Connect 4I

Marie Catharina Hartwell Pors
Business Analytics
Hanyang University
The Technical University of Denmark
mariechpors@gmail.com

Francesco Stefano Schirinzi
Computer Science
Hanyang University
ZHAW
fschirinzi25@gmail.com

Fabio Matteo Alfonso Motta
Engineering & Management
Hanyang University
Pforzheim University
fabiomotta98@gmail.com

Nicolas Bouillette
Computer Science
Hanyang University
Université Sorbonne Paris Nord
bouillette.nicolas@outlook.fr

Lucas Robidou
Computer Science
Hanyang University
Université Sorbonne Paris Nord
luc.robidou@gmail.com

Luca Ren
Computer Science
Hanyang University
Université Sorbonne Paris Nord
renluca93@gmail.com

Video version :
https://www.youtube.com/watch?v=UDkyj1aMv1o&feature=youtu.be

***Abstract*— We were able to understand the course material better in a fun way. The deep analysis of those two algorithms really helped us understand how a real AI is programmed. Being able to have our own game with an AI that learn by itself was really satisfying.**

As one of our AIs needs a lot of time to think and is still dumb, we are now aware of some AIs' limitations.
Furthermore, AlphaZero needs a huge amount of games played to be good, playing 5000 games already noticeably improved the AI.

I. Introduction

Artificial Intelligence (AI) are becoming more and more advanced and are widely used in many different aspects. The applications of AI are becoming a bigger part of our everyday lives. For example, when you're typing on a smartphone and it gives you a suggestion on the next word based on the first part of a sentence, or when your camera auto adjusts its settings based on the image that it is currently seeing. Only a decade ago it was thought nearly impossible that a machine would be able to beat world champions in advanced strategy games such as Go, but now Go has now been convincingly defeated by AIs without human inputs. Reinforcement learning AIs can now start from random strategies playing against it self and achieve superhuman play tactics in less than 24 hours without any information from domain expects only the using the game rules (reference: https://deepmind.com/research/case-studies/alphago-the-story-so-far).

As the world is opening up for the usefulness of AI, a need for more people understanding and working within this area is also essential. We are a group of students currently studying at Hanyang University that wants to extend our knowledge within this field. We are therefore building an AI playing Connect 4. This game is a turn-based game, which involves two players trying to connect four of their checkers in a row while preventing their opponent (the other player) from doing the same. This game will be playable through a web interface, and the outcomes will be saved on the server. Thus, we will be able to let the AI play against itself and train itself until a certain time has passed. Players will then be able to face the AI in the game.

The goal of this project is to obtain a Connect 4 AI as good as possible inside the timeframe and scope of this project to obtain as much experience working with multiple algorithms with different levels of difficulty as possible.

II. Datasets

The data necessary for making this project consist of the information about the board and the rules of the game and some human knowledge of good strategy when playing Connect 4. The remaining data used for this project is obtained by letting the AIs play against themselves over and over until a specified time runs out, or a specified number of iterations is met. Iterations are saved as models so that they can be loaded at any time, without the need to recompute them.

III. Methodology

Two different algorithm methodologies are implemented and analysed in this project - Minimax and AlphaZero. The Minimax algorithm is a heuristic that is fine tuned to perform well in the Connect 4 game, whereas AlphaZero is a more advanced AI that trains Neural Network using a search heuristic method. The two methods are explained below starting with the Minimax algorithm.

A. MiniMax Algorithm

Minimax (MM) is a backtracking decision rule that is implemented for outcome optimization of decisional problems in various fields, such as decision theory, game theory, statistics, but also (and more importantly for us) in artificial intelligence. MM is based on the assumption that each player always plays optimally in his own interest and in perfect rationality (“*homo economicus”)*.
This algorithm is applicable to the Connect 4 game because it is a turn-based two players zero-sum game. Zero sum game represents a situation in which each participant's (player) gain or loss (in this case the score of the board state, and eventually the win or loss outcome) is exactly balanced by the losses or gains of the other participants (player). In other words, the choices a player can make are directly dependant from the previous choice(s) of the other player, and vice versa.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

