than the ones of $\mathbf{p}$. In a nutshell, this is because the MCTS search averages over several evaluations of the neural network on positions of probable lines of play from s.

Agent A is therefore used to generate self-play games, each game state s of those games is labeled with the move probabilities $\boldsymbol{\pi}$ resulting from the MCTS search and the final winner of the game z.

The old neural network f_θ is then trained further on this data, obtaining a new configuration f'_θ. Such update makes the move probabilities and value $(\mathbf{p}, v) = f'_\theta(s)$ closer to the improved search probabilities and self-play winner $(\boldsymbol{\pi}, z)$.

This process can be repeated to obtain every time a new neural network and a stronger agent, this is the essence of the reinforcement learning algorithm used in AlphaGo Zero.

4.2 The algorithm in detail

In this section we get into the details of the reinforcement learning algorithm used in AlphaGo Zero. The training pipeline consists in the repetition of three stages, executed sequentially or in parallel.

- **Self-play**. A new training set is created.

- **Training of the neural network**. The current neural network is retrained on the new data.
- **Evaluation**. The new neural network is tested against the old one.

4.2.1 Self-play

This stage consists in the generation of games by the best current agent $A(f_\theta)$.
We are going to see the selection of a single move through the MCTS search and then the general details about the games generation.

Selecting a move using the MCTS

Given an initial game state s_0, we are going to build a tree with s_0 as root through the MCTS search.

Each edge of the tree (s, a) represents a legal action from the starting node s and stores a set of statistics.

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\} \tag{4.1}$$

where $N(s,a)$ is the number of visits of the edge during the search, $W(s,a)$ is the total action value measured in the search, $Q(s,a) = W(s,a)/N(s,a)$ is the action value normalized on the number of visits and $P(s,a)$ is the prior probability of selecting that edge, i.e. a first estimate of the action value given by the neural network.

The search algorithm proceeds by executing a series of simulations, each simulation consists of the classical four steps of a Monte Carlo tree search, presented in section 3.3.2. Here we will focus on the details of the implementation of the algorithm used in AlphaGo Zero.

1. **Selection**, a leaf node s_L of the tree is selected. Starting from the root s_0, recursively a child is selected until a leaf node is reached. The child chosen is the one corresponding to the edge that maximizes the quantity $Q(s,a)+U(s,a)$. U is defined as:

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{4.2}$$

   Where b is any legal move from s and c_{puct} is a constant parameter. This search control strategy shifts dynamically with the visit count N from the preference of nodes with high prior probability P and low visit count (exploration) to nodes with high action value (exploitation). The c_{puct} parameter offers additional control on the exploration threshold.

2. **Expansion**, if s_L is not a terminal node, i.e. it does not end the game, s_L's children are added to the tree.

3. **Evaluation (Simulation)**, if s_L is a terminal node, it is evaluated with the game score, $v = \{+1, 0, -1\}$ for a victory, a draw or a defeat of the current player.

   If s_L is not a terminal node, s_L and its children are evaluated by the neural network $f_\theta(s_L) = (\mathbf{p}, v)$. Each edge to a child is initialized to $\{N(s_L,a) = 0, W(s_L,a) = 0, Q(s_L,a) = 0, P(s_L,a) = p_a\}$. The value v is backpropagated to the root.

4. **Backpropagation**, the statistics of the edges crossed in the descent to s_L are updated considering the new value v of s_L. For each of those edges (s, a), the visit count is incremented $N(s, a) = N(s, a) + 1$, as well as the total action value $W(s, a) = W(s, a) + v$ and finally is updated the mean action value $Q(s, a) = W(s, a)/N(s, a)$.

Implementation note. Since a call of the neural network is computationally expensive but several states can be evaluated in a single call (see Figure 4.2), the presented search policy can be extremely accelerated with a small variation of the algorithm known as *virtual losses*. The goal is to call the neural network horacle f_θ on multiple states at once. When a leaf state s_L is encountered, it is added to a stack instead of performing immediately the evaluation. Since the algorithm is deterministic we should change the statistics of the tree nodes to not always get the same leaf s_L. So, after the insertion in the stack, the backpropagation step is executed but it updates only the visit count N. When the stack reaches a certain size, it is fed to the neural network horacle and all the predictions are used to complete the statistics updates on the leaf's children and ancestors.

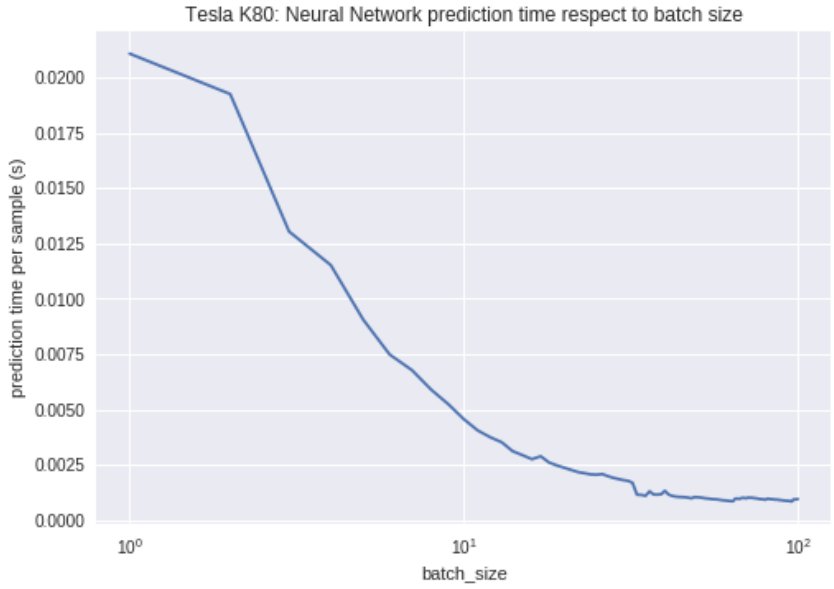


Figure 4.2. Prediction time per sample vs batch size on the OLIVAW neural network. The GPU used is a Nvidia Tesla K80.

At the end of the search we remain with an asymmetric tree with a set of statistics on each edge. We select the move to play in the root position s_0 extracting at random a legal move according to the following probability distribution

$$\pi(a|s_0) = \frac{N(s_0, a)^{1/\tau}}{\sum_b N(s_0, b)^{1/\tau}} \tag{4.3}$$

where τ is a temperature parameter, the probability of selecting the most visited action increases asymptotically when the temperature goes to 0, $\lim_{\tau \to 0} \pi(a|s_0) = \text{argmax}_a(N(s_0, a))$, while the probability distribution asymptotically flattens on an uniform one when the temperature goes to ininity $\lim_{\tau \to \infty} \pi(a|s_0) = 1/B$, where B is the number of legal moves.

Once a move is selected, the corresponding child becomes the new root node, the subtree below is retained with all the statistics while the remainder of the tree is discarded. This process is repeated until the end of the game. All the positions s encountered in the game are then matched to the final winner z along with the move probabilities used to decide the move $\boldsymbol{\pi}$ (temperature $\tau = 1$), finally the sample $\{s, (\boldsymbol{\pi}, z)\}$ is added to the training set.

OLIVAW *difference*. To speed up the learning, OLIVAW saves not only the positions encountered in the game but even positions much visited during the MCTS search but not played. This choice has two benefits:

- **At the same computational cost the average number of positions collected in a game is higher**. With a visit threshold of 50 and a MCTS performing 400 simulations per move, the positions collected grow approximately by 100%.
- **Those states are matched with their Q value instead of the final outcome z**. This fact might seem disadvantageous, z encodes the true expected result while Q is only an estimate based on a certain number of simulations. However z has a big drawback, in a game with multiple errors an evaluation of an early position based on the final outcome is extremely random, while Q offers a better approximation. On the other hand Q suffers from the limited horizon of the simulations, an early position with positive or negative consequences far ahead in the game may not be properly evaluated.

  Abrams et al. [18] show in some experiments on Connect-4 that while the learning can be completed both using Q and z, it is accelerated when using a combination of the two (even the simple average).

  In OLIVAW instead of computing the explicit average, we are collecting both states with z and states with q, exploiting states not played but sufficiently visited during the MCTS search, killing two birds with one stone.

General details about the games generation

In each iteration A plays a thousands of games against itself (25000 in AlphaGo Zero, ~ 2500 in OLIVAW). Each move is selected after several MCTS simulations from the current game state s (1600 simulations in AlphaGo Zero, 100, 200 or 400 in OLIVAW). The first moves of each game are selected using a temperature of 1 to favor exploration, the remaining moves are selected using a temperature $\tau \to 0$, i.e. the best move is always selected.

Dirichlet noise is added to the prior probabilities in the root node $P(s_0, a) = (1-\epsilon)p_a + \epsilon x_a$, where $\boldsymbol{x}$ is a point sampled from the Dirichlet probability distribution X_α ($\alpha = 0.03$ in AlphaGo, $\alpha = \min(1, 10/B)$ in OLIVAW, where B is the number of legal moves. $\epsilon = 0.25$ in both programs).

The Dirichlet distribution is a family of probability distributions $X_{\boldsymbol{\alpha}} = (X_1, ..., X_k)$ parametrized by a vector $\boldsymbol{\alpha}$ of positive reals. The higher value of α_i the greater the weight assigned to that component X_i. If $\alpha_1 = \alpha_2 = ... = \alpha_k = \alpha$ the distribution is symmetric, X_α. If $\alpha < 1$ the *mass* is pushed towards the extremes, if $\alpha > 1$ it is pushed around a central value. X_1 is the uniform distribution, see Figure 4.3 to see some examples.

The Dirichlet noise with an $\alpha < 1$ tends to unbalance the move probability distribution $\pi(a|s_0)$ towards a specific action, favoring a depth exploration of the

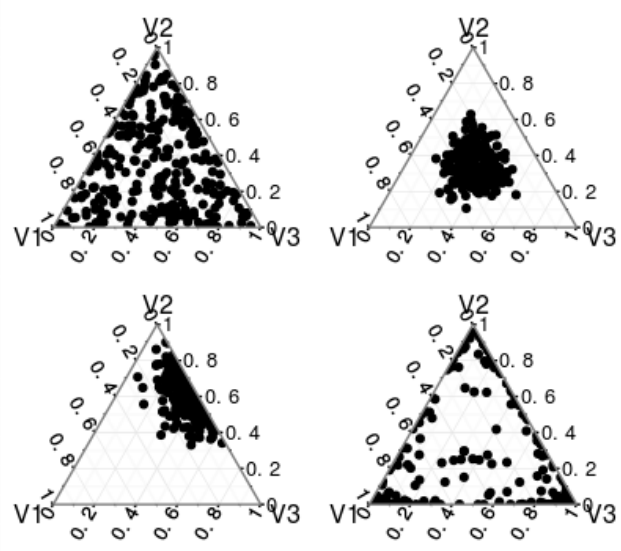


Figure 4.3. Trivariate Dirichlet distributions parameterized by:
a. $\alpha_1 = \alpha_2 = \alpha_3 = 1$. (Uniform distribution)
b. $\alpha_1 = \alpha_2 = \alpha_3 = 10$.
c. $\alpha_1 = 1;\ \alpha_2 = 10;\ \alpha_3 = 5$.
d. $\alpha_1 = \alpha_2 = \alpha_3 = 0.2$.
Image taken from this answer on Cross Validated (Statistics section of Stack Exchange)

subsequent variant. In games with high branching factor, this type of noise preserve the asymmetrc nature of the MCTS search, without excessively affecting the depth reached. The higher the branching factor of the game, the lower the α parameter of the Dirichlet noise used, as pointed out in the AlphaZero paper that presents the implementation of a similar algorithm for Chess, Shogi and Go at the same time. [10].

Furthermore during self-play not all games are played until the end to save computation. If during a game a player values a position under a resignation threshold v_{resign}, the game ends and the opponent is considered the winner. v_{resign} is chosen automatically playing a fraction of the games until the end, so that less than 5% of those games could have be won if the player had not resigned. (AlphaGo Zero plays 10% of the games until the end, Olivaw starts with 10% and increases progressively this fraction from the 16th generation to improve its strength in finals).

4.2.2 Training of the neural network

This stage consists in the retraining of the neural network f_θ on the new data generated by the self-play games.

We are going to see the details about the training, a brief introduction to residual networks and finally the specific architecture of the residual network used in AlphaGo Zero and Olivaw.

DNN training details

The end product of the self-play stage is a training set composed by entries $\{s, (\boldsymbol{\pi}, z)\}$, where s is a game state, while $\boldsymbol{\pi}$ are the move probabilities and z is the final winner of the game that generated the game state.

In order to increase the size of the training data, we perform a *data augmentation* process that exploit the symmetries of the game before feeding the data to the neural network. Both in Go and Othello we can exploit 8 possible symmetries for each position, given by the 4 possible rotation of the board plus the reflection.

Then the neural network f_θ is trained on the games generated by the most recent self-play executions. Games of old generations are used to reduce the possibility to forget useful information. AlphaGo Zero uses as training set a sample of 2.048.000 positions from the last 500.000 games, so drawing from the games generated by the last 20 generations. Olivaw uses a sample of 16.384.000 positions from the games generated by the last 2 to 5 generations, the shift from 2 to 5 is progressive with generations to accelerate the learning process as pointed out in [18], in this way games played by the first weak generations are excluded quickly.

Neural network parameters θ are optimised using gradient descent with momentum and learning rate annealing, to slow down the learning rate with the pregress of generations (starting from a learning rate of 0.003, Olivaw switched to 0.001 in the fourth training and to 0.0001 in the eleventh training). Remembering that $f_\theta(s) = (\boldsymbol{p}, v)$, the loss function is a combination of mean squared error for the value and of cross-entropy for the move probabilities:

$$L(\boldsymbol{\pi}, z, \boldsymbol{p}, v) = (z - v)^2 - \boldsymbol{\pi}^T \log \boldsymbol{p} + c||\theta||^2 \tag{4.4}$$

Where the last term is the L2 regularization that penalize large weights in the neural network. Regularization is used to prevent overfitting and to keep the neural network elastic, since we are performing many training phases.

Residual Networks

AlphaGo Zero and Olivaw use a very deep convolutional network architecture, known as residual network introduced in 2015 by He et al. [20].

In recent years state-of-the-art network architectures have become deeper, reaching even hundreds of layer. This is not so surprising, neural networks are an old idea from the 1950s but only in the last twenty years with the adding of multiple hidden layers their usage has become mainstream in many fields.

As a matter of fact, the deeper the neural network, the more complex the function represented and the higher the levels of abstraction to learn features. However not all that glitters is gold. The training of very deep neural networks can be extremely slow due to vanishing gradients. The gradient signal from the output layer exponentially goes to zero during backpropagation, meaning that weights of the first layers are tuned very slowly. See figure 4.4

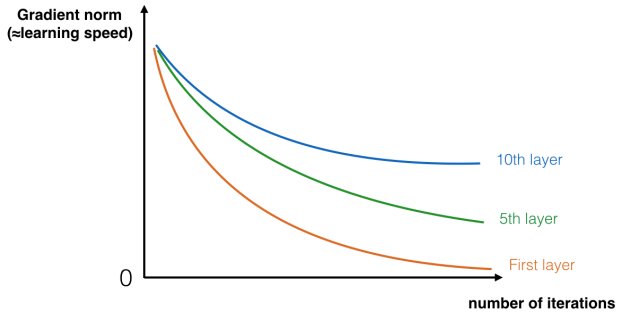


Figure 4.4. Vanishing gradients, learning speed decreases very quickly witht the layer distance from the output. *Image taken from Coursera course on Convolutional Neural Networks*

The intuituion behind residual neural networks is the insertion of shortcuts through layers to provide the gradient signal directly to first layers during backpropagation. Another benefit of the shortcut in a residual block is the ease of learning the identity function, i.e. returning in output the same data received in input, this behaviour makes the addition of a further block less harmful when the model to learn is simpler. A prototype of residual block is shown in figure 4.5

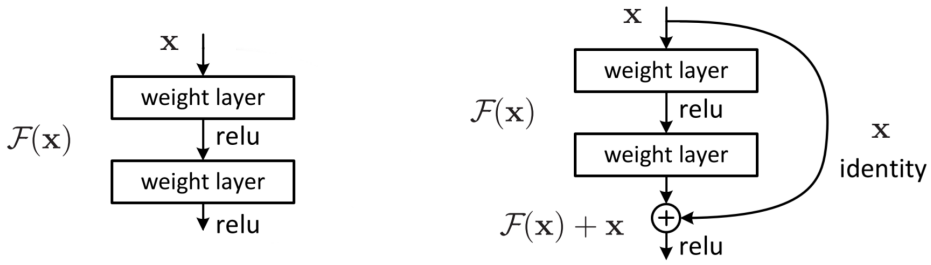


Figure 4.5. Standard block and residual block showing a skip-connection

Neural Network architecture

The game state input s is a $19 \times 19 \times 17$ stack in AlphaGo Zero (as showed in Figure 4.1) and a $8 \times 8 \times 2$ stack in Olivaw, we do not need layers that represent past positions in Othello since the game state is fully observable from the board position, even the information on the player's turn is not necessary since Othello is symmetrical respect to the player, we can assume that in each position is always white to move, flipping all the discs if it is the turn of black.

