COMP 767: Final Project

Playing Tic-Tac-Toe with AlphaZero

Peter Quinn
260689207
peter.quinn@mail.mcgill.ca

Mahyar Bayran
260833355
mahyar.bayran@mail.mcgill.ca

Due: April 30, 2020

All the code for this project can be found at the Github link:
github.com/PeterQuinn396/COMP-767-AlphaZero

The video overview of the project can be found at:
https://youtu.be/ib5hBltclFU

Contents

1 Intro

1.1 Motivation

AlphaZero is an algorithm created by Google Deepmind for training an agent to play various games using reinforcement learning. It is intended to be a general algorithm for any 2 player, zero sum, perfect information game (a category of games that is often referred to as classical games). These types of games include Chess, Go and Tic Tac Toe. The agent is able to learn entirely from self-play. It only needs access to an environment that allows it to simulate games, providing the agent with rules for legal moves and determining the end of the game and the winner.

Our goal was to understand the AlphaZero algorithm by implementing it ourselves, and having it learn to play a simple game. The game we chose was Tic Tac Toe. This game should have small enough state and action spaces that we should be able to create an agent that can learn to play without requiring an extraordinary amount of computation resources.

1.2 Background on AlphaZero and Our Implementation

There are 3 main components in the AlphaZero algorithm that allow it to learn to play games:

a. Neural Network

A neural network is used as the value function and the policy function. In the AlphaZero paper, the authors use a deep neural network with many convolution layers and residual layers. For efficiency, the neural network combines policy prediction and value prediction into a single base network with two separate output heads for the policy and value.

Based on the state of the game, the neural network (with parameters θ) predicts a continuous state value $v_\theta \in [-1, 1]$ and a probability distribution on possible actions p_θ. The state values represent the probability of current player winning from that state.

The neural network plays both players simultaneously, by playing against itself. This is done by having an additional input to the network that indicates which side the network is determining the next move for.

We tried two different NN models for learning this task: a simple fully connected

(FC) architecture, and a more complication architecture with convolutional layers, residual layers and batch norm layers.

The simple FC model had 52 234 trainable parameters. The convolution model had 120 074 training parameters. Diagrams of the construction of the NN models can be seen in the Appendix B.

b. Monte Carlo Tree Search

A technique called Monte Carlo Tree Search (MCTS) is used to explore new actions and to improve the policy. An example of how an MCTS step is done can be found in Appendix A.

When we are doing the tree search, we need to keep track of a few things:

- $Q(s, a)$: the expected return from s when taking action a
- $N(s, a)$: the number of times we have seen the (s, a) across simulations
- $P(s, \cdot) = p_\theta(s)$: the probabilities for taking each action as determined by the NN

For each state-action pair, an upper confidence bound (UCB) value on the return is calculated:

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

The $c_{puct} > 0$ term controls the exploration. The square root term helps to prioritize actions that we have not taken often from the state as it gives less weight to actions we have taken frequently.

From the current game state, a tree based on possible moves and transitions to the next game state is grown. The (move, next state) pair represents a new node in the tree. The NN is used to determine a value for this new node. We use this value in an upper confidence bound calculation to determine the trajectory that will be taken, selecting the action with the highest UCB value to explore.

From each state, a certain number of trajectories are tested. In the original implementation, some Dirichlet noise is added to the root node each time we transition states, in order to ensure that a variety of moves are played and states explored. Instead of adding noise to the probabilities, we allow for there to be an $\epsilon = .1$ chance that a random move is performed.

The authors of the paper identify MCTS as being the most important factor for ensuring that the learning of the algorithm is stable. Learning exclusively from self play is typically unstable. Combining self play with the tree search that explores many possible actions and outcomes, rather than just what the NN identifies as strong, allows unexpectedly strong actions to be found and used to improve the policy.

c. Policy Improvement

The paths through the tree that we simulated are then used to update the policy. The new policy then plays against the old policy a few times, and the better one is kept, and the process repeats. Alternatively, we can forgo this "face off" step and just keep improving the model, which works just as well typically.