The intrinsic perfect information of this game, allows to evaluate (assign a score) each possible board state and, more importantly, all possible consequent board states derived from all possible future states! If we represent this as a tree, the root would represent the current state of the board, and from there, for each turn (i.e. placing a piece) seven children would be generated. Each child node represents a possible next move (and board states which would result from it) that a player could make, one child for each one of the seven columns where the player could drop the next piece. Each of these seven children, would have further seven children evaluating the possible moves derived from them (see figure below). With this technique, the computer is able to pre-play all possible moves and choose the move with a score maximising strategy (path from root - current state- to leaf representing winning the game) for each of the opponent's moves. This is true from a theoretical point of view, in our case, however, due to the lack of computational power, we cannot implement a minimax which "pre-plays" all possible moves. The superior computational power of the computer allows it to think ahead of all possible moves, and the opponent (human being) is left with the task of minimizing his loss. The MiniMax algorithm will maximize the score for each player in each turn starting from the leaves of the tree. The AI will chose the best option.

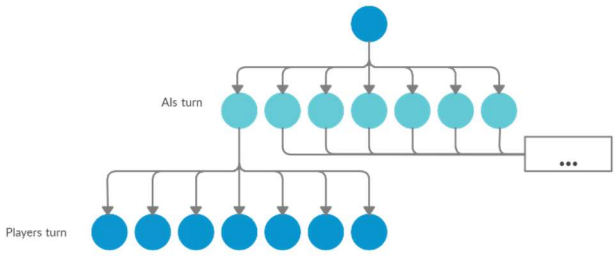

Figure 1: example of a tree

B. MiniMax for Connect 4

The amount of possible moves that the computer can think ahead depends on the computational power of the computer and how much time is appropriate to wait each turn when playing the game.

The time complexity of this algorithm is exponential in terms of the depth of the tree and constant in terms of possible moves over time. The AI has to run through many possible moves to obtain an optimized strategy e.g. with a window length of 4 results in 74=2401 leaf nodes where the score will be calculated. The scoring system is made based on human intuition and knowledge of a good game strategy.

For the purpose of this project we have used Keith Galli's open source code (https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py) and implemented it with the front-end system. The MiniMax algorithm made in this code is recursive and have a Depth First Search strategy going through all nodes in the tree. The depth of the tree is adjustable and set by the user in out code. The higher the depth search the longer each turn will take and the move will be closer to optimal. The first part of the recursive algorithm is to check whether or not a terminal state is reached or if the node is a leaf node. A terminal state can be one of three types: the AI wins, the opponent wins or the board is full. If the current node is terminal the recursion will return the score of +10000000, -10000000 or 0 respectively. If the node is a leaf in the search tree the scoring system will be used to produce a score for the moves made. The number of moves made when reaching the leaf node depends on the depth of the tree. The score is at first set to 0, from here the score is updated depending of how the board looks. First of all a score for more pieces in the middle column is given with a factor of 3 for each piece. This will make the AI more prone to chose the middle column as long as there is room available. This is a good strategy because it will result in more opportunities for obtaining four in a row. The score is also added points when there are two or three pieces in a horizontal, vertical and/or diagonal line of four slots with otherwise empty slots. If there are two pieces and two empty slots the score will be added 2 points and for three pieces and one empty slot the score will be added 5 points. If there are four pieces in a horizontal, vertical and/or diagonal the score will be added 100 points. The score will be subtracted 4 points if the opponent has any line of four slots with three pieces and one empty slot. This scoring system checks through all horisontal, vertical, positive diagonal, and negative diagonal lines of four slots to obtain the final score of the potential moves made. In the next section we will show an example of how the minimax algorithm perform with different window length i.e. different depths of the tree.

C. Exemple of MiniMax

First we will show how the scoring system works when the AI is only looking at one window length ahead. We start from the current state shown in the figure below.