The input s is processed by a single convolutional block and then a residual tower of 39 residual blocks in Alphago Zero (the strongest version) and 10 residual blocks in Olivaw. The output of the residual tower is then processed by the value head and the policy head that output respectively the value v of the position and the move probabilities $\boldsymbol{p}$. An overview of the full alphaGo Zero network is showed in Figure 4.6. The structure of each block is:

- **Convolutional block**. AlphaGo Zero (Olivaw)
  1. A convolution of 256 filters of kernel size 3×3 with stride 1 (same)

2. Batch normalization, see [21] (same)
3. ReLU non-linearity function (same)

- **Residual block**. AlphaGo Zero (OLIVAW)

1. A convolution of 256 filters of kernel size 3×3 with stride 1 (same)
2. Batch normalization (same)
3. ReLU non-linearity function (same)
4. A convolution of 256 filters of kernel size 3×3 with stride 1 (same)
5. Batch normalization (same)
6. A skip connection adding the input to the block (same)
7. ReLU non-linearity function (same)

- **Value head**. AlphaGo Zero (OLIVAW)

1. A convolution of 1 filter of kernel size 1×1 with stride 1 (same)
2. Batch normalization (same)
3. ReLU non-linearity function (same)
4. A fully connected layer to a hidden layer of size 256 (same of size 96)
5. ReLU non-linearity function (same)
6. A fully connected layer to a scalar (same)
7. Tanh non-linearity function outputting a scalar in the range $[-1, 1]$ (same)

- **Policy head**. AlphaGo Zero (OLIVAW)

1. A convolution of 2 filters of kernel size 1×1 with stride 1 (same)
2. Batch normalization (same)
3. ReLU non-linearity function (same)
4. A fully connected layer that outputs a vector of size $19^2 + 1 = 362$ corresponding to move probabilities (same of size $8^2 + 1 = 65$)

4.2.3 Evaluation

This stage consists in the evaluation of the neural network trained on the new games against the starting one.

Agents use the MCTS search as in the self-play phase. In AlphaGo Zero are played 400 games, if the agent using the new neural network wins more than 55% of the games, it is declared the best one and used for the next self-play phase, otherwise the new network is discarded and the training proceeds with the old neural network. In OLIVAW are played only 40 games to save computation.

4.3 The full algorithm in pseudocode

Here it is presented the complete sequential training algorithm of AlphaGo Zero in python pseudocode. It is generalized to any zero sum perfect information game. We are using:

- `class ZerosumPerfinfGame()`: This class encodes the rules of the game, it should have the following methods:
  - `__initialization__()`: creates an instance of the game.
  - `initial_state()`: returns the starting game state s.
  - `next_state(s,a)`: given a game state s and a legal action a, it returns the next state s'.
  - `game_phase(s)`: given a game state s, it returns `None` if the game is in progress, 1 if the winner is white, -1 if the winner is black, 2 if there is a draw.
- `class DeepNeuralNetwork()`: This class interfaces the general algorithm with the neural network, it should have the following methods:
  - `__initialization__(game_rules)`: creates a neural network with an architecture compatible with the rules of the game.
  - `copy()`: returns another instance of the neural network.
  - `training(T)`: given a training set T, it trains the network on T updating the network weights θ.
  - `predict(s)`: given a game state s, it returns the neural network predictions on value v and move probabilities $\boldsymbol{p}$.
- `class MCTS()`: This class handles the Monte Carlo Tree Search, it should have the following methods:
  - `__initialization__(game_rules, f_theta)`: initializes the horacle as f_θ and it will use the game rules to expand properly the tree.
  - `run_simulation_from(s, turn)`: given a starting game state s, it runs several simulations from s updating the statistics on the edges. The turn information is used to add Dirichlet noise if needed.
  - `move_probabilities_form(s)`: given a game state s, it returns the move probabilities $\boldsymbol{\pi}$.
- `class Player()`: This class encodes a playing agent A, it should have the following methods:
  - `__initialization__(game_rules, f_theta)`: initializes the agent to the chosen game and its evaluation function as f_θ.
  - `play(s)`: given a game state s, it returns the action a chosen by the agent.
- `function action_selection(pi, turn)`: Given a move probabilities vector, it returns the chosen action. The turn information is needed to set the temperature parameter τ.
- `function compute_value(winner, s)`: Given a game state s, it returns the value z of the position for the current player knowing the outcome of the game.
- `function data_augmentation(training_sample)`: Given a training sample $(s, \boldsymbol{\pi}, z)$, it returns list of training samples generated from the input one using game symmetries.

- `function duel(player1, player2)`: Given two playing agents, it performs a series of games returning `True` if player1 won more than 55% of games and `False` otherwise.

This is the full algorithm in pseudocode:

```
# Initialization
game_rules = ZerosumPerfinfGame()
f_theta = DeepNeuralNetwork(game_rules)
T = []

for iteration in range(number_of_iterations):

  # self-play
  for game in range(number_of_games):
    s = game_rules.initial_state()
    T_game = []
    mcts_game = MCTS(game_rules, f_theta)
    turn = 0
    winner = None

    while not winner:
      turn += 1
      mcts_game.run_simulation_from(s, turn)
      pi = mcts_game.move_probabilities_from(s)
      T_game.append((s, pi))
      a = action_selection(pi, turn)
      s = game_rules.next_state(s, a)
      winner = game_rules.game_phase(s)

    for sample in T_game:
      s, pi = sample
      value = compute_value(winner, s)
      T = T + data_augmentation((s, pi, value))

  # Training of the neural network
  f_theta_new = f_theta.copy()
  f_theta_new.training(T)

  # Evaluation
  new_player = Player(game_rules, f_theta_new)
  old_player = Player(game_rules, f_theta)
  new_player_is_stronger = duel(new_player, old_player)
  if new_player_is_stronger:
    f_theta = f_theta_new.copy()
```

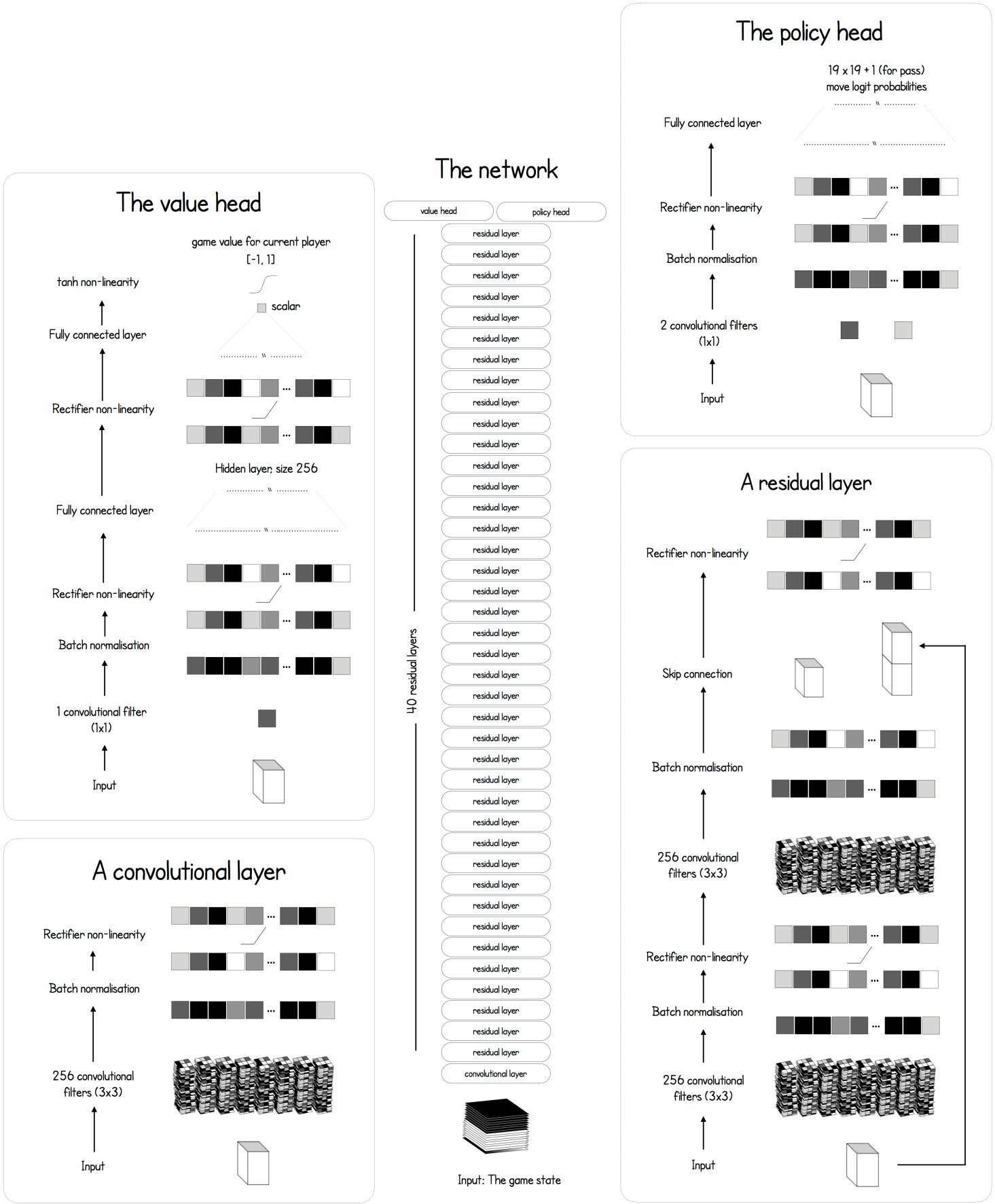


Figure 4.6. AlphaGo Zero Residual Network architecture. *Image taken from [19].*

Chapter 5

Training OLIVAW

Despite the ultimate goal of AlphaGo Zero and OLIVAW is reaching a superhuman level of play, all the insights from the journey to this result are definetely of interest. We are going to look to discoveries rather than emulations, analyzing a pure learning system, provided of no more information on the game than the simple rules.

In this chapter are presented all the insights from the training phase, starting with a brief discussion on the hardware used.

All the data used to generate charts was obtained using the code in the Appendix.

5.1 Hardware used

AlphaGo Zero reached the level of play of AlphaGo Lee (the version that defeated the world champion Lee Sedol) in just 3 days. The hardware cost to reach this result has been quoted as around 25 million dollars [22].

The ambitious goal of OLIVAW was to reach the same result in Othello, within a time frame of few days and using only free resources available to anyone with a laptop and an internet connection.

This was achieved using Colaboratory[1], a Google service for machine learning education and research. Colaboratory was launched in early 2018 and provides free cloud virtual machines equipped with hardware accelerators to code in python (Jupyter notebooks) to Google users. It requires no setup to use and provides an easy and fast connection to Google Drive for memory.

A notebook can be run using a standard CPU or an hardware accelerator that is crucial when using neural networks. The accelerators available are a GPU Nvidia Tesla K80 or a TPU v2 [2]. The virtual machine can be used freely for 12 hours, then it is restarted. The detailed harware specs of a virtual machine are:

- CPU: 1 × single core hyper threaded Xeon Processor, 2.3Ghz, 2 threads.
- RAM: ≈ 12.6 GB.
- Disk: ≈ 33 GB.
- (if used) GPU: 1 × Nvidia Tesla K80, 2496 CUDA cores, 12GB GDDR5 VRAM.

[1] https://colab.research.google.com

[2] Comparable to ∼ 10 Nvidia K80 GPUs, Wikipedia entry for Tensor Processing Unit

- (if used) TPU v2: Google Tensor processing unit, to date details on the specific chip available in Colaboratory are unknown.

Among the three phases of the Olivaw's algorithm, the generation of the games is the most computationally expensive due to the high number of neural network calls, even using GPU (Neural network predictions account for 85% of the computational time in a single MCTS simulation, a single prediction take ≈ 0.01 s).

However, game generation can be performed naturally in parallel by running several instances of the self-play code. This result has been achieved by stimulating a crowdfunding project that exploits the ease of sharing of Colaboratory. [3]

Even the training of the neural network is extremely computationally expensive and moreover can not be executed in parallel. Nevertheless this process can be highly accelerated using a TPU, the acceleration factor is approximately 10 respect to a GPU K80. A full training requires ≈ 1.5 h.

The evaluation phase consists in 40 games between agents of different generations and can be run in less than an hour using a GPU K80.

5.2 Training process

Olivaw's training phase began on 4 November 2018, with the first games generated using a Neural Network with random weights. The eighteenth generation of Olivaw, the one that defeated the Italian Othello champion Alessandro Di Mattei, has been successfully evaluated on 3 December 2018, after ≈ 50.000 games against itself.

We refer to the *i-th generation* as the *i-th successful update of the weights.*

The trend of the training loss over all the generations is showed in Figure 5.1

[3] Colaboratory notebook to automatically generate games from your Google account.

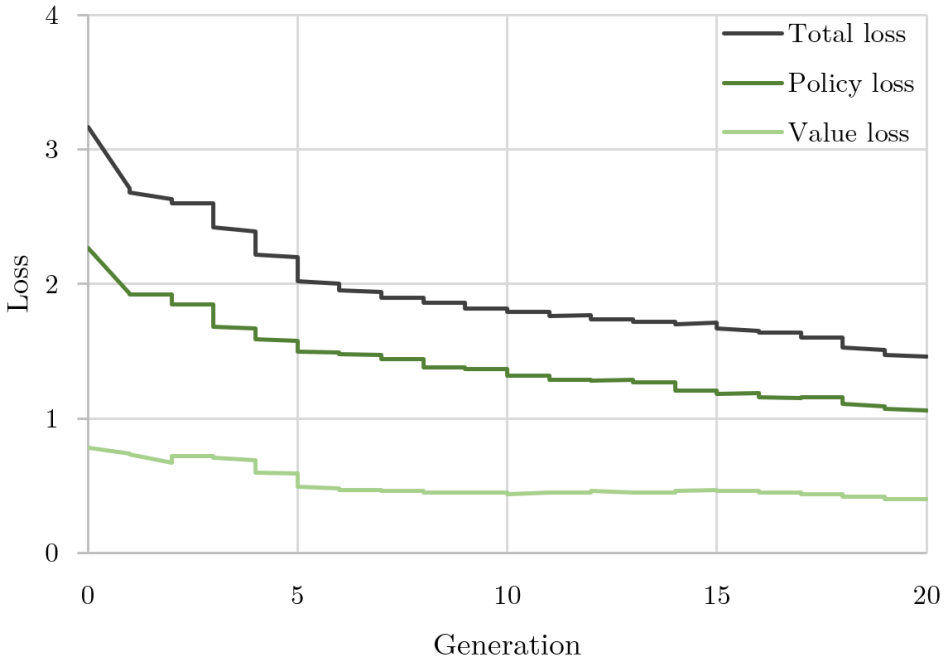


Figure 5.1. Neural network training loss over all the generations. We report the loss at the beginning and at the end of each of the twenty training phases

5.3 Training insights

5.3.1 Elo ratings

We evaluated the relative strength of Olivaw by measuring the Elo rating of each generation. Elo rating system is a method for calculating the relative skill levels of players in zero-sum games. The difference in ratings between two players serves as a predictor of the final outcome, for example if the difference is 100 points, the stronger player should win 64% of the games, if 200 he should win 76%, in accordance with the following formula:

$$p(a \text{ defeats } b) = \frac{1}{1 + \exp(c_{\text{elo}}(e(b) - e(a)))} \tag{5.1}$$

using the standard $c_{\text{elo}} = 1/400$. The ratings were computed using the results of the evaluations stages and the result of the first match of generation 14 vs the italian champion Alessandro Di Mattei, rated 2139[4] at that time. The ratings of each generation of Olivaw are showed in Figure 5.2:

[4]See World Othello Federation ratings.

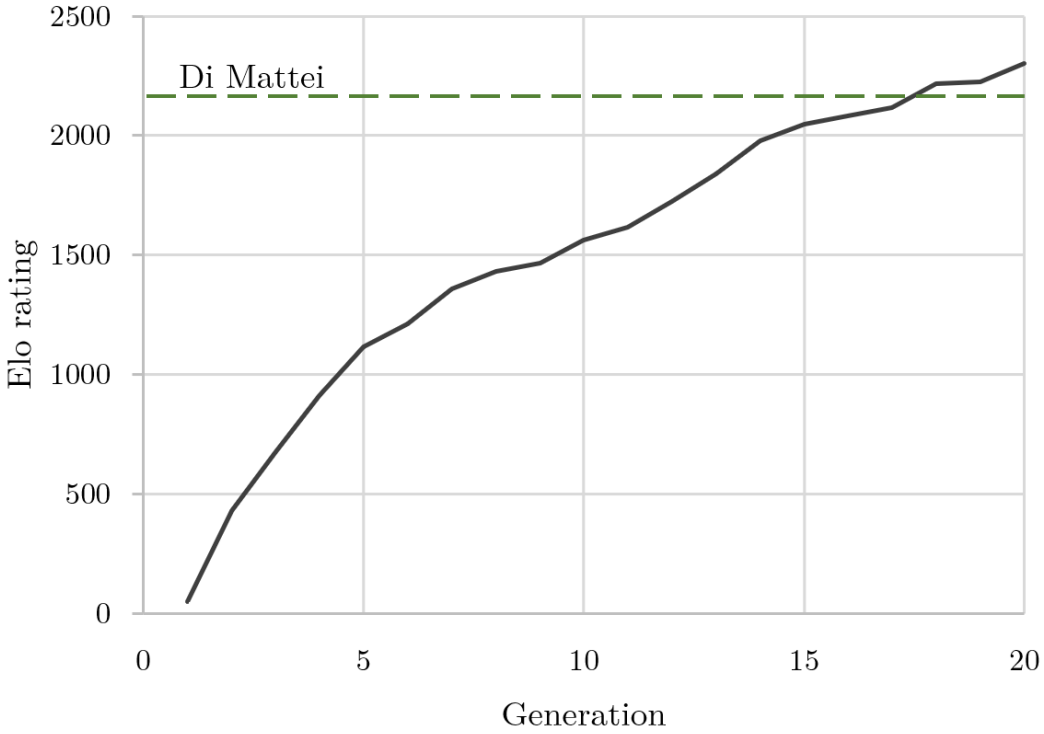


Figure 5.2. Elo rating of each generation of Olivaw, the orange line represents the Elo Rating of the italian champion Alessandro Di Mattei.