The loss function that that we try to minimize is:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \pi_t \log(p_\theta(s_t))$$

This loss has two terms. The first is the mean square error between the value estimate for the states and the final result of the game $z \in \{-1, 1\}$. The other term is the categorical cross entropy of predicting π_t (the target values) from p_θ, the policy the NN outputs for the given state. The $\pi_t(s, a)$ term is proportion of times we took action a from state s, which we saved during the tree search. We found that modifying $\pi_t(s, a)$ to assign a probability of 1 to the action that was found to best during MCTS and 0 to all others made the agent learn better.

1.3 Baseline Methods

We will compare our results with classical SARSA and off-policy Monte Carlo. In these methods, one is trying to learn Q-values but they were developed for single-agent environments. In a game where the agent tries to learn the optimal action through self-play, at each step the agent is trying to maximize its expected return whilst minimizing the expected return of its opponent.

Let's assume that there are two roles playing the game: X and O. Let's assume that at the beginning of the game, the agent takes the role of O and makes the first move. It evaluates the current state and chooses an action according to the policy. For the next move, it's X's turn, meaning the role of the agent has changed to X from O. To make our training process more efficient, we evaluate actions always assuming

that it's O's turn. This assumption requires inverting the state ($Inv(s)$) when it's X's turn and pass the inverted state to the policy. If $s \in S$, then $Inv(s) \in S$ is an state such that every X in s is replaced by O and every O in s is replaced by X.

For the rewarding process, a reward of 100 is given to the role who wins the game and a reward of 0 is for the rest of the state-action pairs. In the classical SARSA, the update for a Q-value is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(St, At)\right]$$

However, in training a game by self-play, the action at time $t+1$ is taken by a different player than the player in time t. We know that $Q(S_{t+1}, A_{t+1})$ is the expected return of taking the action A_{t+1} at S_{t+1}. Therefore, during the update phase of $Q(S_t, A_t)$, it's required to negate $Q(S_{t+1}, A_{t+1})$, meaning that if it's expected that the opponent is going to win the game, then it should have a negative impact on the state-action pair of the last time-step. The described difference in the models can be captured by negating the discount factor γ.

In addition, to ensure exploration during the training phase, we used Upper Confidence Bounds (UCB) policy for training SARSA and ϵ-greedy behavior policy for off-policy Monte Carlo. For parameters, we set $\gamma = -0.9$, $\alpha = 1$, $\epsilon = 0.1$ and we ran the training for 20 million episodes.

2 Results and Discussion

This technique is quite computationally expensive. The AlphaZero results for chess and Go are trained on many of Google's TPU clusters simulating games in parallel. The training uses 5000 1st gen TPUs for generating games, and 64 TPUs 2nd gen for training the NN [1]. Each step of the MCTS uses 800 simulations. For chess 44 million games were simulated during training, and 12 million for Go.

Our tic tac toe game was coded to run on the CPU, so we were not able to take advantage of GPU/TPU hardware acceleration for simulating games and the MCTS. This step consumes the majority of the running time. We were able to use some GPUs for the training of the NN, which did speed up the training. However since our NN was not too large (on the order of 100 000 parameters) it also trained reasonably well on a CPU.

The parameters we used were, $c_{puct} = 2$, $lr = .0001$, 20 games per batch, 60 search steps per node in game, and 150 optimization steps using the latest training

data. These parameters lead to be around 150 states being generated for each batch of games. We performed 300 iterations of generating games / training data and optimizing to fit the latest training data.

We experimented with decaying the learning rate during the training in order to reduce the instability issues and find a better minimum, but we found that this did not have a significant effect. We also experimented with having weight decay on the NN parameters ($\lambda = .001, .0001$) but found that this did not have a significant effect on the agent.

We found that having a replay buffer of states generated helped the agent to learn more consistently. Instead of only taking the states from the last batch of games, we use the replay buffer to hold the 500 states most recently seen.

We observed that having a low loss does not always mean that the agent was performing well in actual games. We tested our agent against a simple agent programmed to follow some simple heuristics that allow it to play well (namely winning when it can, preventing the opponent from winning, playing in the center and preferring corners over edges). Our AlphaZero agent struggled against the heuristic agent. As can be observed in Figure 2 and 4, the agent would sometimes manage to tie all the games played (as would be expected from optimal play from both sides) but the policy would shift over time and the heuristic agent would then be able to beat the AlphaZero agent.

