Learning to Play Othello using Deep Reinforcement Learning and Self Play

Master Thesis

Thomas Steinmann

University of Bern

September 2018

Abstract

Reinforcement Learning enables us to tackle tasks that until recently have been beyond our reach, such as autonomous vehicles or interactive multi agent systems executing complex strategies. But before solving these tasks the underlying techniques have to be studied and matured. With their limited complexity and well defined interfaces, computer games provide the perfect training grounds for both scientists and their training algorithms. Two such games as well as multiple reinforcement learning players interacting with them are implemented in this work. The games are specifically chosen so that players can be trained using the limited resources available to students and private researchers. All players developed for this thesis are trained both against traditional, fixed opponents and using self play. None of the strategies rely on any expert knowledge other than the game's rules which guarantees that the learned strategies are not restricted or biased by the implementation. The experiments show that a policy with reasonable performance can be learned in both training modes. When playing Othello against the *Heuristic Player*, a reference player often used to evaluate Othello implementations and the best performing of the evaluation players used in this work, the best performing reinforcement learning player achieves win rates of approximately 75% and 50% when training against traditional opponents and using self play respectively.

Supervisor: Paolo Favaro, Computer Vision Group, Institute of Computer Science, University of Bern

Contents

1 Introduction

Board games have fascinated people of all ages for as long as we can remember. Ancient board games are in fact so old that they predate writing and most of their rules can only be guessed at today. In Egypt as well as Norse regions and later Europe, board games were played by nobles and kings and being good at playing them was a sign of status. Being a strong player implied intelligence, leadership or even the blessing of gods [9] [2]. It is no wonder that even today children and adults alike see board games as both a passionate past time as well as a way of training their wits.

The inherent fascination with games comes largely from the way they are constructed. Due to their materials and mediums, most games are restricted to having a limited set of rules as well as a well defined environment such as a board or a table on which they are played. The actions that can be taken in the game are equally restricted which makes them both challenging and fair because both players play by the same rules. Lastly games are simulations and simplified models of their real world counterpart. Many games have themes such as railroads, business deals or epic battles of medieval knights or futuristic space ships. Others are more abstract and present themselves as mental exercises to the players. The goal remains the same: to outsmart their opponents with clever strategy and tactics. They all bring something to our life that is usually beyond our reach by restricting themselves to a simplified representation and a limited set of interactions.

It is those very properties that make games interesting for the domain of Machine Learning. They can act as models of the real world. Simplified simulations of things that are yet too difficult to solve in the real world. They can act as training grounds for new techniques to be tested and matured before they are applied to the vastly more complex systems that are our daily life. The interactions with most games are deterministic and discrete, making it easy to predict how each possible action will influence the games state. Meanwhile the world is continuous and even if it is deterministic it would be far too complicated to understand in its entire complexity. Games

can serve as benchmarks in order to compare different algorithms and motivate developers to excel in them, resulting in even better algorithms. They also provide a safe environment for experimentation. If a Machine Learning agent makes a mistake playing TicTacToe it will lose the game and learn from the experience. In the real world a simple error can result in large financial losses or serious injuries if a stock exchange agent wrongly predicts future values or an autonomous car does not recognize a pedestrian.

As an example let us investigate three recent major breakthroughs. Firstly, Alpha Go defeating the reigning world champion of the ancient game of Go is one of the most influential recent works on modern Reinforcement Learning. The Alpha Go team showed the world the potential of what can be achieved using these new and exciting techniques. When Alpha Go started defeating well known Go professionals the community was in disbelief. Many Go players were certain that a machine could not beat a professional human player and would not be able to do so for several decades. They argued that Go was an art form and required intuition rather than calculations. They even questioned if the three times European Go champion Fan Hui simply had been away from China for too long and therefore was not sufficiently connected to the game anymore after he lost against the first iteration of Alpha Go. When asked why they played a certain move, players would often say it just felt right, not able to state an exact reason. How could a machine imitate human intuition. Traditional game theory never reached more than an average amateur level due to the complexity of the game, after all Go is said to have more possible legal game states than the universe has atoms. During a show match against the reigning world champion and one of the most famous professional Go players in the world Lee Sedol, both the commentators as well as other players were speechless as Alpha Go played a move no human would ever have thought of playing. It even recognized as much but still decided on playing it, showing that it had learned something that went beyond what it could have copied from anyone or anything else and instead created something unique. Alpha Go beat Lee Sedol four to one but even more impressively it is said that the way it had played revolutionized how humans played the game. The community accepted Alpha Go as something remarkable and now strive to learn from it [1].

OpenAI Five follows a similar approach but in an arguably even more complex scenario. They train five separate agents both independently and as a team to play the popular online multiplayer game DOTA 2 in which strategy and teamwork are crucial. A single game of DOTA lasts an average of 45 minutes, making it very difficult to trace back which actions had a beneficial effect towards the outcome of a game. Teamwork between agents is another skill that was considered to favour humans over machines. All the same, OpenAI Five beat a very strong amateur team two to zero before going on to play at the International, a major professional DOTA tournament, where it lost both matches against professional teams in full DOTA without any restrictions that simplified the game. OpenAI Five has learned many concepts from scratch that are well known within the DOTA community. Since it has only ever trained against itself it must have redeveloped these concepts on its own and sometimes learned strategies that caught human players off guard. Strategies and concepts like this include relatively straight forward ones such as focusing abilities on a single target in order to score a fast and easy reward. Others are a lot more difficult or controversial, such as baiting an opponent into a trap or sacrificing a player in order for the rest of the team to score a more valuable objective. OpenAI concludes that the understanding and performance of their system is a clear step towards AI systems that can handle the complexity of

the real world. The training algorithm used for OpenAI Five, Proximal Policy Optimization [7], is considered one of the most stable to date and was developed for application in games but is being used for various other applications as well [17] [18]. The OpenAI team has since reported major improvements to their players and is challenging strong teams for show matches again.

Alpha Go and OpenAI Five are very impressive and not only mark a great achievement for Reinforcement Learning as a domain but also made people realize its potential. However it is still difficult to see how their success affects everyday life. This changes fundamentally when looking at our next example, a Google spin off called Waymo. Waymo builds autonomous cars and has already deployed them as a taxi service in Arizona, USA. While their cars are ultimately deployed into the real world and interact with very real people, their journey starts in a sort of game. Their agent is trained mostly from simulations, based on both real and dynamically generated driving situations. In contrast to most games this one is modelled very carefully to represent the task of driving in traffic and interacting with other road users. While it is not a game most humans would find interesting to play on a Sunday night, it provides the same benefits other games inherently have, resulting in the perfect environment for autonomous car agents to learn. That is, the simulation and its parameters as well as the complexity of the problem can be adjusted exactly to the developers needs. The number of experiences generated surpasses by far what could be experienced in the real world. And the risk to other road users is nonexistent in the simulation. Once the agent has progressed far enough so that it is safe to release it onto the streets, training is continued with both simulated data as well as data gained in the real world to continually improve the agents understanding. [21] [22] [20].

With Alpha Go's and OpenAI Five's impressive successes as well as fast progress towards fully autonomous cars by Waymo among others, it is clear that Reinforcement Learning offers a solution to many problems that could not be tackled with more traditional techniques. Many of these powerful implementations rely on equally powerful machines in order to train them, often requiring hundreds of CPUs and GPUs for weeks on end. Inspired by these grand achievements and the technology behind them but lacking comparable resources we settled on a goal that is not only suitable for large companies but can be reached using resources most students would have access to: Self Play Reinforcement Learning for General Board Games. In particular the two games TicTacToe and Othello struck a particular interest because they consist of similar board layouts and action spaces. Moreover the only legal actions for both games is to place a stone on the board, so encoding their rules is straight forward. Othello has accompanied the authors since the beginning of their studies in computer science when they, as many others before and since them, were tasked to implement a search based Othello player. During their master studies they suspected that a superior player could be created using machine learning techniques. This thesis documents how such a player can be built and highlights its key components. It is our hope that this work can serve as a platform for students and enthusiasts to build their own players and use it to experiment with reinforcement learning in a similar fashion to the projects that have inspired it.

2
Introduction to Othello

Modern Othello's invention was credited to Goro Hasegawa who published the game in Japan in 1973. It is named after the Shakespearean play "Othello" in which a black general and his white wife are entrapped in intrigue and jealousy. The stones placed by players of the game are also black and white in order to reflect this. The green background signifies general Othello's battlefield [10], [15], [16]. Othello bears a strong resemblance to an older British game called Reversi the origin of which is still disputed. Reversi rules are basically the same as Othello with slightly different rules concerning the starting position.

Othello is played on a square board, usually made up of eight by eight tiles [3]. The opening position is shown in Figure 2.1 As mentioned before, Othello stones are black on one side and white on the other. Players take turns in placing one of these stones in their respective color on the board. Stones can only be placed in such a way that the new stone *traps* one or more of the opponents stones between itself and any other stone of the same color. All opposing stones that were trapped by placing the new stone in this way are flipped and now belong to the player who trapped them. If a player cannot perform a legal move he simply passes his turn and his opponent places the next stone. The game ends if both players pass successively or there are no free tiles left on the board. The player who controls more stones when the game ends is declared the winner.

Most games simply run until the board is full, giving each player 30 moves in a game. Some games may end sooner when both players run out of legal moves. Others might take slightly longer when one player has to pass and therefore the other gets to play more moves because there are more free spaces available on the board. 30 moves per player is a very good estimate for game length, or depth, however. The estimation of legal moves, the game's breadth, is more complicated. While for the first move there are always four legal moves per player and there can never be more legal moves than free tiles, there can be up to 20 legal moves in the mid game. It is estimated that the average breadth is around 10 moves, based on an average of tournament games.

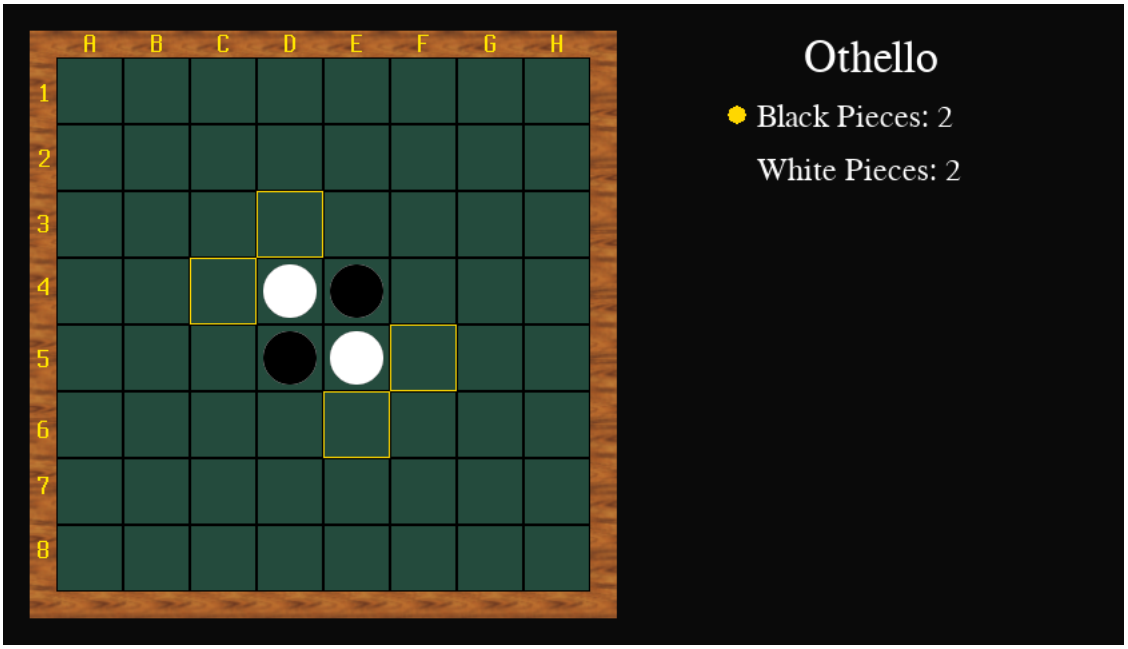

Figure 2.1: Othello opening position. Black traditionally moves first, choosing between three highlighted legal moves each trapping one white stone.