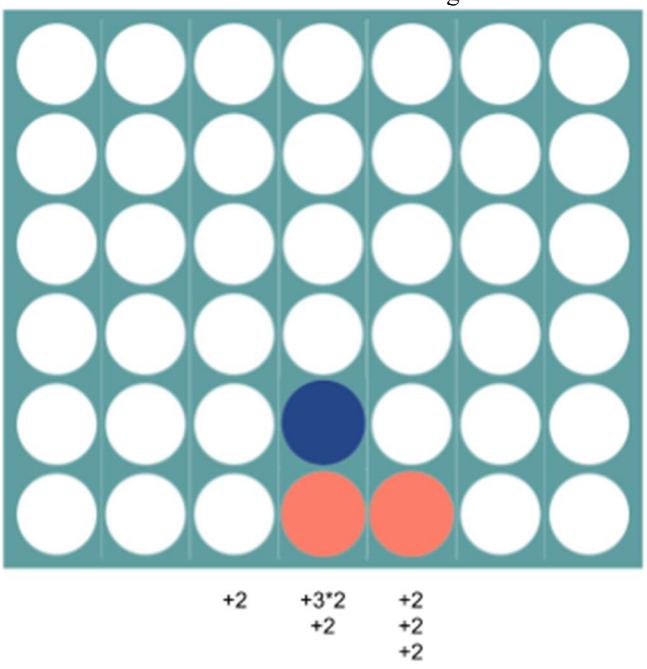

Figure 2: example of score given by min-max for one step only

The points is calculated for each column. In column 1, 2, 6, and 7 the score is initialized as 0, and since there are

no points gained according to this scoring system for these moves. The total score is 0 for making any of these moves. If we place a piece in column 3 we will have a positive diagonal with 2 pieces and 2 empty slots which gives us +2 points. If we place the piece in the 4th column we would have 2 pieces in the middle column that gives us 2 times 3 points i.e. 6 points and in addition we would also obtain 2 points because in a vertical line we obtain two pieces and two empty slots. If we put the piece in the 5th column we would obtain three horizontal lines with two empty slots and two pieces. When only looking at this one step it appears that the middle column is the best option.

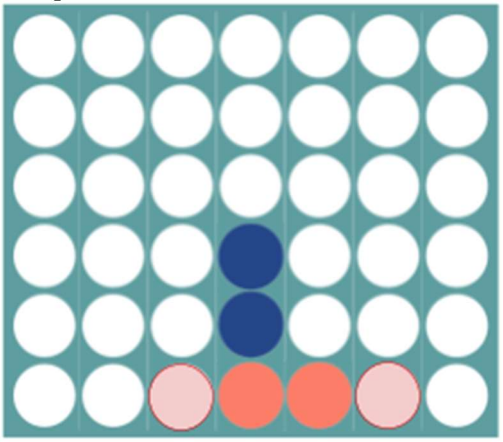

Figure 3: if the AI plays like this, the player will be able to win

If the AI choose the middle column in this scenario the opponent would now have the opportunity to win in the next round if he/she places a piece in either column 3 or 6. This shows the need for thinking further ahead than one step with the score system that we are using.
In the figure below a further evaluation is made with the window length of 3. This is shown in a graph structure. Not all possible states is evaluated, but the most promising moves are chosen for a deeper evaluation.

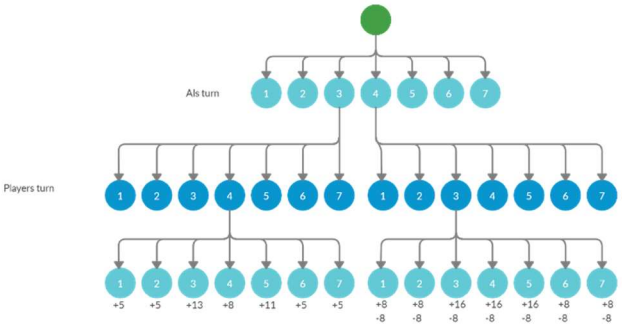

Figure 4: tree search for 3 steps

In the figure above the total scores of the system if these moves were to be made is calculated for each leaf node. This is how the MiniMax code that we are using works just with examining all possible moves in the entire net (that is where all nodes are expanded with 7 child nodes until the 4th layer). If we were to place the first piece in the third column instead of the middle column our highest score would be +11 instead of a total score of +16-8=8 which since we are maximizing the score is preferable. If we looked at the next depth level this would become even more paramount since the player would have the opportunity to win in the next round in the right side of the tree resulting in a -1000000 score.

D. AlphaZero

AlphaZero is a single system that can teach itself from scratch how to master games such as connect-four, chess, shogi, and Go at superhuman level. AlphaZero was developed by DeepMind (Google) in 2017 and it gained a lot of attention because of the revolutionary results it obtained within the reinforcement learning (RL) area. In just 4 hours of training, the AlphaZero AI, was able to beat Stockfish (the former AI chess master) in chess starting from a random strategy only playing against itself (reference: https://science.sciencemag.org/content/362/6419/1140). The AlphaZero system is better at aiming its search for the best possible strategy than former systems.

