CS 581

Alpha Zero for Connect Four

Julia Briden

May 9, 2021

1 Introduction

The AlphaGo Zero program has successfully beat human professionals in the game of Go, using reinforcement learning techniques. To play additional games using the same program structure, the AlphaGo Zero was generalized into the program known as AlphaZero. AlphaZero trains a convolutional neural network (CNN) using Monte Carlo Tree Search (MCTS) and policy iteration. In this project, I implemented the AlphaZero program to play the game of Connect Four.

2 Methods

2.1 Project Completion and Testing

My project plan, Figure 1, was followed to complete the AlphaZero implementation in less than two months. The milestones on the left side of the chart show each project component and the degree of completion. Currently each individual component has been integrated and tested. The total training time for the CNN has been three to four hours without a GPU.

An overview of the program is shown in Figure 2. Each box is a class and the text is the main methods and inputs. To ensure the functionality of the AlphaZero implementation, I tested each component individually and then tested the full integration. To test the Connect

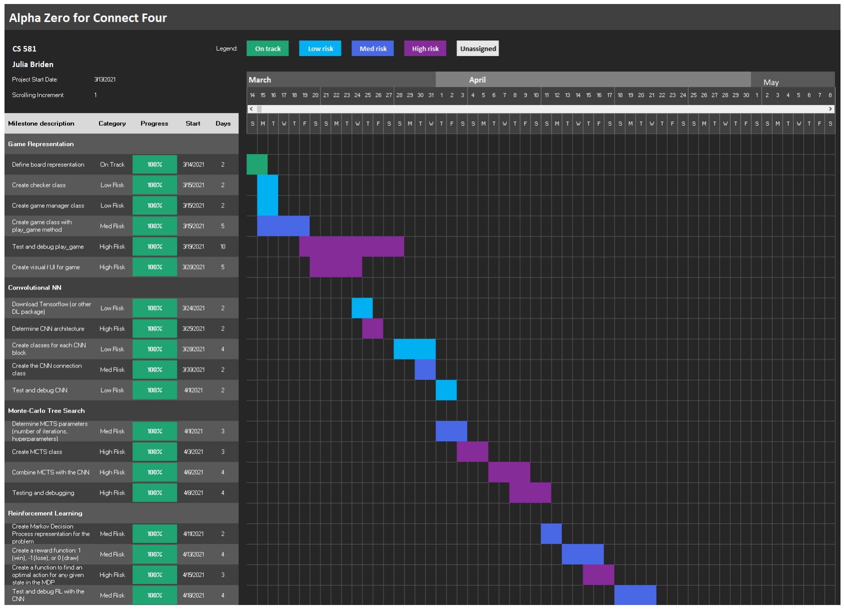


Figure 1: Project plan.

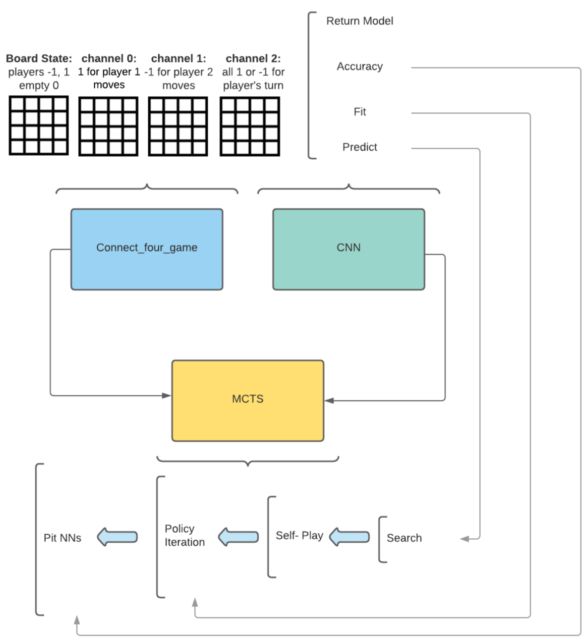


Figure 2: Overview of AlphaZero for Connect Four.