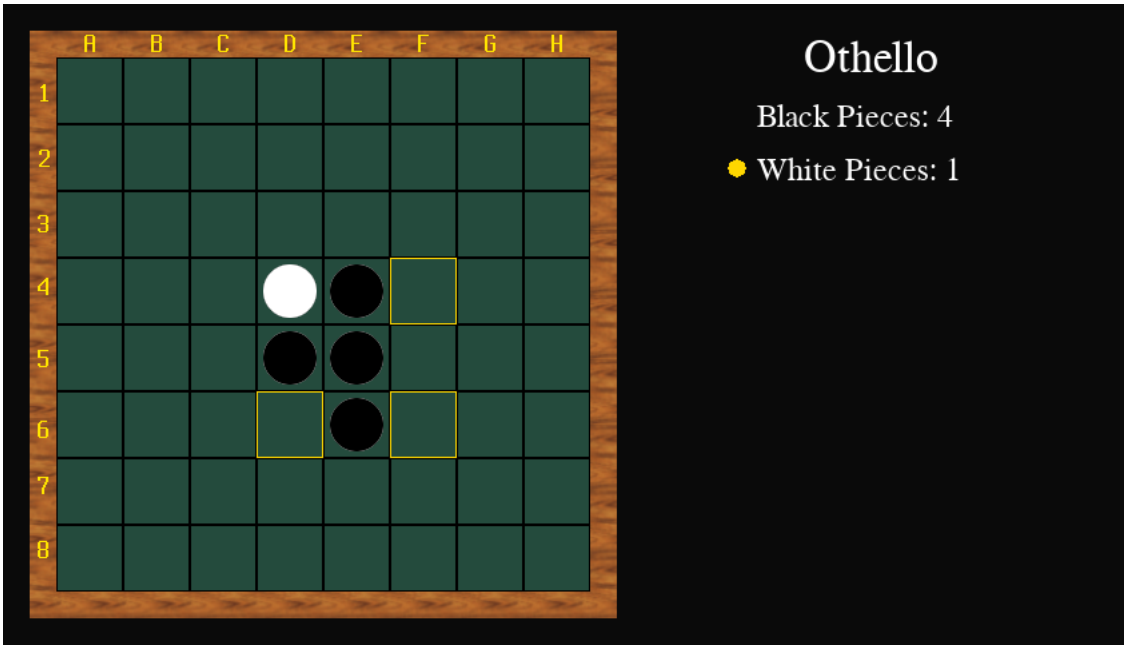

Figure 2.2: This state results from black playing its first move on **F5** which traps whites stone on **E5**

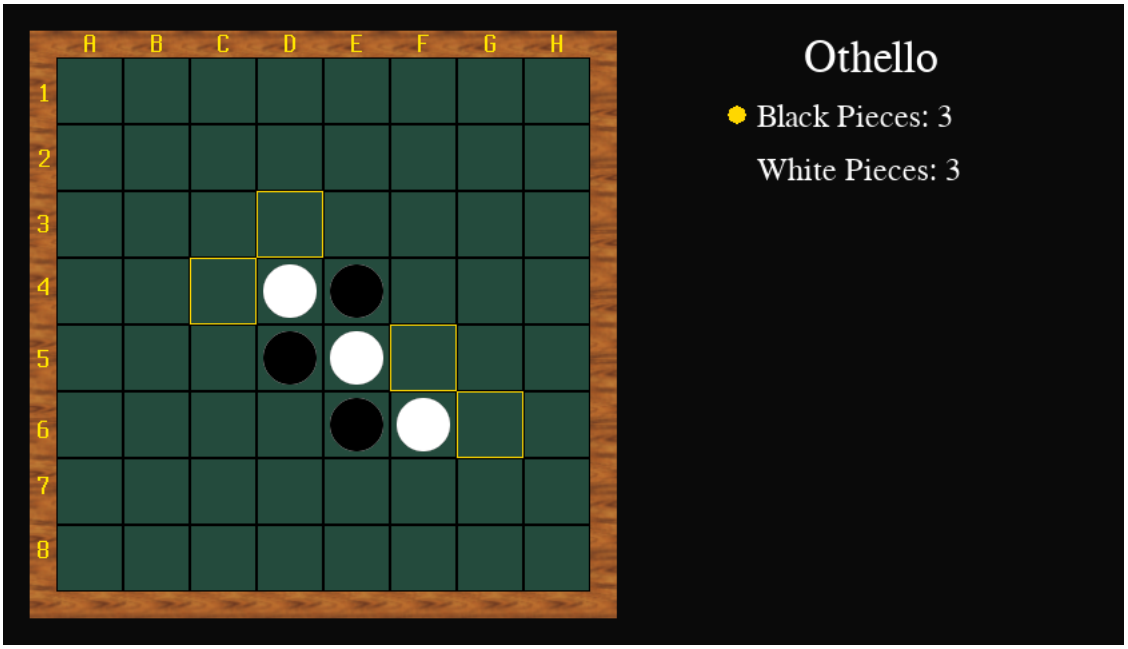

Figure 2.3: White played its stone on **F6** and trapping **E5** again.

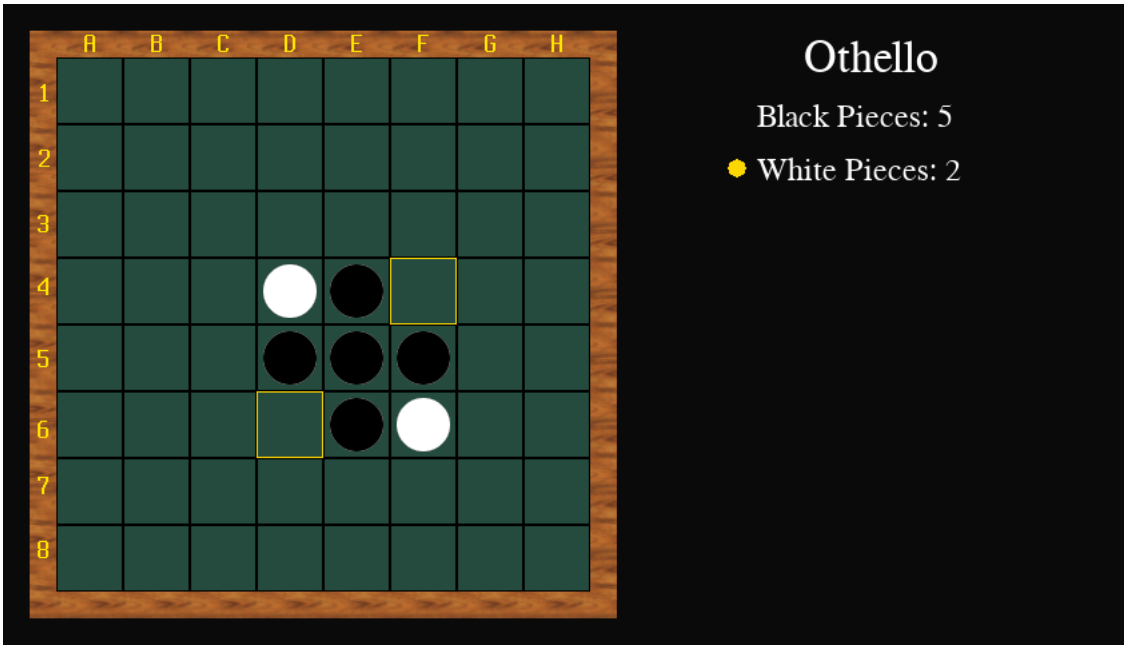

Figure 2.4: Black played **F5** and trapped **E5**

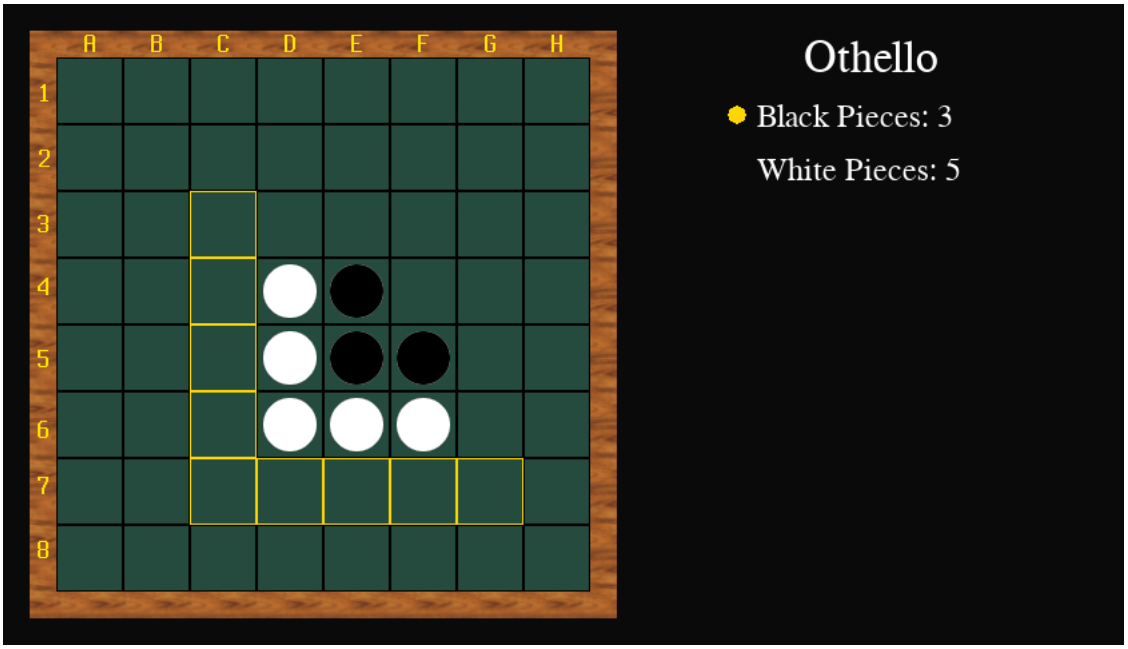

Figure 2.5: White played **D6** and trapped both **D5** and **E6** between the newly placed **D6** and **D4** and **F6** respectively. All trapped stones have to be flipped.

Figures 2.1 to 2.5 show the first four moves of a regular Othello game. Legal moves for the current player are highlighted in yellow. Tiles are referenced using letters **A-H** horizontally and numbers **0-8** vertically, same as for chess boards.

2.1 Othello Players

2.1.1 Heuristic Othello Player

The player we call the *Heuristic* player in this thesis is often used for benchmarking because of its decent performance, low computational cost and deterministic nature. It utilises a heuristic table which assigns a value to each tile on the board. These values are symmetrical for eight axes through the board. The table gives high values to tiles such as the edges that are of high value because they lead to stones that cannot be captured later in the game. Tiles around the edges on the other hand receive a low value because occupying one of them may allow the opponent to occupy the adjacent corner.

2.2 Search based Algorithms

Othello is widely used to teach search based game theory algorithms, most notably the min-max algorithm and its optimization, the alpha-beta pruning. [14] The general idea of these methods is to build a search tree of all relevant moves for both players and the resulting game states. Ideally

a complete graph for such a game could be computed and a computer player would just choose a move that results in a win for every single turn. However the search space for most games is so big that such a graph is not feasible with the computational resources available. The ancient Chinese game of Go for example is said to have more possible game states than there are atoms in the universe. The challenge lies therefore in approximating a complete graph as close as possible and with maximal efficiency. For this the search tree is usually truncated at a certain depth or after a given time has passed and the rest of the tree is approximated with a so called value function that evaluates the value of the state where the tree stopped. This value function traditionally consists of a set of given features such as possible moves for each player, save stones that cannot be captured again or just greedily the number of stones a player controls. Search based algorithms can also be used to improve the *Heuristic Player* described in Section 2.1.1. However, in most search based algorithms this would make the player non deterministic.

2.3 Machine Learning based Algorithms

Machine learning techniques for Othello are related to search based algorithms but often do not perform a search at play time. A straight forward approach is to improve a search based algorithm by learning its value function. [4] used machine learning to learn his own heuristics function, similar to the one used by the *HeuristicPlayer*, in his alpha-beta search algorithm. A more advanced technique basically moves the search from play time to training time by simulating thousands or millions of games and uses them to train an agent. This agent learns a value function that captures the knowledge gained during the search and predicts the value of a possible move without having to perform another search at play time. Such an approach was used by [6] to master Go. They went one step further still and used training time search in order to train an agent and then improved that agents decision again with another search at play time.

2.4 TicTacToe

TicTacToe is a popular mini game played by all ages. It consists of a square board with side length three. Players take turns in putting marks on free spaces of the board until a player has placed three marks in a straight line and wins the game. If no free spaces are available both players draw. TicTacToe shares many characteristics with Othello such as the form of the board and the action space where players can only place stones on free tiles rather than move them as in other board games. Due to its much smaller complexity and similarity to Othello it is uniquely qualified to serve as a proof of concept for techniques and architectures before applying them to Othello.