5.3.2 Black and White Scores

Does Olivaw prefer playing with Black or White? It depends from the generations. During the self-play games until 7th generation Olivaw wins more with white, then with black. Number of draws slowly increases as expected, we can see the trends in Figure 5.3:

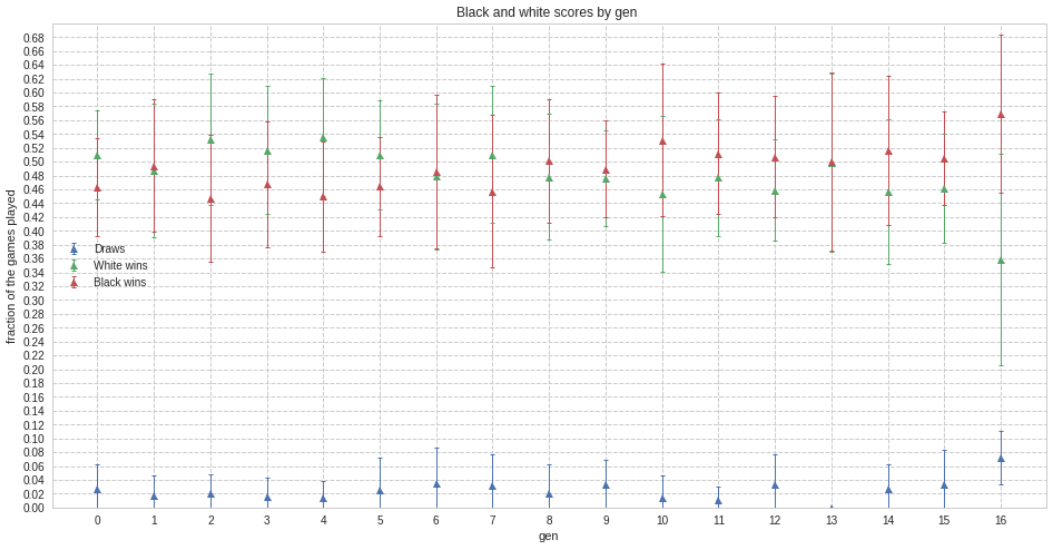


Figure 5.3. Black and White scores for each generation of Olivaw, the bars indicate standard deviations.

5.3.3 Resignation thresholds

Until generation 16, Olivaw plays only 10% of games until the end to save computation. If during a game a player values a position under a resignation threshold v_{resign}, the game ends and the opponent is considered the winner. v_{resign} is fixed automatically using the bunch of complete games, so that less than 5% of those games could have be won if the player had not resigned.

The distribution of the resignation thresholds is an interest measure to monitor learning, we expect that the standard deviation of the resignation threshold distribution decreases over the generations, because a stronger player makes more significant and consistent evaluations of the value of a position, i.e. he can trust his judgment more. We can see the decreasing trend in Figure 5.4:

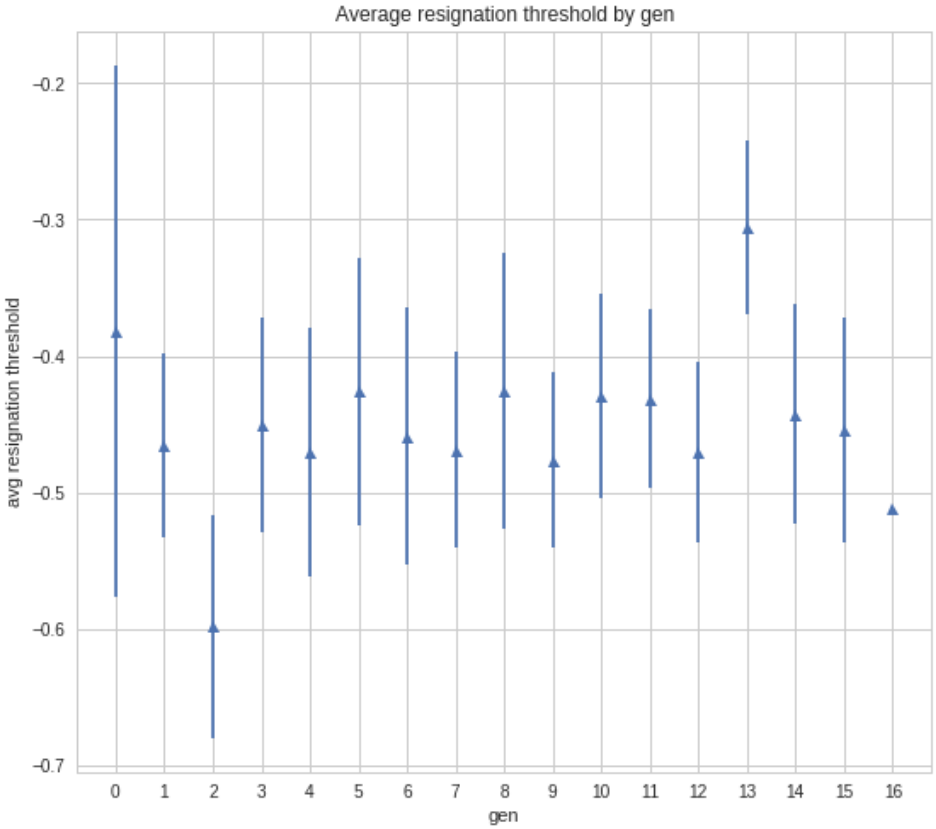


Figure 5.4. Average resignation threshold of each generation of Olivaw, the bars indicate standard deviations. The standard deviation of the 16th generation's games is omitted because of the few measures.

5.3.4 Largest value drops

A drop in the trend of values in a game corresponds either to a bad move or to a over-optimistic evaluation of previous positions, i.e. an error.

Monitoring the trend of average largest value drops in lost games over generations is a good measure to check Olivaw's learning. We expect a decreasing trend, the stronger a player is, the more difficult is to surprise him.

We can see the decreasing trend in Figure 5.5:

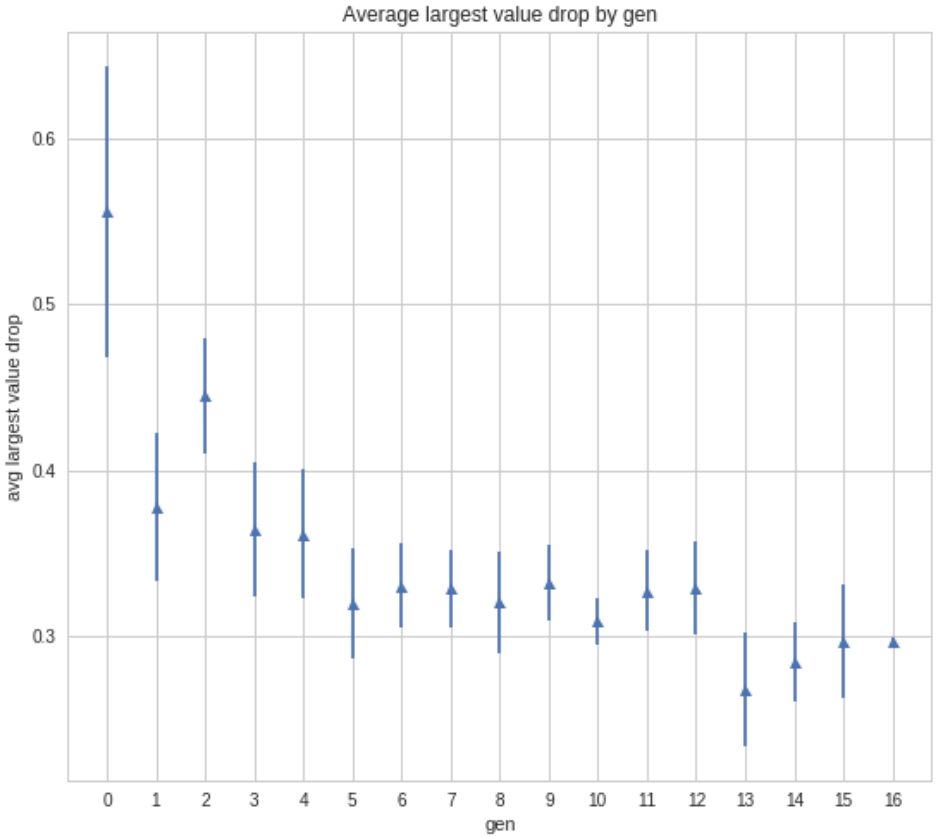


Figure 5.5. Average largest value drop of each generation of Olivaw, the bars indicate standard deviations. The standard deviation of the 16th generation's games is omitted because of the few measures.

5.3.5 Location of the crucial move

What was the opponent's move that caused the biggest drop in the evaluations of the loser? In a sense we can think to this move as the crucial one, the beginning of the defeat. This data can help to understand the hottest spots on the board for the early generations of Olivaw, then the relevance of the single box lose significance in favor of more complex patterns. Nevertheless it is a trace of the strategies adopted by the first generations, for instance we can appreciate very well the evolution of importance of the corners, then of the accesses to the corners, and so on towards an uniform distribution on the board.

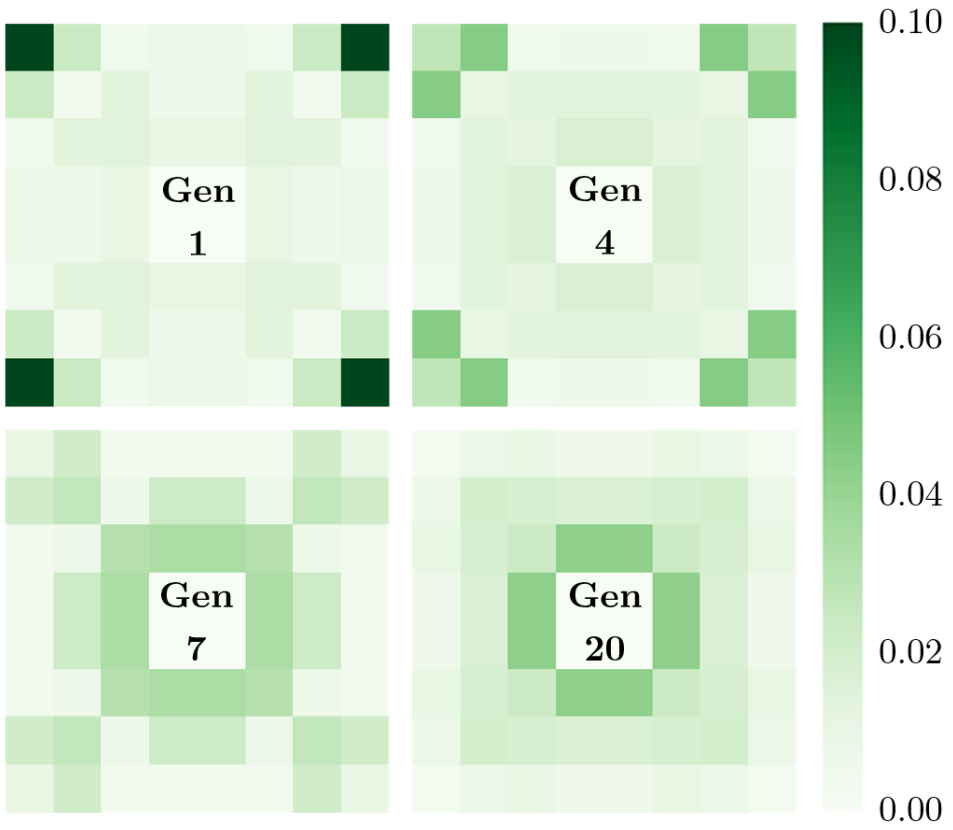


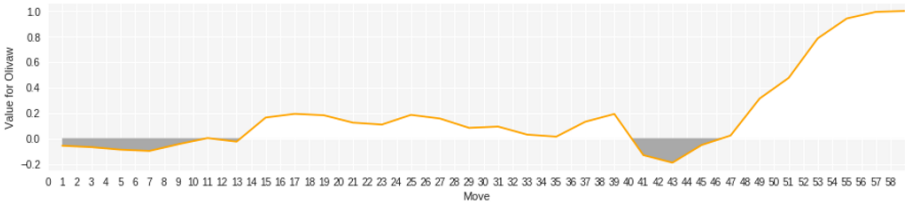
Figure 5.6. Location of crucial moves by generation. Notice the high relevance attributed to the conquest of the corners in the early games, similarly to human beginners. This shifts to the accesses to the corners, and then towards the center of the board, as we would expect from a more experienced player.

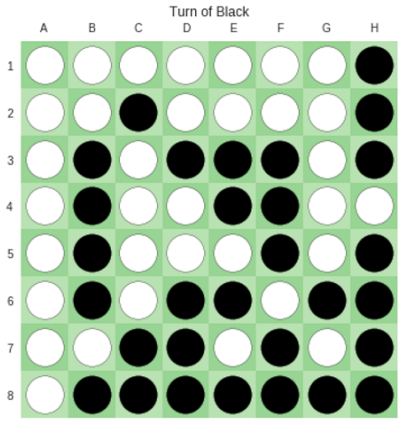
5.4 Games against other engines

This section shows relevant games against other Othello engines, Zebra and Saio. The engines run without using the library at a fixed search depth, for example Zebra 4 looks forward to four moves at most. The engines always play the best move, i.e. they are deterministic. Olivaw is not deterministic because it analyzes a position in one of its eight possible randomly chosen symmetries, but often comes to the same solution, especially the more advanced generations.

Olivaw analyzes 1000 positions for each move (number of simulations in the MCTS), a traditional engine analyzes approximately 5^d positions, where d is the maximum search depth. Zebra4 analyzes $\approx$ 1000 positions per move, Saio 10 analyzes $\approx$ 10.000.000 of positions per move.

5.4.1 First victory against Zebra 4



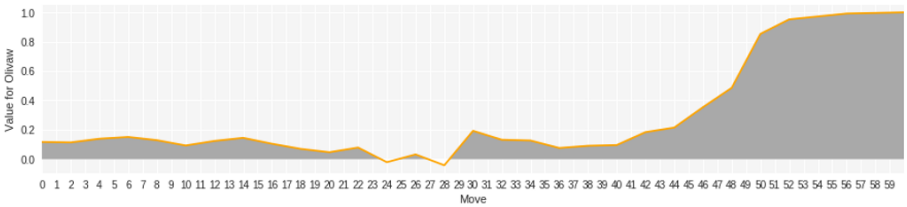


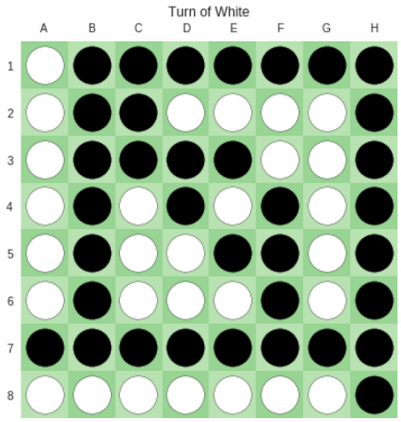
Zebra 4 (**B**) vs Olivaw gen 8 (**W**) – **30 - 34**

1. C4, C3	13. E8, G5	25. H7, G7
2. D3, E3	14. G6, H6	26. H2, G2
3. D2, C6	15. D8, C8	27. B1, A2
4. F3, E2	16. B8, G8	28. H5, H4
5. F1, C5	17. B6, A6	29. H1, G1
6. D6, C2	18. B3, B5	30. B2, A1
7. F2, D1	19. B7, B4	
8. C1, E1	20. A3, G4	
9. F5, F6	21. H3, G3	
10. F4, E7	22. A4, A5	
11. D7, C7	23. A7, F8	
12. F7, E6	24. H8, A8	

Figure 5.7. Olivaw generation 8 vs Zebra 4

5.4.2 First victory against Zebra 6



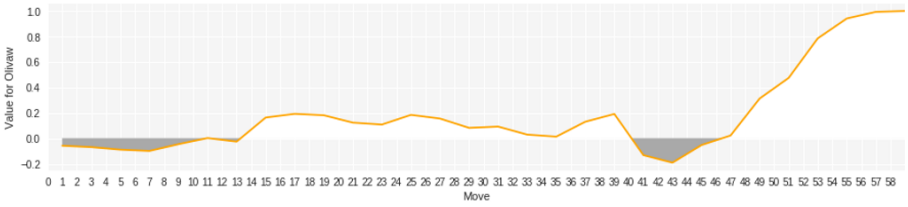


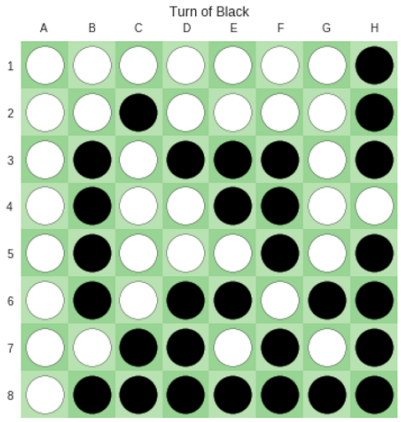
Olivaw gen 11 (**B**) vs Zebra 6 (**W**) – **36 - 28**

1. F5, D6	13. C8, D2	25. H2, G8
2. C5, F4	14. E7, F2	26. F8, B8
3. E3, C6	15. E2, D8	27. B7, G7
4. D3, F6	16. E8, F3	28. H1, G1
5. E6, D7	17. C7, F1	29. B1, A8
6. G4, C4	18. B2, H4	30. A7, PASS
7. B4, B3	19. D1, E1	31. H8
8. B6, A3	20. H3, H6	
9. B5, A4	21. G3, H5	
10. G5, C3	22. H7, G6	
11. A5, A6	23. F7, G2	
12. C2, C1	24. A2, A1	