The three key components of this system are the following:

- Deep convolutional residual neural network (ResNet)
- Monte Carlo Tree Search (MCTS)
- Reinforcement Learning (RL) evaluation to improve performance of the neural network (NN)

Deep convolutional residual neural networks are neural networks with convolutional layers and residual layers that are used to detect filters in a network. The convolutional layers use two main principles: translation invariance and locality (reference: https://d2l.ai/chapter_convolutional-neural-networks/why-conv.html).
The translation invariance focuses on the similarity of the objects regardless of the position and the locality focuses on the nearby region to see what is going on in near proximity instead of trying to look at the whole network. This drastically reduces the number of parameters that have to be calculated. ResNet is a popular way of making CNN and it is the one that we are going to use as well. ResNet is a deep convolutional residual neural network which includes a residual block. A deep NN should in theory get better when adding more hidden layers, but in practice we see that adding more layers will at some point start to make the solution different rather than better. ResNet tries to solve this problem by adding a residual block that makes networks strictly more expressive using a form of nested function class with an identity shortcut connection that will skip one or more layers.
References:

- https://d2l.ai/chapter_convolutional-modern/resnet.html
- https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035
- https://medium.com/@14prakash/understanding-and-implementing-architectures-of-resnet-and-resnext-for-state-of-the-art-image-cf51669e1624

Convolutional Block

```
class ConvBlock(nn.Module):
    def __init__(self):
        super(ConvBlock, self).__init__()
        self.action_size = 7
        self.conv1 = nn.Conv2d(3, 128, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(128)

    def forward(self, s):
        s = s.view(-1, 3, 6, 7)  # batch_size x channels x board_x x board_y
        s = F.relu(self.bn1(self.conv1(s)))
        return s
```

Residual Block

```
class ResBlock(nn.Module):
    def __init__(self, inplanes=128, planes=128, stride=1, downsample=None):
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=3, stride=stride,
                     padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                     padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

    def forward(self, x):
        residual = x
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += residual
        out = F.relu(out)
        return out
```

Output Block

```
class OutBlock(nn.Module):
    def __init__(self):
        super(OutBlock, self).__init__()
        self.conv = nn.Conv2d(128, 3, kernel_size=1) # value head
        self.bn = nn.BatchNorm2d(3)
        self.fc1 = nn.Linear(3*6*7, 32)
        self.fc2 = nn.Linear(32, 1)

        self.conv1 = nn.Conv2d(128, 32, kernel_size=1) # policy head
        self.bn1 = nn.BatchNorm2d(32)
        self.logsoftmax = nn.LogSoftmax(dim=1)
        self.fc = nn.Linear(6*7*32, 7)

    def forward(self,s):
        v = F.relu(self.bn(self.conv(s))) # value head
        v = v.view(-1, 3*6*7)  # batch_size X channel X height X width
        v = F.relu(self.fc1(v))
        v = torch.tanh(self.fc2(v))

        p = F.relu(self.bn1(self.conv1(s))) # policy head
        p = p.view(-1, 6*7*32)
        p = self.fc(p)
        p = self.logsoftmax(p).exp()
        return p, v
```

We used the open source code made by Wee Tee Soh (ref: https://towardsdatascience.com/from-scratch-implementation-of-alphazero-for-connect4-f73d4554002a and https://github.com/plkmo/AlphaZero_Connect4) and this was then connected to our front-end system. The ResNet consist of one convolutional block, 19 residual blocks, and one output block.

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that searches through the game tree in a smart and efficient way. The main concept lies in building a search tree node-by-node, according to the results of simulated playouts. It can be thought of as a *"vicious cycle"* in which the NN gives a series of inputs (policies), the MCTS uses them to run simulations which generate game data, and this is then used to train the NN.

