Implementation of AlphaZero Algorithm on Tic-Tac-Toe

Major Project - Semester 7
Group - 8

Introduction

Can Computers Think Like Humans?

- Alan Turing described the very first algorithm by which a computer can be taught to play chess.
- Two approaches:
  - Examining the game tree to a very high depth and evaluating **all possible moves**. (Minimax and *Alpha-Beta* based engines)
  - Examining less number of positions but **selectively looking** at only the **promising moves** (*Human Approach*).
- Until recently, the First approach was only used but even this approach is not feasible in game with large branching factor (e.g. Go).

Can Computers Think Like Humans?

- Also Engines today have **inbuilt domain specific knowledge** i.e Chess Engines and Go engines are very different.
    - Chess engines have evaluation function built by consulting expert chess players.
- No Chess engine can play Go well, or vice versa.

- Key Insight: **Domain specific improvements lead to loss in generality.**

Can Computers Think Like Humans?

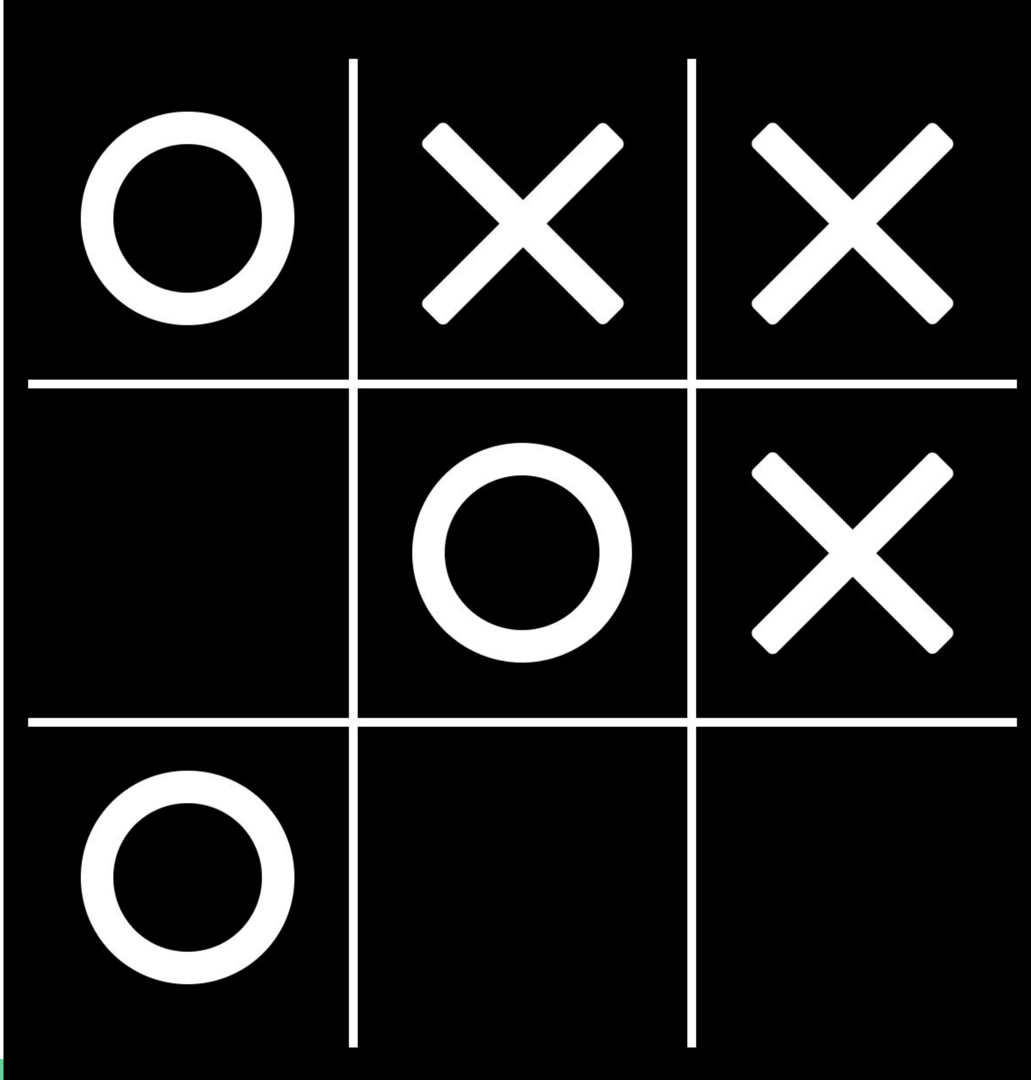
- However, a recent breakthrough in deep learning which goes by the name **AlphaZero** algorithm was announced by DeepMind. It mastered three games - Chess, Go and Shogi and beating existing world champions (humans and engines) in all three domains.

- It started from scratch with **no initial human knowledge** and given only the **rules of the game**.

- The program used **Deep Neural networks** combined with a **Tree search** algorithm to achieve this.

Our Work

- Implementing this algorithm for the game of Tic-Tac-Toe
- Understanding how the tree search works
- Evaluating and Analysing performance of this algorithm



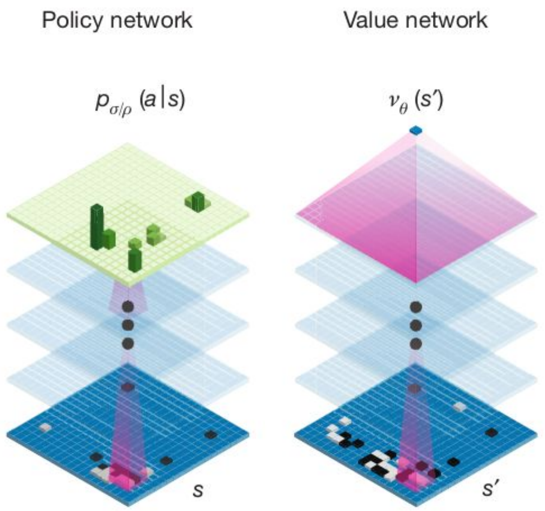
Go Match between AlphaGo and Lee Sedol

AlphaGo won 4 games to 1 against the Go World Champion.

(December 2016)

Research Continues...

- AlphaZero searches **just 80,000 positions** per second compared to **70 million for Stockfish**.
- AlphaZero compensates for the lower number of evaluations by using its **deep neural network** to focus much more selectively on the most promising variation.
- It uses the networks to approximate **Position Evaluation** and **Move Ordering** heuristics.



The problem or challenge

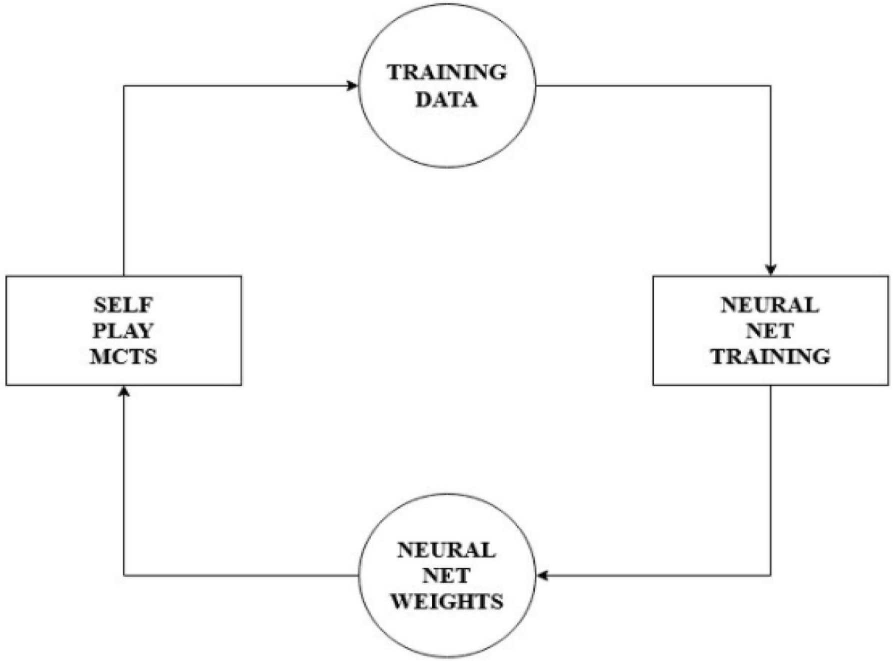
Creating an algorithm that can **learn without human knowledge** can **think like a human** and beyond in a variety of games.