Four Game class, I implemented a play_game method. Where I manually choose the action for each player and the game state is printed after each move. Using the play_game method, I tested each possibility for reaching a game end state and checked that the printed game states were correct.

The CNN was first evaluated for the model creation method, by looking at the input shape, output shapes, and number of parameters. Then the predict method was tested using a test game state as an input. The fit method was tested after the MCTS class and self-play methods were implemented.

The MCTS class includes the methods for search, self-play, and policy iteration. The search method was tested first; a game state, game class, CNN, and current player were used as inputs to search for the game's policy. The game state was printed to ensure that the policy output was consistent with game's outcome. Then the selfPlay method, which calls the search method, was tested. Similarly, the board states, policies, and value were output and checked for correctness. Lastly, the policy iteration method, along with the CNN fit method, was tested to ensure that the CNN properly trains from the self-play data.

Figure 3 shows the results from a single game of manual play using the Connect Four Game class. -1 or 1 is chosen as the first player and the user is prompted for a move column number after each turn. The game ends when a final state is reached and the winner (or a tie) is announced.

The results from a game of self play are shown in Figure 4. In the game, player -1 lost, causing the value to be -1. The state shown is the player 1 locations, player -1 locations, and the next player to go one move before the game ends. The policy values are shown at the bottom for each action. They are all close in value because the policy shown is the first set in the list of policies recorded for the entire game.

A sample policy iteration is shown in Figure 5. After the game reached a final state, the set of game states, policies, and the value for game is used to fit the CNN. The loss for each output and the epoch is displayed as the CNN is training.

The full AlphaZero implementation and detailed instructions on how to run the code can be found in Appendix A.

```
new_game.play_game(-1)
```

```
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Game over player -1 won!
```

Figure 3: Manual Connect Four gameplay.

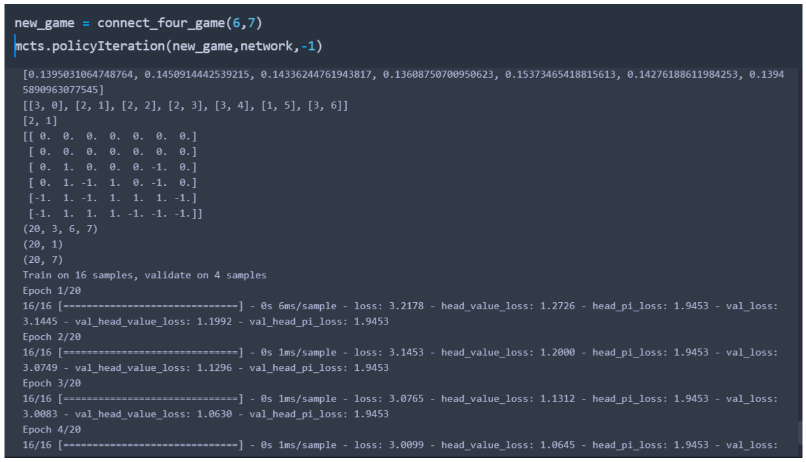
```
new_game = connect_four_game(6,7)
state,value,pi = mcts.selfPlay(new_game,network,-1)
```

```
        [1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1., 1.]]]]), array([[[ 0.,  1.,  0.,  1.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  0.],
        [ 1.,  1.,  0.,  0.,  1.,  0.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  0.],
        [ 0.,  1.,  1.,  0.,  1.,  1.,  0.],
        [ 1.,  0.,  1.,  0.,  0.,  0.,  1.]],

       [[ 1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1.,  1.,  0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  1.,  0.,  1.,  0.],
        [ 0.,  1.,  1.,  0.,  1.,  1.,  0.],
        [ 1.,  0.,  0.,  1.,  0.,  0.,  1.],
        [ 0.,  1.,  0.,  1.,  1.,  1.,  0.]],

       [[-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.]]]])]
[-1]
[[0.14321313798427582, 0.14412978291511536, 0.14227421581745148, 0.14142997562885284, 0.1453661173582077, 0.14163018763065338, 0.14
```

