# Capstone project

Machine Learning Engineer Nanodegree

Daniela Velasquez

April 2021

## 1 Project overview

Artificial Intelligence aimed for years to achieve high performance on games. In its earliest years' it required using domain-specific knowledge to fulfill its objective, this is how in 1997 Deep Blue [2] beat the then-chess champion of the world. Recently AlphaZero [4] captured the attention by conquering one of the most ever devised games: Go [3], a long-standing challenge for the Artificial intelligence community a decade earlier than anticipated starting from tabula rasa and using merely reinforcement learning and deep neural networks. AlphaZero is a generalized algorithm that can self-learn from any domain.

AlphaZero approach will be used to tackle the Robot Reboot game [1], a virtual maze where a robot needs to find its way back home.

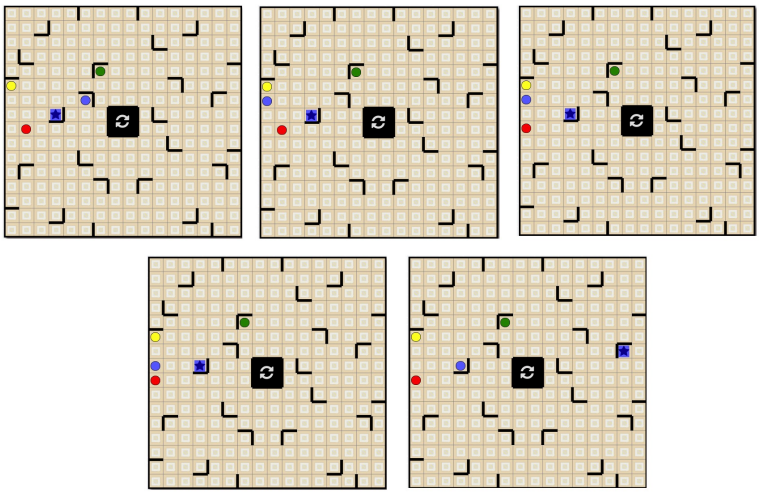

Figure 1: Sample game, states as actions are performed from left to right, The goal is to get the blue robot to its home. Actions performed: Blue moves Left, Red moves Left (It is moved to help to stop the blue robot so it could enter its house), Blue moves Down and Blue moves Right. As soon as the Blue robot gets home the game reset and the house is moved to continue playing.

## 2 Problem statement

The original robot reboot game consists of a 16x16 maze with four robots: Red, Green, Yellow and Blue along with a house for one of the robots. Robots can only move in straight lines (up, down, right and left) and will only stop moving if they hit a wall or another robot. The goal of the game is to get the designated robot to its house. The less movements among all the robots the higher the score. The game has two possible outcomes, either it wins, meaning the robot reached home or it didn't therefore it's a loss.

The objective is to train a neural network [4]

$$(p, v) = f_\theta(s)$$

with parameters $\theta$ that can predict the best action from a game state and the state outcome: either a win or a loss. The neural network will take the game state $s$ as input and predicts a vector $p$ of probabilities of winning per action, and a scalar value $v$ estimating the predicted outcome ($-1$ for a loss or 1 for a win) .

## 2.1 Project design

## 2.2 Robot reboot game

A robot reboot state is defined by a maze, robots positions and a robots houses. The state is represented by a 3-dimensional array where the first layer is the maze, second layer is the first robot's position, third layer is the first robot's house position, fourth layer is the second's robot position, fifth layer is the second's robot's house position and so on for each robot.

The maze is represented by a 2-dimensional array filled with 0's for empty cells and 1's for walls. Mapping walls into their own cells, therefore an $n$ x $n$ maze is defined as a $2n-1$ x $2n-1$ array.

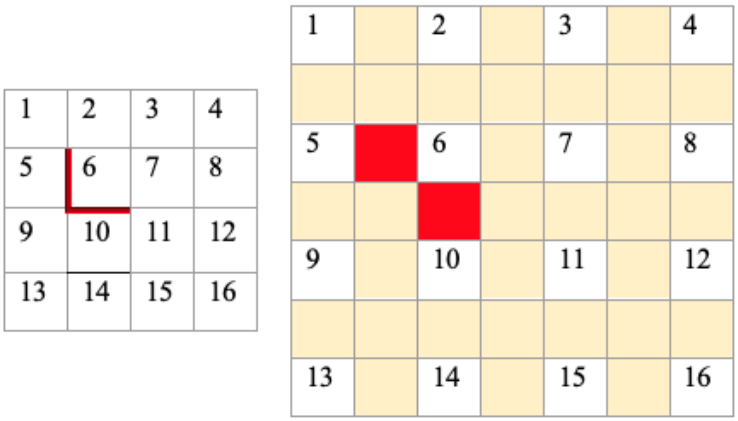


Figure 2: Example on mapping the game maze into an array. A 4 x 4 maze (left side of the image) with a wall to the left and bottom for the cell number 6 is converted into a 7 x 7 matrix (right side of the image) where the red cells represent the maze's walls, yellow cells would be used to portrayed walls if any from the maze.