Description Of The Algorithm

Overview

The algorithm can be divided into these parts :

- **Self-Play using MCTS Algorithm**
- **Neural Net training**



Self Play

- Self play improves quality of training data by searching to higher depth.
- Self play works by running MCTS, then choosing the move according to the probabilities.

Neural Network

$$(\mathbf{p}, v) = f_\theta(s)$$

- Neural Network is concerned with heuristics of MCTS.
- Input to DNN is the **Game State.**
- MCTS relies on Neural Net for expansion.
- The predictions of this network are used to guide a tree search in a series of self play games.

MCTS

- It is heuristic driven search algorithm.
- **MCTS** = **Classic Tree Search** + **Principles Of Reinforcement Learning.**
- MCTS relies on exploration and exploitation strategies.
- Exploration can be thought as searching for new strategies.
- Exploitation is choosing best strategies.
- MCTS only searches **fewer positions** but gives priority to important parts of tree to explore.
- It simulates the outcome rather than exhaustively spending the search space.

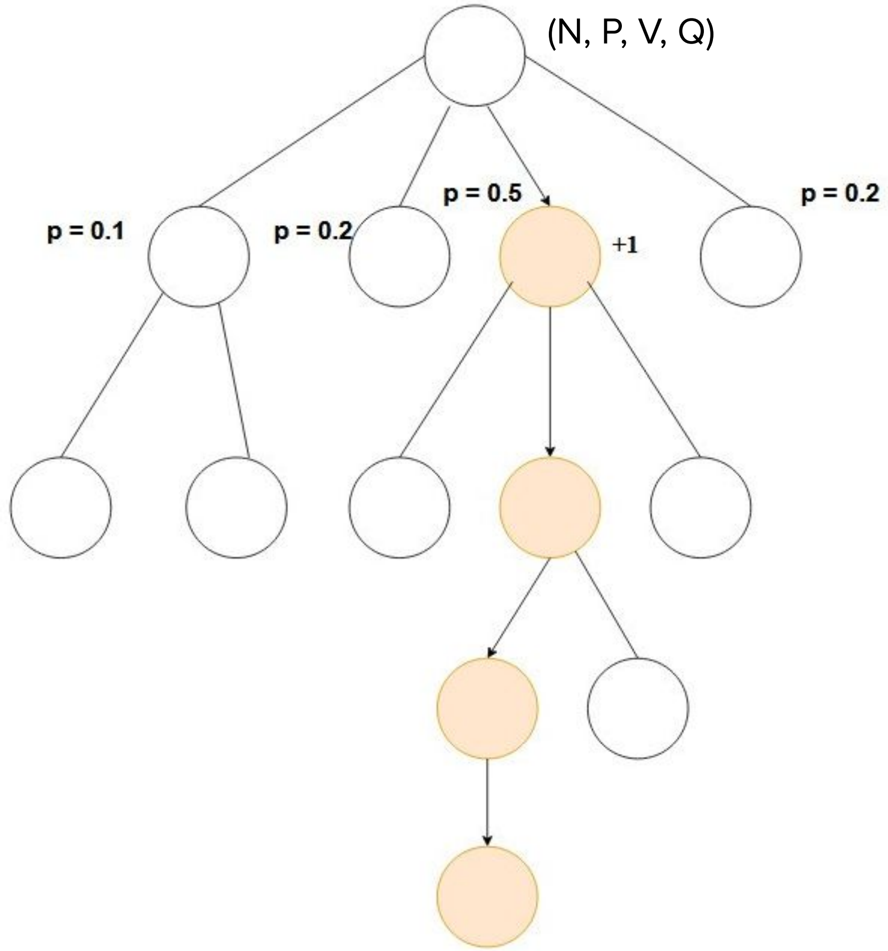
MCTS ...

There are 4 distinct phases of MCTS :

- **Simulation:** Selecting a path through the search tree.
- **Evaluation:** Once a path is being selected, the search tree is expanded and explored.
- **Backpropagation:** After expansion, simulation is done to figure out what it is we are supposed to be doing to make selections and what's being learned through simulation is backed up which will allow us to select what to do next.
- **Play:** The above mention steps are being repeated a number of times and finally the best method is being chosen.

MONTE CARLO TREE SEARCH

1. Simulation
2. Evaluation/Expansion
3. Backpropagation
4. Play



MCTS...

$$\text{UCT Score(action } a) = Q_a + c_{\text{puct}} \cdot P_a \frac{\sqrt{N_b}}{1 + N_a}$$

Architecture And Implementation

Two prototypes :

- C++ w/o neural networks.
- Python with neural networks.

C++ Prototype

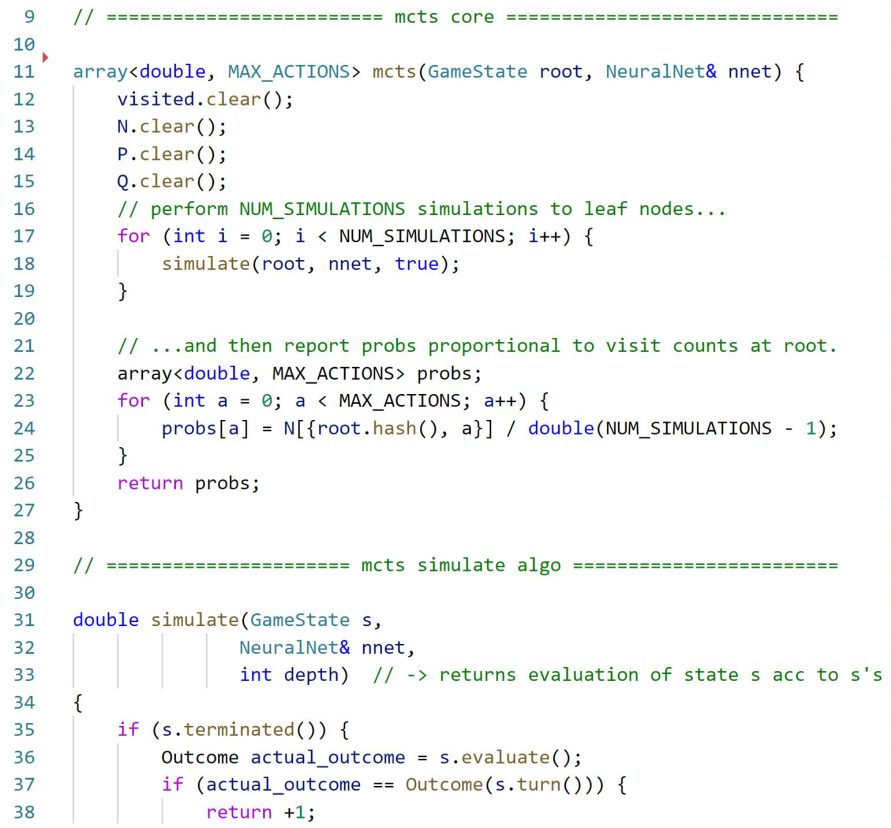
- Prototype version - Used lookup tables instead of neural network to speed up training.
  - Commonly used to test algorithms In Reinforcement Learning
- Self Play is a CPU intensive task.
- C++ is suitable for writing CPU intensive applications.

Limitations :

- Tried using Eigen + MiniDNN libraries for Neural network, but:
- Could Not Use Multi-headed Neural Network.
- Could Not Run On GPU.