One thing that we observed is that the trained agent was not always capable of exploiting its opponents mistakes. If we played a game against it and purposefully blundered such that the agent could win in a single move, it would not always successfully identify the winning move. We suspect this is due to the fact that the agent plays against itself and does not typically make these kinds of blunders, so the agent is not well trained on the states that arise from blunders. The fully connected AlphaZero agent can be played against by running the play_against_agent.py script in the provided Github link. We tried to have this agent available to play in the simple GUI that was built, but we were not able to get the C++ Pytorch library libtorch working correctly.

For evaluating our results from training with SARSA and off-policy Monte Carlo, we played with the agent through the provided graphical interface provided in the github repository. There are two strategical moves in tic tac toe that can be taken into account for evaluation of the performance. One is that if the starting player puts a mark on the corners, then the opponent should always put a mark on the center otherwise, the starting player would have a winning strategy. Another is that when

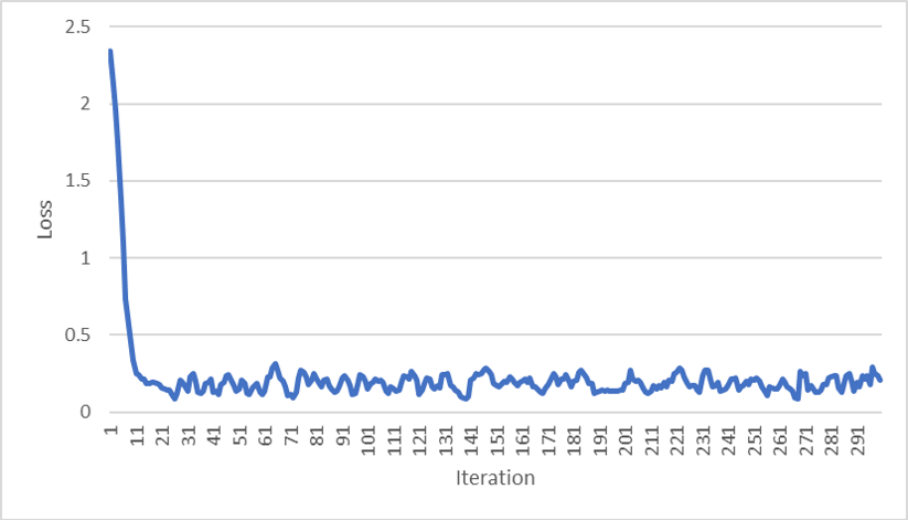

Figure 1: Loss for the fully connected agent over the course of the training. The loss decreases sharply at first but then stays roughly the same.

one player puts 2 marks in a row, column or on the diagonal, the opponent should put a mark to block the player from winning. Based on these evaluation criteria, we can see that the agent performs perfectly and best outcome that the user can get by playing with the agent is going be a tie, i.e. no win is possible for the player and the agent takes advantage of the player's bad moves.

3 Conclusion

This algorithm is extremely computationally intensive. Particularly, simulating many games while examining many possible different actions at each step for the MCTS step is very resource intensive. Generating a large amount of training games is critical for the algorithm to converge to a robust agent. We were not able to train a model that played perfectly every time, but we were able to create an agent that played with at least some understanding of the game. We suspect that by simulating many more games, and many more states in each game, and tweaking some of the hyperparameters, the performance would improve significantly.