Each robot position is described as a $2n-1$ x $2n-1$ array filled with zeros except for the cell where the robot is located.

Each robot's house position is described as a $2n-1$ x $2n-1$ array filled with zeros except for the cell where the robot's house is located, the game only has the house for one robot, if the robot doesn't need to get to it house this layer will be filled with zeros only.

A robot reboot action is represented by a robot and a direction to move that robot, for instance if there are two robots in the game then the actions are [(1, Right), (1, Left), (1, Up), (1, Down), (2, Right), (2, Left), (2, Up), (2, Down)] where each tuple is (robot, direction).

## 2.3 AlphaZero algorithm

AlphaZero generates its own data to train $f_\theta(s)$ from self-played using a Monte-Carlo tree search (MCTS). Overall the steps for one iteration are depicted in Figure 3:

1. Build a dataset using AlphaZero, in other words MCTS guided by $f_\theta(s)$.
2. Train $f_\theta(s)$ to get $f'_\theta(s)$.
3. Select model with better performance after playing $x$ number of times.

At first $f_\theta(s)$ is randomly initialized with $\theta$ values and then the outcome from step 3 is used on step 1.

**Algorithm 1:** Monte-Carlo Tree Search (MCTS)

**Input:** $s$ initial state
**Input:** *actions* game's actions
$p \leftarrow 0$;
**foreach** $a \in actions$ **do**
  $s' \leftarrow s.apply(a)$;
  $p[a] \leftarrow simulations(s')$;
**end**
**return** $p$

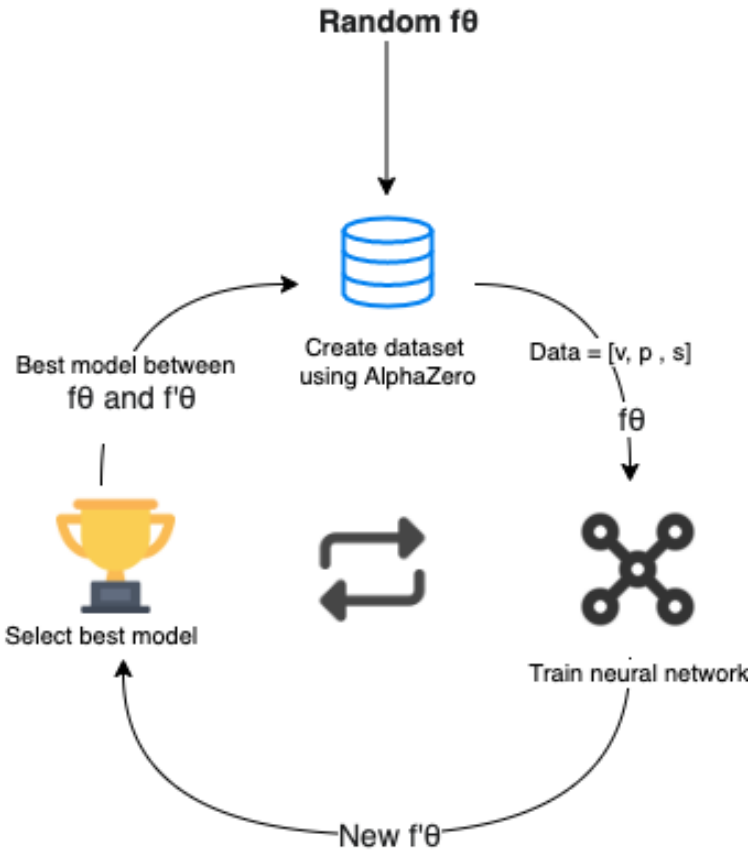


Figure 3: Overall steps for AlphaZero Algorithm

For each next state $s'$ $m$ simulations are played, the branch is explored to a leaf (i.e until the game is over) and the outcome $v$, whether is a win or loss is saved to later on be computed and to estimate how many times the game wins by exploring this branch.