Figure 5.8. Olivaw generation 11 vs Zebra 6

5.4.3 First victory against Zebra 8



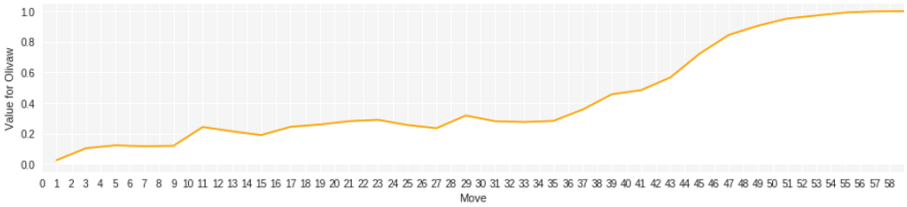


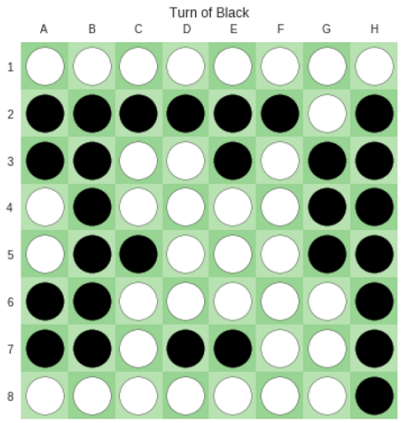
Zebra 8 (**B**) vs Olivaw gen 18 (**W**) – **30 - 34**

1. C4, C3	13. E8, G5	25. H7, G7
2. D3, E3	14. G6, H6	26. H2, G2
3. D2, C6	15. D8, C8	27. B1, A2
4. F3, E2	16. B8, G8	28. H5, H4
5. F1, C5	17. B6, A6	29. H1, G1
6. D6, C2	18. B3, B5	30. B2, A1
7. F2, D1	19. B7, B4	
8. C1, E1	20. A3, G4	
9. F5, F6	21. H3, G3	
10. F4, E7	22. A4, A5	
11. D7, C7	23. A7, F8	
12. F7, E6	24. H8, A8	

Figure 5.9. Olivaw generation 18 vs Zebra 8

5.4.4 First victory against Saio 10





Saio 10 (**B**) vs Olivaw gen 18 (**W**) – **28 - 36**

1. D3, C5	13. F3, D8	25. A7, B8
2. D6, E3	14. B4, A5	26. B1, A1
3. F4, C6	15. A3, B3	27. A2, G8
4. F5, C3	16. B6, E8	28. H3, H7
5. C4, B5	17. A6, A4	29. H8, G7
6. F2, E6	18. B7, H5	30. H2, H1
7. E7, F7	19. H6, G4	
8. C7, F6	20. H4, A8	
9. G5, E2	21. G6, G3	
10. D7, C8	22. F8, G1	
11. C2, D2	23. F1, E1	
12. D1, C1	24. B2, G2	

Figure 5.10. Olivaw generation 18 vs Saio 10

Chapter 6

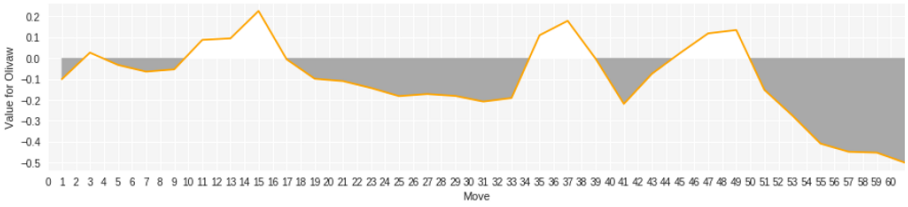
Olivaw meets Elijah

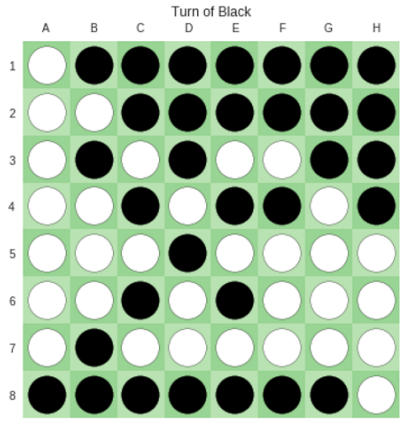
6.1 The challenge with the Italian champion

This section shows the games versus the 2016 and 2017 Italian champion Alessandro Di Mattei. There were two matches in the formula of a best of 5, the first one against the 14th generation of Olivaw, the second one aginst the 18th generation of Olivaw. Olivaw uses always 1000 simulations in its Monte Carlo tree search.

6.1.1 27/11/2018 The first match

The first match ended 3-2 for Di Mattei, from the point of view of Olivaw: Draw - Draw - Defeat - Victory - Defeat.

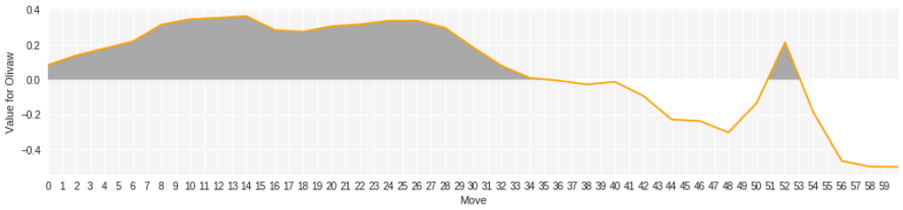


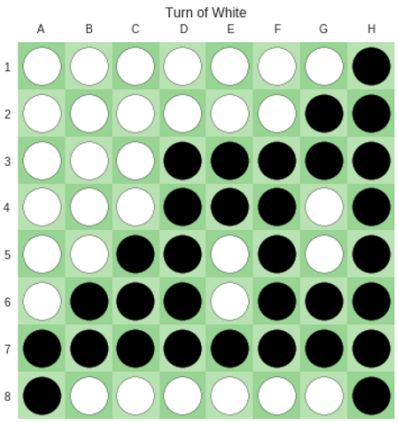


A. Di Mattei (**B**) vs Olivaw gen 14 (**W**) – **32 - 32**

1. C4, E3	13. F7, D1	25. H1, G3
2. F6, E6	14. F1, D7	26. H2, H3
3. F5, C5	15. E1, C1	27. H4, B2
4. C3, C6	16. B1, G6	28. A2, A1
5. D3, D2	17. C7, E7	29. PASS, H5
6. E2, B3	18. F8, D8	30. PASS, H8
7. B4, C2	19. H6, F2	31. G8, H7
8. B6, A4	20. G1, G5	
9. B5, D6	21. C8, B8	
10. A3, A5	22. G7, B7	
11. A6, F3	23. E8, G2	
12. F4, G4	24. A8, A7	

Figure 6.1. Olivaw generation 14 vs Alessandro Di Mattei, first match

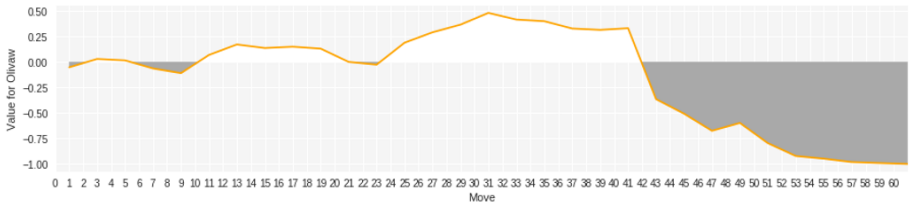


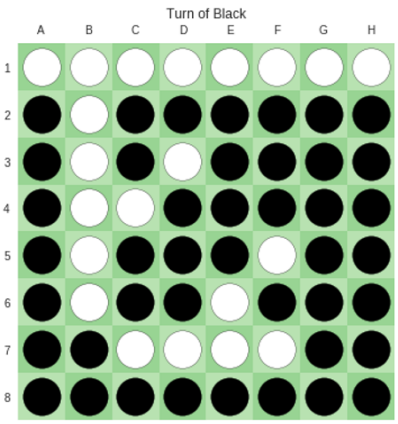


Olivaw gen 14 (**B**) vs A. Di Mattei (**W**) – **32 - 32**

1. D3, C3	13. A2, E8	25. D1, G7
2. C4, C5	14. F7, G5	26. H4, H5
3. B4, D2	15. G6, H6	27. A7, G1
4. D6, C6	16. C2, B7	28. H8, G8
5. E6, D7	17. F3, A5	29. H7, G2
6. B5, A3	18. A8, A1	30. H2, PASS
7. C7, C8	19. E2, B3	31. H1
8. B6, A6	20. D8, B8	
9. E3, E7	21. B2, G4	
10. A4, F2	22. G3, B1	
11. F8, F5	23. C1, H3	
12. F6, F4	24. F1, E1	

Figure 6.2. Olivaw generation 14 vs Alessandro Di Mattei, second match

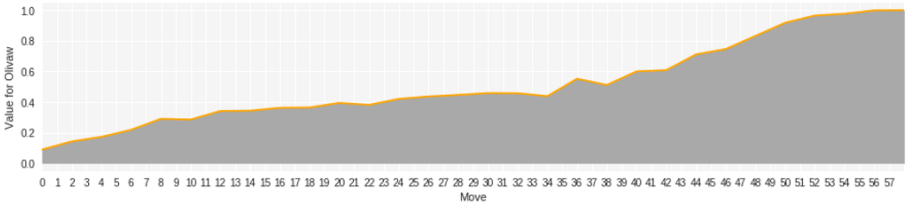


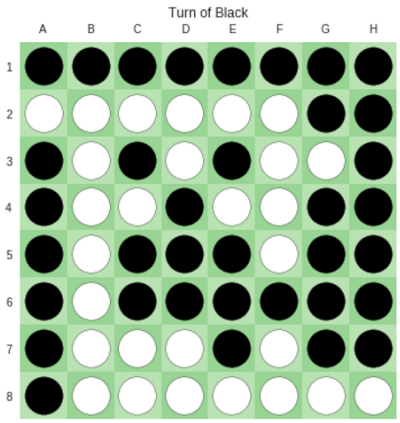


A. Di Mattei (**B**) vs Olivaw gen 14 (**W**) – **43 - 21**

1. C4, E3	13. E1, G5	25. A7, C7
2. F6, E6	14. G4, G6	26. H2, G2
3. F5, C5	15. H5, D6	27. G1, B7
4. C3, C6	16. B1, H3	28. A8, F1
5. D3, D2	17. H4, H6	29. F2, H1
6. E2, B3	18. F7, F8	30. PASS, A1
7. C1, C2	19. D8, A4	31. PASS, B2
8. B4, A3	20. A2, E8	
9. A5, B5	21. G8, G3	
10. A6, F4	22. D7, C8	
11. F3, B6	23. B8, G7	
12. E7, D1	24. H8, H7	

Figure 6.3. Olivaw generation 14 vs Alessandro Di Mattei, third match

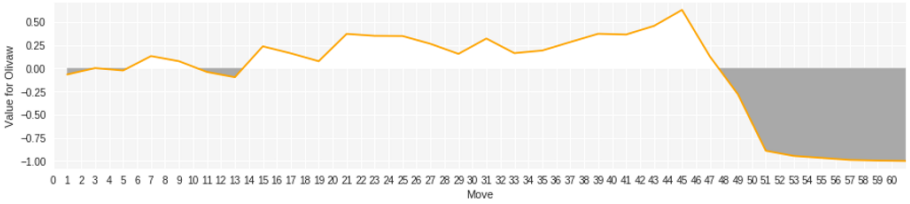


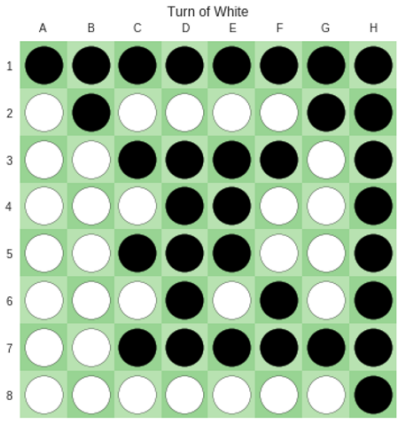


Olivaw gen 14 (**B**) vs A. Di Mattei (**W**) – **28 - 36**

1. F5, F6	13. E3, B3	25. H7, G2
2. E6, D6	14. B4, C3	26. F2, B7
3. E7, G5	15. A5, A6	27. A8, B8
4. C5, C6	16. D3, C2	28. H1, F1
5. C4, F3	17. D2, D1	29. G1, E1
6. D7, D8	18. A3, H2	30. A1, A2
7. C7, C8	19. C1, B1	
8. F4, B6	20. F7, A4	
9. G6, G4	21. A7, B2	
10. H5, H6	22. E2, E8	
11. H4, H3	23. G8, G7	
12. G3, B5	24. F8, H8	

Figure 6.4. OLIVAW generation 14 vs Alessandro Di Mattei, fourth match





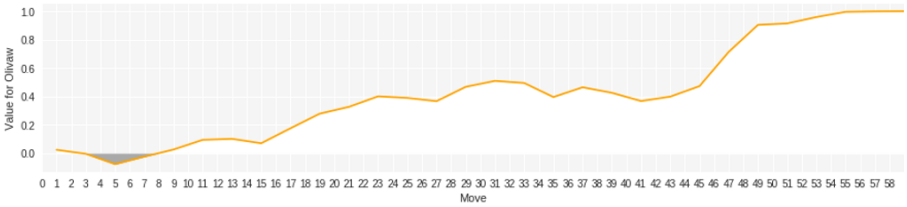
A. Di Mattei (**B**) vs Olivaw gen 14 (**W**) – **33 - 31**

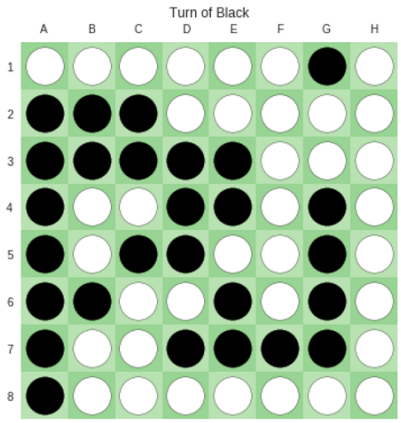
1. C4, E3	13. D2, C7	25. H8, G7
2. F6, E6	14. C8, C2	26. B2, B3
3. F5, C5	15. B1, E8	27. A3, A4
4. F4, G6	16. B8, F1	28. A7, A6
5. F7, C3	17. G1, G8	29. PASS, A2
6. H6, G4	18. H5, H4	30. PASS, G2
7. G3, D7	19. B7, B5	31. H1, PASS
8. E7, F3	20. A5, B4	32. A1
9. F2, H3	21. F8, B6	
10. D3, E2	22. D8, A8	
11. E1, C6	23. H2, C1	
12. D6, G5	24. D1, H7	

Figure 6.5. OLIVAW generation 14 vs Alessandro Di Mattei, fifth match

6.1.2 The second match against generation 18

The second match ended 4-0 for OLIVAW, from the point of view of OLIVAW: Victory - Victory - Victory - Victory.