The MCTS consists of four main steps:

- Selection (Tree traversal)
- Expansion
- Rollout / Simulation
- Backpropagation

When analyzing the MCTS, the game-tree of Connect 4 would have the current state of the game as the root and the seven branches (children) represent the 7 possible moves (states) that the AI could make. For each of these leaf nodes the Upper Confidence Bound (UCB) is calculated. The UCB is found by using the following formula.

$$UCB(Si)=vi+c1\ \sqrt{(ln(N)/ni)} + (c2*Pi)/(ni+1)$$

UCB references:

- https://www.youtube.com/watch?v=UXW2yZndl7U
- https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
- https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
- https://www.youtube.com/watch?v=hmQogtp6-fs&t=650s

Si indicates the states (nodes) in the tree, *vi* is the average value of the state *i*, *c1* is a constant (exploration parameter) which balances the exploitation (*vi*) and the exploration term ($\sqrt{(\ln(N)/ni)}$), N is the number of parent visits, and *ni* is number of visits of the node *i*. The third term introduces probability into the equation, with *Pi* being the prior probability of state *i* (given from the neural net). The node with the highest UCB is chosen. In the first branch all seven leaf nodes will have the same UCB score of infinity (because the denominator *ni* of the state *i* has been visited zero times), so a random node will just be chosen at first. If the leaf node hasn't been visited before i.e. the *ni*=0 there will be a roll out (simulation of game for that state). Otherwise, add a new node as a child of the current node for each available action (where to drop the piece) from current state. Note that the expansion represents the next turn, ergo the opponent's turn. Consequently, proceed with the rollout for all child nodes (one after the other, not contemporary). The leaf node is expanded by evaluating the states associated with the expanded nodes with the neural net to obtain and store *P*. Illegal moves should not be expanded and will be masked (by setting prior probabilities to zero).

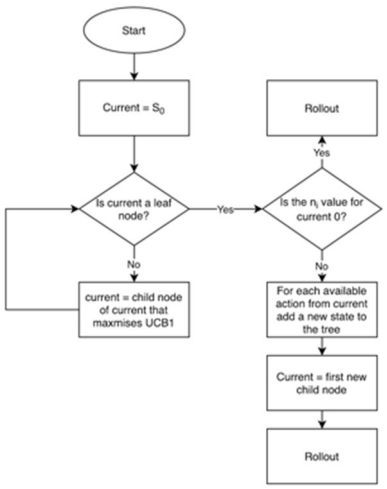

Figure 5: flow chart of Alpha Zero

Consequently, run a simulated playout from the leaf node which was selected in the previous step until a result(value) is achieved. Basically, in the rollout, until a terminal node is reached, randomly choose an action at each step and simulate this action to receive an average reward when the game is over.

Rollout(Si):
loop forever:
if Si is a terminal state:
return value(Si)
Ai = random (available_actions (Si))
Si = simulate (Ai, Si)

The leaf node is then evaluated by the neural net to determine its value v. The value of the terminal state is going to be the estimated value of the leaf node (state) from which the rollout started (*Si*).

This process of updating the current move sequence with the simulation result is known as **Backpropagation**. This means that the value v is then used to update the average value of all parent nodes above it.

Following, an example of the rollout procedure:

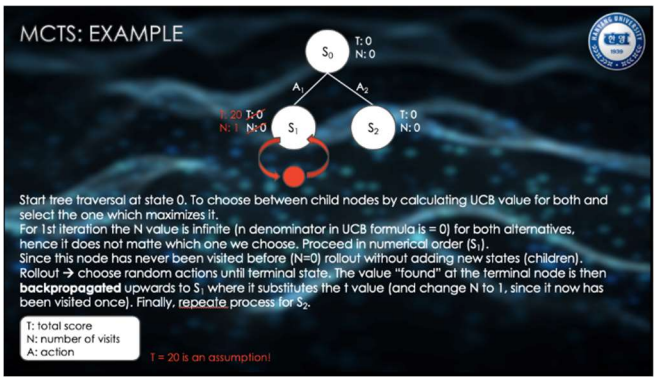

Figure 6: MCTS example

This process is repeated until time runs out or a certain number of iterations are made.

The following image schematically illustrates the process:

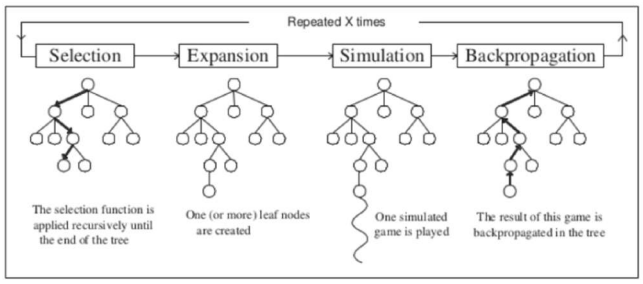

Figure 7: Alpha Zero algorithm

The NN is trained using the MCTS self-play data and the NN will be updated each time a new version of the NN wins against a previous one. The NN guides the MCTS to obtain a better performing system and the best performing NN is always the one that is used.