**Algorithm 2:** Simulations

**Input:** $s$ state to simulate
$v \leftarrow 0$;
**for** $i \leftarrow 0$ **to** $m$ **do**
  $v[i] \leftarrow playout(s)$;
**end**
**return** $mean(v)$

A playout explores a branch until the game is over, selecting the next action based on a heuristic value computed as $q + U(p, n)$ where $q$ is an array for the average of values on the edge from $s$ to each action $a$ and $n$ is an array with the number of visits on the same edge. $U$ is a function that controls how much of exploration the tree will handle in order to give a heuristic value $h$, this determines how good each action is.

**Algorithm 3:** Playout: self-play

**Input:** $s$ state to simulate
**if** $s.end()$ **then**
  **return** $s.value$;
**end**
$p, v \leftarrow f_\theta(s)$;
$h \leftarrow q + U(p, n)$;
$a \leftarrow best(h)$;
$s' \leftarrow s.apply(a)$;
$v \leftarrow playout(s')$;
**return** $v$

Finally to get $v$ (which is the outcome of $s$: win or loss) a self-play will occur taking the best actions predicted by $f_\theta(s)$.

**Algorithm 4:** Play $v$

```
Input: s state
if s.end() then
    return s.value;
end
p, v ← f_θ(s);
a ← best(p);
s' ← s.apply(a);
v ← play(s');
return v
```

### 2.4 Dataset and inputs

Given that a robot reboot game state $s$ is represented by a 3-dimensional array $f_\theta(s)$ is a Convolutional Neural Network (CNN). A sample of the dataset is represented by [$v$,$p$, $s$]. A state $s$ is randomly generated, which means each state situates robots in distinct positions, vary which robot needs to get home, its house spot as well as the maze (it means walls are arranged differently).

$p$ is calculated for a state $s$ from simulated games of self-play building a MCTS guided by $f_\theta(s)$. MCTS algorithm creates a tree to explore the outcomes $v$ for each of the possible actions. Edges on the three collect the number of times they have been visited $n$, the average number of wins, losses and draws after they have been visited $q$. Those values are updated as the tree is explored.

## 3 Metrics

Metrics to consider are: Accuracy, game score $v$ and number of wins, losses and draws. Those metrics will be evaluated when selecting the best model and considered with the same order of relevance described previously.

## 4 Data exploration

The dataset is represented as [$p$, $v$, $s$]. There are four robots in the maze: yellow, green, red and blue and the maze size is 15, therefore a state $s$ is represented by an array with dimensions: 31x31x9. 31 represents the maze rows and columns including the walls in between as cells and each layer in the array represents the following:

1. Maze's walls, walls are represented with a 1, empty cells with a a 0
2. 1 that indicates where the robot yellow is positioned in the maze
3. 1 that indicates where the robot yellow's house is positioned in the maze (if any)
4. 1 that indicates where the robot green is positioned in the maze
5. 1 that indicates where the robot green's house is positioned in the maze (if any)
6. 1 that indicates where the robot red is positioned in the maze
7. 1 that indicates where the robot red's house is positioned in the maze (if any)
8. 1 that indicates where the robot blue is positioned in the maze
9. 1 that indicates where the robot blue's house is positioned in the maze (if any)

Its important to recall that only one robot needs to get home, so all robots except for one will have a zero out layer in their respective house layer.

Probabilities $p$ are represented by an 16-dimensional array where each position in the array represents one action:

1. Robot yellow moves North
2. Robot yellow moves East
3. Robot yellow moves South
4. Robot yellow moves West

5. Robot green moves North
6. Robot green moves East
7. Robot green moves South
8. Robot green moves West
9. Robot red moves North
10. Robot red moves East
11. Robot red moves South
12. Robot red moves West
13. Robot blue moves North
14. Robot blue moves East
15. Robot blue moves South
16. Robot blue moves West

$v$ represents a value of the probability of a win or loss for a state $s$.

# 5 Exploratory Visualization

Given the randomness of the actions performed on the maze to generate the dataset, it is key to analyse the number of scenarios in the dataset that represent a win or a loss, the following plot depicts how those are distributed among the entire dataset:

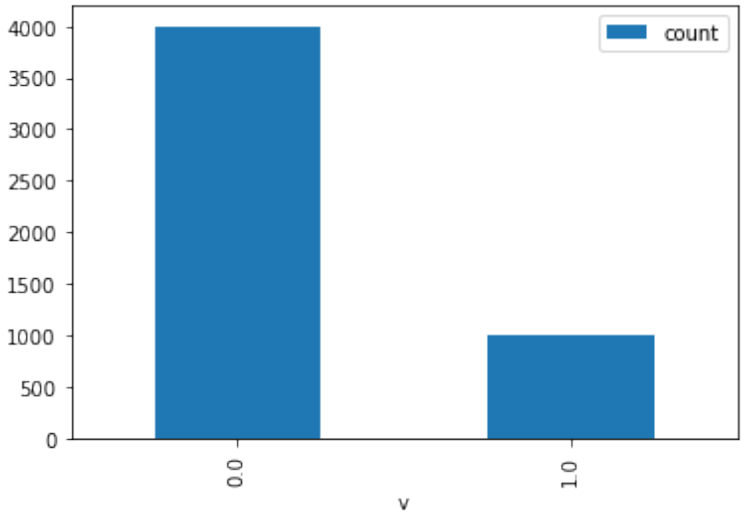


Figure 4: Distribution of wins and losses in the dataset. 0 Represents the number losses and 1 the number of wins

In total there are 3999 loss scenarios and 1001 wins. The dataset is unbalanced, it was split using a stratified approach to keep the same ration of wins and losses between the training and validation dataset.

| Outcome | Train | Validation |
|---|---|---|
| Loss | 2799 | 1200 |
| Win | 701 | 300 |

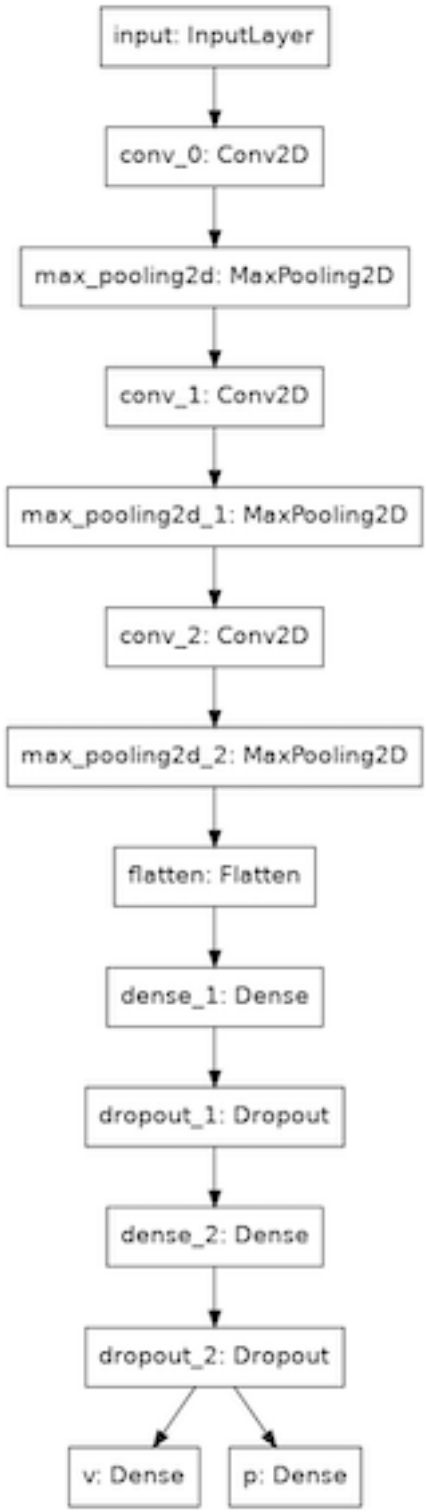


Figure 5: Convolutional Neural network architecture

# 6 Algorithms and Techniques

The neural network architecture is a Convolutional Neural Network (CNN) with two outputs one for $v$ and one for $p$, the former one will be used for classification and $v$ for regression. Dropout is utilized as a regularization technique to prevent over fitting, a common problem faced by CNN.

The following parameters can be tuned into the model:

1. Optimizer.
2. Learning rate for the optimizer.
3. Number of epochs

# 7 Benchmark

A modified version of MCTS known as Upper Confidence Bounds Applied to Trees (UCT) will be compared against the performance of AlphaZero. The UCT algorithm uses a heuristic function that merely rely on the edges visited to calculate the heuristic value that determines how to explore the tree . The heuristic function is

$$v_i = \frac{s_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}}$$

where:

- $s_i$ Aggregate score after applying action $i$
- $n_i$ Number of plays for action $i$
- $N$ Total number of games states simulated
- $C$ Adjustable parameter for exploration

# 8 Data Preprocessing

No preprocessing was needed, data was generated by self play games. In total 5000 samples were generated.