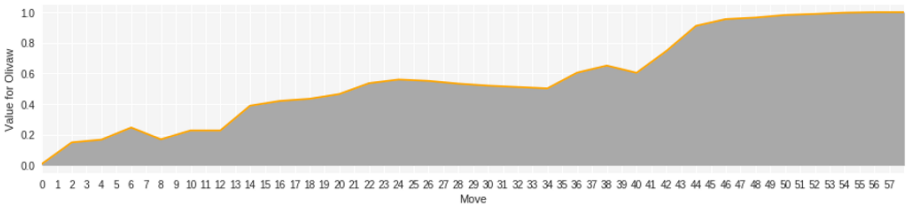


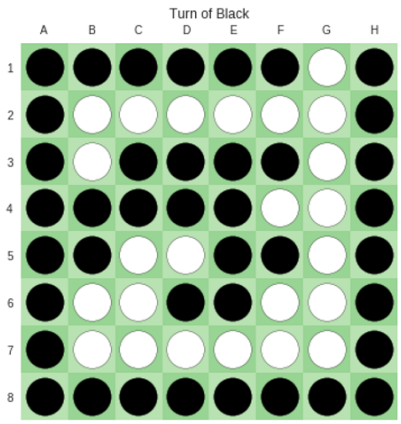


A. Di Mattei (**B**) vs Olivaw gen 18 (**W**) – **27 - 37**

1. C4, E3	13. B2, D7	25. H7, A6
2. F6, E6	14. C8, G5	26. A2, H8
3. F5, C5	15. H4, C7	27. G8, H1
4. C3, C6	16. D8, G6	28. G3, F2
5. D3, D2	17. H6, H5	29. A7, B8
6. E2, B3	18. G4, H3	30. G1, G2
7. B4, A3	19. H2, B7	
8. E7, C2	20. B1, B5	
9. D6, F1	21. A8, A1	
10. D1, F4	22. A5, E8	
11. E1, C1	23. F7, G7	
12. B6, F3	24. A4, F8	

Figure 6.6. OLIVAW generation 18 vs Alessandro Di Mattei, first match

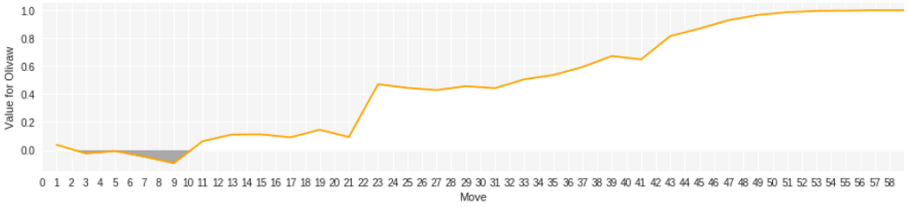


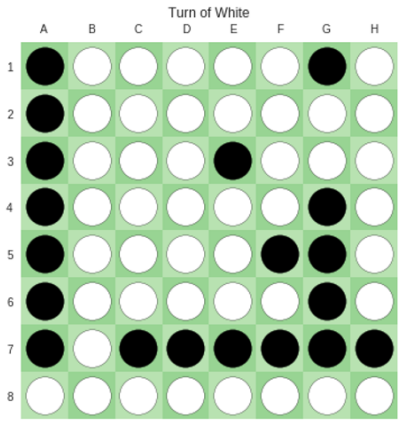


Olivaw gen 18 (**B**) vs A. Di Mattei (**W**) – **24 - 40**

1. E6, F6	13. F4, C7	25. A8, G3
2. F5, D6	14. G6, H6	26. E8, D8
3. E7, G5	15. F7, G8	27. B8, B7
4. C5, C6	16. H4, H5	28. H8, G7
5. E3, C4	17. H7, C2	29. H1, H2
6. D7, F8	18. D2, F2	30. H3, G1
7. B4, D3	19. F3, D1	
8. C3, A3	20. E2, G2	
9. B5, B3	21. E1, C1	
10. B6, C8	22. B1, G4	
11. A4, A5	23. F1, B2	
12. A6, A7	24. A1, A2	

Figure 6.7. OLIVAW generation 18 vs Alessandro Di Mattei, second match

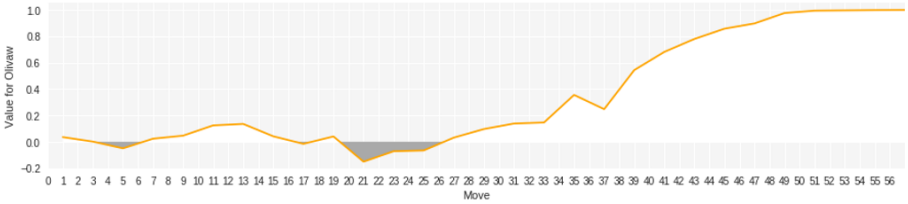


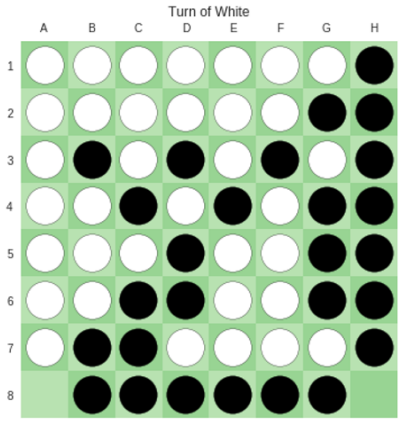


A. Di Mattei (**B**) vs Olivaw gen 18 (**W**) – **19 - 45**

1. C4, E3	13. A3, B6	25. H5, H1
2. F6, E6	14. D7, A4	26. G1, B8
3. F5, C5	15. A5, A6	27. G7, A8
4. C3, B4	16. B3, C1	28. A7, B2
5. D3, C2	17. E1, A2	29. A1, B1
6. D6, F4	18. D1, F1	30. PASS, H8
7. E2, F3	19. G2, E7	31. H7
8. D2, C6	20. E8, F7	
9. G5, G4	21. H2, C8	
10. F2, G3	22. F8, D8	
11. H3, H4	23. C7, G8	
12. B5, H6	24. B7, G6	

Figure 6.8. Olivaw generation 18 vs Alessandro Di Mattei, third match





A. Di Mattei (**B**) vs Olivaw gen 18 (**W**) – **27 - 37**

1. C4, E3	13. D6, D7	25. G8, B2
2. F6, E6	14. C8, F3	26. B8, G7
3. F5, C5	15. C7, F8	27. A2, A1
4. C3, C6	16. F7, G5	28. PASS, G1
5. D3, D2	17. H4, G6	29. G2, H2
6. E2, B3	18. H5, D1	30. H1
7. C1, C2	19. F1, F2	
8. B4, A3	20. B1, H6	
9. A5, B5	21. H7, G4	
10. A6, B6	22. H3, F4	
11. A4, A7	23. G3, E8	
12. E7, E1	24. B7, D8	

Figure 6.9. Olivaw generation 18 vs Alessandro Di Mattei, fourth match

6.2 Conclusions

After one month of training, using only "normal" computational resources, OLIVAW has achieved a superhuman level in Othello, defeating the former Italian champion Alessandro Di Mattei as well as other Othello engines at a world class level.

OLIVAW learned to play Othello knowing only the rules of the game, starting *tabula rasa*, as AlphaGo Zero and AlphaZero, demonstrating once again the universality of this learning paradigm in zero-sum and perfect information board games.

Appendix A

Games generation source code

For the sake of brevity, the reported source code corresponds only to the IPython notebook used for generating games[1]. Anyhow it contains all the functions needed to perform also the training of the neural network and the evaluation phase, even if these are not called.

A.1 Game implementation

```
import numpy as np


class OthelloBoard():
    """An Othello board n x n

    Implementation derived from the original of Eric P. Nichols.
    A game state contains the position on the board and the current player.

    """

    # list of all 8 directions on the board, as (x,y) offsets
    __directions = [(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1),(0,1)]

    def __init__(self, game_state=None, n=8):
        """ Setup board.

        Args:
            game_state: a numpy array representing the board and the current player,
            to set the board in a specific position. None for the initial position.
            n: size of the board if game_state is None

        """

        try:
          self.n = game_state[0].shape[1]
          self.pieces = game_state[0][0] * 1 + game_state[0][1] * -1
          self.state = game_state

        except TypeError:
          self.n = n
          # Create the empty board array.
          self.pieces = [None]*self.n
```

[1] Colab notebook

```
34            for i in range(self.n):
35                self.pieces[i] = [0]*self.n
36            self.pieces = np.array(self.pieces)
37
38            # Set up the initial 4 pieces.
39            self.pieces[int(self.n/2)-1][int(self.n/2)] = -1
40            self.pieces[int(self.n/2)][int(self.n/2)-1] = -1
41            self.pieces[int(self.n/2)-1][int(self.n/2)-1] = 1;
42            self.pieces[int(self.n/2)][int(self.n/2)] = 1;
43
44            self.state = (np.stack([self.pieces == 1, self.pieces == -1]), -1)
45
46        def __getitem__(self, index):
47            """ Add [][] indexer syntax to the Board"""
48            return self.pieces[index]
49
50        def white_advantage(self):
51            """ White pieces - black pieces"""
52            return np.sum(self.state[0][0]) - np.sum(self.state[0][1])
53
54        def get_legal_moves(self):
55            """Returns all the legal moves for the current player"""
56            color = self.state[1]
57            moves = set()  # stores the legal moves.
58
59            # Get all the squares with pieces of the given color.
60            for y in range(self.n):
61                for x in range(self.n):
62                    if self[x][y]==color:
63                        newmoves = self.get_moves_for_square((x,y))
64                        moves.update(newmoves)
65            return list(moves)
66
67        def has_legal_moves(self):
68            """ Returns True if the current player has legal moves"""
69            if np.sum(self.state[0][0]) + np.sum(self.state[0][1]) == self.n * self.n:
70                return False
71            else:
72              for y in range(self.n):
73                  for x in range(self.n):
74                      if self[x][y]:
75                        newmoves = self.get_moves_for_square((x,y))
76                        if len(newmoves) > 0:
77                            return True
78              return False
79
80        def get_moves_for_square(self, square):
81            """ Returns all the legal moves that use the given square as a base.
82            That is, if the given square is (3,4) and it contains a black piece,
83            and (3,5) and (3,6) contain white pieces, and (3,7) is empty, one
84            of the returned moves is (3,7) because everything from there to (3,4)
85            is flipped.
86            """
87            (x,y) = square
88
89            # determine the color of the piece.
90            color = self[x][y]
91
92            # skip empty source squares.
93            if color==0:
94                return None
95
96            # search all possible directions.
97            moves = []
```

```
98          for direction in self.__directions:
99              move = self._discover_move(square, direction)
100             if move:
101                 moves.append(move)
102
103         # return the generated move list
104         return moves
105
106     def execute_move(self, move):
107         """ Perform the given move on the board; flips pieces as necessary.
108         color gives the color pf the piece to play (1=white,-1=black)
109         """
110
111         #Much like move generation, start at the new piece's square and
112         #follow it on all 8 directions to look for a piece allowing flipping.
113         flips = [flip for direction in self.__directions
114                       for flip in self._get_flips(move, direction, self.state[1])]
115         assert len(list(flips))>0
116         for x, y in flips:
117             self[x][y] = self.state[1]
118         self.state = (np.stack([self.pieces == 1, self.pieces == -1]),
            ↪  -self.state[1])
119
120     def _discover_move(self, origin, direction):
121         """ Returns the endpoint for a legal move, starting at the given origin,
122         moving by the given increment."""
123         x, y = origin
124         color = self[x][y]
125         flips = []
126
127         for x, y in OthelloBoard._increment_move(origin, direction, self.n):
128             if self[x][y] == 0:
129                 if flips:
130                     return (x, y)
131                 else:
132                     return None
133             elif self[x][y] == color:
134                 return None
135             elif self[x][y] == -color:
136                 flips.append((x, y))
137
138     def _get_flips(self, origin, direction, color):
139         """ Gets the list of flips for a vertex and direction to use with the
140         execute_move function """
141         flips = [origin]
142
143         for x, y in OthelloBoard._increment_move(origin, direction, self.n):
144             if self[x][y] == 0:
145                 return []
146             if self[x][y] == -color:
147                 flips.append((x, y))
148             elif self[x][y] == color and len(flips) > 0:
149                 return flips
150
151         return []
152
153     @staticmethod
154     def _increment_move(move, direction, n):
155         """ Generator expression for incrementing moves """
156         move = list(map(sum, zip(move, direction)))
157         while all(map(lambda x: 0 <= x < n, move)):
158             yield move
159             move=list(map(sum,zip(move,direction)))
```

```
def hshbl_s(state):
  """ Returns an hashable of a game state (position + current player) """
  hsh = state[0].tostring()
  hsh += state[1].to_bytes(2, 'little', signed=True)
  return hsh

def hshbl_sa(state_action_tuple):
  """ Returns an hashable of a couple game state, action """
  hsh = hshbl_s(state_action_tuple[0])
  hsh += state_action_tuple[1].to_bytes(2, 'little')
  return hsh

class OthelloGame():
    """ Rules of the Othello game

    Implementation derived from the original of Eric P. Nichols.
    A game state contains the position on the board and the current player.

    """
    def __init__(self, n=8, memory_boost=True, cached_predictions_limit=750000):
        """ Generates rules for an Othello game played on a board n x n.

        Args:
            n: size of the board
            memory_boost: if True caches position resulting from played moves,
              to speed up the computation if those moves are
              newly played in a different game
            cached_predictions_limit: if memory_boost, set the limit of
              cached positions

        """
        self.n = n
        self.memory_boost = memory_boost
        if self.memory_boost:
          self.stateaction_nextstate = {}
          self.state_actions = {}
          self.prediction_used = 0
          self.cached_prediction_limit = cached_predictions_limit

    def initial_state(self):
        """ Initialize an OthelloBoard and returns the initial game state """
        b = OthelloBoard(n=self.n)
        return b.state

    def state_size(self):
        """ Returns the shape of a game state """
        return (2, self.n, self.n)  # one channel per color

    def action_size(self):
        """ Returns the total number of different actions admissible in a game"""
        return self.n*self.n + 1  # + 1 for pass

    def next_state(self, state, action):
        """ Given current game state and an action, returns next game state.
        Action must be valid.
        """

        if self.memory_boost:
          try:
            new_state = self.stateaction_nextstate[hshbl_sa((state, int(action)))]
            self.prediction_used += 1
            return new_state
          except KeyError:
```

```
64              pass
65
66          b = OthelloBoard(game_state=state)
67          if action == self.n*self.n:
68              return (state[0], -state[1])
69          else:
70              move = (int(action/self.n), action%self.n)
71              b.execute_move(move)
72          if self.memory_boost and len(self.stateaction_nextstate) <
            ↪  self.cached_prediction_limit:
73            self.stateaction_nextstate[hshbl_sa((state, int(action)))] = b.state
74          return b.state
75
76      def possible_actions(self, state):
77          """ Given a game state,
78          returns the possible actions from the corresponding position as a vector
79          with entry 0 if action not possible and entry 1 if action possible
80          """
81
82          if self.memory_boost:
83            try:
84              possible_actions_vector = self.state_actions[hshbl_s(state)]
85              self.prediction_used += 1
86              return possible_actions_vector
87            except KeyError:
88              pass
89
90          b = OthelloBoard(game_state=state)
91          legal_moves =  b.get_legal_moves()
92          valids = [0]*self.action_size()
93          if len(legal_moves) == 0:
94            if not self.game_phase(state):
95              valids[-1] = 1
96            return np.array(valids)
97          for x, y in legal_moves:
98            valids[self.n*x+y] = 1
99          possible_actions_vector = np.array(valids, dtype=bool)
100
101         if self.memory_boost and len(self.state_actions) <
            ↪  self.cached_prediction_limit:
102           self.state_actions[hshbl_s(state)] = possible_actions_vector
103         return possible_actions_vector
104
105     def game_phase(self, state):
106         """ Given a game state, returns the phase of the game,
107         0 if not ended, 1 if white won, -1 if black won
108         """
109         b = OthelloBoard(game_state=state)
110         if b.has_legal_moves():
111           return 0
112         b = OthelloBoard(game_state=(state[0], -state[1]))
113         if b.has_legal_moves():  # if the other player has legal moves game
            ↪  continue
114           return 0
115         delta_white = b.white_advantage()
116         if delta_white > 0:
117           return 1
118         if delta_white < 0:
119           return -1
120         return 2  # draw signal
121
122     def get_final_state_value(self, state):
123       """ Given a final game state, returns the outcome respect to the player,
124       1 if the current player won, -1 if lost, 2 if draw
```

```
        """
        b = OthelloBoard(game_state=state)
        player = state[1]
        delta_white = b.white_advantage()
        if delta_white > 0:
            return 1 * player
        if delta_white < 0:
            return -1 * player
        return 2 # draw signal

    def get_score(self, state):
        """ Given a game state, returns the istant player advantage
        """
        b = OthelloBoard(game_state = state)
        if state[1] == 1:  # white
            return b.white_advantage()
        else:
            return -b.white_advantage()

    def stats(self):
      """ Prints stats about the cached positions """
      print("### OTHELLO GAME STATS ###\n")
      print("Othello game, board", str(self.n) + "x" + str(self.n))
      size_sans = asizeof.asizeof(self.stateaction_nextstate)
      size_sa = asizeof.asizeof(self.state_actions)

      print("\nMEMORY BOOST, Number of elements cached:",
       ↪  len(self.stateaction_nextstate) + len(self.state_actions))
      print("MEMORY BOOST, Total memory used:", (size_sans + size_sa) / 1000000,
       ↪  "MB")
      print("\nMEMORY BOOST, stateaction_nextstate, memory used:", (size_sans) /
       ↪  1000000, "MB")
      print("MEMORY BOOST, state_actions, memory used:", (size_sa) / 1000000, "MB")
      print("MEMORY BOOST, Time saved (s)", round(self.prediction_used * 0.00041,
       ↪  3))
```

A.2 Plot functions

```
import matplotlib.pyplot as plt


def display(state):
  """ Given a game state, print a beautiful image representing the position"""
  n = state[0].shape[1]
  line1 = np.full(n, 0.3)
  line2 = np.copy(line1)
  line1 [::2] += 0.1
  line2[1::2] += 0.1

  image = np.stack([line1, line2] * int(n/2))

  plt.matshow(image, cmap="Greens", vmin=0, vmax=1)

  y_white, x_white = np.where(state[0][0])
  y_black, x_black = np.where(state[0][1])
  plt.scatter(x_white, y_white, s=1500*6/n, c='w', edgecolors='black')
  plt.scatter(x_black, y_black, s=1500*6/n, c='black')

  row_labels = range(1, n + 1)
  col_labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
  plt.xticks(range(n), col_labels)
```

```
    plt.yticks(range(n), row_labels)
    plt.grid(False)
    player = "White" if state[1] == 1 else "Black"
    plt.title("Turn of " + player)
    plt.show()



b = OthelloBoard(n=8).state
display(b)


from google.colab import widgets

def display_game(states, details=None):
    """ Given a list of game states and eventually a list of details associated to
    each game state, print all in a beautiful sequential way using a colab widget

    Args:
        game_states: list of game states
        details: list of (list of) strings about each game state, should be of the
        same length of game_states

    """
    game_len = len(states)
    if details:
      if len(details) != game_len:
        print("Warning: not printing details, wrong dimension. Expected " +
         ↪  str(game_len) + ", given " + str(len(details)))
        details = None
      try:
        iterator = iter(details[0])
        details_iterable = True
      except TypeError:
        details_iterable = True
    tb = widgets.TabBar([str(i) for i in range(game_len)])
    for i in range(game_len):
      with tb.output_to(i, select=(i < 1)):
        display(states[i])
        if details:
          if details_iterable:
            for item in details[i]:
              print(item)
          else:
            print(details[i])
```

A.3 Data augmentation

```
def symmetric_equivalent(matrix, i):
    """ Given a matrix, returns its symmetric equivalent respect to the i-th
    ↪  symmetry.
    i symmetry is the inverse of the -i symmetry

    Args:
        matrix: a numpy array
        i: goes from -symmetries to symmetries with negative i representing the
        inverse of the corresponding positive symmetry transformation
    Returns:
        new_matrix: matrix transformed according to symmetry i
    """
    if i > 0:
```

```
    if i > 4:
      new_matrix = np.rot90(matrix, i - 4)
      return new_matrix
    else:
      new_matrix = np.rot90(matrix, i)
      new_matrix = np.fliplr(new_matrix)
      return new_matrix
  else:
    if i < -4:
      new_matrix = np.rot90(matrix, i + 4)
      return new_matrix
    else:
      new_matrix = np.fliplr(matrix)
      new_matrix = np.rot90(new_matrix, i)
      return new_matrix


def data_augmentation(sample, symmetries=8):
  """ Given a training sample, returns a tuple with the augmented samples obtained
  exploiting symmetries.

  Args:
      sample: a training sample (game state, pi, value)
      symmetries: how much symmetries you want to exploit
  Returns:
      a tuple in the form: (game_states, move_probabilities, values)
  """

  state, white_pi, white_value = sample
  n = state[0].shape[1]

  if state[1] == -1:  # black to white
    white_state = np.stack([state[0][1], state[0][0]])
  else:
    white_state = state[0]

  white_pi_board = np.reshape(white_pi[:-1], (n, n))
  new_game_states = []
  new_move_probabilities = []
  new_winners = []

  possible_symmetries = np.arange(1,9)
  if symmetries == 8:
    chosen_symmetries = possible_symmetries
  else:
    chosen_symmetries = np.random.choice(possible_symmetries, size=symmetries,
    ↪  replace=False)

  for i in chosen_symmetries:
    new_game_states.append(np.stack([symmetric_equivalent(white_state[0], i),
    ↪  symmetric_equivalent(white_state[1], i)]))
    new_move_probabilities.append(np.array(list(symmetric_equivalent(white_pi_board,
    ↪  i).ravel()) + [white_pi[-1]]))
    new_winners.append(white_value)
  return (new_game_states, new_move_probabilities, new_winners)
```

A.4 Neural Network

```
import sys
sys.path.append('..')