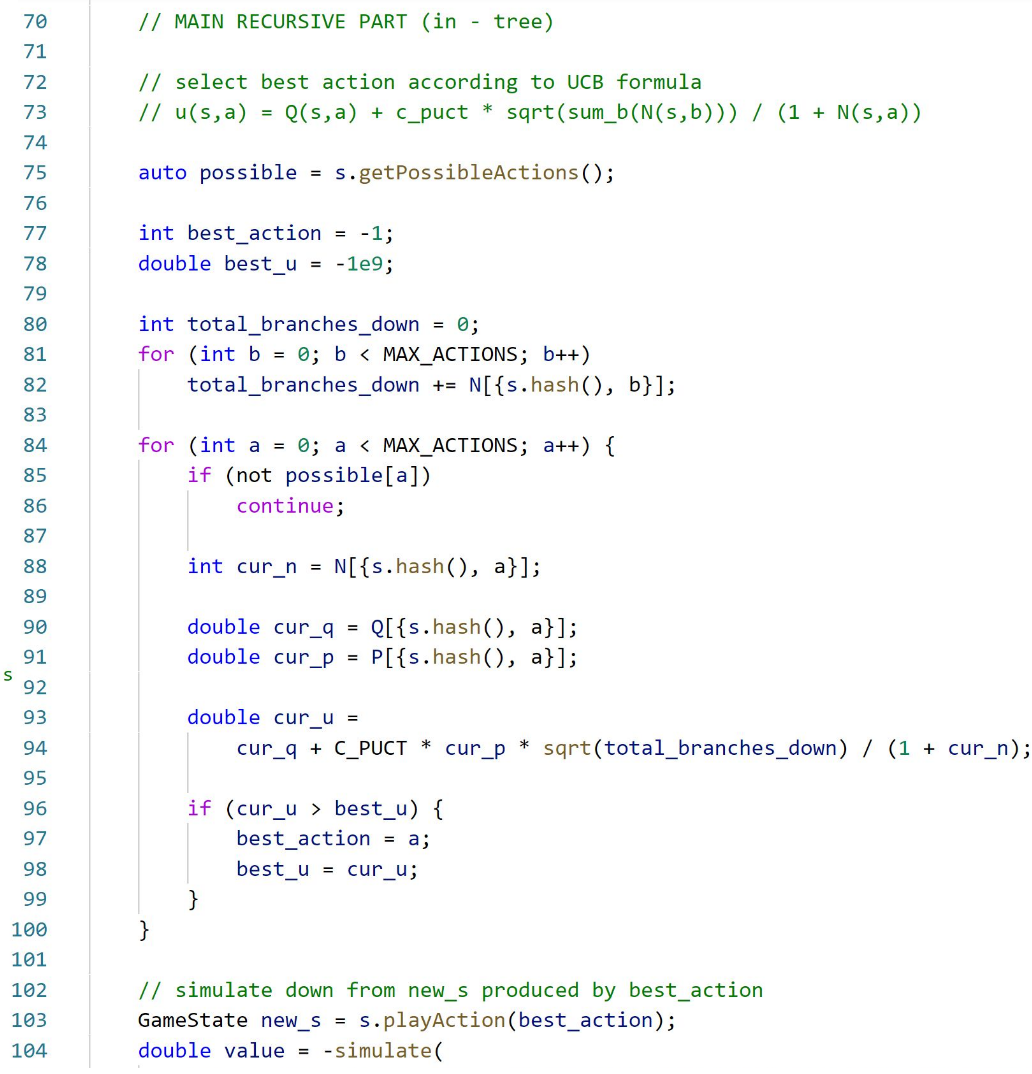
Results

- Using C++ w/o Neural Networks



```cpp
// ========================= mcts core ==============================

array<double, MAX_ACTIONS> mcts(GameState root, NeuralNet& nnet) {
    visited.clear();
    N.clear();
    P.clear();
    Q.clear();
    // perform NUM_SIMULATIONS simulations to leaf nodes...
    for (int i = 0; i < NUM_SIMULATIONS; i++) {
        simulate(root, nnet, true);
    }

    // ...and then report probs proportional to visit counts at root.
    array<double, MAX_ACTIONS> probs;
    for (int a = 0; a < MAX_ACTIONS; a++) {
        probs[a] = N[{root.hash(), a}] / double(NUM_SIMULATIONS - 1);
    }
    return probs;
}

// ====================== mcts simulate algo ========================

double simulate(GameState s,
                NeuralNet& nnet,
                int depth)  // -> returns evaluation of state s acc to s's
{
    if (s.terminated()) {
        Outcome actual_outcome = s.evaluate();
        if (actual_outcome == Outcome(s.turn())) {
            return +1;
```

MCTS Code

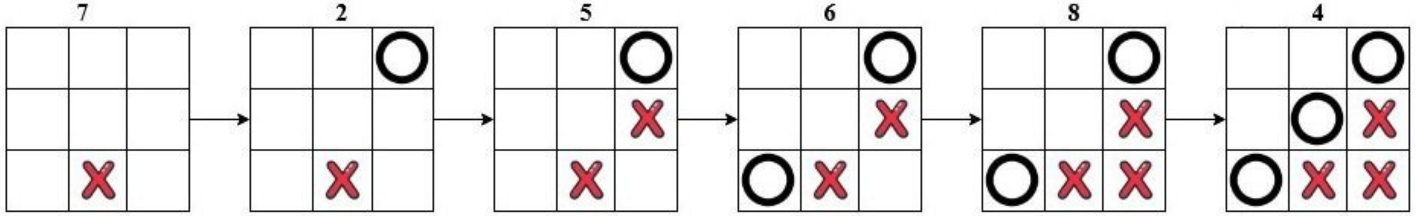


```cpp
    // MAIN RECURSIVE PART (in - tree)

    // select best action according to UCB formula
    // u(s,a) = Q(s,a) + c_puct * sqrt(sum_b(N(s,b))) / (1 + N(s,a))

    auto possible = s.getPossibleActions();

    int best_action = -1;
    double best_u = -1e9;

    int total_branches_down = 0;
    for (int b = 0; b < MAX_ACTIONS; b++)
        total_branches_down += N[{s.hash(), b}];

    for (int a = 0; a < MAX_ACTIONS; a++) {
        if (not possible[a])
            continue;

        int cur_n = N[{s.hash(), a}];

        double cur_q = Q[{s.hash(), a}];
        double cur_p = P[{s.hash(), a}];

        double cur_u =
            cur_q + C_PUCT * cur_p * sqrt(total_branches_down) / (1 + cur_n);

        if (cur_u > best_u) {
            best_action = a;
            best_u = cur_u;
        }
    }

    // simulate down from new_s produced by best_action
    GameState new_s = s.playAction(best_action);
    double value = -simulate(
1     725684 -> o
2     26130 -> x
3     52483 -> x
4     560412 -> o
5     0712346 -> x
6     147523 -> o
7     4380652 -> x
8     3712680 -> x
9     83024 -> x
10    720153486 -> .
11    28160 -> x
12    8061274 -> x
13    20486 -> x
14    34527160 -> o
15    28476 -> x
16    40672 -> x
17    678451 -> o
18    4357802 -> x
19    24160 -> x
20    3106475 -> x
```

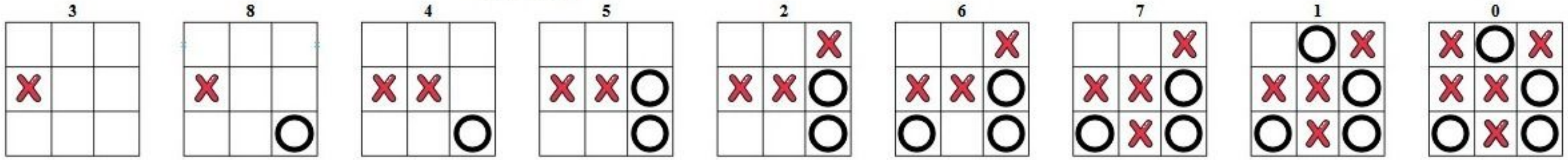
```
99981    042173586 -> .
99982    042175368 -> .
99983    042175368 -> .
99984    042175368 -> .
99985    042175368 -> .
99986    042175368 -> .
99987    042175368 -> .
99988    0312457 -> x
99989    042137 -> o
99990    042175368 -> .
99991    304526178 -> .
99992    38415 -> x
99993    348176250 -> .
99994    3485607 -> x
99995    042175368 -> .
99996    384520167 -> x
99997    321406 -> o
99998    042175368 -> .
99999    38067214 -> o
100000   384526710 -> .
```

