Alpha-zero Project Report

Student: Stefano Iannicelli 247485

Introduction

AlphaGo Zero is an algorithm presented in 2017 that makes use of a neural network, trained through reinforcement learning, to play Go 'without human knowledge'. Prior to this, AIs made use of techniques such as minimax, which, thanks to heuristics suitably calibrated to the given problem, made it possible to create automatic players. With alphazero, on the other hand, it is only necessary to define the problem and launch the training of the neural network.

Alpha-zero

As just mentioned in the introduction, the algorithm learns automatically, in fact, once the rules of the game have been defined, games, called episodes, are simulated where the algorithm plays against itself. This phase automatic play (self-play) serves to collect the examples that will later be used to train the network. The neural network underlying the algorithm receives the current state of the board as input and outputs the optimal policy. The self-play phase is guided by Monte-Carlo Tree Search, which allows the policy to be improved.

Neural Network

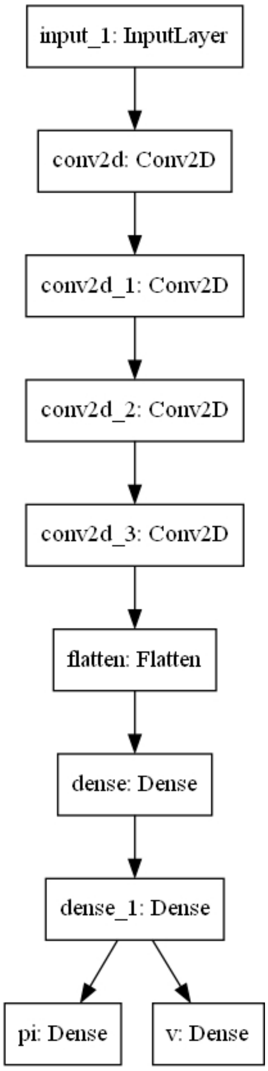
The network on the input side receives the board status and on the output side returns:

- $v_\theta \in$ [-1, 1]: the value of the board from the perspective of the player who is to perform the move;
- $\vec{p}_\theta \rightarrow$ a probability vector containing the policy; it

consists of:

- 4 convolutional layers with 128 filters, relu activation function and kernel size of 3;
- 2 dense layers consisting of 1024 neurons with relu activation function, dropout at 0.3 and batch normalisation.

The following figure shows the architecture of the network.



The examples will be tuples $(s_t, \vec{\pi}_t, r_t)$ where s_t is the current state, $\vec{\pi}_t$ is the result of the Monte Carlo Tree Search algorithm and $r \in \{-1, 1\}$ is the reward. Thus, the loss function is given by the sum of the categorical crossentropy for the policy network and the quadratic error for the v network.

$$l=\sum_t \left((v_\theta(s_t)-r_t)^2 - \vec{\pi}_t \log(\vec{p}_\theta(s_t)) \right)$$

The optimiser chosen is Adam, adaptive momentum estimation, which is based on stochastic gradient descent but at each step stores an exponentially decreasing average of the squared gradients that it uses to adapt the learning rate, and an exponential average of the gradients that can be compared to the momentum. Essentially, the gradient descent will follow a path comparable to a ball having a certain angular momentum and falling along a surface with friction, this is done to try to local minima. Adam is one of the best algorithms, in fact it is among the most popular at the moment, as it offers good learning speed.

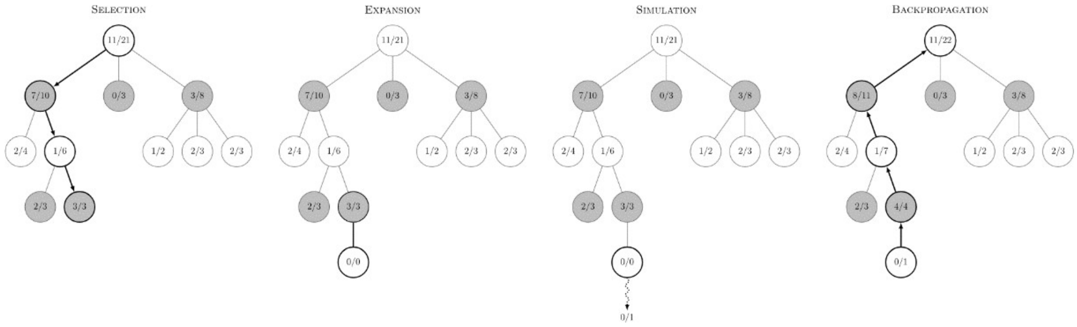
Pure MCTS

MCTS is an algorithm that is used on adversarial games, it is based on the simulation of matches in which each move is randomly selected from the space of permissible actions, as the algorithm iterates a tree formed by the visited states is created, the end result of each match is propagated to the root. At the end of the simulations, the move that has led the player to victory the most times will be the chosen move.