```

```
4  import argparse
5  from keras import layers
6  from keras.models import *
7  from keras.layers import *
8  from keras.optimizers import *
9  from keras.preprocessing import image
10 from keras.utils.data_utils import get_file
11 from keras.applications.imagenet_utils import preprocess_input
12 from IPython.display import SVG
13 from keras.utils.vis_utils import model_to_dot
14 from keras.initializers import glorot_uniform
15 from keras.models import model_from_json
16 from keras import regularizers
17
18
19
20 import scipy.misc
21 from keras.utils import plot_model, layer_utils
22
23 import keras.backend as K
24 K.set_image_data_format('channels_last')
25 K.set_learning_phase(1)
26
27 def identity_block(X, f, filters, l2_regularization, stage, block):
28     """ Identity block of the neural net, derived from the implementation presented
29     in Andrew Ng. Coursera Deep Learning specialization.
30
31     Args:
32       X: input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
33       f: integer, specifying the shape of the middle CONV's window for the main
    ↪  path
34       filters: python list of integers, defining the number of filters in the CONV
    ↪  layers of the main path
35       stage: integer, used to name the layers, depending on their position in the
    ↪  network
36       block: string/character, used to name the layers, depending on their position
    ↪  in the network
37
38     Returns:
39       X: output of the identity block, tensor of shape (n_H, n_W, n_C)
40     """
41
42     # defining name basis
43     conv_name_base = 'res' + str(stage) + block + '_branch'
44     bn_name_base = 'bn' + str(stage) + block + '_branch'
45
46     # Retrieve Filters
47     F1, F2 = filters
48
49     # Save the input value. You'll need this later to add back to the main path.
50     X_shortcut = X
51
52     # First component of main path
53     X = Conv2D(filters=F1, kernel_size=(f, f), strides=(1, 1), padding='same',
        ↪  name=conv_name_base + '2a', data_format='channels_first',
        ↪  kernel_initializer=glorot_uniform(seed=0),
        ↪  kernel_regularizer=regularizers.l2(l2_regularization))(X)
54     X = BatchNormalization(axis=3, name=bn_name_base + '2a')(X)
55     X = Activation('relu')(X)
56
57     # Second component of main path
```

```
58      X = Conv2D(filters=F2, kernel_size=(f, f), strides=(1, 1), padding='same',
        ↪  name=conv_name_base + '2b', data_format='channels_first',
        ↪  kernel_initializer=glorot_uniform(seed=0),
        ↪  kernel_regularizer=regularizers.l2(l2_regularization))(X)
59      X = BatchNormalization(axis=3, name=bn_name_base + '2b')(X)
60
61      # Final step: Add shortcut value to main path, and pass it through a RELU
        ↪  activation
62      X = Add()([X, X_shortcut])
63      X = Activation('relu')(X)
64
65      return X
66
67  def make_residual_net(game_rules, architecture_parameters):
68      """ Given the game rules and architecture parameters, builds and returns the
69      model of the residual net
70
71      Args:
72        game_rules: rules of the game used to derive input and output shapes
73        architecture_parametrs: tuple containing the number of filters for
74          each conv layer, the learning rate, and the l2 reg parameter
75
76      Returns:
77        model: the compiled keras model of the residual neural net
78      """
79
80      number_of_filters_convlayers, learning_rate, l2_regularization =
        ↪  architecture_parameters
81      # Neural Net
82      input_board = Input(shape=game_rules.state_size())    # s: batch_size x board_x
        ↪  x board_y x 2 channels (black, white)
83
84      X =
        ↪  Activation('relu')(BatchNormalization(axis=3)(Conv2D(number_of_filters_convlayers,
        ↪  3, padding='same', data_format='channels_first',
        ↪  kernel_regularizer=regularizers.l2(l2_regularization))(input_board)))
        ↪  # batch_size  x board_x x board_y x num_channels
85      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=2, block='b')
86      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=3, block='b')
87      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=4, block='b')
88      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=5, block='b')
89      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=6, block='b')
90      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=7, block='b')
91      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=8, block='b')
92      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=9, block='b')
93      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=10, block='b')
94      X = identity_block(X, 3, [number_of_filters_convlayers,
        ↪  number_of_filters_convlayers], l2_regularization, stage=11, block='b')
95      pi = Activation('relu')(BatchNormalization(axis=3)(Conv2D(2, 1, padding='same',
        ↪  data_format='channels_first',
        ↪  kernel_regularizer=regularizers.l2(l2_regularization))(X)))
96      pi = Flatten()(pi)
97      pi = Dense(game_rules.action_size(), activation='softmax', name='pi',
        ↪  kernel_regularizer=regularizers.l2(l2_regularization))(pi)
```

```
    v = Activation('relu')(BatchNormalization(axis=3)(Conv2D(1, 1, padding='same',
     ↪  data_format='channels_first',
     ↪  kernel_regularizer=regularizers.l2(l2_regularization))(X)))
    v = Flatten()(v)
    v = Dense(int(1.5 * game_rules.n**2), activation='relu', name='v1',
     ↪  kernel_regularizer=regularizers.l2(l2_regularization))(v)
    v = Dense(1, activation='tanh', name='v',
     ↪  kernel_regularizer=regularizers.l2(l2_regularization))(v)

    model = Model(inputs=input_board, outputs=[pi, v])
    model.compile(loss=['categorical_crossentropy','mean_squared_error'],
     ↪  optimizer=Adam(learning_rate))
    return model
```

```
class DeepNeuralNetwork():
  """ This class interfaces the general algorithm with the chosen neural
  network architecture
  """
  def __init__(self, game_rules, clone=None, model_file=None, weights_file=None,
   ↪  architecture_parameters=(256, 0.001, 0.0001), memory_boost=True,
   ↪  cached_prediction_limit=750000):
    """ Setup a Residual Neural Network from scratch or clone an existing one.

    Args:
        game rules:  rules of the game
        clone: a DeepNeuralNetwork object to clone, None for initializing the
          neural network from scratch
        model_file: a string with the path to a h5 file containing a model to load
 ↪  it
        weights_file: a string with the path to a h5 file containing weights to
 ↪  load it
        architecture_parameters: tuple containing the number of filters for
          each conv layer, the learning rate, and the l2 reg parameter
        memory_boost: if true caches predictions of the neural net to seed up the
 ↪  computation
        cached_predictions_limit: if memory_boost, set the max number of cached
 ↪  predictions
    """
    self.game = game_rules
    _, learning_rate, _ = architecture_parameters

    self.memory_boost = memory_boost
    if self.memory_boost:
      self.boardstate_prediction = {}
      self.prediction_used = 0
      self.predictions_failed = 0
      self.cached_prediction_limit = cached_prediction_limit

    if clone:
      self.model = clone_model(clone.model)
      self.model.set_weights(clone.model.get_weights())
      self.boardstate_prediction = clone.boardstate_prediction
      self.prediction_used = clone.prediction_used
      self.predictions_failed = clone.predictions_failed
      self.model.compile(loss=['categorical_crossentropy','mean_squared_error'],
       ↪  optimizer=Adam(learning_rate))
    elif model_file:
      self.model = load_model(model_file)
      self.model.compile(loss=['categorical_crossentropy','mean_squared_error'],
       ↪  optimizer=Adam(learning_rate))
    elif weights_file:
      self.model = make_residual_net(self.game,
       ↪  architecture_parameters=architecture_parameters)
```

```
41        self.model.load_weights(weights_file)
42      else:
43        self.model = make_residual_net(self.game,
           ↪  architecture_parameters=architecture_parameters)
44
45
46    def training(self, training_set, steps_per_epoch=1600, epochs=20,
       ↪  extract_from_last=20000000, batch_size=512,
47                 absolute_path=ABSOLUTE_PATH):
48      """ Trains the neural network using the given training set.
49
50        Args:
51            training_set: a TrainingSet object containing the generator
52            steps_per_epoch: number of generator calls that constitutes an epoch
53            epoch: number of epochs
54            extract_from_last: use data from the last n positions
55            absolute_path: path to the folder with the data
56      """
57      gen = training_set.training_generator(absolute_path=absolute_path,
         ↪  extract_from_last=extract_from_last, batch_size=batch_size)
58      h = self.model.fit_generator(gen, steps_per_epoch=steps_per_epoch,
         ↪  epochs=epochs, shuffle=False)
59
60
61    def save(self, iteration_tag, absolute_path=ABSOLUTE_PATH):
62      """ Save the model in the absolute_path folder.
63
64        Args:
65            iteration_tag: generation number, nedded for the file name
66            absolute_path: folder where to save the file
67      """
68      self.model.save(absolute_path + 'NNet_it.{0:03d}'.format(iteration_tag))
69
70    def predict(self, states):
71      """ Given a list of states, returns the neural network prediction for those
         ↪  states.
72
73        Args:
74            states: list of game states for which you want to make a prediction
75        Returns:
76          A tuple containing the move probabilities as numpy vector and the values
77          as numpy vector
78
79      """
80      if len(states) == 1:  # single state
81        board_state, player = states[0]  # state preparation for prediction
82        if player == -1:
83          board_state = np.stack([board_state[1], board_state[0]])
84
85        sym = np.random.randint(1,9)  # we pass to the neural network one of the
           ↪  equivalent symetric states
86        board_state = np.stack([symmetric_equivalent(board_state[0], sym),
           ↪  symmetric_equivalent(board_state[1], sym)])
87
88        if self.memory_boost:  # prediction
89          try:
90            prediction = self.boardstate_prediction[board_state.tostring()]
91            self.prediction_used += 1
92          except KeyError:
93            self.predictions_failed += 1
94            prediction = self.model.predict(board_state.reshape((1, 2, self.game.n,
               ↪  self.game.n)))
95            if self.memory_boost and len(self.boardstate_prediction) <
               ↪  self.cached_prediction_limit:
```

```
96              self.boardstate_prediction[board_state.tostring()] = prediction
97          else:
98            prediction = self.model.predict(board_state.reshape((1, 2, self.game.n,
                ↪  self.game.n)))
99
100         move_probabilities_sym, value = prediction  # we put things back in order
              ↪  with the symmetries
101         move_probabilities =
              ↪  symmetric_equivalent(move_probabilities_sym[0][:-1].reshape((self.game.n,
              ↪  self.game.n)), -sym)
102         move_probabilities = np.concatenate((move_probabilities.ravel(),
              ↪  [move_probabilities_sym[0][-1]]))
103         return (move_probabilities, value[0])
104
105       else:  # multiple states
106         board_states =[]
107         syms = []
108         for s in states:
109           board_state, player = s  # state preparation for prediction
110           if player == -1:
111             board_state = np.stack([board_state[1], board_state[0]])
112
113           sym = np.random.randint(1,9)  # we pass to the neural network one of the
                ↪  equivalent symetric states
114           board_state = np.stack([symmetric_equivalent(board_state[0], sym),
                ↪  symmetric_equivalent(board_state[1], sym)])
115           syms.append(sym)
116           board_states.append(board_state)
117
118         predictions = self.model.predict(np.array(board_states))
119
120         batch_move_probabilities = []
121         batch_values = []
122         for i in range(len(board_states)):
123           move_probabilities_sym, value = predictions[0][i], predictions[1][i]  # we
                ↪  put things back in order with the symmetries
124           move_probabilities =
                ↪  symmetric_equivalent(move_probabilities_sym[:-1].reshape((self.game.n,
                ↪  self.game.n)), -syms[i])
125           move_probabilities = np.concatenate((move_probabilities.ravel(),
                ↪  [move_probabilities_sym[-1]]))
126           batch_move_probabilities.append(move_probabilities)
127           batch_values.append(value[0])
128
129           if self.memory_boost and len(self.boardstate_prediction) <
                ↪  self.cached_prediction_limit:
130             self.boardstate_prediction[board_states[i].tostring()] =
                  ↪  (move_probabilities_sym.reshape(1, move_probabilities_sym.size),
                  ↪  value.reshape((1, 1)))
131
132         return (batch_move_probabilities, batch_values)
133
134
135     def stats(self):
136       """ Neural network stats about architecture and cached predictions. """
137       print("### DEEP NEURAL NETWORK STATS ###\n")
138       print("Number of layers:", len(self.model.layers))
139       print("Trainable parameters:", self.model.count_params())
140
141       print("\nMEMORY BOOST, Number of predictions cached:",
            ↪  len(self.boardstate_prediction))
142       size_bp = asizeof.asizeof(self.boardstate_prediction)
143       print("MEMORY BOOST, memory used:", (size_bp) / 1000000, "MB")
```

```
144     print("MEMORY BOOST, prediction used/failed:", self.prediction_used,
        ↪  self.prediction_failed)
145     print("MEMORY BOOST, Time saved (s)", round(self.prediction_used * 0.01063, 3))
```