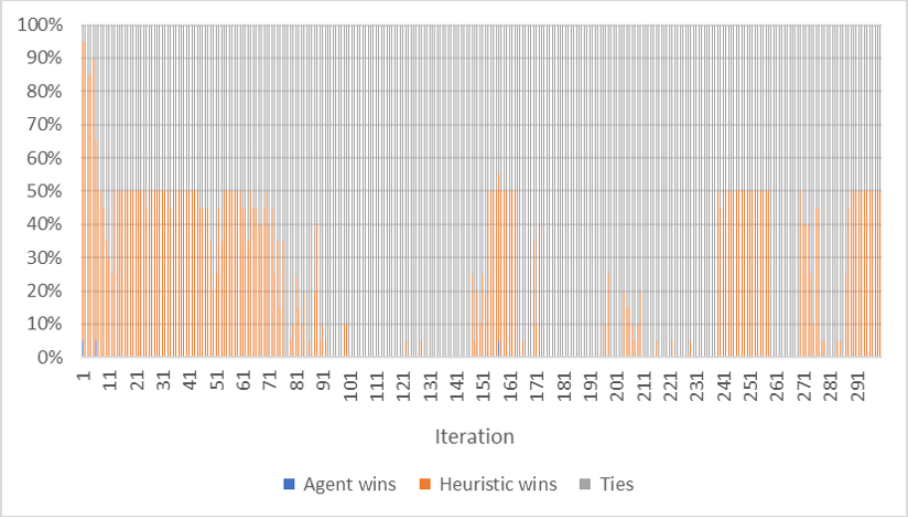

Figure 2: Win rate for the fully connected model agent over the course of the training. After every iteration, the agent being trained is pitted against the simple heuristic bot. 20 games are played in total, each bot playing first half the time.We can see that the policy of the agent is evolving over time, but does not always improve as the heuristic bot is able to win games.

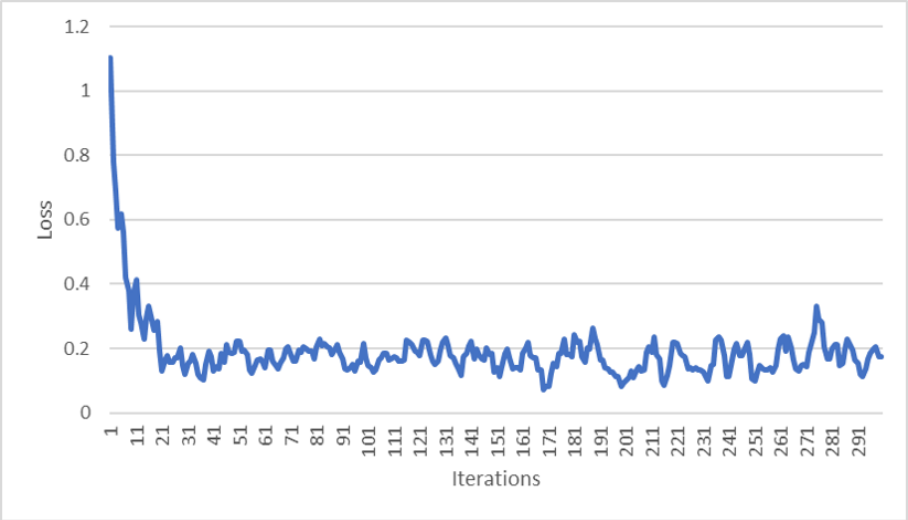

Figure 3: Loss for the convolutional agent over the course of the training.

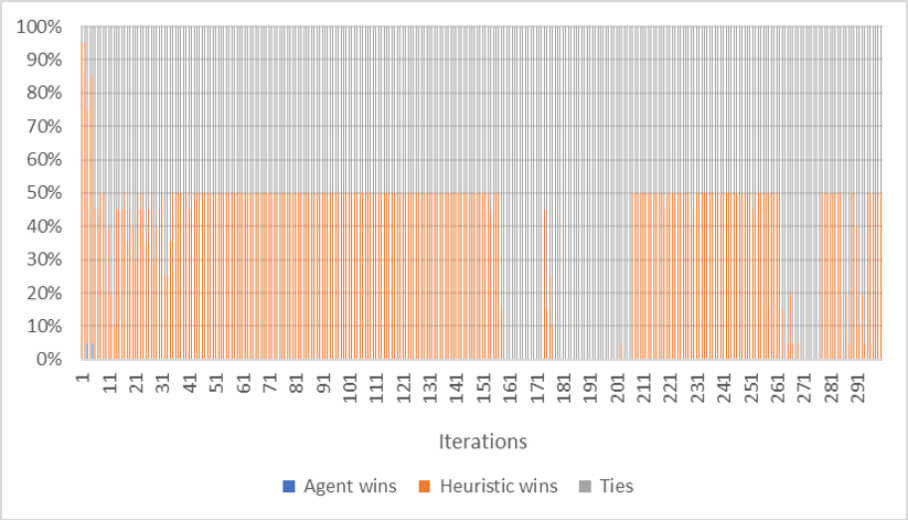

Figure 4: Win rate for the convolution model agent over the course of the training.