Mtcs consists of four steps:

- **selection**: start at root R, the current state of the game, and select successive actions up to a leaf node L, a node that has not yet been visited in any simulation;
- **expantion**: if the leaf node L found in the selection phase is not an end state of the game, the tree is expanded with a child C obtained from a random but permissible move of the game;
- simulation: complete a random game simulation starting at node C;
- **backprobagation**: uses the result of the simulation to update the information in the nodes in the path from C to R.

The following figure shows an example of the four stages just described.



In the selection phase, actions (nodes) are chosen that maximise a coefficient called UCB1, which is obtained from the following formula:

$$UCB1(= \frac{U(n)}{N(n)} + c\sqrt{\frac{\ln N(parent(n))}{N(n)}}$$

Where:

- $U(n)$: number of wins that passed through node n;
- $N(n)$: number of simulations that passed through node n;
- c: is an exploration parameter.

Monte Carlo Tree Search for Policy Improvement

Unlike pure MCTS in this version used in AlphaGo for each arc in the search tree we save:

- $Q(s, a)$ expected value of the reward obtained performing action **a** in state **s**;
- N(s, a) number of times action **a** was chosen in state **s** during the simulations;
- P(s, .) a probability vector representing the a priori probability of choosing a particular action from state **s** according to the policy returned by the neural network.

From these values is calculated$U$ $(s,a$) which is an upper bound of the value of Q for that particular arc:

$$U+(s,\ )a=Q\ \ (s,\ )ac \cdot P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

where:

- there is a hyper-parameter controlling the degree of exploration;
- $\sum_b N(s, b)$ number of times action **a** was not chosen in state **s**.

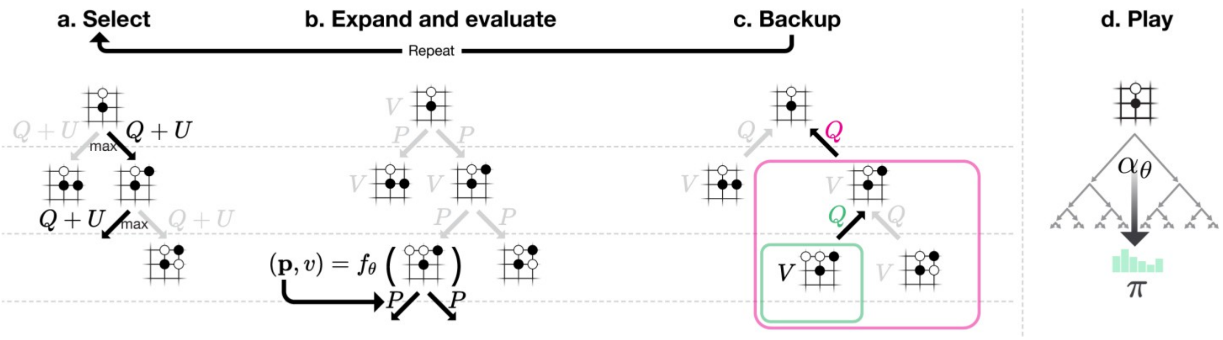
The algorithm starts by creating a search tree by entering state s as the root, at each step the action a that maximises the value $U(s, a)$ is calculated, if the new state is already present in the tree we continue by choosing a new action otherwise a new node is created. The variables of this new node F will be initialised in the

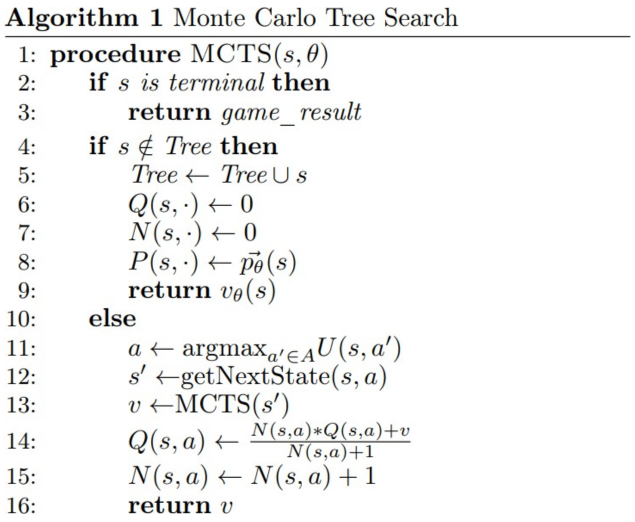
following way:

- $P(s,\ .\ ) = \vec{p}_{\vec{\theta}} \rightarrow (s)$ probability vector obtained from the **pi** network;
- $v = v_{\theta}\ \ (s)$ value representing the expected reward obtained from the **v** network;
- $Q(s, a)$= N(s, a)= 0 for each action **a**.