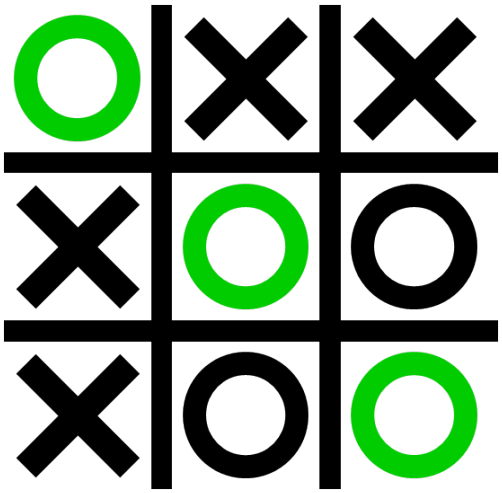

Figure 2.6: TicTacToe board

3
Introduction to Reinforcement Learning

Reinforcement Learning leverages special algorithms to train an agent on a task while performing it at the same time. Generally Reinforcement Learning can be seen as a special case of *supervised learning* where both the *training samples* and their respective labels, called *rewards*, are dynamically generated by the environment. The environment contains expert knowledge on the target task. Using the reward, the reinforcement agent reinforces actions that lead to positive rewards while discouraging actions that lead to negative rewards, hence the name Reinforcement Learning.

Due to the ability to learn from a task for which there is no fixed training set, Reinforcement Learning has the potential to solve very difficult tasks that would not be solvable using either supervised or unsupervised learning. This applies mostly for tasks where the action space is very large and sufficient training data is hard to come by. Instead of a fixed training set, Reinforcement Learning methods rely on an environment which takes an action and returns a reward whenever one is available. The environment is a crucial part of the learning system as it defines the task to be learned and separates it from the inner workings of the agent which it can control. That said, it is often significantly simpler to dynamically generate examples for expected behaviour using expert knowledge than to create a program which behaves in such a way. Ideally a reward would be generated for every action selected by the agent, resulting in a situation very similar to regular supervised learning. However, depending on the task a reward may not be available after every action. We call these rewards *delayed*. Rewards are often delayed until a certain goal is accomplished or a clearly unwanted situation arises. In board games the reward often has to be delayed until the end of the game when a winner can be determined. Assigning an unbiased reward during the game is impossible since judging how well a game is going is dependent on the strategies applied by either player. Consider Figure 3.1 where both players are winning according to their strategy. Introducing a predefined reward would therefore restrict the strategy. How the delayed reward is distributed over all training samples is a defining factor between different

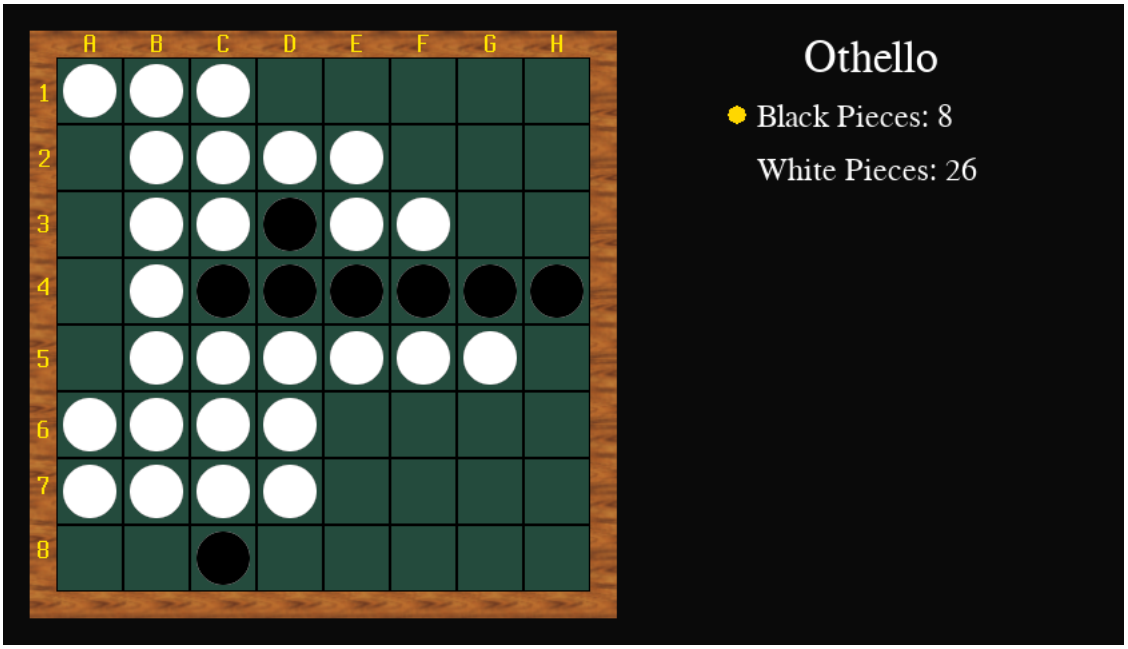

Figure 3.1: White follows a greedy strategy, trying to keep the stone count difference as high as possible in its favour, by which it is clearly winning with 26 to 8 stones. Black on the other hand is playing a mobility based strategy. It tries to maximize the difference of legal moves. According to this strategy Black is clearly winning with 14 legal moves while White only has 3. Without defining the strategy by which to judge it is impossible to assign an unbiased reward.

Reinforcement Learning methods.

Reinforcement Learning methods are made up of two main components, the *policy function* and the *state value function*. The *policy function* chooses which action to take in the next time step while the *value function* assigns a value to a given board state, rating how well the board state reflects the given policy. The agent is trained in a constant cycle, first updating the policy then adjusting the value function accordingly or vice versa. In theory either function can be derived from the other. Given a policy, the perfect value function assigns values according to how well the agent has been following the strategy. Given a value function, the according strategy chooses actions in order to maximize the resulting values. Therefore depending on the learning algorithm either the Policy function, the value function or both can be implemented and trained as function approximators.

There are two families of Reinforcement Learning algorithms relevant to this thesis: *Value Function* methods, sometimes called *Monte Carlo* methods, and *Policy Gradient* methods. Value Function methods train a value function which assigns values to *game states* or *(game state, action)* pairs. In the domain of board games a game state usually describes the positions of pieces on and off the board as well as potentially some history of moves if necessary. An *action* is any interaction a player has with the environment other than receiving rewards. Action examples are making a move, passing or conceding a game. After predicting a value for each action using

the value function, actions can be ranked according to those values. The Policy function is not learned but chooses an action according to the list ranked by the value function. Policy Gradient methods on the other hand forego explicit assignment of values and instead train their Policy function to predict a vector containing the probability of taking each action. Actions can then be sampled according to these probabilities. Policy Gradient methods can work without a value function or use it in order to improve the loss values.

All methods used in this thesis are *on policy* methods and *episodic*, meaning they optimize the same policy that is used to make decisions on which action to take and wait for a reward signal from the environment before updating the policy.

3.1 Notation

For the remainder of this thesis let $board(t)$ be the board state at time t and $a(i)$ the i-th valid action, such as setting a stone, moving a piece, etc. t is measured in moves, therefore $board(0)$ indicates the board state before the first move is made. P_a denotes the probability of taking action a, also called the *action probability*. $reward_t$ and $reward_a$ are the reward received for the sample at time t and for taking action a respectively. $statevalue_t$ denotes the value assigned to the game state at time step t using a *value function*. An *episode* denotes all actions between two rewards. For board games this usually means all moves of a single game before a winner can be determined.

3.2 Exploration vs Exploitation

A major concern when applying Reinforcement Learning techniques is the balance between *exploration* and *exploitation* during training. The term *exploration* describes an agents need to explore new states and explore unseen or unfamiliar situations in order to learn from them. Exploitation on the other hand allows the agent to apply previously learned knowledge and perform better at the given task. This is important to limit the search space, progress in the task and allow the agent to reach new game states to explore in the first place. After training, an agent should minimise Exploration and maximise Exploitation in order to maximise performance. During training however, a certain amount of exploration is necessary in order to allow the policy to be updated and improved. The amount of exploration needed, as well as the manner of how it is achieved, depends strongly on the learning method.

3.3 Reward discounting

Delayed rewards lead to a problem where most training samples do not have a reward signal. Their reward is usually set to zero during the simulation of an episode. In order to update the network, each state must have a reward/loss/label. A reward of zero would lead to a zero gradient and no change in policy for that specific sample. Reward discounting solves this problem by applying the final reward to all game states of the episode. However it assumes that the further away an action is from the end of the episode, the smaller its contribution to the final reward. At

the same time game states later in the episode are visited less often, hence their updates must be weighed greater to compensate for this fact. Thus the reward is propagated backwards through the episode, reducing it for every game state. The discounting function is a hyper parameter and needs to be determined experimentally.

3.4 Bootstrapping

Bootstrapping is an alternative to reward discounting. A *bootstrapping* algorithm bases its updates on intermediary predictions rather than on a final reward. It therefore does not have to wait for the final reward before being updated. Additionally, every prediction is already based on the game state and therefore does not have to be discounted further. The simplest bootstrapping methods are based on Dynamic Programming. There a table of predictions is filled one field at a time. Every prediction is based on parts of the table that have already been filled in by previous predictions. A similar concept can be used for function approximator methods such as Value Function or Policy Gradient methods. Instead of table values, the prediction of previous state values are used to bootstrap the policy. Bootstrapping generally reduces variance but introduces a bias.

3.5 Value Function Methods

Value Function methods predict a value given an input state and sometimes also an input action. This means a value function is a function $board(t), a(i) \rightarrow value$. In order to build a policy a player samples an action according to their $value$ after evaluating $board(t), a(i) \rightarrow value, \forall\, i$. A *greedy* player selects the action resulting in the highest value after applying it to the current board state. The *greedy* policy is the base of most Value Function policies. In order to force the agent to explore the state action space it is usually modified with a parameter ε which determines the possibility of performing a random action instead of the greedy one during training. This is commonly referred to as the ε-greedy policy. ε is a hyper parameter and must be determined during evaluation.

3.5.1 Monte Carlo Learning

Monte Carlo Learning is the basic implementation of a Value Function method. It leverages a value function in order to predict state values according to which an action is selected. The game states are saved for use as training samples. The delayed reward is discounted as explained in Section 3.3. At the end of an episode the value function is updated using each pair of game states and discounted rewards. Monte Carlo Learning is unbiased as it only relies on the final reward but generally suffers from high variance.

3.5.2 Temporal Difference Learning

The Temporal Difference (TD) algorithm is a variant of the basic Monte Carlo algorithm described above. In contrast to its predecessor it uses a form of bootstrapping rather than discounting the

final reward. This can be better understood when analysing the algorithm from the end of an episode, which is called the *backward view*. The final reward is still the result of the game just like in the basic Monte Carlo algorithm. The rewards of all other states $state(n)$ however are computed based on the state values $value(n)$ and $value(n+1)$ computed from their respective game states. Intuitively this assigns each $(state, action)$ pair the difference in state value that the input action facilitates.

Since TD Learning is a bootstrapping method it has a lower variance than Monte Carlo Learning but introduces a bias since each reward is based on prior predictions.

3.6 Policy Gradient Methods

Policy Gradient methods forego explicit scoring of board states and actions and instead directly predict the likelihood of taking a certain action given a board state. They therefore model a function $board(t) \rightarrow P$ where P is a vector and P_i is the likelihood of taking action i. Intuitively P_i is related to the likelihood of gaining a positive reward if taking action i. Exploration is naturally built into Policy Gradient methods since their output is a distribution. Exploration is guaranteed as long as every valid move is assigned a positive value.

3.6.1 Reinforce Learning

The Reinforce algorithm [23] is the base implementation of Policy Gradient methods and serves as the Policy Gradient equivalent of Monte Carlo Learning. It trains a Policy function to take a game state as an input and predict a list of action probabilities. The next move is selected by sampling from the action probabilities outputted by the policy function. It does not require a Value Function. Just like Monte Carlo Learning (Section 3.5.1) it uses reward discounting and updates its policy at the end of each episode.

3.6.2 Baseline Methods

When passing a very beneficial game state to a Reinforce Learning algorithm the action probabilities are likely to all be high because even when choosing a bad action, the probability of gaining a positive reward is still high. If the final reward is positive, the reward for this particular bad action is still positive and taking it would be reinforced positively. In order to prevent this, a baseline is subtracted from the reward of each action. This baseline can be a constant or a function depending on the game state, as long as it is independent of the action. A widely used baseline is the Value function already known from Monte Carlo methods. After subtracting the baseline each action should be given a reward independent of the value of the state only depending on how much it improves the game state and therefore the probability of gaining a positive reward.