A.5 Monte Carlo Tree Search

```
1   import math
2
3
4   class MCTS():
5     """
6     This class handles the Monte Carlo Tree Search.
7     the nodes of the tree are state-action couples, the structure of the tree is
    ↪  encoded
8     in two dictionaries: parent_children and child_parent, while the parameters
9     associated to every node are in the node_parameters dict.
10    The core functions are run_simulation_from() and
11    move_probabilities_and_value_from().
12
13    """
14
15    def __init__(self, game_rules, nnet, c_puct=1):
16      """ Setup MCTS.
17
18      Args:
19          game_rules:  rules of the game
20          nnet: a DeepNeuralNetwork object to invoke as oracle
21          c_puct: a constant determining the level of exploration (as in the AGZ
    ↪   paper)
22      """
23      self.game = game_rules
24      self.nnet = nnet
25      self.c_puct = c_puct
26
27      self.parent_children = {}
28      self.child_parent = {}
29
30      self.node_parameters = {}
31
32
33    def next_state(self, state, action):
34      """ Given a node (game state and a legal action) returns the resulting state,
35      common to all its children (that differ only in action)
36      """
37      return self.parent_children[hshbl_sa((state, int(action)))][0][0]
38
39
40    def move_probabilities_and_value_from(self, state, temperature=1,
    ↪   play="exploratory"):
41      """ Given a game state, reads the tree and returns the move probabilities
42      and the value from that state,
43
44      Args:
45          state: desired game state
46          temperature: temperature parameter used to flat or spike the pi
47          play: game mode, if competitive temperature is not used
48            (the action with the highest probability is selected)
49      Returns:
50          a tuple containing the move probabilities as a numpy vector and the
51          value as a float
52      """
```

```
53      possible_actions_vector = self.game.possible_actions(state)
54      n_vector = np.zeros(len(possible_actions_vector))
55      q_values = []  # value of the state is the max of the state-action values
56      ns = []
57      ws = []
58      ps = []
59      if not possible_actions_vector.size:
60        n_vector[-1] = 1
61      for i, entry in enumerate(possible_actions_vector):
62        if entry:
63          n, w, q, p = self.node_parameters[hshbl_sa((state, i))]
64          n_vector[i] = n
65          q_values.append(q)
66          ns.append(round(float(n), 3))
67          ws.append(round(w, 3))
68          ps.append(round(p, 3))
69      if play[0] == 'c':  # competitive
70        return n_vector / np.sum(n_vector), max(q_values)
71
72      if play[0] == 'e':  # exploratory
73        n_vector_temper = np.power(n_vector, 1/temperature)
74        return  n_vector_temper / np.sum(n_vector_temper), max(q_values)
75
76
77    def run_simulation_from(self, state, to_leaf_iterations=100, dirichlet=True,
      ↪  virtual_losses=True, batch_size=8):
78      """ Given a game state, run n simulations to expand the tree from that state
79
80      Args:
81          state: desired game state
82          to_leaf_iterations: number of simulations, i.e. how many times
83            the horacle is called
84          dirichlet: if True, dirichelet noise is added to the move probabilities
85            in the first simulation to promote exploration of different lines (see
    ↪   AGZ paper)
86          virtual_losses: if true virtual losses are used to optimize the prediction
87            time of the neural net horacle (predicting more states at once)
88          batch_size: If virtual_losses, how many states to predict at once
89      """
90      self.prune(state)
91      self.to_leaf_descent(state, dirichlet=dirichlet)
92      to_leaf_iterations -= 1
93
94      if virtual_losses:
95        while len(self.parent_children) < batch_size * 5 and to_leaf_iterations:  # 5
96          self.to_leaf_descent(state)
97          to_leaf_iterations -= 1
98
99        while to_leaf_iterations:
100         leaves = []
101         paths = []
102         possible_actions_vectors = []
103         batch_in_progress_size = 0
104
105         while batch_in_progress_size < batch_size and to_leaf_iterations:
106           growing_branch = self.to_leaf_descent(state, virtual_loss=True)  # insert
              ↪  leaf in batch only if the leaf need the nnet call
107           to_leaf_iterations -= 1
108
109           if growing_branch:
110             leaf, path, possible_actions_vector = growing_branch
111             leaves.append(leaf)
112             paths.append(path)
113             possible_actions_vectors.append(possible_actions_vector)
```

```
            batch_in_progress_size += 1

        if leaves:
          try:
            batch_values = self.batch_leaves_expansion(leaves, paths,
             ↪  possible_actions_vectors)
          except:
            print(leaves)
            raise ValueError
          for j, leaf_value in enumerate(batch_values):
            self.from_leaf_ascent(leaf_value, paths[j])

    else:
      for i in range(to_leaf_iterations):
        self.to_leaf_descent(state)


  def prune(self, state):
    """ Prune the tree maintaining only the subtree starting from the given state
     ↪  """
    if not self.child_parent:  # if the tree is empty
      pass

    else:
      possible_actions_vector = self.game.possible_actions(state)
      possible_actions = np.nonzero(possible_actions_vector)[0]

      if hshbl_sa((state, int(possible_actions[0]))) not in self.child_parent:
        self.parent_children = {}
        self.child_parent = {}
        self.node_parameters = {}

      else:  # create a copy of the good branches and del the rest

        new_root = (state, 255)
        self.new_parent_children = {}
        self.new_child_parent = {}
        self.new_node_parameters = {}

        self.new_parent_children[hshbl_sa(new_root)] = []
        for action in possible_actions:
          child = (state, int(action))
          child_hshbl = hshbl_sa(child)
          self.new_parent_children[hshbl_sa(new_root)].append(child)
          self.new_child_parent[child_hshbl] = new_root
          self.new_node_parameters[child_hshbl] = self.node_parameters[child_hshbl]

          self.update_progeny(child)

        self.parent_children = self.new_parent_children
        self.child_parent = self.new_child_parent
        self.node_parameters = self.new_node_parameters

  def update_progeny(self, node):
    """ Recursive function used by the prune() function to build a new tree
    containing all the progeny of a given node
    """
    node_hshbl = hshbl_sa(node)
    if node_hshbl in self.parent_children:
      self.new_parent_children[node_hshbl] = []
      for child in self.parent_children[node_hshbl]:
        child_hshbl = hshbl_sa(child)
        self.new_parent_children[node_hshbl].append(child)
        self.new_child_parent[child_hshbl] = node
```

```
176         self.new_node_parameters[child_hshbl] = self.node_parameters[child_hshbl]
177
178         self.update_progeny(child)
179
180       else:
181         return
182
183
184     def to_leaf_descent(self, state, dirichlet=False, virtual_loss=False, eps=0.75):
185       """ A single descent to discover new leaves.
186       If virtual losses are not used this function calls the horacle, creates new
187       nodes, initializes their parameters and updates the parameters of all visited
      ↪  nodes.
188       If virtual losses are used this function performs exclusively the descent,
189       returning the final leaf, the path and the possible actions from the leaf,
190       for a later call at the horacle and subsequent tree update
191
192       Args:
193           state: desired game state, beginning of the descent
194           to_leaf_iterations: number of simulations, i.e. how many new leaves will
195             have the tree
196           dirichlet: if True, dirichelet noise is added to the p of the new nodes
      ↪  (the leaves)
197           virtual_losses: if true the question for the neural net horacle
198             (pi and value for an unseen position) waits the filling of a batch of
199             questions that will be subjected all at once to speed up the prediction
      ↪  time.
200             The node parameters are updated at a later time
201           eps: fraction of dirichlet noise to use (see AGZ paper)
202       Returns:
203           None if virtual_loss is False
204           A tuple containing the leaf state, the path and the possible actions from
205             the leaf state if virtual_loss is True
206       """
207       root = (state, 255)  # root action is a placeholder
208       descent = [root]
209       stop_simulation=False
210       while not stop_simulation:
211         possible_actions_vector = self.game.possible_actions(state)
212         possible_actions = np.nonzero(possible_actions_vector)[0]
213         draw = False
214
215         if len(possible_actions) == 0:
216           stop_simulation = True
217           value = self.game.get_final_state_value(state)
218           virtual_loss = False
219           if value == 2:
220             draw = True
221
222
223         elif hshbl_sa((state, int(possible_actions[0]))) not in self.child_parent:
224           stop_simulation = True
225
226           if not virtual_loss:
227             move_probabilities, value = self.nnet.predict([state])
228             value = np.asscalar(value)
229             move_probabilities = move_probabilities * possible_actions_vector
230             move_probabilities /= np.sum(move_probabilities)
231
232             parent = descent[-1]
233             self.parent_children[hshbl_sa(parent)] = []
234             if dirichlet:  # alpha is proportional to the inverse of legal moves
235               alpha = min([1, 10 / len(possible_actions)])
236               dirichlet_noise = np.random.dirichlet([alpha] * len(possible_actions))
```

```
237             for i, action in enumerate(possible_actions):
238               self.parent_children[hshbl_sa(parent)].append((state, int(action)))
239               self.child_parent[hshbl_sa((state, int(action)))] = parent
240               if dirichlet:
241                 self.node_parameters[hshbl_sa((state, int(action)))] = (0, 0., 0., (1
                    ↪  - eps) * move_probabilities[action] + eps * dirichlet_noise[i])
242               else:
243                 self.node_parameters[hshbl_sa((state, int(action)))] = (0, 0., 0.,
                    ↪  move_probabilities[action])
244
245         else:
246           if len(possible_actions) == 1:
247             action = possible_actions[0]
248           else:
249             q_values = np.zeros(len(possible_actions))
250             u_values_nosqrt = np.zeros(len(possible_actions))
251             n_tot = 0
252             max_q_plus_u = 0
253             for i, action in enumerate(possible_actions):
254               n, w, q, p = self.node_parameters[hshbl_sa((state, int(action)))]
255               q_values[i] = q
256               u_values_nosqrt[i] = self.c_puct * p / (1 + n)
257               n_tot += n
258             if n_tot:
259               q_plus_u_values = q_values + u_values_nosqrt * math.sqrt(n_tot)
260             else:
261               q_plus_u_values = q_values + u_values_nosqrt
262             action = possible_actions[np.argmax(q_plus_u_values)]
263           descent.append((state, int(action)))
264           state = self.game.next_state(state, action)
265
266
267       for node in reversed(descent[1:]):
268         n, w, q, p = self.node_parameters[hshbl_sa(node)]
269         n += 1
270         if not virtual_loss:
271           if draw:
272             value = -0.5  # value of a draw for both players
273           else:
274             value = -value
275           w += value
276           q = w / n
277         self.node_parameters[hshbl_sa(node)] = (n, w, q, p)
278
279       if virtual_loss:
280         return state, descent[1:], possible_actions_vector
281       else:
282         return None
283
284
285     def from_leaf_ascent(self, leaf_value, path):
286       """ If virtual losses are used, this function updates the node parameters after
287       the horacle batch prediction
288
289       Args:
290           leaf_value: predicted value of the leaf position
291           path: list of nodes visited in the simulation from the root to the leaf
292       """
293       for node in reversed(path):
294         leaf_value = -leaf_value
295         n, w, q, p = self.node_parameters[hshbl_sa(node)]
296         w += leaf_value
297         q = w / n
298         self.node_parameters[hshbl_sa(node)] = (n, w, q, p)
```

```
299
300
301   def batch_leaves_expansion(self, leaves, paths, possible_actions_vectors):
302     """ If virtual losses are used, this function submit the leaf postion batch to
303     the oracle, initializes the children of the leaf nodes with the obtained
304     move probabilities of the batch and returns the values of the batch
305
306     Args:
307         leaves: leaf nodes in the batch
308         paths: lists of nodes visited in the simulation from the root to the leaves
309         possible_actions_vectors: possible action from each of the leaf positions
310     Returns:
311         batch_values: list of the values of the leaf node positions
312     """
313     batch_move_probabilities, batch_values = self.nnet.predict(leaves)
314     batch_move_probabilities = np.array(batch_move_probabilities) *
        ↪  np.array(possible_actions_vectors)
315     batch_move_probabilities = batch_move_probabilities /
        ↪  np.sum(batch_move_probabilities, axis=1)[:, np.newaxis]
316
317     for i, leaf in enumerate(leaves):
318       parent = paths[i][-1]
319       self.parent_children[hshbl_sa(parent)] = []
320
321       for j, entry in enumerate(possible_actions_vectors[i]):
322         if entry:  # if entry, j corresponds to the action
323           self.parent_children[hshbl_sa(parent)].append((leaf, int(j)))
324           self.child_parent[hshbl_sa((leaf, int(j)))] = parent
325           self.node_parameters[hshbl_sa((leaf, int(j)))] = (0, 0, 0,
              ↪  batch_move_probabilities[i][j])
              ↪  ########################################
326
327     return batch_values
328
329
330
331   def stats(self):
332     """ Stats about the MCTS size, useful to test the prune impact
333     """
334
335     print("### MCTS STATS ###\n")
336     print("Number of nodes:", len(self.child_parent))
337     size_pc = asizeof.asizeof(self.parent_children)
338     size_cp = asizeof.asizeof(self.child_parent)
339     size_np = asizeof.asizeof(self.node_parameters)
340     print("\nTotal size:", (size_pc + size_cp + size_np) / 1000000, "MB\n")
341     print("parent_children", size_pc / 1000000, "MB")
342     print("child_parent", size_cp / 1000000, "MB")
343     print("node_parameters", size_np / 1000000, "MB")
```

A.6 Training Samples

```
1  from os import listdir
2
3  class TrainingSet():
4    """ Through a TrainingSet object it is possible to collect positions data,
5    save it to disk automatically and create a generator with the last collected data
6    """
7    def __init__(self, save_every=16384):
8      """ TrainingSet initialization
9
```

```
    Args:
        save_every: save data to disk every n positions
    """
    self.game_states = []
    self.search_probabilities = []
    self.winners = []
    self.save_every = save_every

  def extend(self, samples, label=0):
    """ Extend TrainingSet object with new data

    Args:
        samples: tuple of lists containing states, move probs and values
        label: iteration number to use as label when saving the data
    """
    new_game_states, new_search_probabilities, new_winners = samples
    if new_game_states:
      self.game_states.extend(new_game_states)
      self.search_probabilities.extend(new_search_probabilities)
      self.winners.extend(new_winners)
    if len(self.winners) > self.save_every:
      self.save(label)

  def save(self, iteration, absolute_path=ABSOLUTE_PATH):
    """ Save to disk all the positions in memory and frees the memory

    Args:
        iteration: iteration number used as tag for the data files
        absolute_path: path to the folder with the data
    """
    i = 0
    file_tag_i = '{0:03d}_{1:04d}.npz'.format(iteration, i)
    while np.DataSource().exists(absolute_path + 'it.' + file_tag_i):
      i += 1
      file_tag_i = '{0:03d}_{1:04d}.npz'.format(iteration, i)
    np.savez_compressed(absolute_path + 'it.' + file_tag_i,
     ↪  game_states=self.game_states, search_probs=self.search_probabilities,
     ↪  winners=self.winners)
    self.game_states = []
    self.search_probabilities = []
    self.winners = []

  @staticmethod
  def training_generator(absolute_path=ABSOLUTE_PATH, extract_from_last=2000000,
   ↪  batch_size=1024):
    """ Generator that yields a batch from the last collected data
    Args:
      absolute_path: path to the folder with the data
      extract_from_last: use data from the last n positions
      batch_size: size of a batch of data
    Yields:
      a tuple in the form: (game_states, move_probabilities, values)
    """
    arrays_collection = []

    # loading files
    available_triples = [f[3:] for f in listdir(absolute_path) if f[0] == 'i']
    available_triples.sort()
    positions_loaded = 0
    while positions_loaded < extract_from_last and available_triples:
      triple_id = available_triples.pop()
      arrays_loaded = np.load(absolute_path + 'it.' + triple_id)
      positions_loaded += len(arrays_loaded['winners'])
      arrays_collection.append(arrays_loaded)
```

```
71
72      game_states = np.vstack(tuple(arrays_loaded['game_states'] for arrays_loaded in
        ↪  arrays_collection))
73      search_probs = np.vstack(tuple(arrays_loaded['search_probs'] for arrays_loaded
        ↪  in arrays_collection))
74      winners = np.concatenate(tuple(arrays_loaded['winners'] for arrays_loaded in
        ↪  arrays_collection))
75
76      indices = np.arange(min([extract_from_last, positions_loaded]))
77      while True:
78        extracted_indices = np.random.choice(indices, size=batch_size, replace=False)
79        yield (game_states[extracted_indices], [search_probs[extracted_indices],
          ↪  winners[extracted_indices]])
```