Afterwards, the reward **v** is propagated backwards in the tree by updating the Q(s, a) values of the nodes on the path from **F** to the **root** node. If an end state is encountered the simulation,

the final utility of the board will be propagated. Figure Algorithm 1 shows the pseudocode of the algorithm just explained.





Algorithm 1 Monte Carlo Tree Search

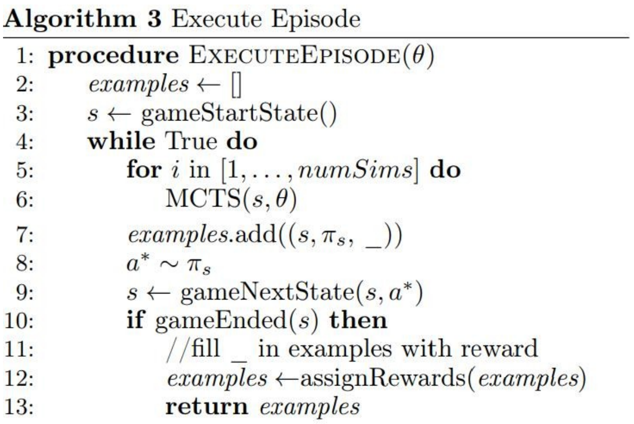
```
procedure MCTS($s, \theta$)
    if $s$ is terminal then
        return $game\_result$
    if $s \notin Tree$ then
        $Tree \leftarrow Tree \cup s$
        $Q(s, \cdot) \leftarrow 0$
        $N(s, \cdot) \leftarrow 0$
        $P(s, \cdot) \leftarrow \vec{p_\theta}(s)$
        return $v_\theta(s)$
    else
        $a \leftarrow \mathrm{argmax}_{a' \in A} U(s, a')$
        $s' \leftarrow$getNextState$(s, a)$
        $v \leftarrow$MCTS$(s')$
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
        return $v$
```

Once this process is finished, we start again by running simulations from the root, after a few simulations the value of

N(s, a) will return a good approximation of the optimal policy for each action taken in state **s**.

Finally, the chosen action in state **s** will be randomly sampled from $N(s, a)^{\frac{1}{\tau}}$ whereτ is the temperature, a hyper-parameter that regulates the degree of exploration learning. Ifτ is 0 the best action will always be chosen, otherwise ifτ has a high value there will be an even distribution, so the action will be chosen randomly. In this phase the train examples are collected and saved in the replay memory, the examples will be tuples (s,π_s , r) where **s** is the current state,π_s is the result of the MCTS and r∈{-1, 1} is

the reward that is awarded at the end of the game from the perspective of the current player, -1 if in the game that generated the example (s,π_s , _) the current player lost, 1 otherwise. Algorithm 3 shows the pseudocode.

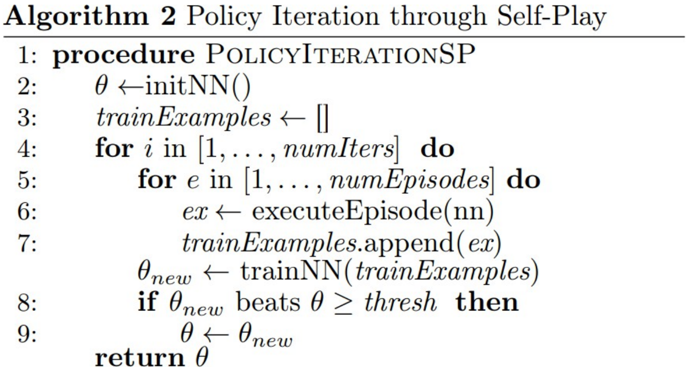


Algorithm 3 Execute Episode

```
procedure EXECUTEEPISODE($\theta$)
    $examples \leftarrow []$
    $s \leftarrow$ gameStartState()
    while True do
        for $i$ in $[1, \ldots, numSims]$ do
            MCTS($s, \theta$)
        $examples$.add(($s, \pi_s$, _))
        $a^* \sim \pi_s$
        $s \leftarrow$ gameNextState($s, a^*$)
        if gameEnded($s$) then
            //fill _ in examples with reward
            $examples \leftarrow$assignRewards($examples$)
            return $examples$
```

Train

At the beginning the neural network is initialised with random weights, at each iteration of the algorithm 60 self-play episodes were played using MCTS, and each step of each episode was saved in the replay memory. At the end of the self-play phase, the weights of the neural network are updated using 100000 random examples sampled from the replay memory, the new network obtained is against an old network called the best model, if the new network beats the best model by 60%, the new network will become the new best model, we continue with the next iteration resetting the MCTS tree. Otherwise, the new network will be discarded and at the next iteration training will restart from the current best model. The following figure shows the psudocode.