0	1	2
3	4	5
6	7	8

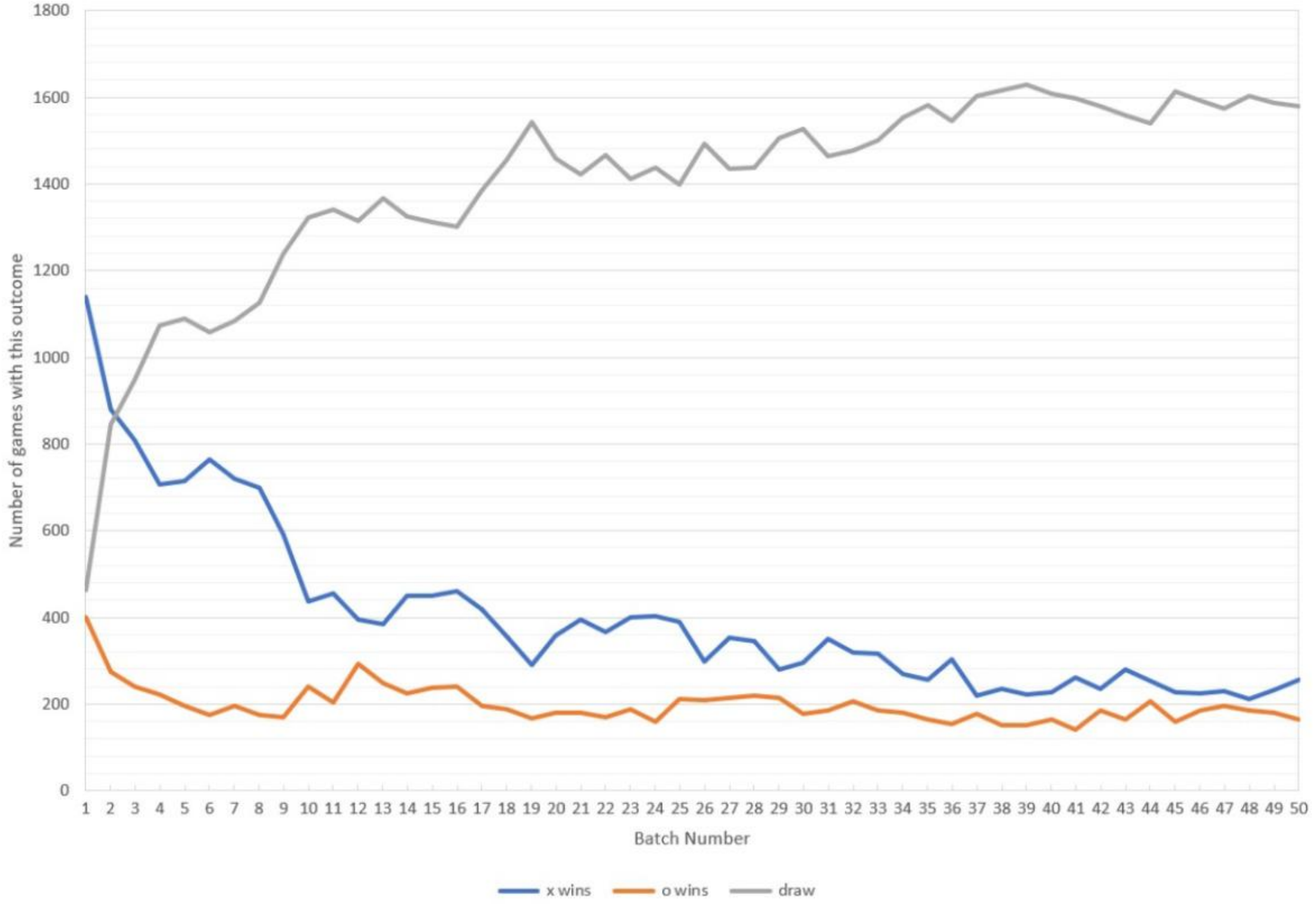
GAME-I



GAME-II

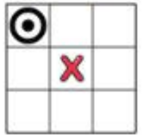


Self-Play game outcomes (100,000 games, batch size = 2000)

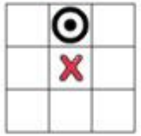


COMBINATIONS OF FIRST TWO MOVES (UPTO ROTATION AND REFLECTION)