3.6.3 Actor Critic Methods

Actor Critic methods further build on Baseline methods and represent the Policy Gradient equivalent of TD Learning as they leverage the same bootstrapping principles. Simply speaking they make use of an action dependent baseline using a similar architecture as baseline methods.

More explicitly, they use a separate network or network head to predict the difference in state value an action will result in and subtract this value from the reward. This introduces a bias but further reduces variance and often leads to faster convergence.

4 Related Work

4.1 Othello

This section highlights related work on the topic of applying Machine Learning techniques to various components of Othello players. This is supposed to not only give perspective on where this thesis fits in the research domain but also introduce influential contributions that this work is based on.

4.1.1 An Intelligent Othello Player combining Machine Learning and Game Specific Heuristics

The player in [4] checks for various high impact situations called *exploits*: A move that will win the game, occupy a corner or prevent the opponent from having a valid move the next turn will always be taken immediately. The player also keeps a blacklist of moves that would allow the opponent to make an equally good move on the next turn and prevents itself from taking one of those moves.

If none of those exploits are possible the player searches the current and future action space using Minimax. The center piece of any Minimax implementation is the evaluation function or state value function. It assigns each game state a score using which it can be compared to other game states. Rather than using a hand crafted state value function, [4] leverages genetic algorithms in order to learn a heuristic table as explained in Section 2.1.1 to use as the state value function.

4.1.2 Playing Othello by Deep Learning Neural Network

No expert knowledge was used in [11]. Instead they collected a training set of almost 1200 games from various sources. They trained a custom state value function using supervised learning

and used it as an evaluation function for Minimax and Alpha-Beta pruning searching. During evaluation they realized that their player tended to overfit due to their small training set relative to the vast action space of Othello.

4.1.3 Reinforcement Learning in the Game of Othello: Learning against a Fixed Opponent and Learning from Self-Play

Three different reinforcement learning algorithms are implemented and analysed in [19]: Temporal Difference and its two extensions Sarsa and Q-Learning. All of them belong to the family of Monte Carlo Learning methods introduced in Section 3.5.1. They also use three different opponents: Two variations of the deterministic Heuristic Player explained in Section 2.1.1 as well as a Random Player. Finally they evaluate the difference between training their agents against the target opponent and self play. They achieve strong performance when training against the target opponent and reasonably strong results when relying on self play. The network architecture is not mentioned in the paper.

4.1.4 Systematic N-tuple Networks for Position Evaluation

Jaśkowski's [8] work relies on recognizing patterns called n-tuples. His best performing player uses all possible 2-tuples - two horizontally, vertically or diagonally neighbouring tiles - on an Othello board. He learns weights for these 2-tuples using coevolutionary methods and achieves a 96% win rate against the Heuristic Player when playing ε-Othello. In ε-Othello players are forced to take a random move with probability $p = 0.1$ in order to make games non deterministic. He concludes that 2-tuples work best, mostly because larger tuples rapidly increase the search space which cannot be searched efficiently using evolutionary methods and available resources.

4.1.5 Othello by rgruener

This Python based Othello framework provides a game board, a list of players and a game class organizing it all together. The strategy resides in the code of the players and they can be swapped in and out freely. It also provides a Graphical interface and allows human players to play against the computer players.

4.2 Alpha Go

Alpha Go has had a huge impact on Reinforcement Learning and Deep Learning as a whole. Their games against Lee Sedol in March 2016 brought the eyes of the world to this new promising technology when they beat the reigning World Champion in 4 out of 5 Games and astonished experts with a completely new move that no human has ever considered worth playing before. Alpha Go evolved into what it is over several years and was documented in two separate papers [5] and [6]. Both of which were influential to this thesis.

4.2.1 Alpha Go

In [5], Deepmind trained their player in three steps. First they trained a Policy Network to predict human expert moves using supervised learning and a data set of human expert moves. They call this network the SL network. In the second step the SL network was trained using the Reinforce algorithm and playing against older versions of itself to create what they called the RL network. The only difference between the two is the further improved weights of the RL network. In the a Value Network is trained to predict a board states likelihood of winning using the same Reinforce algorithm. The value assigned to a position depends on the strategy and is therefore based on the RL network.

Using only the RL network Alpha Go already beat the previous best performing players consistently. In order to improve its performance both networks are combined to control a Monte Carlo Tree Search, a search based method that searches only the most relevant parts of the action space by reducing the breadth of the search using the RL network and truncating the depth using the Value network. The first major version of Alpha Go is therefore a mix between a search based and a Machine Learning player as the learned components improve the action search. The encoding of the game state also includes some expert knowledge.

4.2.2 Alpha Go Zero

In [6] Alpha Go combined the Policy and Value functions into one network using two output heads. This way the learned representation can be shared between both tasks. This implementation also does not rely on supervised training or multiple training phases but is trained only using self play. MCTS is used in a unique and clever way to improve performance. Rather than applying it after training and only using it for move selection, Alpha Go leverages the power of MCTS for move selection during training. The Policy function is then trained to predict the output of the MCTS. Meanwhile the Value function is updated to reflect the Policy function and lead to better MCTS results during the next game.

4.3 Simple statistic Gradient-Following Algorithms for Connectionist Reinforcement Learning

Williams describes a family of learning algorithms called REINFORCE, an acronym for *REward Increment = Nonnegative Factor times Offset Reinforcement times Characteristic Eligibility* which describes the form of the algorithm. REINFORCE algorithms, nowadays also called policy gradient methods, directly learn to optimize a policy function from a given reward. Formally they search the weight space for a combination of weights w so that $E(r|w)$ is maximal, where r is the reward allotted by the environment. Williams concludes that these algorithms are useful in their own way but even more importantly, they could serve as a foundation on which to build more powerful algorithms [23].

4.4 Discussion of Prior Work

Most prior Othello implementations focus either on supervised learning from a relatively small data set or use reinforcement learning to improve single components of their search or heuristic based solutions. All of them require some form of expert knowledge. In order to not restrict the learned strategy during training it would be ideal not to incorporate any expert knowledge or train against predefined players. The only way of achieving this is a system relying fully on self play.

Interestingly Alpha Go made a similar transition between their two main implementations. [5] uses expert knowledge in the board encoding and the supervised learning step and leveraging MCTS. Two years later [6] does not rely on either of those and trains only against older versions of itself. The second paper strongly encouraged us to further pursue our current approach when it was released.

5
Learning to play Othello

When building a system in order to train any reinforcement agent there are four main components that need to be considered. Some implementations may combine some or all of them into a few files or classes but in this thesis they will be introduced, discussed and implemented separately. These components are the *Framework* which allows for multiple agents and environments to be organized and experiments to be run, the *Environment* which allows for agents to interact with, receive observations and rewards and finally the *Agent* which learns to interact with the environment in a way that optimizes the reward it receives. The last components are the *Experiments* which combine an environment and one or more agents. Experiments can be used to train, evaluate or analyze agents in the respective environments.

5.1 Framework

The framework provides a base structure for the project and organizes agents, environments and experiments. The framework's most important qualities are its structure and modularity. They allow for easy understanding and future expansion such as adding new environments, agents and experiments. For this purpose the framework defines interfaces for these three child components. It also handles repeating tasks such as documenting results, which is used by all experiments regardless of what agents or environments are used. Finally the framework is itself expandable, allowing for future improvement and additional features. Figure 5.1 shows how Agents, Environments and Experiments are all independent of each other and the framework serves as an organizational structure.

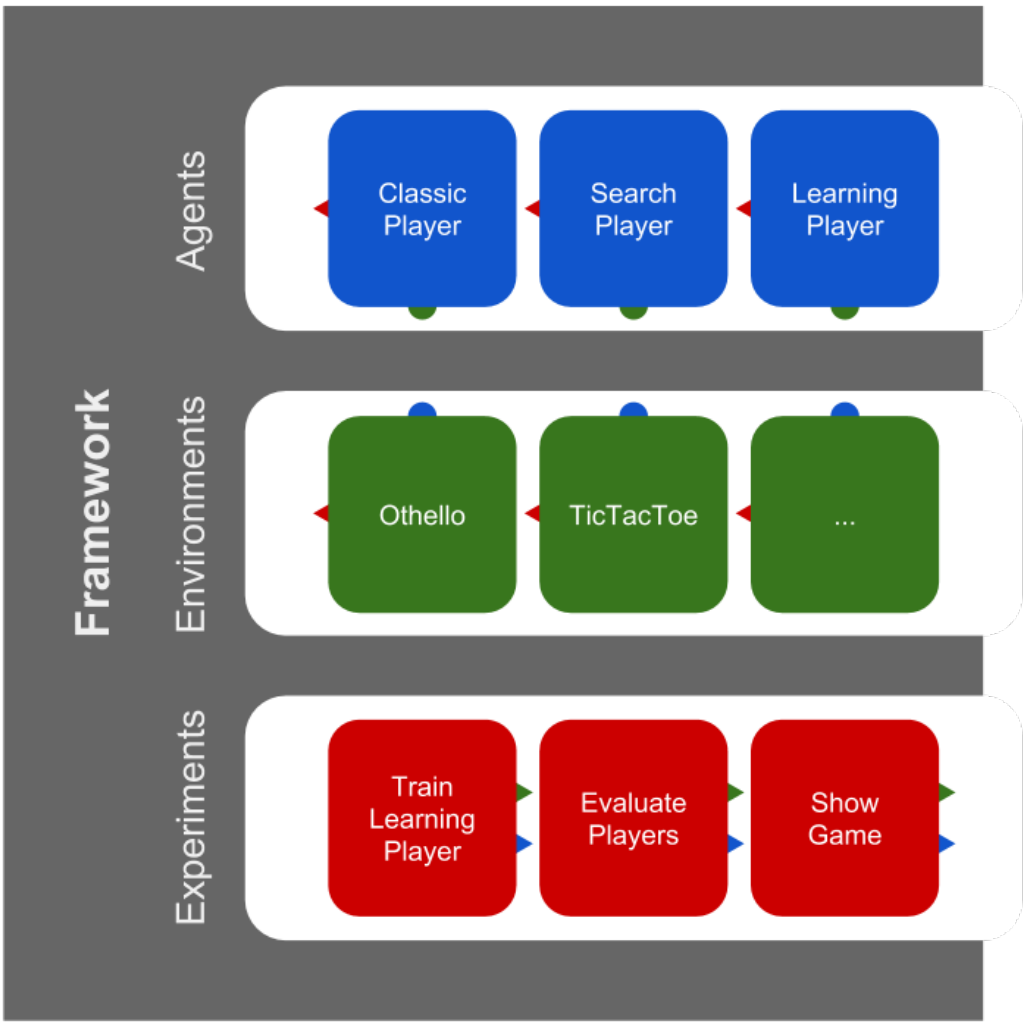

Figure 5.1: The framework serves as an organizational structure containing multiple agents, environments and experiments. Each one has well defined interfaces depicted as colorful tips. The color indicates which components they interact with.

5.2 Environment

When learning to play Othello, the environment an agent should interact with is the game of Othello itself. We therefore also call the environment the game. The environment encodes and enforces the rules of the game and provides an interface for the agents to interact with. Two environments will be implemented in the scope of this thesis, Othello and TicTacToe. Both are *complete knowledge*, *zero sum* games with *delayed rewards*. *Complete knowledge* means there is no hidden information and both players have the same information about the game at all times. Both games are also *episodic*, meaning the decision they have to make does not rely on past

moves or any other historic data but only the current game state. A *zero sum* game is one where any advantage for one player results in a disadvantage for the other player(s). The only unbiased reward for both games is the final result of the game. Since there is no obvious metric that rates the performance of a player independent of its strategy, any other reward would introduce a bias towards certain strategies and would effectively limit the player in discovering novel strategies. Because the reward is only available at the end of the game we say these games have a *delayed reward*.

The first game to be implemented is TicTacToe. It is a rather simple game due to its small board size of 9 tiles and the resulting small number of valid moves called the action space. With a maximal game length of 9 moves between both players, an episode is very short and rewards are not far delayed. TicTacToe can therefore be used as a proof of concept by developing both a working environment as well as an agent which are both simple to debug and able to be trained in a relatively short time with readily available resources. Othello is much more complex in all of the above mentioned ways. Its board size is $8x8 = 64$ and thus about seven times as big as TicTacToe's. This results in a much larger action space. The same is true about its episode length and how long the reward is delayed. While TicTacToe has been mathematically solved a long time ago and the optimal strategy is relatively simple, most Othello implementations still rely on search based algorithms and board value estimations based on hand crafted features.