IV. Evaluate & Analysis

A. Connect 4 AI Arena

It would be tedious, prone to errors, and time-consuming to change the options of an AI and run the script manually every time. That is why we created the file *'connect4_arena.py'* in which one can add to the '*games'* array a list of game configurations to run in sequence. A game configuration is a battle between two AI's/Algorithms with specific settings.

From the next screenshot it's not visible, but for all games, except the one in the screenshot, was used the *'c4_current_net_trained2_iter7.pth.tar'* model file for the Alpha-Zero AI. The mentioned model was trained with roughly 5000 MCTS self-play games.

Example Configuration for one battle:

```
games = [
    {
        'title': 'AZ_vs_p1w1_MM_iter8', # MiniMax with windowLength 1 and iter8 file  vs AlphaZero
        'starting_player': 'player_one',
        'iterations': 1,
        'player_one': {
            'AI': 'alpha-zero',
            'options': {}
        },
        'player_two': {
            'AI': 'min-max',
            'options': {
                'windowLength': 1
            }
        }
    }
]
```

After each battle, the JSON-File containing the statistics to the battle(s) can be found in the folder *'{project-root-folder}/src_backend/minxmax_vs_alpha_sav'.*

This is the statistic of the script after running with the configuration above.

```
{
  "game_history": {},
  "AZ_vs_p1w1_MM_iter8": {
    "iterations_count": 1,
    "player_one": {
      "won_count": 0,
      "settings": { "AI": "alpha-zero", "options": {} }
    },
    "player_two": {
      "won_count": 1,
      "settings": { "AI": "min-max", "options": { "windowLength": 1 } }
    },
    "game_ids": ["925421"]
  }
}
```


For each game, a separate JSON-File is created, which contains the following information:


```
{
  "moves_count": 24,
  "player_one_settings": { "AI": "alpha-zero", "options": {} },
  "player_two_settings": { "AI": "min-max", "options": { "windowLength": 4 } },
  "time_elapsed_player_one": 225.69207119999555,
  "time_elapsed_player_two": 6.759364599995024,
  "tab": [
    ["", "", "", "red", "black", "", ""],
    ["", "red", "black", "red", "black", "", ""],
    ["", "red", "black", "black", "red", "", ""],
    ["", "red", "red", "red", "black", "", ""],
    ["", "red", "black", "red", "black", "", ""],
    ["red", "black", "black", "red", "black", "", "black"]
  ],
  "first_player": "player_one",
  "winner": "player_two",
  "list_of_moves": {
    "1": { "row": 5, "col": 4, "color": "black" },
    "2": { "row": 5, "col": 3, "color": "red" },
    "3": { "row": 4, "col": 4, "color": "black" },
    "4": { "row": 4, "col": 3, "color": "red" },
    "5": { "row": 3, "col": 4, "color": "black" },
    "6": { "row": 2, "col": 4, "color": "red" },
    "7": { "row": 1, "col": 4, "color": "black" },
    "8": { "row": 3, "col": 3, "color": "red" },
    "9": { "row": 2, "col": 3, "color": "black" },
    "10": { "row": 1, "col": 3, "color": "red" },
    "11": { "row": 5, "col": 6, "color": "black" },
    "12": { "row": 0, "col": 3, "color": "red" },
    "13": { "row": 5, "col": 1, "color": "black" },
    "14": { "row": 4, "col": 1, "color": "red" },
    "15": { "row": 0, "col": 4, "color": "black" },
    "16": { "row": 3, "col": 1, "color": "red" },
    "17": { "row": 5, "col": 2, "color": "black" },
    "18": { "row": 5, "col": 0, "color": "red" },
    "19": { "row": 4, "col": 2, "color": "black" },
    "20": { "row": 3, "col": 2, "color": "red" },
    "21": { "row": 2, "col": 2, "color": "black" },
    "22": { "row": 2, "col": 1, "color": "red" },
    "23": { "row": 1, "col": 2, "color": "black" },
    "24": { "row": 1, "col": 1, "color": "red" }
  }
}
```


The reason why we save so much information, is that amongst other reasons we want to check if the AI is deterministic or not. Since we know how Minimax works, we know that it is deterministic. It is so because it is a fully deterministic world in which perfect information about the states for each player is guaranteed, the amount of possible states is finite, and because we can exactly determine the effects a certain move has on the state of the world (i.e. the game).