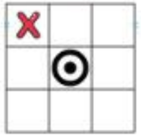
1.CR->CO



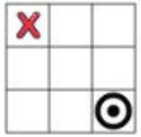
2.CR->ED



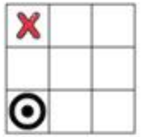
3.CO->CR



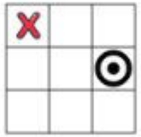
4.CO->CO



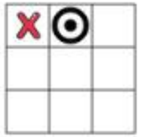
5.CO->CO



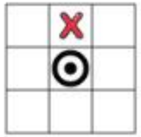
6.CO->KN



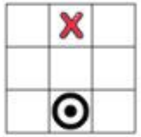
7.CO->AD



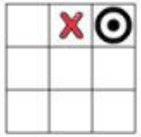
8.ED->CR



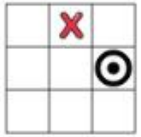
9.ED->OE



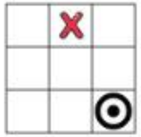
10.ED->AD



11.ED->DL

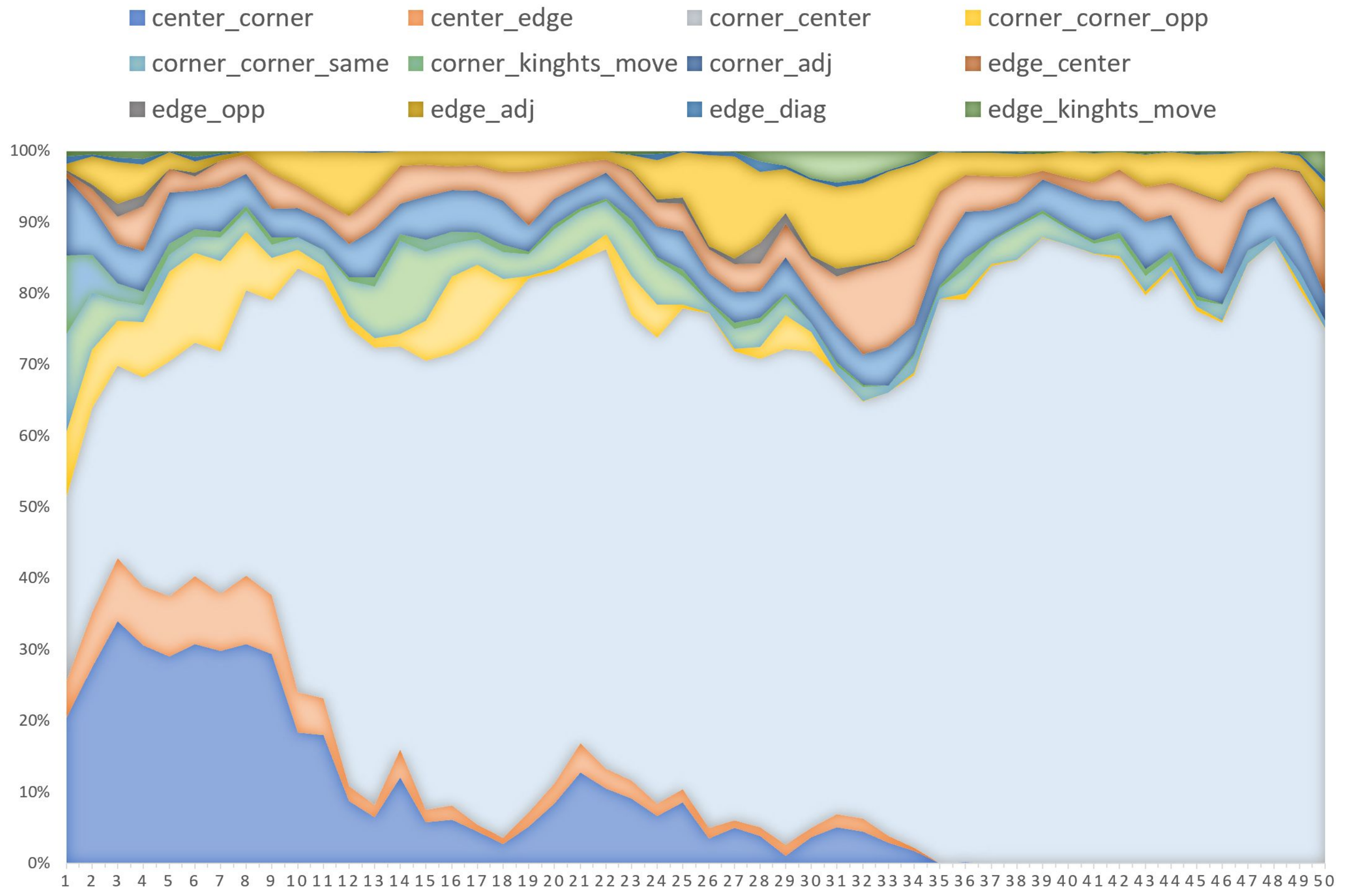


12.ED->KN



ACRONYMS USED
CR - CENTER
EDGE - ED
CORNER - CO
KNIGHT'S MOVE - KN
ADJACENT - AD
DIAGONAL - DL
OPPOSITE EDGE - OE

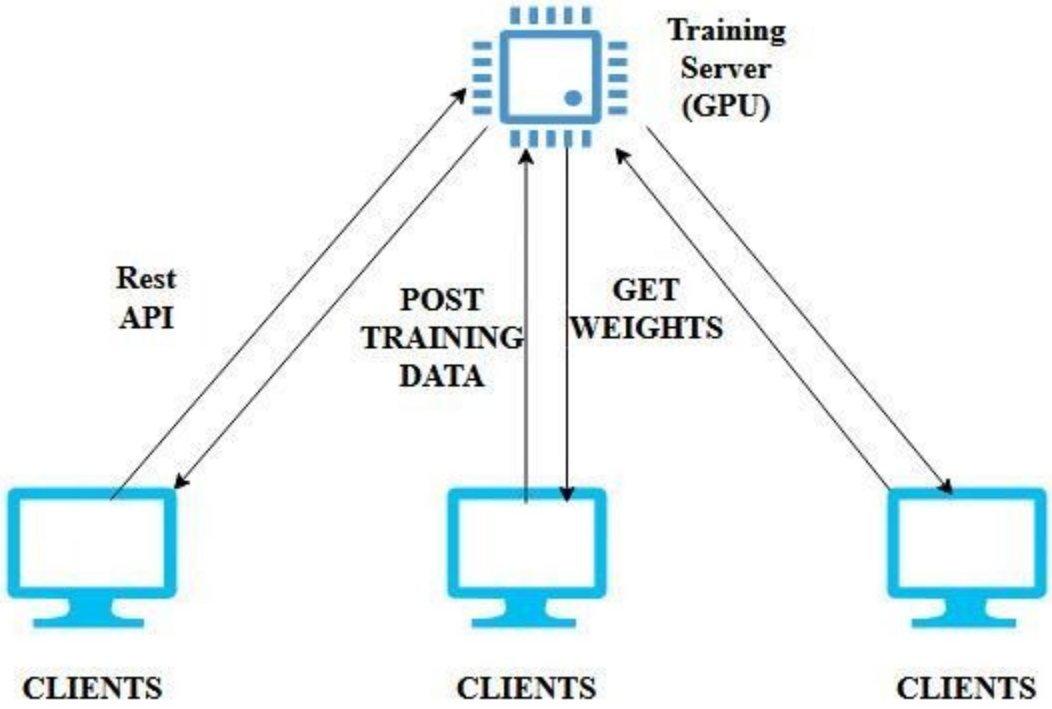
FIRST AND SECOND MOVE (PERCENTAGE OF GAMES)



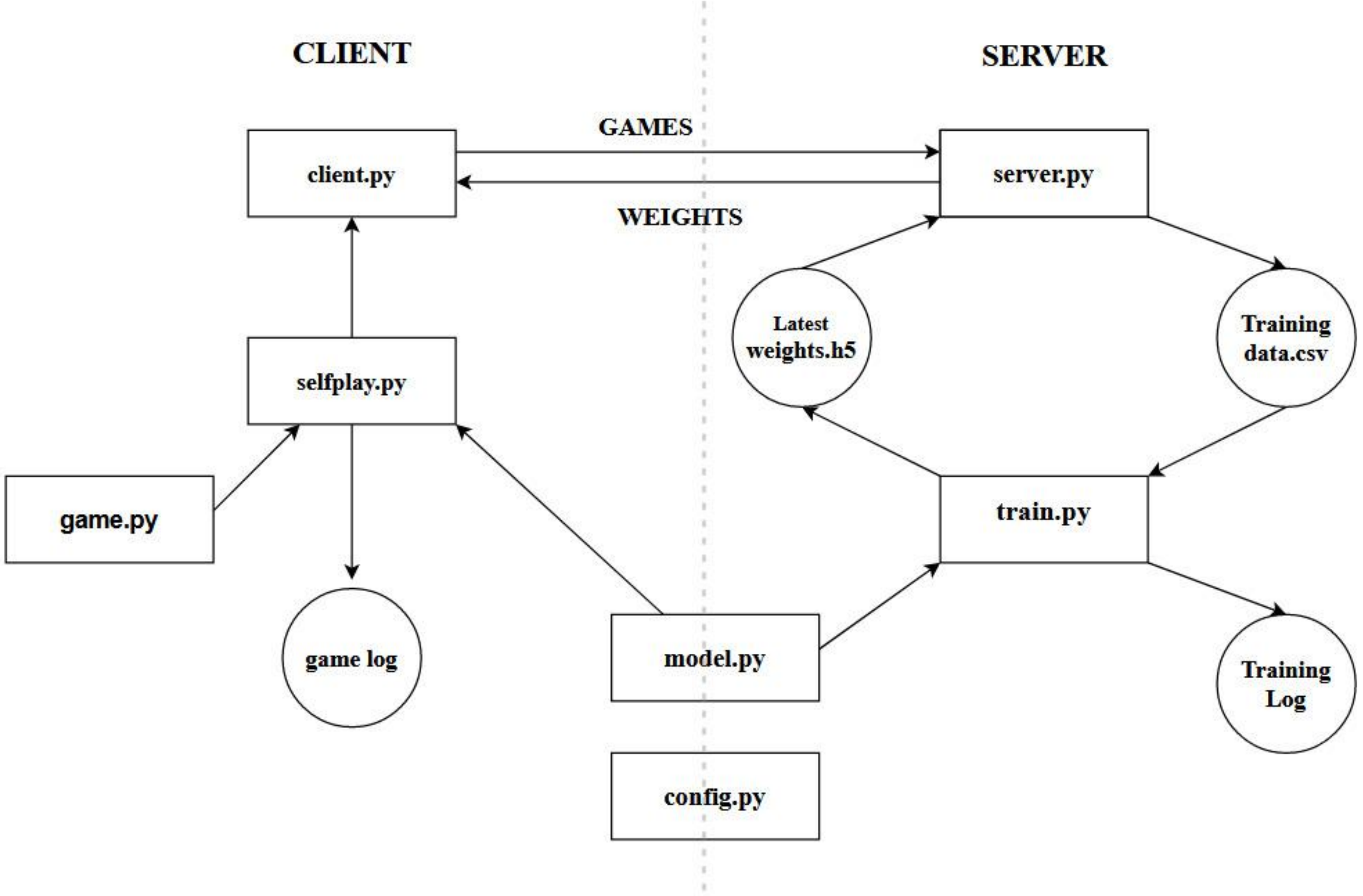
Python Based Implementation

Python

- Used A Client Server Architecture.
- Training Process can be divided into two parts :
  a) **GPU Intensive :** Training of deep neural network.
  b) **CPU Intensive :** Self Play Game Generation.
- Server process can collect training data and train neural network on it.
- Server then sends weights of trained net back to client.
- Client receives the neural network from server.
- Client then plays fixed number of games and sends data of each game. The whole cycle is then repeated.