To ensure the presence of wining scenarios with a higher frequency the robot that needs to get home is purposely located in a position close to its home.This approach was considered since the MCTS maximum depth was set to 20 due to time and computational resources limitations to generate a dataset with a significant amount of samples.

The steps to obtain one sample are:

1. Generate a game state $s$.
2. Get probabilities $p$ by applying MCTS.
3. Calculate the state output $v$.

## 8.1 Generate a game state

A game state consist of a maze, the robot house and the four robots positions.

### 8.1.1 Create maze

There are 15 quadrants stored in memory, each of them with different walls and robots houses to be reached (Diagonal walls are not taken into consideration and only one robot house is selected when a state is created). A maze is combination of 4 different quadrants put together ($q1$ for upper left, $q2$ for upper right, $q3$ for lower left and $q4$ for lower right), note that based on the location of the quadrant in the maze, the quadrant is properly rotated.

### 8.1.2 Select a robot house

Each quadrant $q1$, $q2$, $q3$, $q4$ have a set of possible robot houses determined, one robot house is randomly selected, this determines the robot that needs to get home based on the color and the robot's house position in the maze.

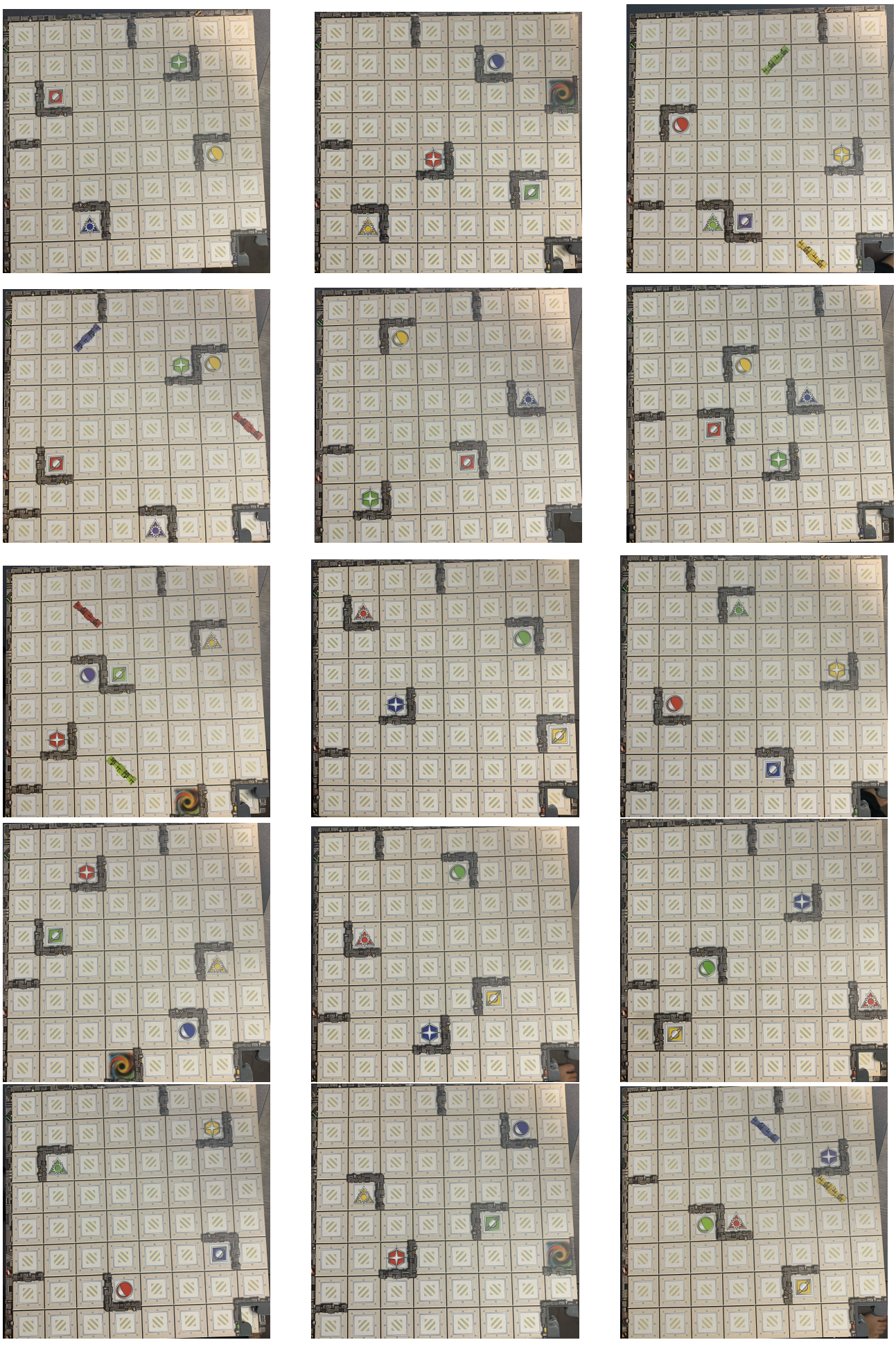

Figure 6: Quadrants used to generate a maze

### 8.1.3 Place robots in the maze

All the robots, except for the one that needs to get home, are randomly place in the maze making sure they don't overlap. The robot selected to get home is placed in its house and then $x$ number of actions (that involved this robot)are applied to the state, $x$ is a random value between one and six. The purpose is to perform backward movements that warranty this robot it is located close to its house and the chances of getting winning scenarios for the dataset increase.