5.3 Agent

The agent is at the core of the implementation. It can represent any kind of strategy such as one step heuristic evaluations, search algorithm based implementations, learned agents using neural networks, etc. In this work we call the agent *player* since they interact with a game. In the case of a machine learning player this is where training data is acquired, stored and the network is trained. The player is a self contained module that interacts with the framework, the game and the experiments using well defined interfaces. These interfaces are well defined and minimal in their extent. This allows for great independence and freedom of implementation in how to determine its next move. Additional players can be built and added to the project by other contributors.

5.4 Experiment

The experiment defines how the players and environments are used in order to achieve a specified goal. A simple experiment may only let two players compete against each other in a predefined game and display the results. An experiment like this may be helpful when analyzing players or serve as a simple training experiment. A more complex experiment may sample an opponent from multiple players, train against the selected one and then evaluate against all of them in order to analyze generalization. At the same time the experiment serves as documentation for how the experiment was performed and what hyper parameters were used.

5.5 Challenges

Othello, like most other board games, presents two major challenges in the form of delayed rewards, which are discussed in Chapter 3, and dynamic environments. Rewards are delayed because an unbiased estimate of the game state is impossible to provide for the environment as this estimate is directly related to the strategy. The only unbiased reward the environment can give is therefore the result at the end of the game. The problem then becomes how to assign credit for the final reward to each training prediction. Several approaches to this are introduced in Chapter 3 and implemented in this thesis. The second challenges, dynamic environments, describe the fact that the the task is constantly changing. The only way to avoid this would be to play against a single fixed opponent for the entire duration of the training. This is not an interesting experiment in many cases as generalization from it would be minimal. Usually one of several players is sampled randomly when training against fixed opponents or, e.g. in self play, the opponent constantly changes dynamically with the player. While more difficult to adjust to, these dynamic environments force the player to generalize rather than learn to only solve a very narrow task.

6 Implementation

6.1 Framework

The framework is an integral part of the application and makes up a large part of the code base. After some research we decided on using [13] as a base framework. It allowed both human and computer players to compete against each other. Each player object contained its own strategy which allowed for experiments with a large amount of different approaches. The included computer player was implemented using an alpha-beta search and evaluating positions on the basis of several hand crafted heuristics. This was useful as these search based players represent a very well performing baseline. First the framework was extended to allow many games in short succession and improved overall performance which is vital due to the high computational costs given the amount of games played. Another vital component is documentation and monitoring which was added in form of several stats and plots kept by a Plotter class. Finally experiments must be documented and run given specific parameters depending on their purpose. For this purpose an *Experiment* class was introduced which wraps the basic framework and passes the required parameters to the game and players before the experiment begins. Experiments are executable scripts which simplifies running and managing different experiments in parallel. During the course of this work the framework was continuously changed and does not share much resemblance with the original other than the basic structure. Figure 6.1 highlights the structure of the framework and where other parts of the implementation fit in.

6.2 Game

Each game comes with its own game board, players, config file and experiments. While most of the implementation can be shared between games due to their similarity it is still preferable to maintain separate standalone code bases for each game so that changes to one do not interfere

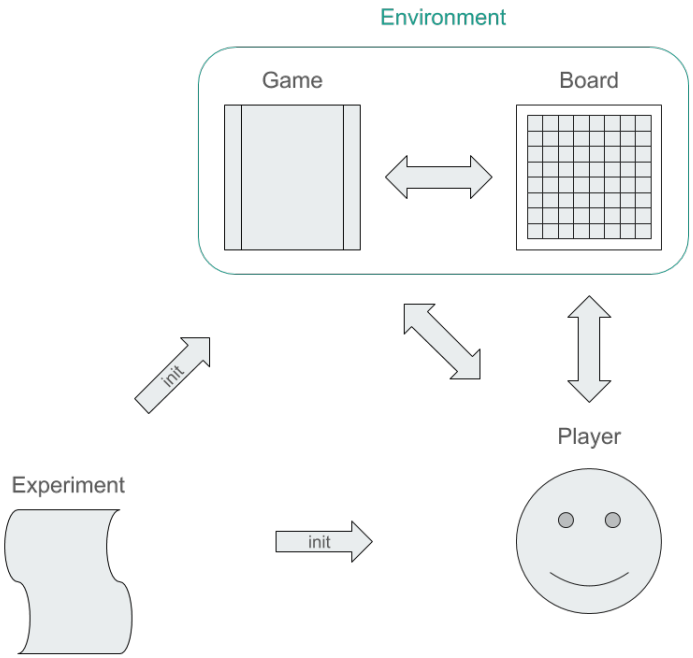

Figure 6.1: **Framework Architecture** The Game and Board classes make up the environment with which the player interacts during both training and evaluation. Both the environment and the player implement base classes and can be extended and swapped with other different versions. The experiment initializes both environment and players and controls which player is training and being evaluated against which opponent at any time, e.g. is the player trained against fixed opponents or using self play.

with the others. This also helps keeping parameterization in reasonable bounds.

A games rules, defined by its legal actions and winning condition, are contained in its board class. The board is responsible for setup, all actions taken during the game as well as determining the winner. It also punishes the players for taking illegal actions or prevents them from doing so.

6.3 Player

The player is where different strategies are implemented. It provides a standardized interface between a strategy and the framework. For machine learning agents this is where most of the learning algorithm is implemented. First however, we introduced several non learning players as a baseline. A *RandomPlayer* as well as a search based *ComputerPlayer* were already available from the framework. We introduced three more: a *HeuristicPlayer*, a *MobilityPlayer* and a *SaveStonesPlayer*, the last two of which rely on search based algorithms as well.

100	-25	10	5	5	10	-25	100
-25	-25	2	2	2	2	-25	-25
10	2	5	1	1	5	2	10
5	2	1	2	2	1	2	5
5	2	1	2	2	1	2	5
10	2	5	1	1	5	2	10
-25	-25	2	2	2	2	-25	-25
100	-25	10	5	5	10	-25	100

Table 6.1: The Heuristic Table of an Othello *HeuristicPlayer*

6.3.1 HeuristicPlayer

The HeuristicPlayer uses a Heuristic Table of tile values for its value function. It then allots points to the player holding these tiles according to the table. The value of a board state is determined by the difference between the player's points and its opponent's. This rather simple player is often used as a baseline because it is fast to execute, deterministic as well as easy to understand and debug. The Strategy of the Heuristic player is given by the Heuristic Table. We used table 6.1 as found in [19] as our strategy for the *HeuristicPlayer*. By looking at the values allotted to each tile we can see that the corners are regarded as the most important tiles on the board, therefore whenever the *HeuristicPlayer* has the chance to take a corner, he will do so. Additionally the tiles around a corner have negative values. This prevents the player from taking those tiles because doing so would allow its opponent to take the corner. Furthermore the table is symmetrical by eight axes which guarantees that rotations and reflections of a game state are equal in value.

6.3.2 Computer Players

The *ComputerPlayer* consists of two main parts: the alpha-beta cut algorithm that unrolls the game multiple moves ahead of the current game state [14], and the value function that scores each game state in order to determine its value. The original player available in the framework used the following features and weighted them depending on how far the game has progressed:

- **Stone count**
  The number of stones each player controls. This is the most important feature in the endgame since it determines the winner of the game.
- **Corner count**
  The number of edges each player controls. This is most important in the mid game since corner stones cannot be taken by the opponent.
- **Mobility**
  The number of valid moves each player has. Restricting the opponents possibilities may interfere with the opponents strategy and allow the player to take many stones at a time in the endgame.

- **Edge count**
  The number of edges each player controls. Edge stones are valuable because they are harder to overturn and may lead to winning a corner. They are most valuable early and mid game, depending on the game state.

- **Corner edge count**
  The number of stones directly surrounding an empty corner each player controls. These stones are valued negatively since they may allow the opponent to occupy a corner.

- **Stability count**
  The number of stones connected to a corner through stones of the same color. These stones are protected from four sides, three from the edge of the board and one from the corner stone or a number of other stones that are connected to the edge stone, and therefore cannot be taken by the opponent.

We created two additional players for more variety and to assess if our agents can learn the concepts behind them:

- **MobilityPlayer**
  A player implementing a strategy based purely on maximizing the mobility in the early and mid game. As soon as the end of the game can be simulated the stones count is considered instead.

- **SaveStonesPlayer**
  This player maximizes its own save stones and minimizes its opponent's. A save stone is any stone that cannot be taken by the opponent for the rest of the game. The computation of save stones starts at a corner stone which is the first save stone. The number of save stones can then be expanded along one of the two edges adjacent to the corner. Each stone along the edge and directly adjacent to a save stone is also save as it is protected from four adjacent sides, covering half at least one direction of any line. The same can then be applied for the next row where every stone that is protected by four adjacent save stones becomes a save stone as well. Just with any other player in the endgame only the stone count is considered for scoring.

6.3.3 Reinforcement Learning Players

Finally Reinforcement Learning players are where the main focus of this thesis lies. In Section 3 we introduced two families of Reinforcement Learning methods. A first approach based on Value Function methods such as Monte Carlo and Temporal Difference methods did not yield acceptable results and revealed many complications related to those methods. There are several important advantages to Policy Gradient methods. The most obvious is that while a Value Function's output is a scalar, a Policy Gradient's output is a distribution. In practical terms a Value Function has to be evaluated for every state action combination in order to get its value whereas a Policy Gradient only has to be evaluated once in order to output the likelihood of each action as a vector. This both simplifies the policy and reduces computational costs. The second major advantage to Policy Gradients is that they have a natural tendency towards exploration which does not necessarily

have to be controlled by a hyper parameter. For these reasons we chose Policy Gradient methods as a family of training algorithms for my Reinforcement Learning players.

- **Reinforce Player** The Reinforce player is based on 3.6.1 and represents the simplest implementation of a Policy Gradient player. It only utilizes the policy head of the network. Each turn the player is passed the game state as well as the set of valid actions. These inputs are run through the policy network in order to predict the action probabilities. From these action probabilities the player samples an action according to their likelihoods. The action is returned as the players move and the probability is stored for training purposes. This process is repeated for an entire episode. After the game has finished the winner is announced to both players. This is the only reward signal the player receives as any other signal would be biased and would limit the space of possible strategies. After the game a single loss is calculated adding the likelihoods of each action multiplied by the discounted reward.

  $L = \sum_a P_a * reward_a$

  The loss can then be backpropagated, upgrading the policy before the next game begins.

- **Baseline Player** The Baseline player extends the Reinforce player with an additional component, the Value Function Baseline. This baseline compensates for game states that are already tipped in favour of one of the two players. The choice of action in those states may not be as important as in earlier, more balanced game states since even a bad choice of action may still lead to victory. For example if the Othello Board is already mostly full, filling in the last few stones without turning many stones often does not make much of a difference to the final result. In fact even a bad choice where much better ones might be available may still let the player win the game. In predicting a state value, or likelihood of players to win, based on the game state and subtracting this score from the reward, the reward for a bad action should now be negative even if the player is still winning after taking it. The state value is computed using a second head of the Policy Network called the Value Head. A separate loss is computed for the the Value Head using the predicted state values and the final result. The two losses are added together to form a single loss before backpropogating over it.

  $L_{policy} = \sum_i -P(a)_t * (reward_t - statevalue_t)$

  $L_{statevalue} = \sum_t L1loss(statevalue_t,\ reward_t)$

  $L = L_{policy} + L_{statevalue}$

  Because the Policy Head and the Value Head share most of the policy network but are trained on two separate losses, the network practically learns two related tasks at the same time, either improving the representation used by the other. This further improves generalisation and therefore performance.

  Both of these factors contribute to significantly reduce the variance and thus increase convergence speed during training.

- **Actor Critic Player** The Actor Critic player builds on the architecture of the Baseline player and augments it with Bootstrapping (3.4). In case of the Actor Critic player the policy loss is modified as follows:

  $L = \sum_i -P(a)_t * (statevalue_{t+1} - statevalue_t)$

  Intuitively the reward is given given as the difference between the predicted state value after the move and the predicted state value before the move, where the last predicted state value is the actual result of the game. This method allows for allocating greater rewards to important actions, specifically those that improve the predicted state value by a large amount and vice versa.