SELF-PLAY GAME GENERATION CLIENTS (CPU)


CLIENT
SERVER
GAMES
WEIGHTS
client.py
server.py
selfplay.py
Latest weights.h5
Training data.csv
game.py
train.py
game log
model.py
Training Log
config.py

Description

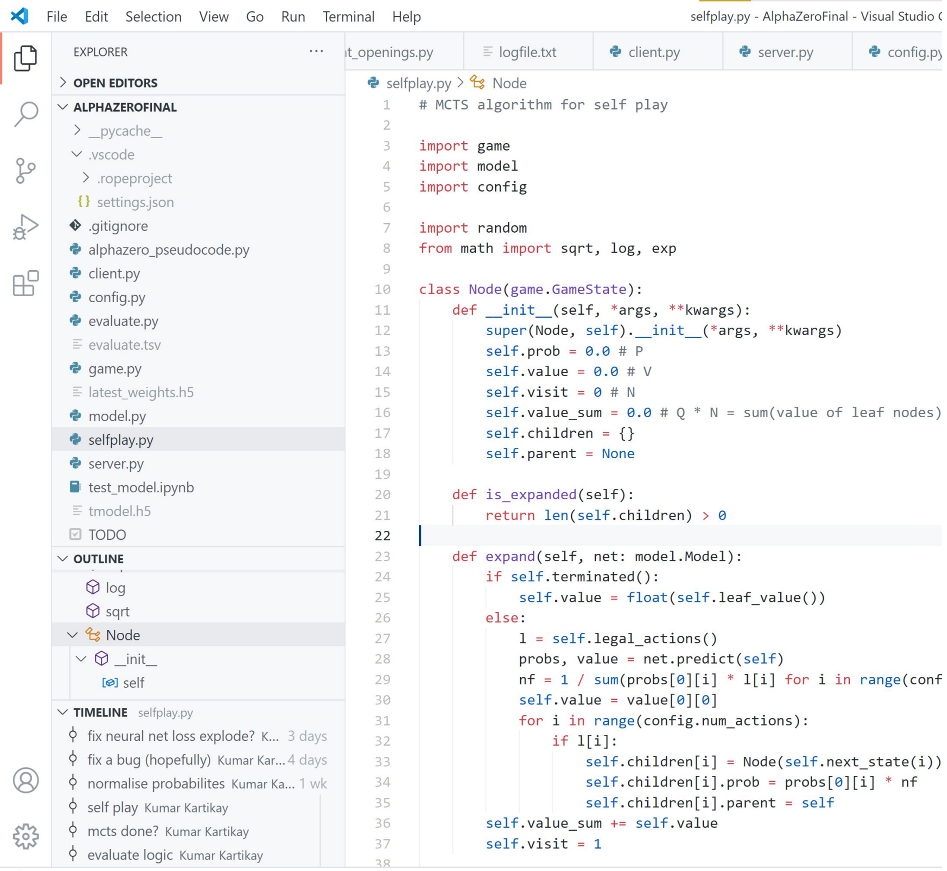
- **Server.py :** Handles HTTP requests for :
  a) Fetching Latest Model Weights
  b) Appending New Training Data
  c) Communicates with Training Process (**train.py**) via files latest_weights and training_data.
- **Client.py :** Gets the latest neural network from the server, plays many games of self play with the MCTS algorithm, thereby generating training data, which it reports to the server upon termination of the games. Also writes the self play games to a log file in a shortened format, for later analysis.

Description...

- **Evaluate.py :** Gets the latest neural net from the server and evaluates it by playing against an opponent that always plays a random legal move. Reports win/loss rate to a file 'evaluate.csv'.
- **Train.py :** Starts initially with random initialised network. It then reads incoming training data and trains the neural network, trains the neural network using Stochastic Gradient Descent, and writes the updated weights to a le `latest-weights.h5'. This final file is sent to clients by the server.
- **Model.py :** Contains details of the neural network architecture. This file is needed by both server and client processes.
- **Game.py :** Contains game specific rules.

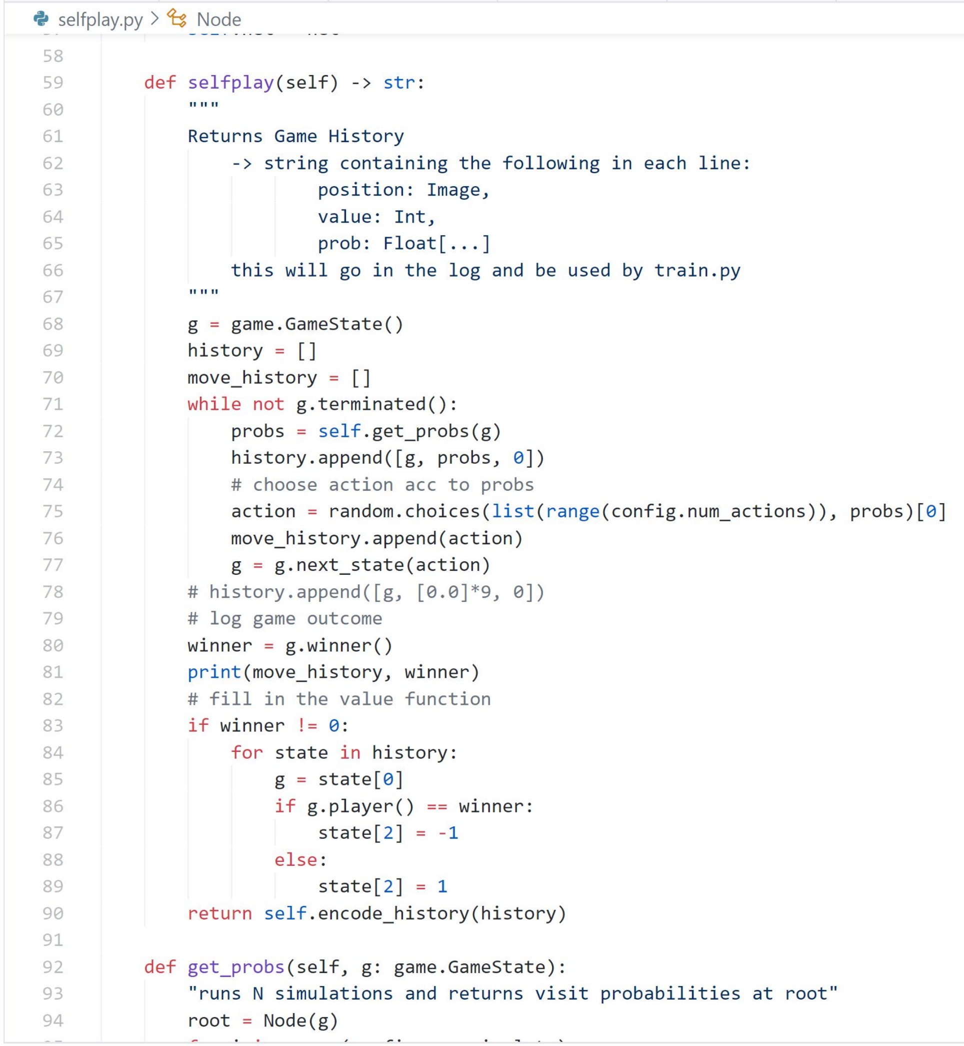
Description...

- **Selfplay.py :** It contains the main MCTS search procedure in an MCTS class. It also has a class Node which represents a node in the game tree.
- **Config.py :** contains various constants relevant to both the server and client, such as, number of simulations per move in MCTS, learning rate of the neural network, etc.