A.7 Self-play

```
1  def compute_value(winner, s):
2    """ Given a position, returns the value of the position for the current player
3    given the outcome of the game
4      """
5    player = s[1]
6    if winner != 2:
7      return player * winner
8    else:
9      return 0  # value of a draw
```

```
1  from tqdm import tqdm
2
3  def self_play(game_rules=None, f_theta=None, training_set=None,
   ↪  number_of_games=1000, random_seed=False, simulations=100,
4                resignation_threshold_used=None, virtual_losses=True, c_puct=3,
                 ↪  collect_multiple_positions=True,
5                collection_threshold=50, iteration_tag=0, show_games=10):
6    """ This function executes the self play games.
7      Args:
8        game_rules: rules of the game
9        f_theta: DeepNeuralNetwork to be used with the MCTS
10       training_set: TrainingSet to be used to manage the storage of the positions
11       number_of_games: number of self play games executed by the function
12       random_seed: if True a random seed is used to make the generated games
   ↪   replicable,
13         useful for debug, be sure to set False during real training
14       simulations: how many times the MCTS calls the horacle to evaluate a single
15         position
16       resignation_threshold_used: float or None, if float a player resigns when
17         evaluates the position under the resignation threshold, if None all games
18         are played until the end and a usable resignation threshold is calculated
19         and returned by the function. See details in AGZ  paper.
20       virtual_losses: if true virtual losses are used to optimize the prediction
21           time of the neural net horacle (predicting more states at once)
22       c_puct: a constant determining the level of exploration (as in the AGZ paper)
23       collect_multiple_positions: if True positions not played but very visited
24         during the MCTS simulations are collected in addition to the positions of
25         the game, the value used is their Q value instead of the final winner
26       collection_threshold: if collect_multiple_positions is True this number set
27         the visiting threshold to collect a not played position
28       iteration_tag: iteration number used as tag for the data files
29       show_games: how many self played games are showed in output
30     Returns:
31       stats: a dictionary containing stats about the self played games
```

```
32          (e.g. number of white/black victories, avg number of moves, an usable
33          resignation threshold...)
34      """
35      if not game_rules:
36        game_rules = OthelloGame(n=8)
37      if not f_theta:
38        f_theta = DeepNeuralNetwork(game_rules)
39      if not training_set:
40        T = TrainingSet()
41      else:
42        T = training_set
43      if random_seed:
44        np.random.seed(42)
45      stats = {'iteration': iteration_tag, 'games': number_of_games, 'winner': {1:0,
        ↪  -1:0, 2:0},
46              'resignation_threshold': resignation_threshold_used,
                ↪  'avg_number_of_moves':0, 'final_turn': [0] * (game_rules.n *
                ↪  game_rules.n + 30),
47              'avg_largest_valuedrop': 0, 'move_causing_largest_valuedrop': [0] *
                ↪  game_rules.action_size()}
48
49      min_values_winner = [] # for resignation threshold
50
51      # Self play
52      for game_index in tqdm(range(number_of_games), position=0):
53        s = game_rules.initial_state()
54        T_game = []
55        mcts_game = MCTS(game_rules, f_theta, c_puct=c_puct)
56        winner = 0
57        turn = 0  # to adjust temperature after n moves
58        min_value = {1:1, -1:1}  # for resignation threshold
59        predicted_values = []  # q values predicted in each position
60        largest_valuedrop = 0  # largest q value drop
61        decisive_move = 0  # move corresponding to the largest q value drop
62        if collect_multiple_positions:
63          T_game_more = []
64
65        resignation = False
66        while not winner:
67          turn += 1
68          mcts_game.run_simulation_from(s, to_leaf_iterations=simulations,
            ↪  virtual_losses=virtual_losses)
69          pi, value = mcts_game.move_probabilities_and_value_from(s)
70
71          if resignation_threshold_used:
72            if value < resignation_threshold_used:
73              winner = - s[1]
74              resignation = True
75              break
76            else:
77              winner = game_rules.game_phase(s)  # 0: game not ended, 1: white won, -1:
                ↪  black won, 2: draw
78          else:
79            if value < min_value[s[1]]:
80              min_value[s[1]] = value
81
82          predicted_values.append(value)
83
84          T_game.append((s, pi))
85          if turn <= game_rules.n * game_rules.n / 3:
86            a = np.random.choice(game_rules.action_size(), p=pi)
87          else:  # temperature -> 0
88            a = np.argmax(pi)
89
```

```
90        if collect_multiple_positions:
91          for possible_action in np.nonzero(pi)[0]:
92            possible_action_visits = mcts_game.node_parameters[hshbl_sa((s,
              ↪  int(possible_action)))][0]
93            if possible_action_visits > collection_threshold:
94              try:
95                collectable_state = mcts_game.next_state(s, possible_action)
96                collectable_pi, collectable_value =
                  ↪  mcts_game.move_probabilities_and_value_from(collectable_state)
97                num_next_possible_actions = len(np.nonzero(collectable_pi)[0])
98                avg_totalvisits_per_nextpossibleaction = int(possible_action_visits /
                  ↪  num_next_possible_actions)
99                if avg_totalvisits_per_nextpossibleaction > collection_threshold:
100                 T.extend(data_augmentation((collectable_state, collectable_pi,
                    ↪  collectable_value),
101                                            symmetries=min(8,
                                               ↪  avg_totalvisits_per_nextpossibleaction
                                               ↪  % collection_threshold)),
                                               ↪  label=iteration_tag)
102                 if show_games:
103                   T_game_more.append((collectable_state, collectable_pi,
                      ↪  collectable_value, possible_action_visits))
104
105             except KeyError:
106               pass
107
108       if turn > 2:
109         value_drop = value - predicted_values[-3]
110         if value_drop > largest_valuedrop:
111           largest_valuedrop = value_drop
112           decisive_move = a
113           decisive_turn = turn
114
115       s = game_rules.next_state(s, a)
116       winner = game_rules.game_phase(s)
117
118     if resignation:
119       final_pi, _ = mcts_game.move_probabilities_and_value_from(s)
120     else:
121       final_pi = np.array([0.] * (game_rules.action_size() - 1) + [1.])
122
123     T_game.append((s, final_pi))
124     predicted_values.append(compute_value(winner, s))
125
126     if not resignation_threshold_used:
127       if winner == 1 or winner == -1:
128         min_values_winner.append(min_value[winner])
129       else:
130         min_values_winner.append(min_value[-1])
131         min_values_winner.append(min_value[+1])
132
133     stats['winner'][winner] += 1
134     stats['final_turn'][turn] += 1
135     stats['avg_largest_valuedrop'] = (stats['avg_largest_valuedrop'] * game_index +
        ↪  largest_valuedrop) / (game_index  + 1)
136     stats['move_causing_largest_valuedrop'][decisive_move] += 1
137
138     if show_games:
139       print_states = []
140       print_details = []
141       print_states_more = []
142       print_details_more = []
143
144     for i, sample in enumerate(T_game):
```

```
      s, pi = sample
      value = compute_value(winner, s)
      predicted_value = predicted_values[i]

      if show_games:
        banner_decisive_move = '' if i != decisive_turn else '   !!! CRUCIAL MOVE !!!'
        print_states.append(s)
        print_details.append(("Value: " + str(value) + banner_decisive_move,
                              "\nPredicted Value: " + str(round(float(predicted_value), 3)),
                              "\n\nPi: (pass action = " + str(pi[-1]) + ")",
                              np.around(np.asarray(pi[:-1]).reshape(game_rules.n, game_rules.n), decimals=2)))
      T.extend(data_augmentation((s, pi, value)), label=iteration_tag)

    if show_games:
      for i, sample in enumerate(T_game_more):
        s, pi, value, visits = sample

        if show_games:
          print_states_more.append(s)
          print_details_more.append(("Value: " + str(value) + '\t Visits: ' + str(visits),
                                "\nPredicted Value: " + str(round(float(value), 3)),
                                "\n\nPi: (pass action = " + str(pi[-1]) + ")",
                                np.around(np.asarray(pi[:-1]).reshape(game_rules.n, game_rules.n), decimals=2)))

    if show_games:
      print("\nExample of a game in self play, collected states: ", len(T_game), '+', len(T_game_more))
      display_game(print_states, details=print_details)
      print("\nAdditional collected states")
      display_game(print_states_more, details=print_details_more)
      show_games -= 1

  stats["avg_number_of_moves"] = sum([i * occurrences for i, occurrences in enumerate(stats['final_turn'])]) / number_of_games
  if not resignation_threshold_used:
    resignation_threshold = np.percentile(min_values_winner, 5)
    stats["resignation_threshold"] = resignation_threshold
    print("\nUsable resignation threshold:", resignation_threshold)
  return stats
```

A.8 Players

```
class GeneralPlayer():
  """ Class prototype for a player """

  def __init__(self, game):
    """ Initializes the player, providing the game_rules and whatever is needed
    by the specific player
    """
    self.game = game
    self.name= "PlayerName"
    pass

  def new_game(self):
    """ This method executes the necessary instructions at the beginning of a game
```

```
14        """
15        pass
16
17    def play(self, state):
18        """ Given a game state (position + current player) this method returns the
19        player action
20        """
21        pass
22
23    def stats(self):
24        """ This method returns meaningful metadata about the last player decision,
25        """
26        pass
27
28
29  class RandomPlayer():
30    def __init__(self, game, seed=False):
31      self.game = game
32      self.name= "Random"
33      self.seed = seed
34      if self.seed:
35        np.random.seed(seed)
36
37    def new_game(self):
38      pass
39
40    def play(self, state):
41      valids = self.game.possible_actions(state)
42      a = np.random.choice(np.nonzero(valids)[0])
43      return a
44
45    def stats(self):
46      return None
47
48
49  class HumanOthelloPlayer():
50    def __init__(self, game):
51      self.game = game
52      self.name = "Human"
53
54    def new_game(self):
55      print(" ### NEW GAME ###")
56      pass
57
58    def play(self, state):
59
60      display(state)
61      valid = self.game.possible_actions(state)
62      while True:
63          a = input()
64
65          y, x = [x for x in a.split(' ')]
66          y = 'ABCDEFGH'[:self.game.n].find(y.upper())
67          x = int(x) - 1
68          if x != 255 and y != -1:
69            a = self.game.n * x + y
70          else:
71            a = self.game.n * self.game.n
72          if valid[a]:
73              break
74          else:
75              print('Invalid')
76
77      return a
```

```
78
79    def stats(self):
80      return None
81
82
83  class GreedyOthelloPlayer():
84    def __init__(self, game):
85      self.game = game
86      self.name = "Greedy"
87
88    def new_game(self):
89      pass
90
91    def play(self, state):
92      color = state[1]
93      valids = self.game.possible_actions(state)
94      candidates = []
95      for a in range(self.game.action_size()):
96          if valids[a]==0:
97              continue
98          possible_next_state = self.game.next_state(state, a)
99          score = self.game.get_score(possible_next_state)
100         candidates += [(score, a)]
101     candidates.sort()
102
103     return candidates[0][1]
104
105   def stats(self):
106     return None
107
108
109 class OlivawOthelloPlayer():
110   def __init__(self, game, nnet=None, mcts=True, simulations=20,
      ↪  possible_actions_mask=True, tag=''):
111     self.game = game
112     self.name= "Daneel" + tag
113     self.to_leaf_iterations = simulations
114     self.nnet = nnet
115
116     self.possible_actions_mask = possible_actions_mask
117     self.last_move_probabilities = None
118     self.last_value=None
119
120     if not self.nnet:
121       print("No neural net passed as argument, initializing a new one with memory
          ↪  boost")
122       self.nnet = DeepNeuralNetwork(game)
123     if mcts:
124       self.mcts = MCTS(game, self.nnet)
125     else:
126       print("MCTS set to False, playing directly with the nnet")
127     if possible_actions_mask:
128       print("Warning, the neural net may output illegal moves that are masked out")
129
130   def new_game(self):
131     self.mcts = MCTS(self.game, self.nnet)
132
133   def play(self, state):
134     if self.mcts:
135       self.mcts.run_simulation_from(state,
          ↪  to_leaf_iterations=self.to_leaf_iterations, dirichlet=False)
136       move_probabilities, value = self.mcts.move_probabilities_and_value_from(state,
          ↪  play='competitive')
137       self.last_move_probabilities = move_probabilities
```

```
      self.last_value=value
      a = np.argmax(move_probabilities)
      if self.possible_actions_mask:
        possible_actions = self.game.possible_actions(state)
        if not possible_actions[a]:
          a = np.argmax(move_probabilities * possible_actions)
      return a
    else:
      move_probabilities, value = self.nnet.predict([state])
      self.last_move_probabilities = move_probabilities
      self.last_value=value
      a = np.argmax(move_probabilities)
      if self.possible_actions_mask:
        possible_actions = self.game.possible_actions(state)
        if not possible_actions[a]:
          a = np.argmax(move_probabilities * possible_actions)
      return a

  def stats(self):
    return (self.last_move_probabilities, self.last_value)
```

A.9 Duel

```
def duel(player1 ,player2, games=1, threshold=0.55, show_game=4,
 ↪  input_control=True):
  """ This function executes a series of competitive games between two players.

    Args:
      player1: the first Player
      player2: the second Player
      games: number of competitive games executed by the function
      threshold: a float, player 1 is considered stronger if its number of
        victories satisfy  wins / (total games - draws) > threshold
      show_game: how many games are showed in output
      input_control: if True the showing of a game can be decided contestually
        by the user
    Returns:
      a bool, True if player 1 is stronger, False otherwise
  """

  print("Duel: " + player1.name + " vs " + player2.name + ", " + str(games) + "
   ↪  games\n")
  game_rules = player1.game
  player1_score = [[0, 0], [0, 0], [0, 0]]  # black/white wins, draws, defeats
  player2_score = [[0, 0], [0, 0], [0, 0]]

  for i in range(games):
    player1.new_game()
    player2.new_game()
    black_player, white_player = (player1, player2) if i%2 else (player2, player1)

    state = game_rules.initial_state()
    winner = 0
    if show_game:
        rec_game = [state]
        stats = []

    while not winner:
      try:
        a1 = black_player.play(state)
        state = game_rules.next_state(state, a1)
```

```
37          winner = game_rules.game_phase(state)
38          if show_game:
39            rec_game.append(state)
40            stats.append(black_player.stats())
41          if not winner:
42            a2 = white_player.play(state)
43            state = game_rules.next_state(state, a2)
44            winner = game_rules.game_phase(state)
45            if show_game:
46              rec_game.append(state)
47              stats.append(white_player.stats())
48        except:
49          display_game(rec_game)
50          print("Error caught during this game")
51
52
53      if winner == 1:
54        if i % 2:
55          player1_score[2][0] += 1
56          player2_score[0][1] += 1
57        else:
58          player2_score[2][0] += 1
59          player1_score[0][1] += 1
60      elif winner == -1:
61        if i % 2:
62          player1_score[0][0] += 1
63          player2_score[2][1] += 1
64        else:
65          player2_score[0][0] += 1
66          player1_score[2][1] += 1
67      else:
68        if i % 2:
69          player1_score[1][0] += 1
70          player2_score[1][1] += 1
71        else:
72          player2_score[1][0] += 1
73          player1_score[1][1] += 1
74
75      if show_game:
76        stats.append((np.zeros(game_rules.action_size()), [999]))
77        if winner == 1:
78          winner_name = "White is the"
79        elif winner == -1:
80          winner_name = "Black is the"
81        else:
82          winner_name = "Draw, there is no"
83
84        print("Sample game. Black: " + black_player.name + "   White: " +
          ↪  white_player.name + "\t" + winner_name + " winner" )
85        try:
86          print_details = [("Value: " + str(np.around(value,decimals=3)),
87                            "\nPi: (pass action = " + str(pi[-1]) + ")",
88                            np.around(np.asarray(pi[:-1]).reshape(game_rules.n,
                              ↪  game_rules.n), decimals=2)) for pi, value in
                              ↪  stats]
89          display_game(rec_game, details=print_details)
90        except TypeError:
91          display_game(rec_game)
92        if input_control:
93          control = input("Show another game? ")
94          if control[0] != 'y':
95            show_game = 0
96        else:
97          show_game -= 1
```

```
98
99    print("FINAL SCORE\nPlayer | black/white wins, draws, defeats")
100   print(player1.name + ":", player1_score)
101   print(player2.name + ":", player2_score)
102   if (sum(player1_score[0]) / (games - sum(player1_score[1]) + 0.0001)) > threshold:
       ↪  # wins / (total games - draws) > threshold
103     return True
104   else:
105     return False
```

A.10 Games generation

```
available_triples = [f for f in listdir(ABSOLUTE_PATH) if f[0] == 'N']
available_triples.sort()
nnet_weights_file_name = available_triples.pop()
iteration_start = int(nnet_weights_file_name[8:11]) + 1
log_file_name = 'log' + nnet_weights_file_name[8:11] + '.txt'
print(nnet_weights_file_name + " will be loaded")
with open(ABSOLUTE_PATH + 'search_depth.txt', 'r') as f:
  simulations = int(f.read())

number_of_games = 200

positions_save_every = 8192
self_play_show_games = 1

game_rules = OthelloGame(n=8)
f_theta = DeepNeuralNetwork(game_rules, memory_boost=True,
 ↪  weights_file=ABSOLUTE_PATH + nnet_weights_file_name)
T = TrainingSet(save_every=positions_save_every)
```

```
import os.path
log_exist = False

for iteration in range(iteration_start, iteration_start + 1):

  print('')
  print('#'*50 + '#'*14 + '#'*50)
  print('#'*50 + ' ITERATION ', iteration, '#'*50)
  print('#'*50 + '#'*14 + '#'*50)

  # Self play
  print('')
  print('_'*51 + ' SELF PLAY ' + '_'*51)

  stats_fullgames = self_play(game_rules=game_rules, f_theta=f_theta,
   ↪  training_set=T, number_of_games=int(number_of_games * 0.7),
                                    simulations=simulations, random_seed=False,
                                     ↪  virtual_losses=True,
                                     ↪  iteration_tag=iteration,
                                     ↪  show_games=self_play_show_games)
  T.save(iteration)
  stats_resgames = self_play(game_rules=game_rules, f_theta=f_theta, training_set=T,
   ↪  number_of_games=int(number_of_games * 0.7), random_seed=False,
          simulations=simulations, iteration_tag=iteration, show_games=2,
           ↪  resignation_threshold_used=stats_fullgames["resignation_threshold"])
  T.save(iteration)

  if os.path.isfile(ABSOLUTE_PATH + log_file_name):
    log_exist = True
```

```
24    with open(ABSOLUTE_PATH + log_file_name, 'a+') as f:
25        if not log_exist:
26          f.write("number of games | white wins | black wins | draws |
             ↪  resignation_threshold | avg_number_of_moves | avg_larges_valuedrop |
             ↪  top ten crucial moves\n")
27        for stats in [stats_fullgames, stats_resgames]:
28          f.write(str(stats["games"]) + "\t" +
29                  str(stats["winner"][1]) + "\t" +
30                  str(stats["winner"][-1]) + "\t" +
31                  str(stats["winner"][2]) + "\t" +
32                  str(stats["resignation_threshold"]) + "\t" +
33                  str(stats["avg_number_of_moves"]) + "\t" +
34                  str(stats["avg_largest_valuedrop"]) + "\t" +
35
                     ↪  str(list(np.array(stats["move_causing_largest_valuedrop"]).argsort()[-10:][::-1]))
                     ↪  + '\n')
```

Bibliography

[1] C. E. Shannon, "XXII. Programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, Mar. 1950. DOI: 10.1080/14786445008521796.

[2] H. v. d. (J. Herik and J. Roerade, *Computerschaak, schaakwereld en kunstmatige intelligentie*. Academic Service, 1983.

[3] L. V. Allis, "Searching for Solutions in Games and Artificial Intelligence," PhD thesis, 1994. DOI: citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.99.5364.

[4] J. Schaeffer and J. v. d. Herik, *Chips challenging champions : games, computers and artificial intelligence*. Elsevier, 2002, p. 362.

[5] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2-3, pp. 273–289, Feb. 1992. DOI: 10.1016/0004-3702(92)90074-8.

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

[7] Buro Michael, "Methods for the evaluation of game positions using examples," PhD thesis, University of Paderborn, Germany, 1994. DOI: skatgame.net/mburo/ps/mics{_}dis.pdf.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," Dec. 2013.

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

[10] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.," *Science (New York, N.Y.)*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018. DOI: 10.1126/science.aar6404.

[11] M. Campbell, "Mastering board games," *Science*, vol. 362, no. 6419, pp. 1118–1118, Dec. 2018. DOI: 10.1126/science.aav1175.

[12] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "StarCraft II: A New Challenge for Reinforcement Learning," Aug. 2017.

[13] Sharpe Lynda, "So You Think You Know Why Animals Play...," *Scientific American Blog Network*, 2011. DOI: blogs.scientificamerican.com/guest-blog/so-you-think-you-know-why-animals-play/.

[14] A. A., M. Abdel, M. Gadallah, and H. El-Deeb, "A Comparative Study of Game Tree Searching Methods," *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 5, 2014. DOI: 10.14569/IJACSA.2014.050510.

[15] Romano Benedetto, "SAIO: un sistema esperto per il gioco dell'othello," PhD thesis, University of Naples Federico II, 2009. DOI: romanobenedetto.it/tesi.pdf.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. DOI: 10.1038/nature14539.

[17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017. DOI: 10.1038/nature24270.

[18] Y. A. Prasad Aditya Abrams Ishaya, "Lessons From Implementing AlphaZero," Tech. Rep., 2018. DOI: medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191.

[19] Foster David, "AlphaGo Zero Explained In One Diagram," *Applied Data Science – Medium*, DOI: /medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Dec. 2015.

[21] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Feb. 2015.

[22] Alpert Bill, "Artificial Intelligence's Winners and Losers," *Barron's*, 2017. DOI: barrons.com/articles/artificial-intelligences-winners-and-losers-1509761253.