6.3.4 Network Architectures

Several network architectures are used in this thesis. Players generally share the same networks but apply different learning algorithms. Network size depends on the game and the size of the game board. All networks share a similar structure shown in figure 6.2. The first layers are shared and generate a representation of the game state. After the representation layers the policy head and the value head are used to predict the action probabilities and the state value respectively.

6.4 Optimization

In order to optimize both player performance and training time we implemented three optimizations improving specific components of our learning system. These optimizations are not technically required for the success of this work but play a notable role nonetheless.

6.4.1 Legal Softmax

When a player is initialized its weights are random and therefore so is its strategy. If it is then trained against another player it starts out by taking random moves without any understanding of which moves are legal and which ones are not. This is a problem as it leads to very slow learning both against fixed opponents and in self play. Against fixed opponents, which only ever perform legal moves, winning a game would first require to only select legal moves 30 times in a row, which is the average game length. After that it would still most likely lose because the fixed opponent has some form of strategy other than just taking random moves, leading to a negative reward even when only legal moves are taken. A similar situation arises in a self play setting where both players would randomly select legal or illegal moves. In this training mode a player would still win around half of all games played but in the early stages most of those wins would stem simply from the opponent, and therefore a version of the player itself, playing an illegal move. Of course there are several solutions to solving this issue such as curriculum learning, separate loss terms etc. However, even using these techniques it is still possible for the player to select illegal moves later in the training process. In order to prevent this we introduced the *LegalSoftmax* module. At the very last step of the policy head's forward pass, Softmax is applied to the move probabilities. After that the results are masked so that only legal moves have a nonzero move probability. The legal move mask is provided by the board and is therefore given

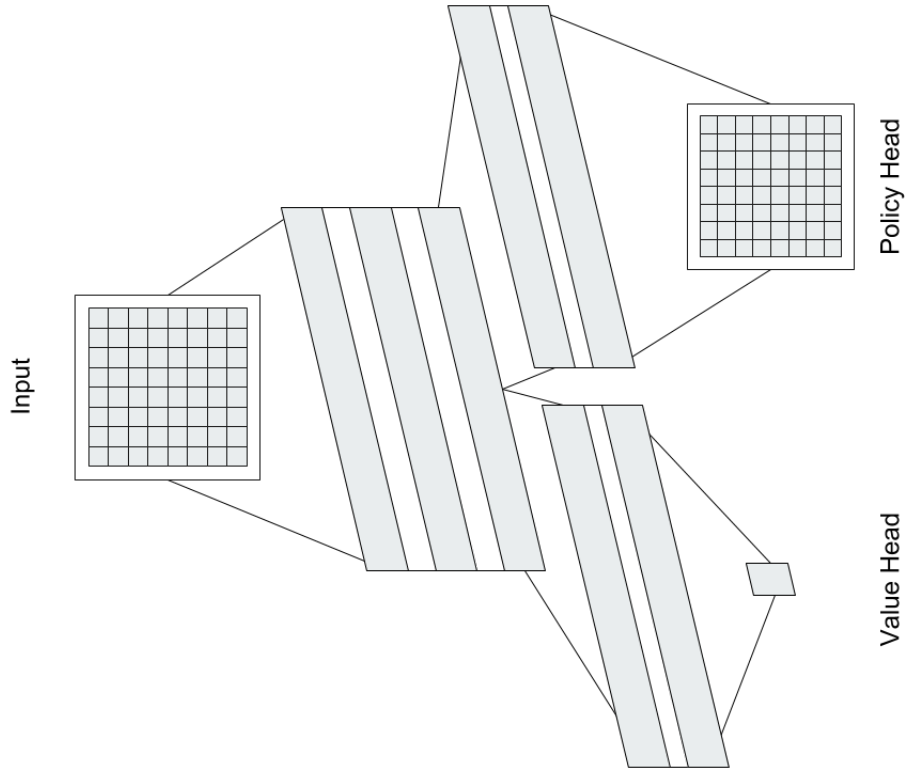

Figure 6.2: **Network Architecture** All players make use of the same network, even though not all of them make use of the value head. As an input the network takes the board state, represented as two two dimensional arrays. One of them represents the stones on the board while the other provides a binary map of legal moves.

by the environment. Giving the player the legal moves as an input can be considered using expert knowledge of the game. It does not restrict the strategy of the player however and therefore does not interfere with our goals. It also arguably allows more of the network's capacity to be used for developing an actual strategy rather than learning the rules. This in turn leads to smaller networks and faster training time.

6.4.2 Milestones

As discussed in Section 5.5, all training environments for this work are dynamic, especially in self play mode. When playing against an earlier version of itself it is possible for the player to find and exploit a weakness and therefore overfit to the current opponent. The next iteration would then learn to do the same thing to the current one. In order to prevent this we implemented *milestones*. These are previous versions of the player that are saved in regular intervals during training. At the start of every epoch there is a chance that the opponent is replaced with a randomly sampled milestone. This forces the player to generalize and remain able to beat previous versions where potential exploits do not lead to winning.

6.4.3 Just in Time Compilation

The environment is a crucial part of the implementation and its performance is highly relevant to the overall training speed. Most of the computational resources for the framework are used by the board classes in order to manipulate and interpret the game state. With the help of the numba package we were able to improve performance significantly using just in time compilation. Since the package is still in development, we had to compromise on some code style principles but the gained performance is well worth the cost [12].

7 Experiments

This section introduces the experiments performed on several players using multiple modes of training as well as the reasoning behind the training methods. Three strategies and Two modes of training evolved over the course of this work: *Reinforce*, *Baseline* and *Actor Critic* players, which are all described in Section 6.3.3, as well as the learning modes *playing against traditional opponents* and *self play*.

7.1 Traditional Opponents

Certain training methods, as well as all evaluation methods rely on so called *traditional opponents*. They get their name from the fact that they rely on older mechanisms and have been known for much longer than the Deep Reinforcement methods discussed here. Learning to beat deterministic opponents would be a rather simple task and merely require learning one sequence of moves per opponent. For this reason all *traditional opponents* are non deterministic. The following list introduces all *traditional opponents*:

- **Random Player**
  The Random Player randomly samples one of the legal moves given a board. It is considered the simplest player to beat and is non deterministic by design.
- **Novice Player**
  The Novice Player improves on the Random Player by choosing moves that lead to winning a game immediately. If no such move is available it samples a random move and is therefore non deterministic as well. In TicTacToe the Novice Player outperforms the Random Player significantly where in Othello the last move often does not make a huge difference.
- **Experienced Player**
  The Experienced Player implements the Heuristic Player described in Section 2.1.1. In

TicTacToe it also selects winning moves immediately and prevents opponents from winning on the next turn if possible. Both the Othello and TicTacToe version of Experienced Players are rather strong players while still being computationally cheap. The TicTacToe version is very close to the perfect TicTacToe Player.

- **Expert Player**
  The Expert Player represents a perfect player if one is available. If not, it is the state of the art player and the one to beat.

7.2 Training modes

Throughout this thesis many modes of training were tested and discarded until finally only the two following ones remained. They also represent two large groups of training modes in the domain of board games.

7.2.1 Playing against Traditional Opponents

When playing against traditional opponent the Learning Player is assigned one of a list of traditional opponents at random on regular intervals throughout the training process. It plays a number of games against the selected opponent and uses the generated data as training samples for policy improvements before being given a new opponent. This mode is generally considered the simpler of the two because the static strategy of the traditional opponents make for a more static environment where the only thing that changes is the chosen opponent.

7.2.2 Self Play

In self play mode the Learning Player plays against slightly older versions of itself. At regular intervals the learning player is evaluated against its current opponent. If it is stronger than the current opponent it replaces the current opponent and is trained against the newer version until the next evaluation. This potentially improves straight up self play for two reasons. Firstly it ensures that the player is continuously improving since otherwise the opponent is not updated. Secondly it creates a momentarily static environment during each interval where the opponent is fixed. This allows the Learning Player to adjust to the environment and make progress towards the goal before it is changed and the level of play is improved, both of the opponent and the learning player. We call this mode training *vs Best*. The alternative where a player is trained against an opponent which shares its weights directly and is therefore instantly updated whenever the player makes an update is termed training *vs Self*.

In order to prevent forgetting previously learned strategies there is a chance that a much older player is selected as an opponent. These so called *Milestones* are players saved on regular intervals during training. Since saving a player is expensive, this is only done a few times during training but [5] suggests that this measure prevents strategy forgetting and improves performance.

While self play is the more challenging of the two tasks it allows for more freedom in learning a strategy than playing against fixed traditional opponents. In theory this should allow players

trained through self play to perform better against the known players and generalize a lot better against unknown ones since they were never trained on them in the first place.

7.3 Evaluation

Evaluation in Reinforcement Learning settings can be tricky as there are no fixed training and test data sets. Usually we want to evaluate an agent on an environment and measure its performance on a given task. But in board games the environment and a player's performance change drastically with the choice of opponent. We decided therefore to create a bench mark that can be used for any game and all players regardless of the mode they were trained in. Evaluation games are held against all available traditional opponents. All players are set to evaluation mode, reducing exploration to a minimum and yielding maximal results. This evaluation method obviously gives an advantage to players trained against Traditional Opponents as they are trained on the same environment as they are tested on. Meanwhile Self Play players have never interacted with Traditional Opponents during training. During both training and evaluation, a loss is rewarded with -1, a draw with 0 and a win with 1. The following plots show the number of episodes on the x scale and the trained player's average evaluation values against each of the traditional opponents on the y scale. The *SearchPlayer* represents an Expert or perfect player which can not be beaten but only drawn against.

7.4 TicTacToe Experiments

All strategies and learning methods were first tested on TicTacToe as it is computationally less expensive and intuitively easier to understand due to its size. In TicTacToe given perfect play both players would always draw, regardless of who plays first. This can be proven mathematically as well as empirically by playing two perfect players against each other. Considering this, TicTacToe players *ActorAcritic* and *Baseline* performed reasonably well during evaluation, while *Reinforced* performs significantly worse. As explained in Section 7.3, 1 and -1 on the Y axis represent an average win or loss respective while 0 represents an average draw. Winning against the *ExpertPlayer* is impossible as it is a perfect player which cannot lose under any circumstances, therefore a constant draw would be perfect play from the learned player. *ExperiencedPlayer* approaches a perfect player but performs slightly worse on purpose. It is therefore expected for the learning player to perform slightly better against it.

7.4.1 TicTacToe against Traditional Opponents

In this setting all players perform as expected. The more complex players *ActorCritic* and *Baseline* learn rather quickly and win most games against the two simpler traditional opponents *RandomPlayer* and *NovicePlayer* while drawing most games against the more sophisticated *ExperiencedPlayer* and *ExpertPlayer*.

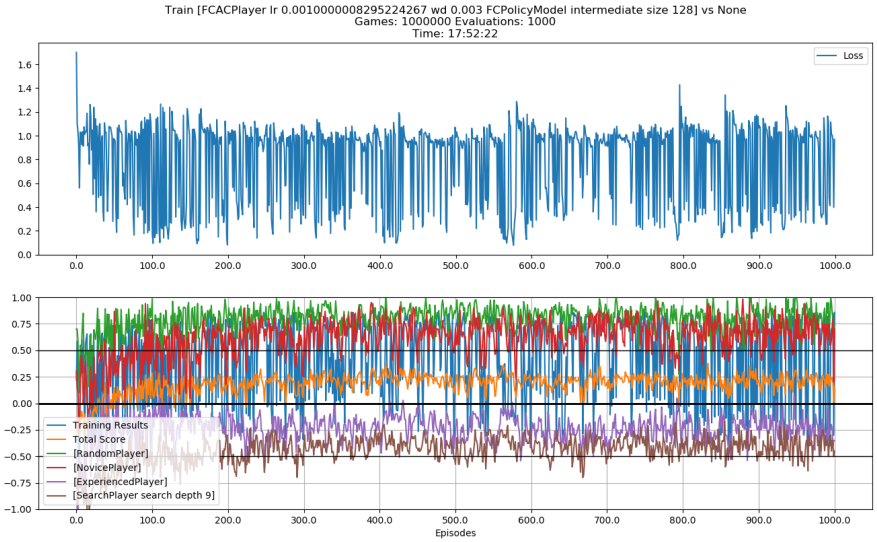

Figure 7.1: **ActorCritic player trained on Traditional Opponents.** ActorCritic player wins most games against simple opponents and mostly draws against very good players which indicates reasonably good performance.