Figure 4: Final state, value, and policy list for a game with self play.



```
new_game = connect_four_game(6,7)
mcts.policyIteration(new_game,network,-1)
[0.1395031064748764, 0.1450914442539215, 0.14336244761943817, 0.13608750700950623, 0.15373465418815613, 0.14276188611984253, 0.1394
5890963077545]
[[3, 0], [2, 1], [2, 2], [2, 3], [3, 4], [1, 5], [3, 6]]
[2, 1]
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1. -1.  1.  0. -1.  0.]
 [-1.  1. -1.  1.  1.  1. -1.]
 [-1.  1.  1.  1. -1. -1. -1.]]
(20, 3, 6, 7)
(20, 1)
(20, 7)
Train on 16 samples, validate on 4 samples
Epoch 1/20
16/16 [==============================] - 0s 6ms/sample - loss: 3.2178 - head_value_loss: 1.2726 - head_pi_loss: 1.9453 - val_loss:
3.1445 - val_head_value_loss: 1.1992 - val_head_pi_loss: 1.9453
Epoch 2/20
16/16 [==============================] - 0s 1ms/sample - loss: 3.1453 - head_value_loss: 1.2000 - head_pi_loss: 1.9453 - val_loss:
3.0749 - val_head_value_loss: 1.1296 - val_head_pi_loss: 1.9453
Epoch 3/20
16/16 [==============================] - 0s 1ms/sample - loss: 3.0765 - head_value_loss: 1.1312 - head_pi_loss: 1.9453 - val_loss:
3.0083 - val_head_value_loss: 1.0630 - val_head_pi_loss: 1.9453
Epoch 4/20
16/16 [==============================] - 0s 1ms/sample - loss: 3.0099 - head_value_loss: 1.0645 - head_pi_loss: 1.9453 - val_loss:
```

Figure 5: Final board state, CNN input sizes, and CNN training for policy iteration.

2.2 Deep Learning

The network architecture for the CNN is shown in Figure 6. The input is a list of (3,n,m) game states, concatenated channels 0 (1 where player 1 has played), 1 (1 where player -1 has played), and 2 (all -1 or 1 based on which player's turn it is). The targets or outputs are the policy, a list of probabilities with a length equal to the number of columns, and the value, [-1,1] (1 if the player of choice wins and -1 if the player of choice loses) based on the expected outcome of the game. The game state was reduced from the 19x19x17 stack used in the original AlphaZero algorithm to a 3xnxm stack to account for limited computational resources and the smaller action space in a game of connect four. Both the policy and value output dimensions are consistent with the original AlphaZero algorithm. The value should be a scalar because it is a prediction for the outcome of the game and the policy should be the size of the action space so it can be used to determine the best action for a specific game state.

The CNN has 10 layers: one input layer, two 7x7 conv2d layers, one 2x2 average pooling layer, two 1x1 conv2d layers, one flatten layer, one dense layer with 120 units, one dense layer with 84 units, and one layer with two dense outputs.

The rectified linear activation function (Relu) was chosen for the conv2d and dense layers because sigmoid and hyperbolic tangent activation functions can cause deep neural networks to fail to receive gradient information [1]. Relu overcomes this drawback because