## 8.2 Apply AlphaZero

Once a state $s$ is configured, AlphaZero is used to get the distribution probabilities $p$ of winning based on the state. AlphaZero has a maximum depth of 20, 50 playouts per action and will select the best action based on the current $f'_\theta(s)$ leading the search for exploitation and the number of visits per edge for exploration.

## 8.3 Predict the state outcome

Apply the best predicted action from the $f_\theta(s)$ used to lead the search over $s$ until the end of the game is reached (or more than 20 moves have been performed ) and check the game outcome $v$ at the final state.

**Algorithm 5:** Play: predict the outcome for a state $s$

**Input:** $s$ state
**if** $s.end()$ **then**
  **return** $s.value$;
**end**
$p, v \leftarrow f_\theta(s)$;
$a \leftarrow best(p)$;
$s' \leftarrow s.apply(a)$;
$v \leftarrow play(s')$;
**return** $v$

# 9 Implementation

Most of the code was written using Objective Oriented Programming. The implementation process had three stages:

- Define game framework
- Define the robot reboot game
- Define AlphaZero and UCT Algorithms
- Apply AlphaZero which means: generate the dataset, train the model and compare the new model against the previous model.

## 9.1 Game framework

A mini framework that defines how a game is represented is in the package $robot - reboot/src/game$. A game is composed by actions that are applied to a state. Figure 7 depicts classes defined, a state belong to a game.

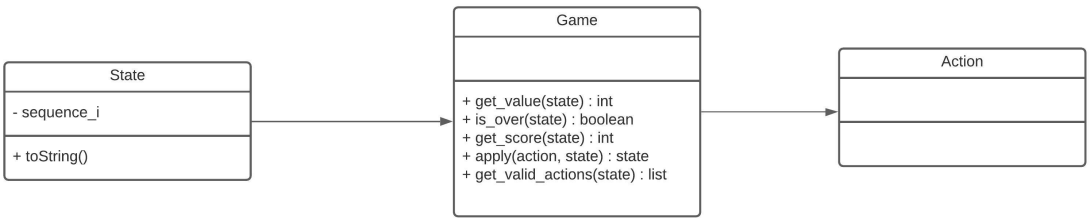


Figure 7: Game diagram

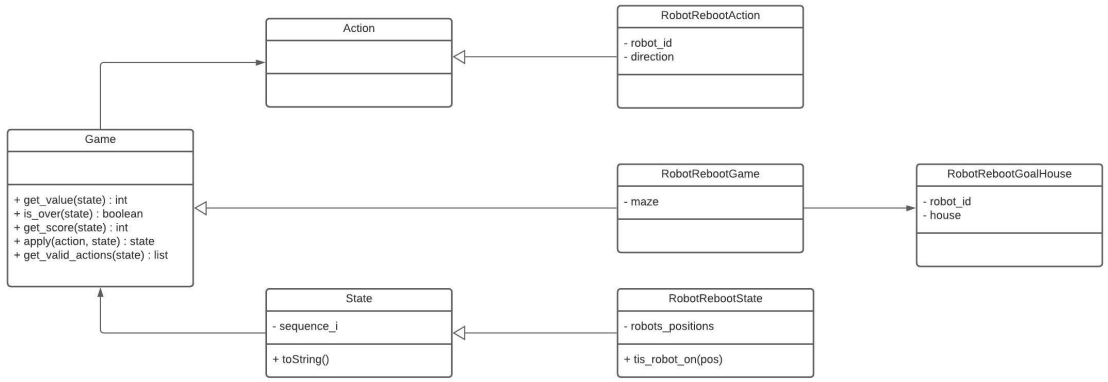


Figure 8: Robot reboot game class diagram

## 9.2 Robot reboot game

Based of the game framework, the robot reboot game is defined, its rules, actions and how a state is represented. Implementation can be found in the package $robot - reboot/src/robot_reboot$. Figure 8 explains in more detail the classes implemented.