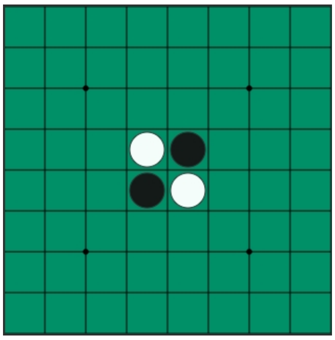


Algorithm 2 Policy Iteration through Self-Play

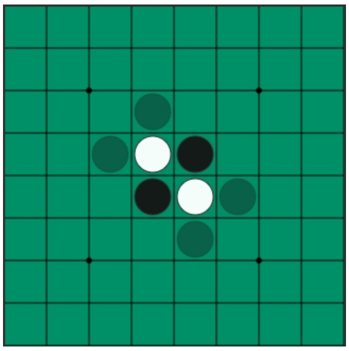
```
procedure PolicyIterationSP
    $\theta \leftarrow$ initNN()
    $trainExamples \leftarrow []$
    for $i$ in $[1, \dots, numIters]$ do
        for $e$ in $[1, \dots, numEpisodes]$ do
            $ex \leftarrow$ executeEpisode(nn)
            $trainExamples$.append($ex$)
        $\theta_{new} \leftarrow$ trainNN($trainExamples$)
        if $\theta_{new}$ beats $\theta \geq thresh$ then
            $\theta \leftarrow \theta_{new}$
    return $\theta$
```

Othello

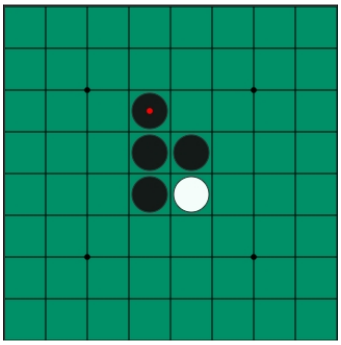
The game chosen for the application of the alphaGo Zero algorithm was Othello, an information-perfect board game for two players invented in 1971 by Gorō Hasegawa. It is played on a chessboard called an Othello board, generally 8×8 in size but there are versions that also use 6×6. In the standard size (8×8), players have 64 pawns, black on one side and white on the other. Before starting the game, four checkers, two white and two black, are placed in the central squares to form an X configuration, as shown in the following figure.



The black player starts by placing a pawn so that it forms a horizontal, vertical or diagonal line between the new disc and an existing black one, with one or more contiguous white discs between the black pawns. The following diagram shows the moves available to the black player.



Once the black checker is placed in an eligible position, all white checkers that are enclosed between the two black checkers are flipped over. Thus, the white checkers become black, as shown in the following example.



The game continues until the ophelliere is full, the player who has the most checkers of his colour wins, it is a draw if the number of black checkers is equal to the number of white ones.

The space of othello states has symmetries, in fact it is invariant respect to rotation and mirror operation; thus, each state has a total of seven symmetries. This was very useful in the train phase because it allowed for **data augmentation**.