At the beginning we were not sure if AlphaZero is deterministic or not. With the battles we could find out that it is not the case - if AlphaZero starts the game, else it will always do the same moves.
Comparing the games in the file *'run_13122019-053731.json'*, we can see that Alpha Zero always starts at the same location. From this observation, we see that some deterministic tendencies can be recognized.

Because of the deterministic tendencies of the Algorithms, 99% of the time the games were identical. Because of that reason, we decided to only list the games that are different from each other. The script runs and the games are saved as JSON-Files in the GitHub repository in the folder *'{project-root-folder}/src_backend/minxmax_vs_alpha_sav/'.*

B. Game History

Game ID	Player 1 (MM [1-4] or AZ)	Player 2 (MM [1-4] or AZ)	AI Started (MM [1-4] or AZ)	Winner AI (MM [1-4] or AZ)	Move Count to Win	Total Move duration Player 1	Total Move duration Player 2
668680	AZ	MM4	(P1) AZ	(P2) MM4	26	218	6
824241	AZ	MM4	(P1) AZ	(P2) MM4	24	195	6
152561	MM4	MM3	(P2) MM3	(P1) MM4	28	8.4	5.1
175506	MM4	MM2	(P1) MM4	(P1) MM4	33	8.6	4.8
250903	MM4	MM1	(P1) MM4	(P1) MM4	33	15	7
263533	MM4	MM4	(P1) MM4	(P1) MM4	15	7.1	6.4
396947	MM4	MM1	(P2) MM1	(P1) MM4	28	8.1	4.2
466239	MM4	MM4	(P2) MM4	(P2) MM4	31	8.5	10.3
873543	MM4	MM2	(P2) MM4	(P1) MM4	28	7.9	4.6
739762	MM4	MM3	(P1) MM4	(P1) MM4	33	7.4	4.7

Figure 8: comparison tab (durations are in seconds)

We have analyzed the performance and reached the conclusion that MM is a powerful algorithm even with a depth of four, while the AlphaZero AI does not have sufficient training to beat a real-world player or the MM AI.

Furthermore, MM is much faster than AlphaZero and a good real-life competitor.

V. Related Work

A. Tool

- Spyder; Visual Studio; SublimeText3
- Conda -> Python Run Environment
- Github
- Discord
- WhatsApp

B. Librarie

- Back-End : Python3
  - os
  - sys
  - random
  - json
  - ai_module
  - timeit
  - time
  - copy
  - numpy
  - sys
  - math
  - bottle
  - torch

- Front-End : JavaScript 1.8
  - Vue from *'vue'*
  - Router from *'vue-router'*
  - *'bootstrap'*
  - axios from *'axios'*
  - VueAxios from *'vue-axios'*
  - router from *'./router'*

C. Blog

- https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
- https://science.sciencemag.org/content/362/6419/1140
- https://deepmind.com/research/case-studies/alphago-the-story-so-far
- https://rossta.net/blog/series/connect-four.html
- https://pytorch.org/docs/stable/nn.html#convolution-layers
- https://d2l.ai/chapter_convolutional-modern/resnet.html
- https://towardsdatascience.com/from-scratch-implementation-of-alphazero-for-connect4-f73d4554002a

D. Video

- https://www.youtube.com/watch?v=y7AKtWGOPAE
- https://www.youtube.com/watch?v=ahkBkIGdnWQ
- https://www.youtube.com/watch?v=UXW2yZndl7U
- https://www.youtube.com/watch?v=MMLtza3CZFM&feature=youtu.be
- https://course.fast.ai/videos/?lesson=6
- https://www.youtube.com/watch?v=hmQogtp6-fs&t=648s

E. Code

- https://github.com/KeithGalli/Connect4-Python/blob/master/connect4_with_ai.py
- https://github.com/plkmo/AlphaZero_Connect4

VI. Conclusion

We were able to understand the course material better in a fun way. The deep analysis of those two algorithms really helped us understand how a real AI is programmed. Being able to have our own game with an AI that learn by itself was really satisfying.

As one of our AIs needs a lot of time to think and is still dumb, we are now aware of some AIs' limitations. Furthermore, AlphaZero needs a huge amount of games played to be good, playing 5000 games already noticeably improved the AI.