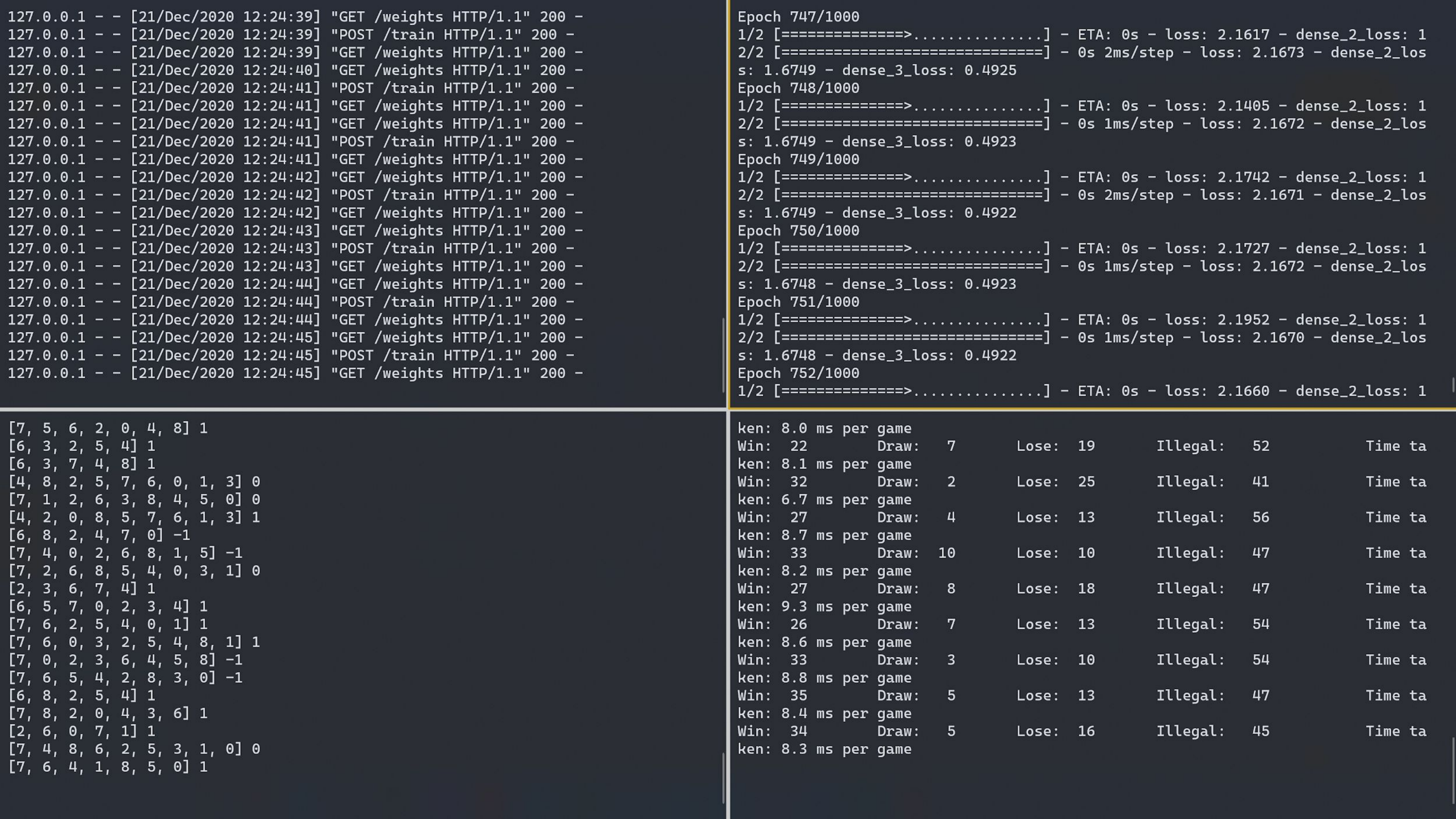


```python
# MCTS algorithm for self play

import game
import model
import config

import random
from math import sqrt, log, exp

class Node(game.GameState):
    def __init__(self, *args, **kwargs):
        super(Node, self).__init__(*args, **kwargs)
        self.prob = 0.0 # P
        self.value = 0.0 # V
        self.visit = 0 # N
        self.value_sum = 0.0 # Q * N = sum(value of leaf nodes)
        self.children = {}
        self.parent = None

    def is_expanded(self):
        return len(self.children) > 0

    def expand(self, net: model.Model):
        if self.terminated():
            self.value = float(self.leaf_value())
        else:
            l = self.legal_actions()
            probs, value = net.predict(self)
            nf = 1 / sum(probs[0][i] * l[i] for i in range(conf
            self.value = value[0][0]
            for i in range(config.num_actions):
                if l[i]:
                    self.children[i] = Node(self.next_state(i))
                    self.children[i].prob = probs[0][i] * nf
                    self.children[i].parent = self
        self.value_sum += self.value
        self.visit = 1
```

Python code



```python

    def selfplay(self) -> str:
        """
        Returns Game History
            -> string containing the following in each line:
                    position: Image,
                    value: Int,
                    prob: Float[...]
            this will go in the log and be used by train.py
        """
        g = game.GameState()
        history = []
        move_history = []
        while not g.terminated():
            probs = self.get_probs(g)
            history.append([g, probs, 0])
            # choose action acc to probs
            action = random.choices(list(range(config.num_actions)), probs)[0]
            move_history.append(action)
            g = g.next_state(action)
        # history.append([g, [0.0]*9, 0])
        # log game outcome
        winner = g.winner()
        print(move_history, winner)
        # fill in the value function
        if winner != 0:
            for state in history:
                g = state[0]
                if g.player() == winner:
                    state[2] = -1
                else:
                    state[2] = 1
        return self.encode_history(history)

    def get_probs(self, g: game.GameState):
        "runs N simulations and returns visit probabilities at root"
        root = Node(g)
```



```
127.0.0.1 - - [21/Dec/2020 12:24:39] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:39] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:39] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:40] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:41] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:41] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:41] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:41] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:41] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:42] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:42] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:42] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:43] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:43] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:43] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:44] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:44] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:44] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:45] "GET /weights HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:45] "POST /train HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2020 12:24:45] "GET /weights HTTP/1.1" 200 -
```