- The game stores the maze and its goal, which represents a position in the maze and the robot that needs to get home.
- An action is defined by a robot id and a direction to move such robot.
- A state contains a list for the position in the maze of each robot.

## 9.3 AlphaZero and UCT Algorithms

Both AlphaZero and UCT are algorithms that inherit their main functionality from MCTS, they differ on how the playouts are carried out, therefore MCTS main functionality is defined as an abstract class and both algorithms are in charge of defining the playouts, see Figure 9 for further details. AlphaZero is defined in the package $robot - reboot/src/alphazero$, whereas UCT can be found in $robot - reboot/src/uct$. Both algorithms use a heuristic function to select the next action, in the case of AlphaZero its heuristic function makes a decision based on the $p$ predictions from the current neural network combined with the heuristic function of the UCT algorithm (this is not the original implementation of AlphaZero but in order to help out the algorithm to generate better data faster this approach was taken). A Monte Carlo Tree Search Algorithm makes a search over a game to determine the probability of winning for each action on the game, it also stores the statistics for a state to know how many times it was visited, how many wins after applying an action in that state and the probability of winning when an action is taken. The AlphaZero class requires a Game player as well, this one is in charge of interpreting the results from a model when predicting the probabilities $p$ for a state $s$.

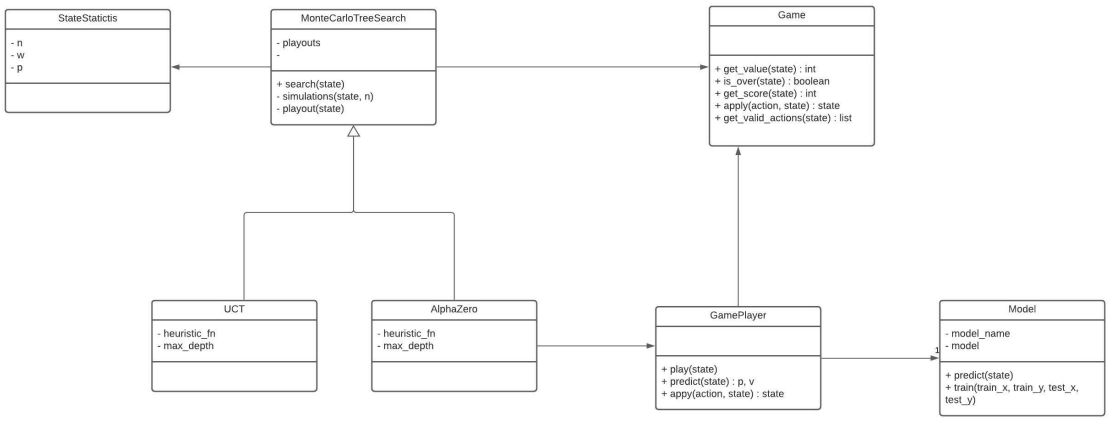


Figure 9: AlphaZero and UCT Algorithms diagram

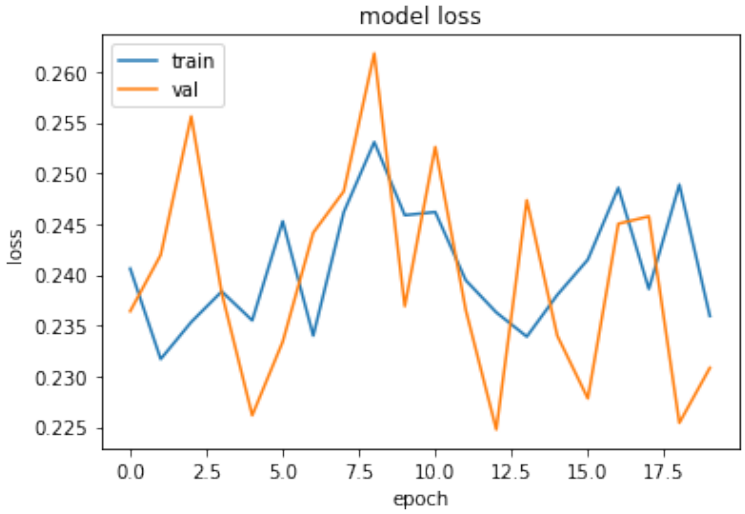


Figure 10: Model loss fluctuation.