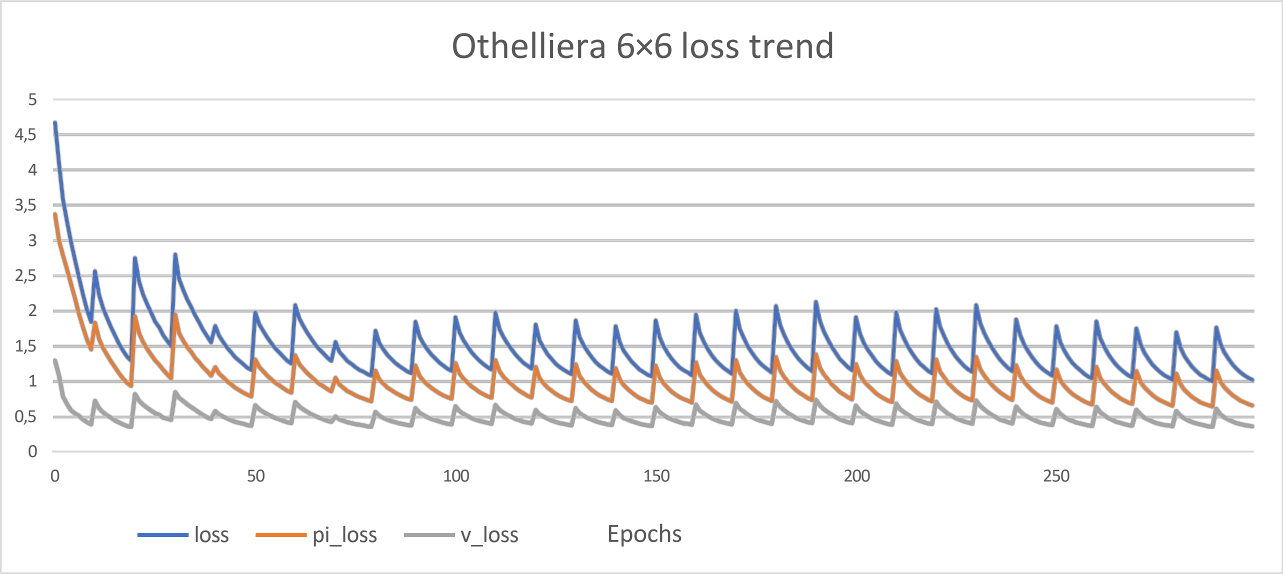
Experiments

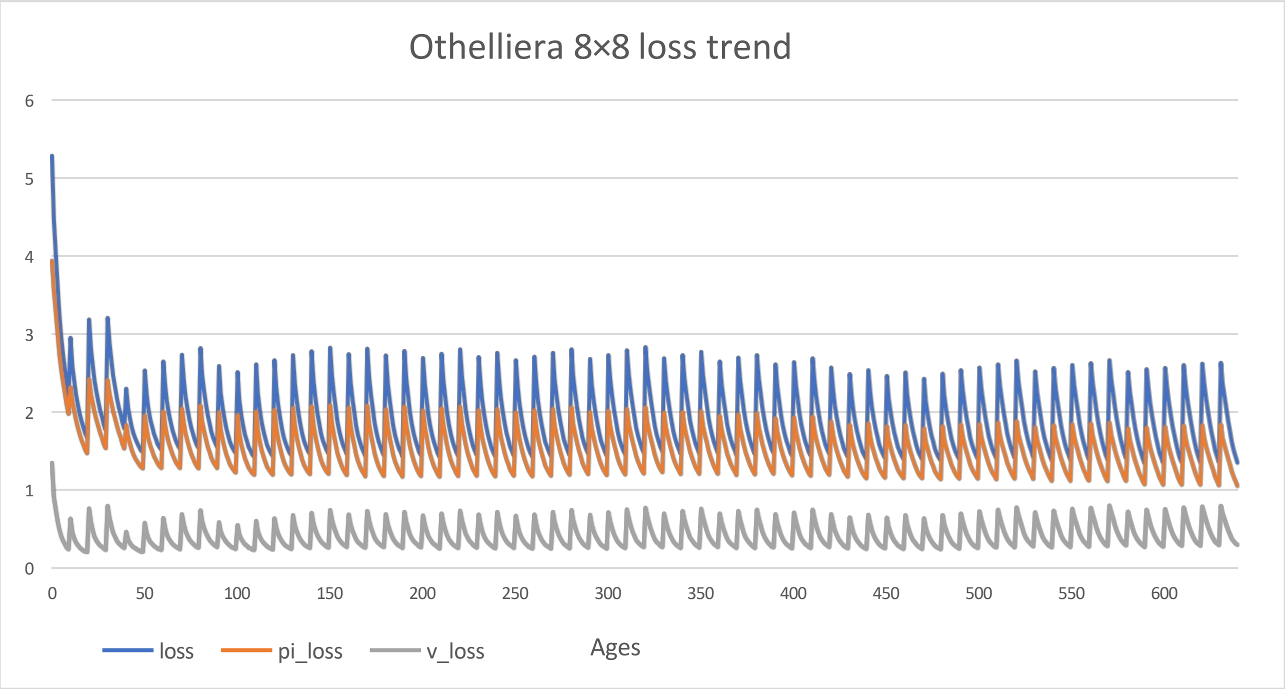
The framework used for testing the algorithm described so far was Alpha Zero General available on github at the following link https://github.com/suragnair/alpha-zero-general. We chose to train two models, one on the 6×6 and a second on the 8×8.

Initially, however, the training times were very high as an episode in the 6×6 board took more than 30 seconds, which would have resulted in unsustainable train times. Therefore, an attempt was made to optimise the code, and two main changes were made which resulted in a speedup of about 10x in the self-play phase. The first was to replace the predict command with predict_on_batch; the two commands perform the same operations, but predict_on_batch is extremely faster (source: https://stackoverflow.com/questions/44972565/), when running predict on a single batch. The second optimisation was to have only the cpu perform the self-play phase, as it was noted that using the gpu took longer to copy the tensor from the ram to the gpu's memory than it would have taken to perform the predict on the gpu.

In order to improve the train time instead, replay memory sampling was implemented; in the original code, this was not present and at each iteration, the model was trained on the entire replay memory, resulting in extremely high train times per epoch.

These modifications made it possible to train the model on a computer with modest computational capabilities. The model was trained on the 6×6 and 8×8, for both the number of monte Carlo simulations was 25, while the temperature parameter was set to 1 for the first k moves of each episode then to 0, this was done to encourage the algorithm to explore new states during the early stages of the game. In the 8×8, k was chosen equal to 20 while in the 6×6 k=10, furthermore, at each iteration of the algorithm, the model trains for 10 epochs saving the loss values in a log file. The models were trained for 30 and 63 iterations respectively, taking 4 hours in the 6×6 and approximately 20 in the 8×8. The following graphs show the loss trends.