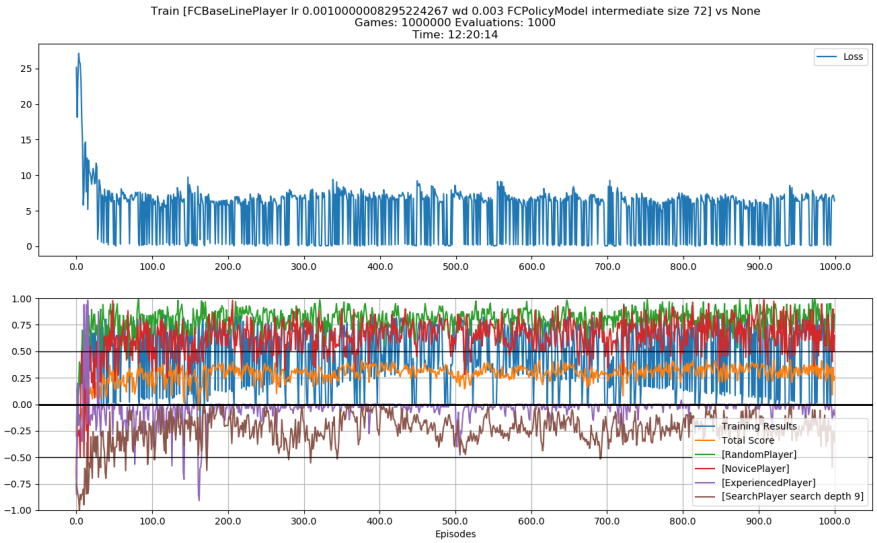

Figure 7.2: **Baseline player trained on Traditional Opponents.** Just like *ActorAcritic*, *Baseline* performs as expected for a player approaching perfect play. Interestingly for most of the training it performs slightly better against the good opponents than the more complex *ActorAcritic* does.

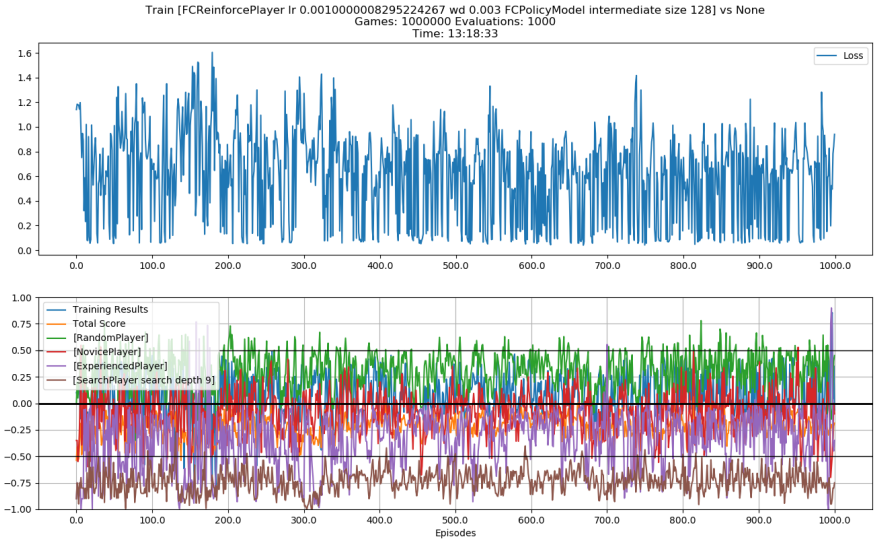

Figure 7.3: **Reinforce player trained on Traditional Opponents.** While still beating the simple *Random* and *Novice* opponents this player struggles significantly more with the perfect *Expert* opponent. Interestingly it performs a lot better against the *Experienced* opponent and even learns how to exploit its weakness at the very end of training.

7.4.2 TicTacToe Self Play

Traditional opponents are interesting for developing and testing our implementation but the goal of this thesis has always been to train players using self play. In self play mode the *ActorAcritic* player performs much better than the other two implementations and about as good as when trained against traditional opponents. This is a valuable finding and supports the claim that self play is valuable even in simple environments such as TicTacToe, especially since this setting is considered much more difficult than training against traditional opponents because learned players have to generalize in order to beat the reference players.

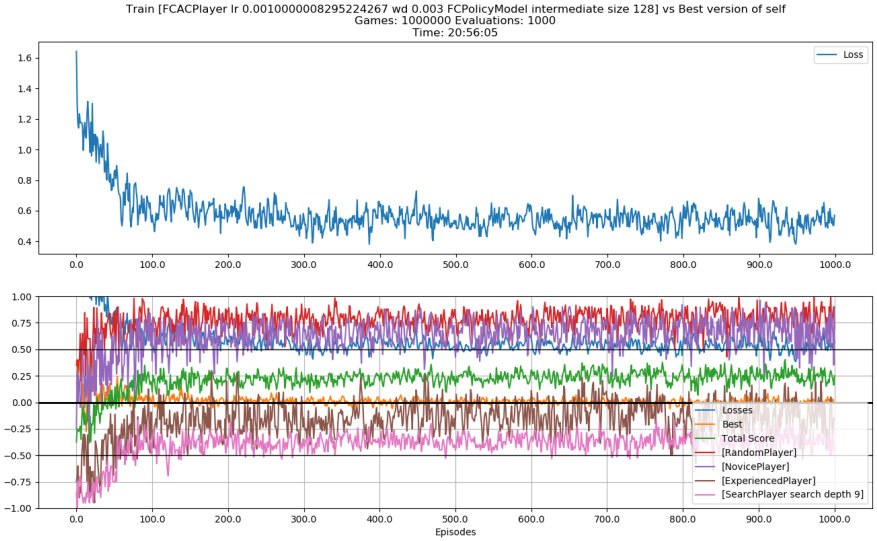

Figure 7.4: **Actor Critic player vs Best iteration of self.** In this setting *ActorCritic* reaches similar performance as when trained against traditional opponents. This is significant as, unlike when training against traditional opponents, it plays the reference players only through generalization without ever having played against them in training.

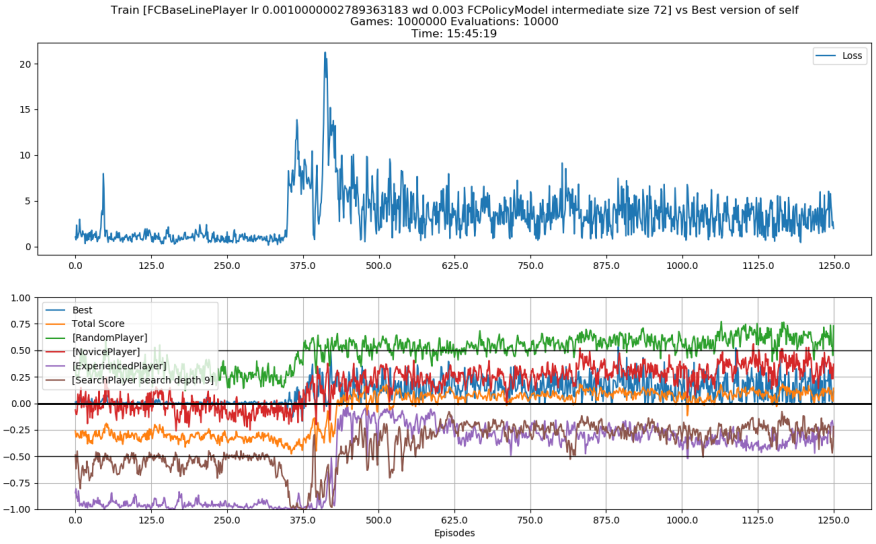

Figure 7.5: **Baseline player vs Best iteration of self.** While the performance is lower than when training against traditional opponents, the policy learned in this experiment is still performing well against the simple opponents and acceptable against the more difficult ones, especially considering as the plots indicate the policy is still improving. Note how performance against the perfect *Expert* opponent is similar if not slightly better than against the slightly imperfect *Experienced* opponent.

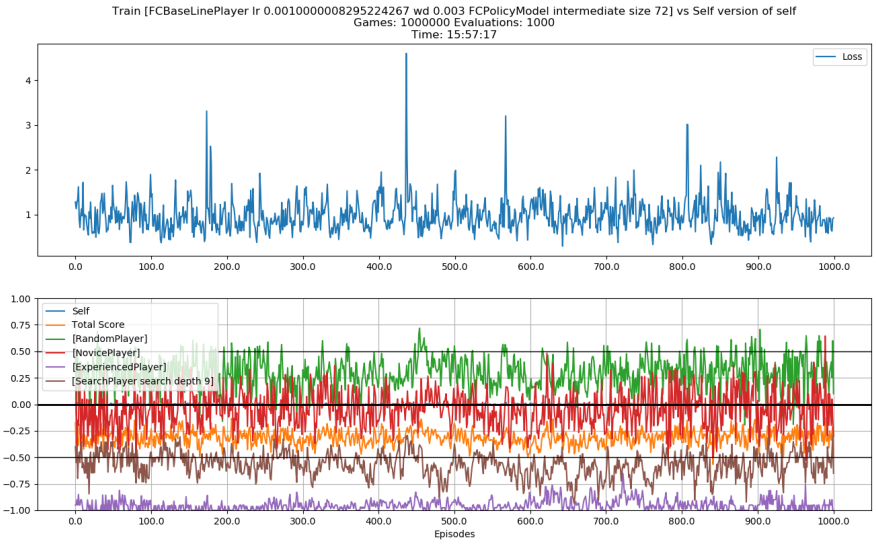

Figure 7.6: **Baseline player vs Self.** When training the *BaselinePlayer* against itself without delaying the opponents update we receive a promising loss plot but performance is still not optimal. Interestingly the player performs much better against the perfect *Expert* opponent than against the *Experienced* opponent

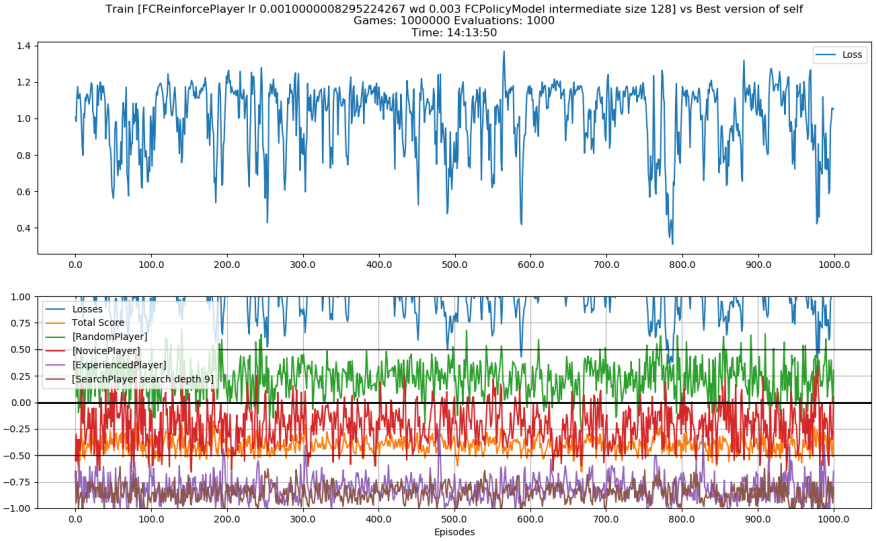

Figure 7.7: **Reinforce player vs Best iteration of self.** In this setting the *ReinforcePlayer* does not seem to learn much at all.

7.5 Othello Experiments

After evaluating the system on TicTacToe the main challenge is training a well performing Othello player. In contrast to TicTacToe, Othello is not solved mathematically and no perfect strategy is known. We also do not know how biased starting positions are and how the result would look like under perfect play. For the following plots the Y axis again indicates win rates and loss values respectively while the X axis shows the number of episodes. During evaluation we focused most of our computational resources on the *BaselinePlayer* which showed the most robust performance during development.

7.5.1 Othello against Traditional Opponents

In this setting Othello players *ActorCriticPlayer* and *BaselinePlayer* are trained against the traditional opponents *Random*, *Novice* and *Experienced* which are described in Section 7.1. Since Othello is not mathematically solved, a perfect player is not available.