### 9.4 Apply AlphaZero

First the convolution neural network is set up with random weights to produce the initial dataset, its weights are store for further use, to ensure all the data is produced equally.

Second, the dataset is produced using multiple processes to speed up the data generation, each process generates one sample and stores it in its own tfrecords file.

Later on, all data samples are put together for data analysis, the main goal is to ensure that the same ratio of winning and loss scenarios are distributed between the training and validation dataset. After this, a dataset for training and validation are created with their corresponding data samples.

Next, the current model is trained using Sagemaker with the following hyper-parameters:

- Optimizer: Adam
- Learning rate: 0.01
- Epochs: 100

The model is setup to reduce the learning rate on Plateau.

The last step for the first iteration consists of making the initial model and the trained model compete using a testing dataset to see which one performs better.

After the best model is selected, the initial model competes against the new trained model and the one with a better performance is selected, ideally this process is repeated to keep improving the search results but the time limitations prevent further iterations.

## 10 Refinement

To produce the dataset initially an approach of multi threading was implemented however due to shared resources like the factory to generate a game and the file to write a sample (the goal was to write all the data samples in one file) caused the time to increase making it an unfeasible approach.

The model was not reducing its learning rate during training which started causing fluctuation in the model's loss throughout the epochs check Figure 10 for a visual reference, this is a symptom that the learning rate was too big, to tackle it the learning rate is reduced during plateau.

## 11 Model Evaluation and Validation

Each CNN for alpha zero was evaluated against a testing data set with 400 states. The results are depicted below, for testing purposes the CNN was merely used to predict games, no the entire MCTS was used.

| Model | Test data | | Validation data | | | Training data | | |
|---|---|---|---|---|---|---|---|---|
| | Average Score | Wins | Loss | Size | Win scenarios | Loss | Size | Win scenarios |
| $f_0$ | -19.53 | 10 | - | - | - | - | - | - |
| $f_1$ | -19.43 | 12 | 0.06 | 900 | 34 | 1.37 | 2100 | 79 |
| $f_2$ | -19.43 | 12 | 1.34 | 897 | 26 | 1.33 | 2091 | 60 |


Figure 11: $f_1$ Model Loss

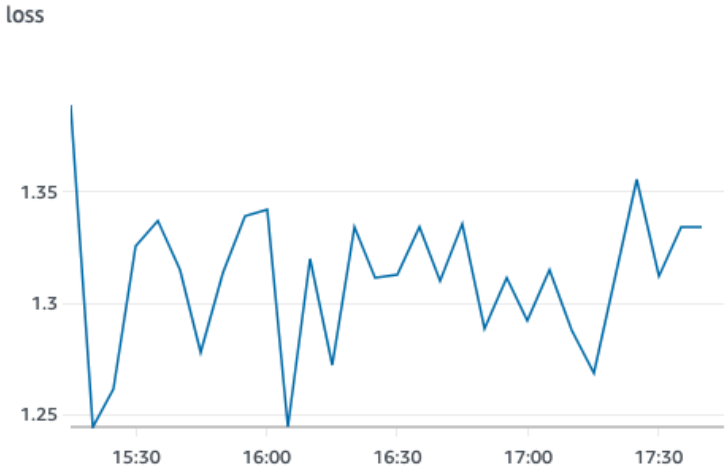


Figure 12: $f_2$ Model Loss

The CNN $f_1$ improved slightly compared to $f_0$ by winning in a couple more of scenarios and getting a better average score. The loss for $f_1$ decreased smoothly throughout the epochs as Figure 11 depicts.

However between $f_1$ and $f_2$ no improvement was achieved overall. When looking at the loss in Figure 12 the loss fluctuated as it was training despite the use of the reduce learning rate technique. Unfortunately before another round of training could be performed, the credit for AWS was maxed out and no further experiments could be performed.

Comparing the results between UCT and Alphazero over a dataset with 30 games (running this simulations take a lot of time) the results were collected on the following table.

| Model | Wins | Average score |
|---|---|---|
| UCT | 18 | -9.13 |
| AlphaZero | 17 | -9.6 |

It is safe to conclude that even thought AlphaZero improved from its first neural network, it requires a lot more data and iterations in order to achieve a super human level. Powerful computer resources are needed to produce big quantities of simulated games in order to achieve a high performing neural network.

# References

[1] Robot reboot, 2016.

[2] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[3] DeepMind. Alphago – the movie.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.