The jagged pattern is given by the sampling done on the replay memory and the new examples added at each iteration. The trends are similar, in both cases it can be seen that the v_loss remains almost constant, while the pi_loss is slowly minimised causing the total loss to be minimised, a sign that the model slowly learns the optimal policy.

The model was tested against various types of automatic players:

- **random**: player who chooses a random move from among those eligible;
- **greedy**: player who chooses the move that maximises the number of tokens of the current player;
- minimax: player using the minimax algorithm with alpha-beta pruning, search depth four and greedy heuristics.

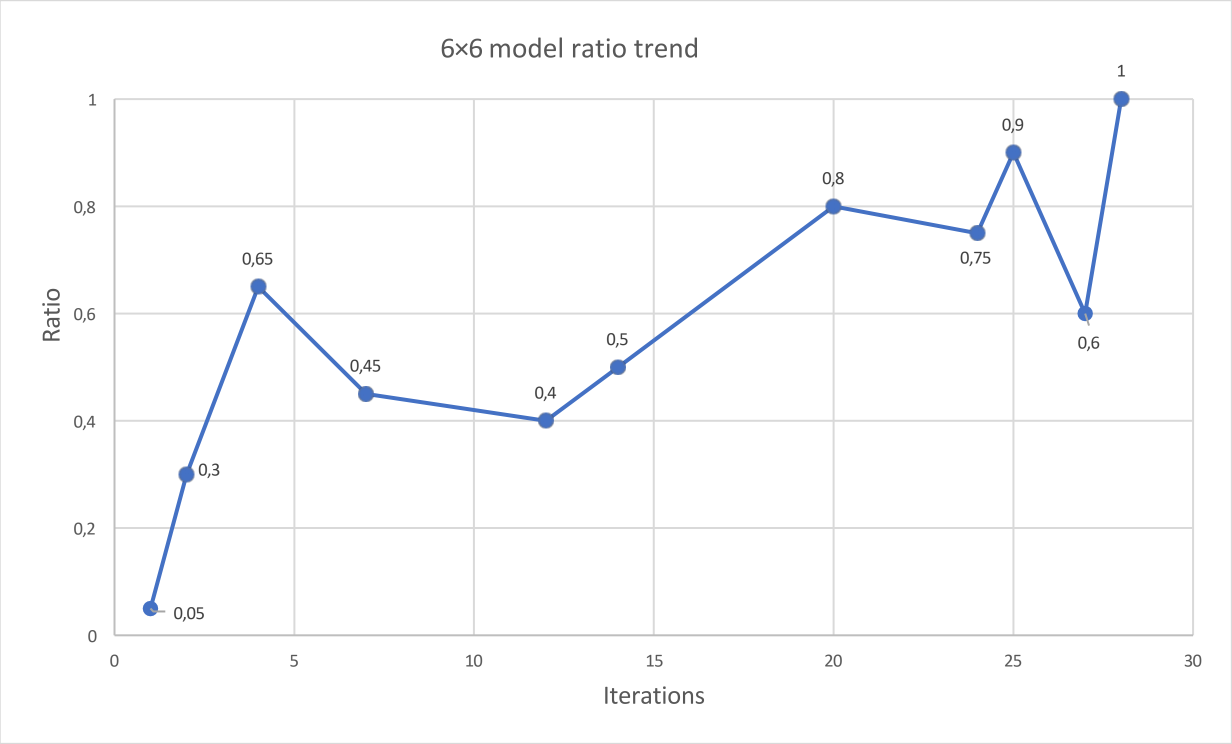
The results of the matches are summarised in the following table showing the number of victories of the neural model against the different players just mentioned, over 20 matches:

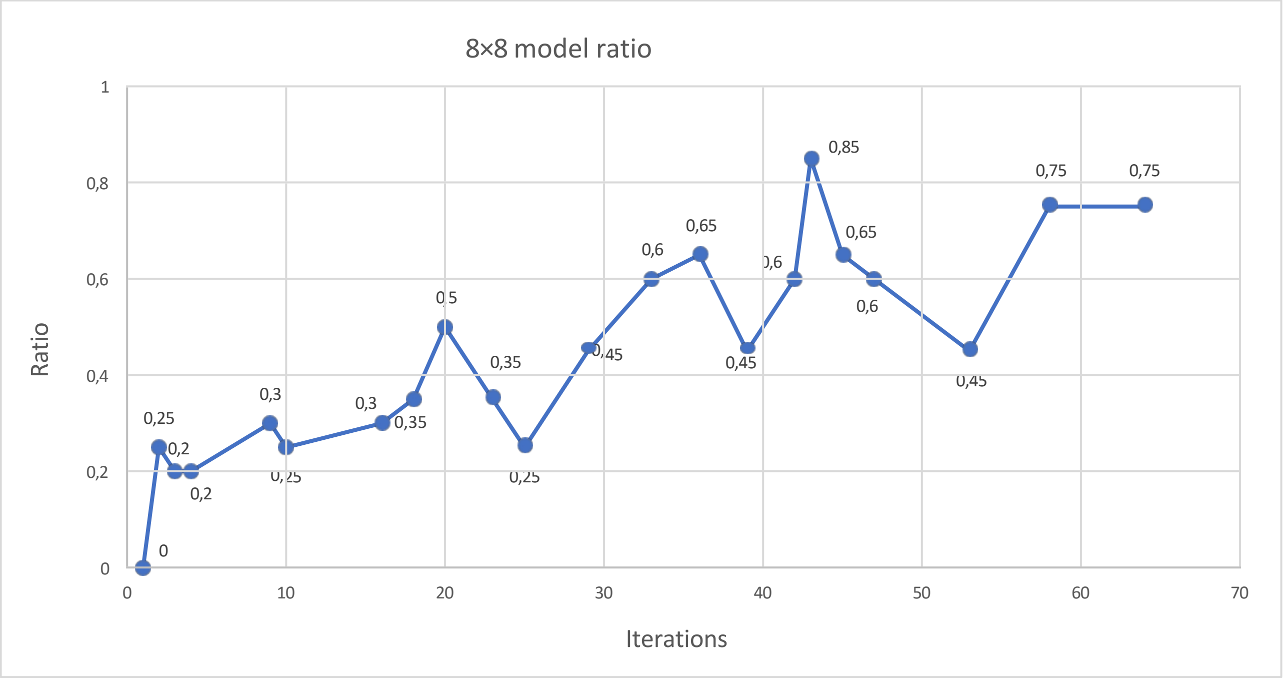
Opposing player	Model 6×6	Model 8×8
Random	20/20	20/20
Greedy	20/20	20/20
Minimax	20/20	17/20

The neural model in the 6×6 octoliter has not lost a single game, while the trained model on the 8×8 always beats the random player and the greedy one, while losing only 3 times out of 20 against the minimax player.

Comparison with minimax

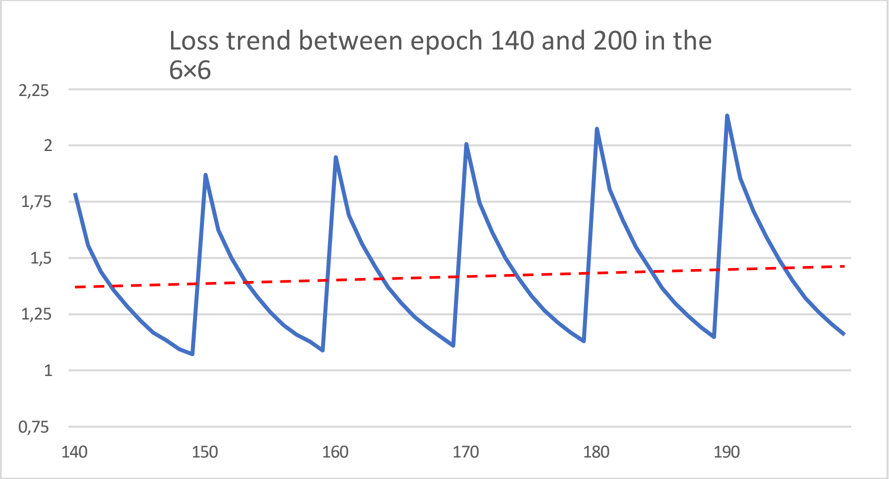
During training, every time the algorithm found a new best model, 20 games were simulated between the new model and minimax, this was done to further proof that the model was learning to play. Below is the trend in the ratio of the neural model's wins to the number of simulations for both boards.