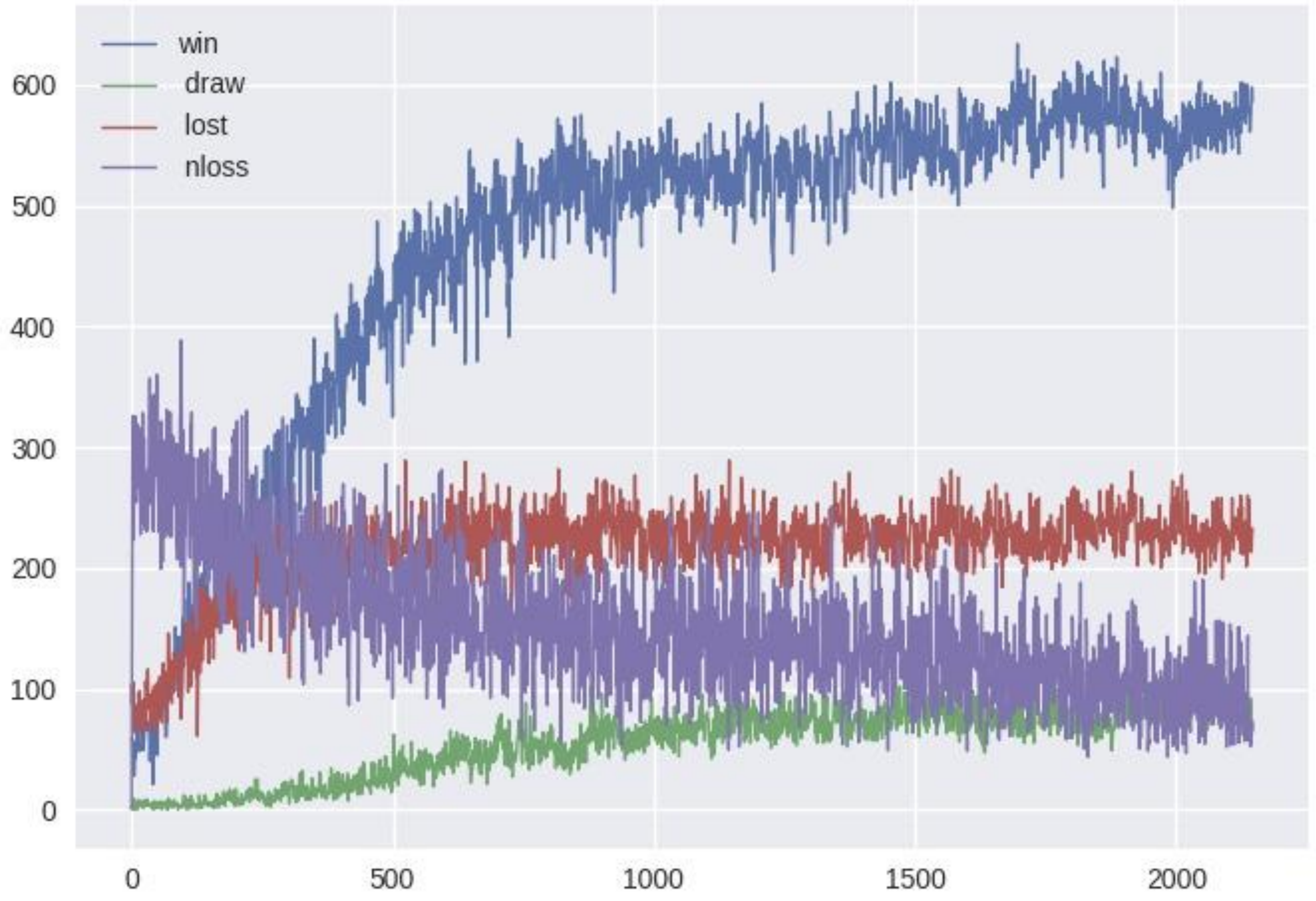
```
Epoch 747/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1617 - dense_2_loss: 1
2/2 [==============================] - 0s 2ms/step - loss: 2.1673 - dense_2_los
s: 1.6749 - dense_3_loss: 0.4925
Epoch 748/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1405 - dense_2_loss: 1
2/2 [==============================] - 0s 1ms/step - loss: 2.1672 - dense_2_los
s: 1.6749 - dense_3_loss: 0.4923
Epoch 749/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1742 - dense_2_loss: 1
2/2 [==============================] - 0s 2ms/step - loss: 2.1671 - dense_2_los
s: 1.6749 - dense_3_loss: 0.4922
Epoch 750/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1727 - dense_2_loss: 1
2/2 [==============================] - 0s 1ms/step - loss: 2.1672 - dense_2_los
s: 1.6748 - dense_3_loss: 0.4923
Epoch 751/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1952 - dense_2_loss: 1
2/2 [==============================] - 0s 1ms/step - loss: 2.1670 - dense_2_los
s: 1.6748 - dense_3_loss: 0.4922
Epoch 752/1000
1/2 [==============>...............] - ETA: 0s - loss: 2.1660 - dense_2_loss: 1
```

```
[7, 5, 6, 2, 0, 4, 8] 1
[6, 3, 2, 5, 4] 1
[6, 3, 7, 4, 8] 1
[4, 8, 2, 5, 7, 6, 0, 1, 3] 0
[7, 1, 2, 6, 3, 8, 4, 5, 0] 0
[4, 2, 0, 8, 5, 7, 6, 1, 3] 1
[6, 8, 2, 4, 7, 0] -1
[7, 4, 0, 2, 6, 8, 1, 5] -1
[7, 2, 6, 8, 5, 4, 0, 3, 1] 0
[2, 3, 6, 7, 4] 1
[6, 5, 7, 0, 2, 3, 4] 1
[7, 6, 2, 5, 4, 0, 1] 1
[7, 6, 0, 3, 2, 5, 4, 8, 1] 1
[7, 0, 2, 3, 6, 4, 5, 8] -1
[7, 6, 5, 4, 2, 8, 3, 0] -1
[6, 8, 2, 5, 4] 1
[7, 8, 2, 0, 4, 3, 6] 1
[2, 6, 0, 7, 1] 1
[7, 4, 8, 6, 2, 5, 3, 1, 0] 0
[7, 6, 4, 1, 8, 5, 0] 1
```

```
ken: 8.0 ms per game
Win:  22        Draw:   7       Lose:  19       Illegal:   52           Time ta
ken: 8.1 ms per game
Win:  32        Draw:   2       Lose:  25       Illegal:   41           Time ta
ken: 6.7 ms per game
Win:  27        Draw:   4       Lose:  13       Illegal:   56           Time ta
ken: 8.7 ms per game
Win:  33        Draw:  10       Lose:  10       Illegal:   47           Time ta
ken: 8.2 ms per game
Win:  27        Draw:   8       Lose:  18       Illegal:   47           Time ta
ken: 9.3 ms per game
Win:  26        Draw:   7       Lose:  13       Illegal:   54           Time ta
ken: 8.6 ms per game
Win:  33        Draw:   3       Lose:  10       Illegal:   54           Time ta
ken: 8.8 ms per game
Win:  35        Draw:   5       Lose:  13       Illegal:   47           Time ta
ken: 8.4 ms per game
Win:  34        Draw:   5       Lose:  16       Illegal:   45           Time ta
ken: 8.3 ms per game
```

Results

- Python with Neural Network


win
draw
lost
nloss
600
500
400
300
200
100
0
0
500
1000
1500
2000

Challenges

- Since MCTS algorithm is a very resource hungry algorithm - it requires a combination of CPU and GPU power to run.
- A long time is needed to generate self play games while using the neural network.
- GPU utilisation was quite low (only 6-10%) during our training.

Review of Related/Similar Work

1. **Leela Chess Zero**:- Low level implementation in **C++ and CUDA**. Optimized for highly distributed computing environments. Plays Chess only.

   Also uses **client-server** architecture with multiple clients throughout the world contributing training data.

2. **AlphaZero.jl**:- Written in **Julia** language. Fast implementation Because of Julia's just-in-time compilation. Uses Asynchronous MCTS to speed up training. Tested on game **Connect4.**

Review of Related/Similar Work

3. **Oracle's implementation:-** Oracle research has also experimented with implementing AlphaZero algorithm on **Connect4** and has published various results on its blog.

   They have extensively experimented with hyperparameter tuning and optimisations to the training algorithm.

Future Scope / Possible Improvements

- Improving the neural network efficiency.
- Exploring the algorithm for other two player board games.
- Future works concerning utilisation of this algorithm for other research work in reinforcement learning.
- Improving Efficiency of MCTS by making it parallelized.
- Improving computing efficiency at the cost of memory, by not discarding the game tree statistics after every move.

Conclusion

- Performed Well as can be seen.
- Gave us an insight of general methodology of combining deep learning and tree search to explore large combinatorial spaces effectively.
- Future works concerning utilisation of this algorithm for other research work in reinforcement learning.

References

- **Silver, D.**, Huang, A., Maddison, C. et al. **Mastering the game of Go with deep neural networks and tree search.** Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961
- **Silver, D.**, Schrittwieser, J., Simonyan, K. et al. **Mastering the game of Go without human knowledge.** Nature 550, 354–359 (2017). https://doi.org/10.1038/nature24270
- **Silver D.**, et al. **A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play** Science 07 Dec 2018: Vol. 362, Issue 6419 https://doi.org/10.1126/science.aar6404

References

- **AlphaZero: Shedding new light on chess, shogi, and Go** - Deepmind blog: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go
- **AlphaZero Explained** - https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/
- **Leela Chess Zero - Open source neural network based chess engine -** https://lczero.org/
- **Lessons From Implementing AlphaZero - Oracle Developers Blog** https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191
- **AlphaZero.jl Package Documentation** https://jonathan-laurent.github.io/AlphaZero.jl/stable/