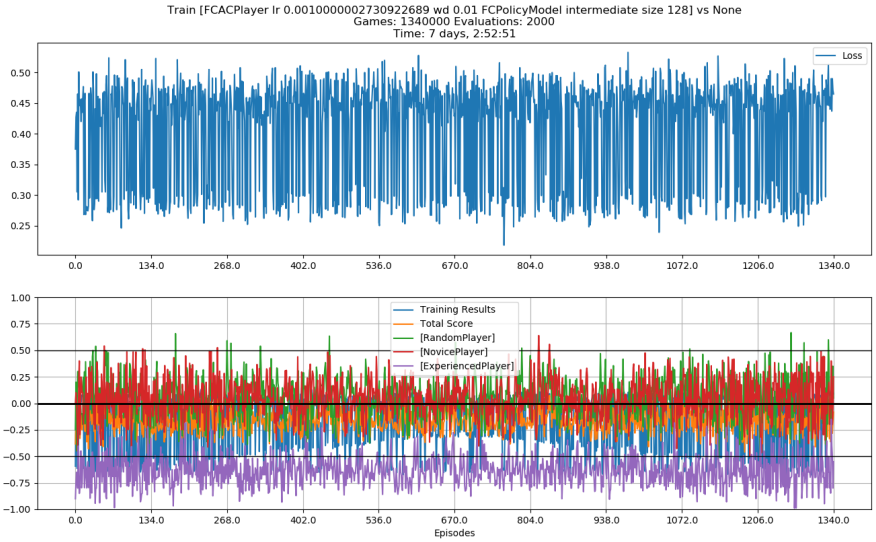

Figure 7.8: **ActorCritic Player vs traditional Opponents.** When looking at the *ActorCritic-Player*'s loss it is both very small and does not seem to decrease during training. We suspect this is due to hyper parameter tuning and balancing between which of the two networks of the policy is updated faster. The bootstrapped reward which is used to train the policy head is computed using only the predictions from the value head which start out very small at the beginning of training. With such a small loss it seems the gradients are rather small as well and the policy updates are minimal which does not allow for learning of a proper policy. If the same experiment on TicTacToe is any indication, much better performances could be achieved in this experiment.

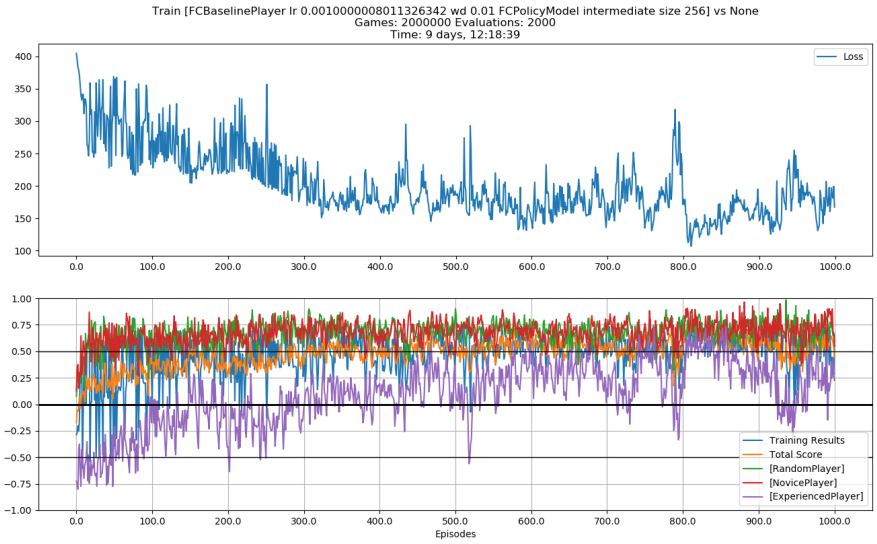

Figure 7.9: **Baseline player vs Self.** When training the *BaselinePlayer* against traditional opponents it shows good performance of over 75% win rates against the *Experienced* opponent at the end of training and better yet against simpler players. It also continues to improve over the entire training, indicating that the training data gained by simulating over 1.5 million games is indeed useful.

7.5.2 Othello Self Play

Both *BaselinePlayer* self play experiments show good policy improvements initially but do not reach performance as high as when training against traditional opponents. This is expected as the self play setting is much more demanding than its traditional opponent counterpart. It is also likely that the policy, and with it performance, is still being improved at this stage of training.

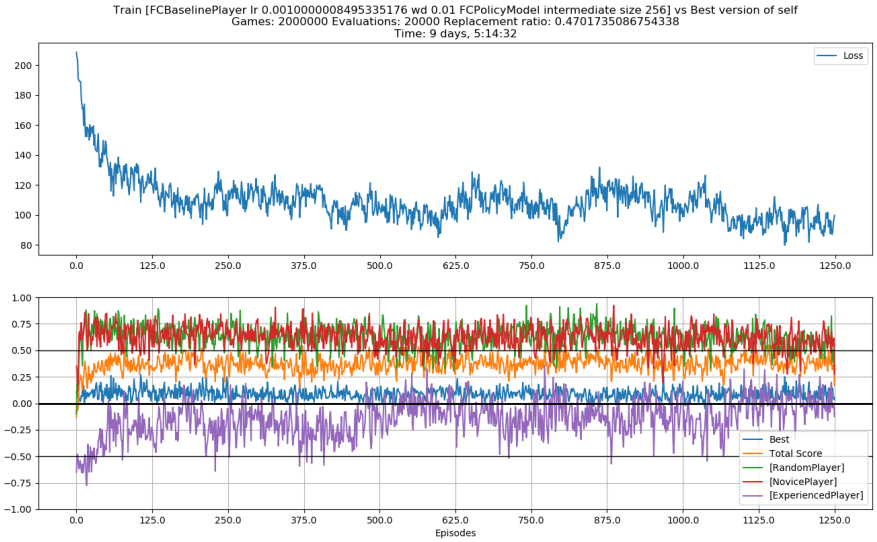

Figure 7.10: **Baseline player vs Best.** In this setting the *BaselinePlayer* continuously gains performance over the *Experienced* opponent. During later stages of training, improvements are slow which is expected as the player only plays against itself, compared to training against traditional opponents where the environment is more constant and the opponent to beat is the same one that is used during evaluation.

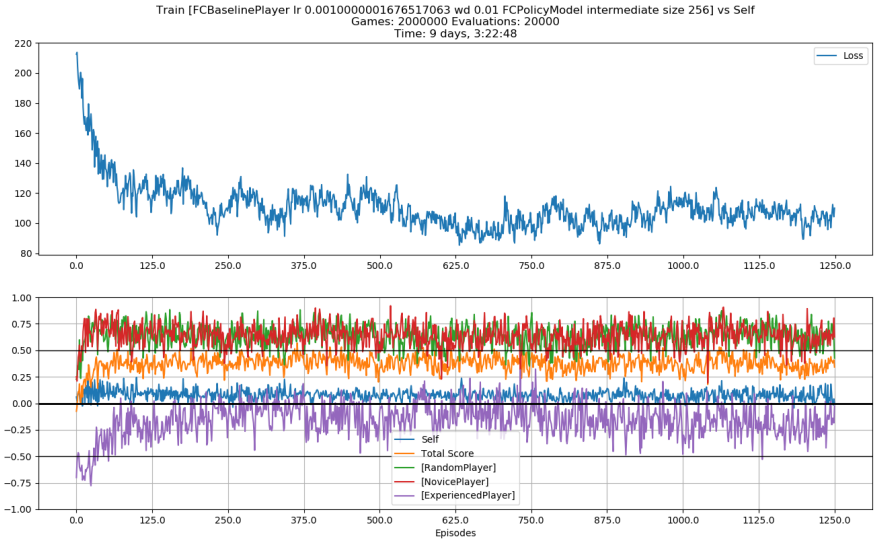

Figure 7.11: **Baseline player vs Self.** When training the *BaselinePlayer* against itself without delaying the opponents updates performance is comparable with training *vs Best* as shown in Figure 7.10 which indicates that delaying the opponents updates does not improve performance significantly in this instance.

8 Conclusion

This work shows that both TicTacToe and Othello can be trained to reasonable performance even with small networks and readily available resources. This is true for both examined training modes: training against traditional, fixed opponents as well as using self play. Three players using different reinforcement learning techniques have been implemented. Of those players the *BaselinePlayer* achieved the most stable performances which are similar to those of the traditional *HeuristicPlayer*, or in some cases clearly surpassing them, which is often used as a baseline to compare Othello players. Another positive finding is that the TicTacToe *ActorCriticPlayer* is capable of achieving the same performance when trained using self play or against traditional opponents. This indicates that a similar feat is possible for the other players and possibly Othello, given the right circumstances. The results of this work serve as a foundation for further improvements such as more capable models, more advanced reinforcement learning algorithms and combination with search based methods. Expanding and improving the framework as well as contributing other games and players to the project is made simple by clean implementation of the framework and it is our hope that it will continue to prove useful in the future.

9 Future Work

9.1 Full scale Experiments

Due to the very large action spaces and long training times, some took several weeks, the number of experiments we were able to run was limited, especially after the implementation was completed. We focused most of our efforts on the *BaselinePlayer* as it showed the most robust performance during development. More experiments on the Actor Critic and pure REINFORCE player would have been interesting, as well as experimenting with larger networks. All of these experiments were run at some point during development but not on the final version of the implementation. Further experiments on other opponents as well as ablation studies were planned but could not be completed due to time constraints. Even though we tried to minimize hyper parameters by using Adam instead of standard SGD and focusing on policy gradient methods rather than value function methods, there are still a few very influential hyper parameters left in the system. Cross validations for tuning those hyper parameters such as learning rate, opponent replacement rates, network architectures etc. separately for each player could yield significant improvements but this was simply not feasible with available resources.

9.2 More advanced Players

For reasons of both computational as well as time resources we had to limit ourselves to three learning algorithms for this thesis. However there are many other promising algorithms available as well as others that have not yet been discovered. One that we would like to implement in particular is Proximal Policy Optimization (PPO) [7], the de facto standard algorithm with OpenAI. Another interesting approach would be to combine learning and search similar to what has been done in [6].

9.3 Community Contribution

From the start of this thesis we hoped to create a framework in such a way that other users might find it intuitive and convenient enough to use it for their own experiments. Ideally the PyTorch community would contribute other games and many more players for both the games we provided as well as community introduced ones. Due to its simplicity it might even be used in teaching settings.

Bibliography

[1] Alpha go, 2017.

[2] Peter Attia. The full history of board games, 2016. https://medium.com/swlh/the-full-history-of-board-games-5e622811ce89.

[3] British othello federation. http://www.britishothello.org.uk/rules.html.

[4] Kevin Anthony Charry. An intelligent othello combining machine learning and game specific heuristics. Master's thesis, Louisiana State University, 2008.

[5] Demis Hassabis et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2015.

[6] Demis Hassabis et al. Mastering the game of go without human knowledge. *Nature*, 2017.

[7] John Schulman et al. Proximal policy optimization, 2017. https://blog.openai.com/openai-baselines-ppo.

[8] Wojciech Jaśkowski. Systematic n-tuple networks for position evaluation: Exceeding 90% in the othello league. *ICGA Journal 37(2)*, 2014.

[9] Jeffrey S. Johnston. *Past Times: Ancient Board Games*. Jeffrey S. Johnston, 2015.

[10] Daisuke Kikuchi. Goro hasegawa, inventor of board game othello, dies at 83. *The Japan Times*, 2016. https://www.japantimes.co.jp/news/2016/06/23/national/goro-hasegawa-inventor-board-game-othello-dies-83/#.W27BChixXCI.

[11] Yip Tsz Kwan. Playing othello by deep learning neural network. Master's thesis, University of Hong Kong, 2017.

[12] Numba package. https://numba.pydata.org/.

[13] Othello by rgruener. https://github.com/rgruener/othello.

[14] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, third edition, 2009.

[15] Daniel E. Slotnik. Goro hasegawa, creator of othello board game, dies at 83. *The New Your Times*, 2016. https://www.nytimes.com/2016/06/26/world/asia/goro-hasegawa-creator-of-othello-board-game-dies-at-83.html.

[16] Othello plot overview. http://www.sparknotes.com/shakespeare/othello/summary/.

[17] OpenAI Team. Openai five, 2018. https://blog.openai.com/openai-five/.

[18] OpenAI Team. Openai five benchmark: Results, 2018. https://blog.openai.com/openai-five-benchmark-results.

[19] Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. *IEEE*, 2013.

[20] Waymo. Google i/o recap: Turning self-driving cars from science fiction into reality with the help of ai, 2018. https://medium.com/waymo/google-i-o-recap-turning-self-driving-cars-from-science-fiction-into-reality-with-the-help-of-ai-89dded40c63.

[21] Waymo. Waymo journey, 2018. https://waymo.com/journey.

[22] Waymo. The world's longest and toughest ongoing driving test, 2018. https://medium.com/waymo/the-worlds-longest-and-toughest-ongoing-driving-test-44464867865b.

[23] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 1992.