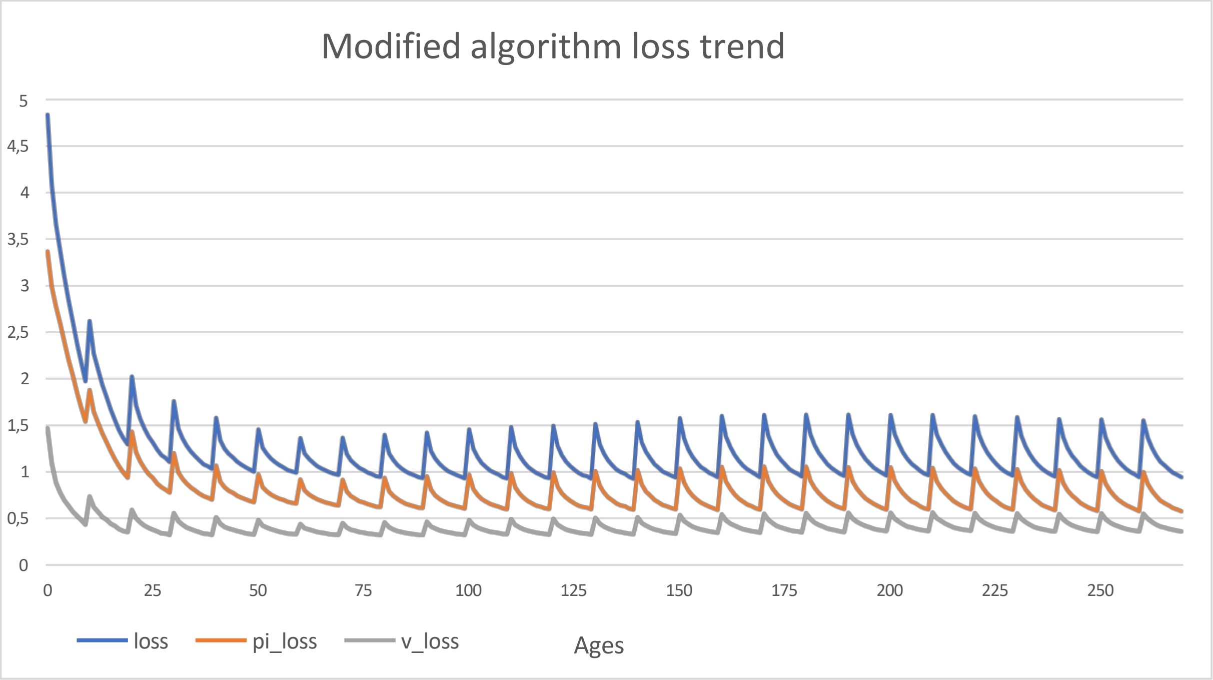
In the 6×6 ophelliary the model updates the best model 10 times and at iteration 28 always manages to beat the minimax player, in the 8×8 on the other hand 21 best models are found and the model that performs best against minimax is the one obtained at iteration 43 which won 17 games out of the 20 played, 85% win rate. Interestingly, the trend is not monotonically increasing, as one might have imagined, but rather is extremely variable. By comparing the best model of the 8×8 model against the model obtained at iteration 43, it was found that the final best model is significantly stronger, in contrast its performance when compared to minimax is lower. The explanation for this phenomenon could be that the model, playing against itself, always develops new strategies for beating itself, but it is not certain that these strategies also apply against the minimax player, perhaps at a certain iteration moves were made that were very useful for beating minimax, but in subsequent iterations they may have been modified as not useful for beating itself. Recent studies have shown, in the game of Go, that making certain moves, which to the eye of a human player may be naive, challenged the neural model by managing to beat AphaGo for the first time in several years. In fact, the greedy heuristics used in minimax are not used for Othello, because having a sloping advantage of one's own colour does not always lead to a higher probability of victory, which may have put the neural model in trouble as minimax does not make moves that it expects.

Further considerations on loss



The following graph shows the trend of the loss between epoch 140, iteration 14, and epoch 200, iteration 20; in particular, it can be seen that the loss does not follow a decreasing trend, but is slowly increasing. This is due to the fact that a new best model is never found between iteration 14 and 20, so the algorithm, at each iteration, will discard the model just trained and start training again from the current best model.

At this point it was decided to train a new model from scratch, modifying the algorithm in the following way, when the new model obtained from the train does not beat the best model it is not discarded, but at the next iteration the new model will continue to be trained until it beats the best model. The following graph shows the trend of the loss obtained from the modification of the algorithm.



The new trend is similar to the previous one, but in this case each epoch has a certain weight for loss minimisation. In fact, at iteration 27 the loss has reached a value of 0.94, whereas in the original algorithm it was 1.08. When the two models are compared, the strength is almost the same, but the modified algorithm seems to find new best models more frequently. This improvement was also used in AlphaZero, a more general version of AlphaGo Zero, to eliminate the best model selection phase altogether and thus save a lot of time during training.

Conclusions

This project presented the AlphaGo Zero algorithm, which makes use of reinforcement learning to learn to play 'without human knowledge'. Prior to this, AIs made use of techniques such as minimax, which, thanks to heuristics appropriately calibrated to the given problem, made it possible to create automatic players. With Alphazero, on the other hand, it is only necessary to define the problem and launch training of the neural network. Furthermore, two neural models were trained on the game Othello, the first on the 6×6 board size and the second on the 8×8 board size, which achieved excellent results when compared to classical algorithms such as minimax. The two neural models are also faster in choosing a move, which means that by visiting fewer states they manage to choose a better move, a sign that the network has learnt an optimal game policy. The training of the models was carried out on a PC with limited computing power, but by optimising the code, it was possible to obtain relevant results in a reasonable time. The various papers describing the algorithm, in fact, use several cpu and gpu for training AlphaZero, e.g. the original paper uses 19 cpu, 4 tpu and 64 gpu.