4 Contributions

- Mahyar
  - Implementation of GUI and training of baseline methods
- Peter
  - AlphaZero training code

References

[1] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: https://science.sciencemag.org/content/362/6419/1140.

A MCTS Example

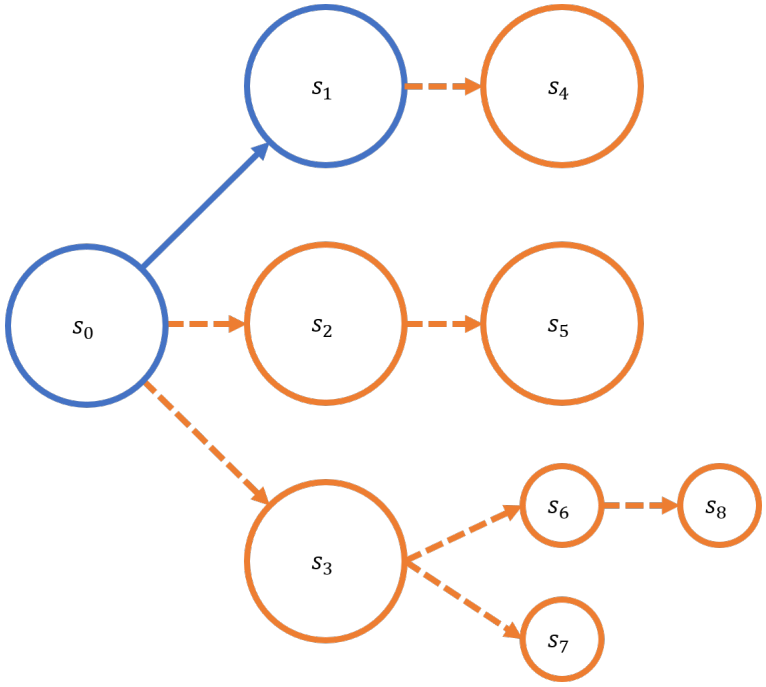

Figure 5: Monte Carlo Tree Search. An example of how one step of MCTS is performed. Starting from the node s_0, we explore a certain number of paths by sampling actions according to our UCB estimate, recording the number of times we have sampled each action. As we go down the tree, if we reach a node we haven't seen before, we use the NN to get a value for the node, and then use it update all the Q values of nodes above it in the tree. If we arrive at a node we have visited before, we recursively explore from this node. If we reach a terminal node, we backpropagate this value up the tree. To determine what actually we actually play in the game we are simulating, we take action with the highest estimated value (blue arrow). We preserve the tree during the game. This search is done for every step of the game until a terminal state is reached.

B NN Architectures

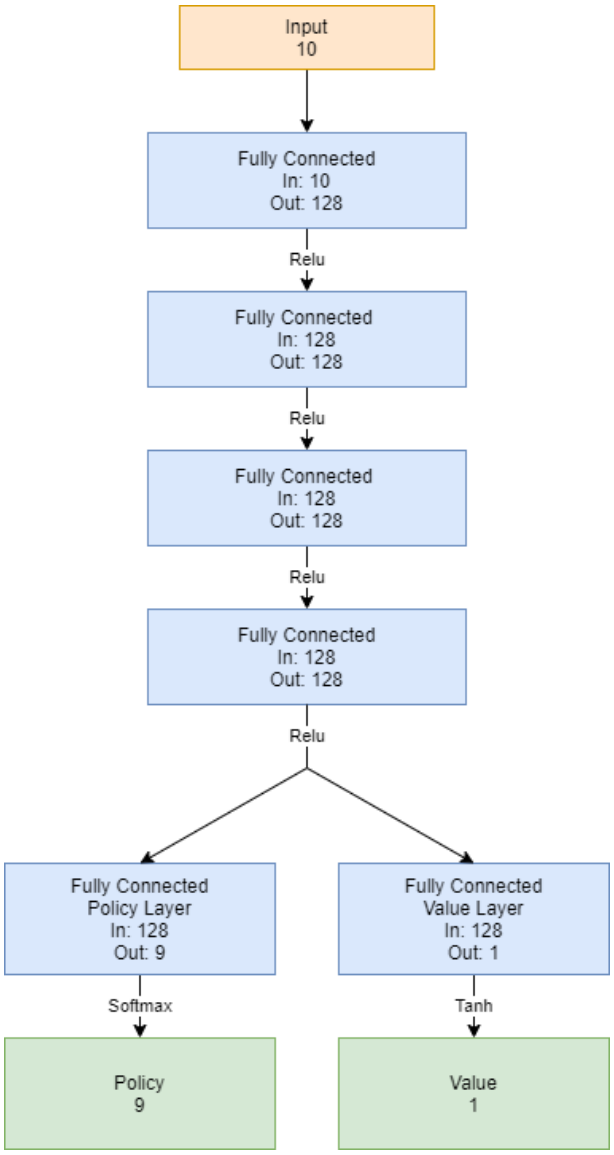

Figure 6: Fully connected model architecture. The input space has dimension 10. The first 9 numbers are the flattened 3x3 grid containing 1 for the 1st players tokens, -1 for the 2nd players tokens, and 0 for an empty space. The final number is the turn, +1 for player 1 and -1 for player 2. The output is the policy (a probability distribution over the 9 possible space to play), and a value (a number between -1 and 1 indicating how the game state favours player 2 or player 1 respectively.

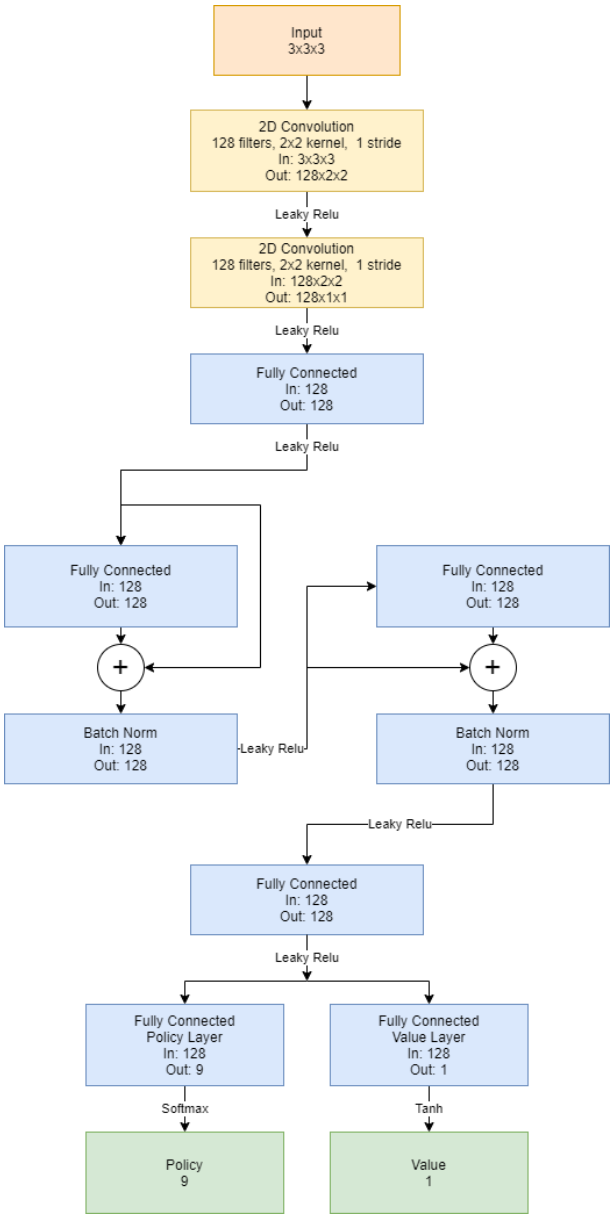

Figure 7: Convolutional model architecture. The input space is 3x3x3. The first layer is the 3x3 game board with one hot encoding for player 1's tokens. The second layer is the 3x3 game with one hot encoding for player 2's tokens. The third layer is the turn (+1 for player 1, -1 for player 2) repeated 3x3. The output is the policy (a probability distribution over the 9 possible space to play), and a value (a number between -1 and 1 indicating how the game state favours player 2 or player 1 respectively.