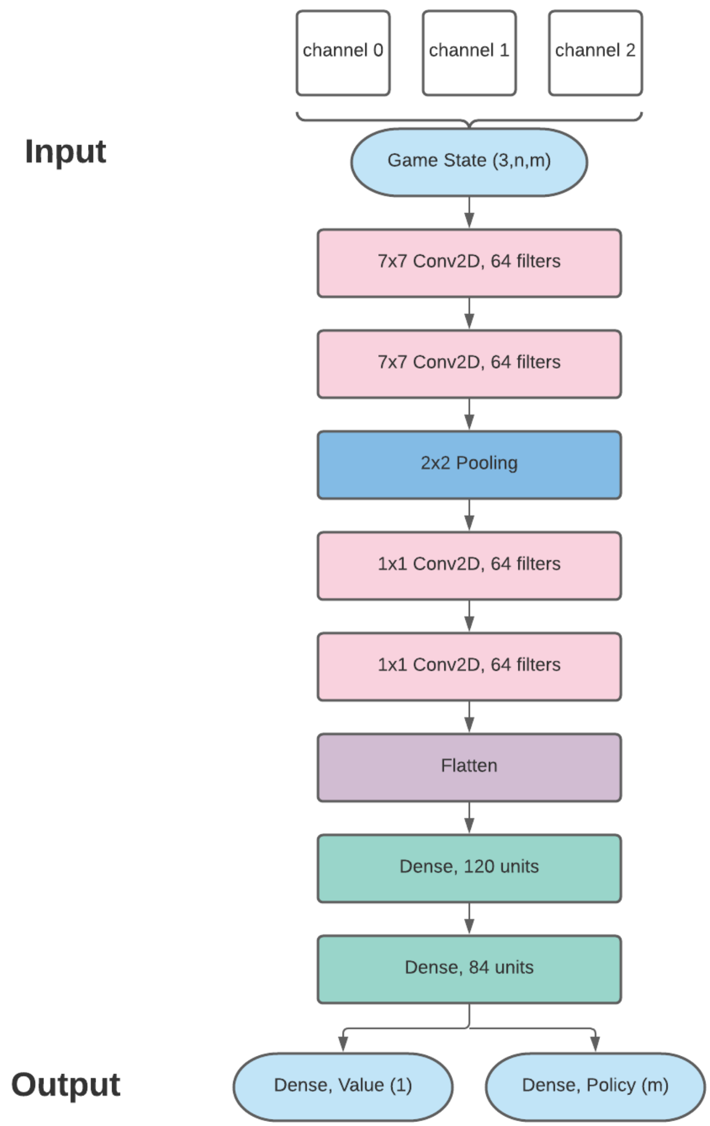


Figure 6: Convolutional Neural Network architecture.

of its linear behavior. Since the gradients are proportional to the node activations, there is no vanishing gradient problem. In addition, Relu is widely used in deep neural networks because it is computationally simple and it is also capable of outputting a true zero value, which can simplify the model.

Average pooling was used to extract features from the map and create a downsampled feature map. This method prevents overfitting by reducing the number of feature values following the pooling layer. Since only the main features are sampled, less computations are required and the chance of overfitting is reduced. Max pooling was not used because it extracts more pronounced features, that would likely not be applicable for a game of Connect Four.

The conv2d layers were added in groups of two to allow the network to extract high-level features from the game states. The additional flattening layer was then used to create a vector input for the dense layers. The final layers in the CNN are dense layers which backpropagate the prediction error to improve the system's performance.

The CNN architecture I chose has 265,108 parameters and less layers than the neural network used for AlphaZero. Since the input states are fairly small in my implementation, I think it is computationally feasible to increase the number of conv2d blocks in my CNN. Increasing the number of parameters would likely result in a better performing network due to more feature extractions.

2.3 Monte Carlo Tree Search

The MCTS method overview is shown in Figure 7. Nodes are initialized as dictionaries, with game states as keys. Each node stores the following attributes: Q, the mean value of the next state, p, the prior probability of selecting an action, N, the number of times an action has been taken from the state, and v, the total value of the next state.

The search method takes an input of a search state, connect four game, CNN, and player preference. Given a state, S, the MCTS is run according to Figure 7: first, the method checks if S is a terminal state. If it is, a value of -1 (adverse player won), 0 (tie), or 1 (choice player won) is returned. If the state is not terminal, the list of explored nodes is checked to see if the state already exists. If S is a new state, a new node is added and

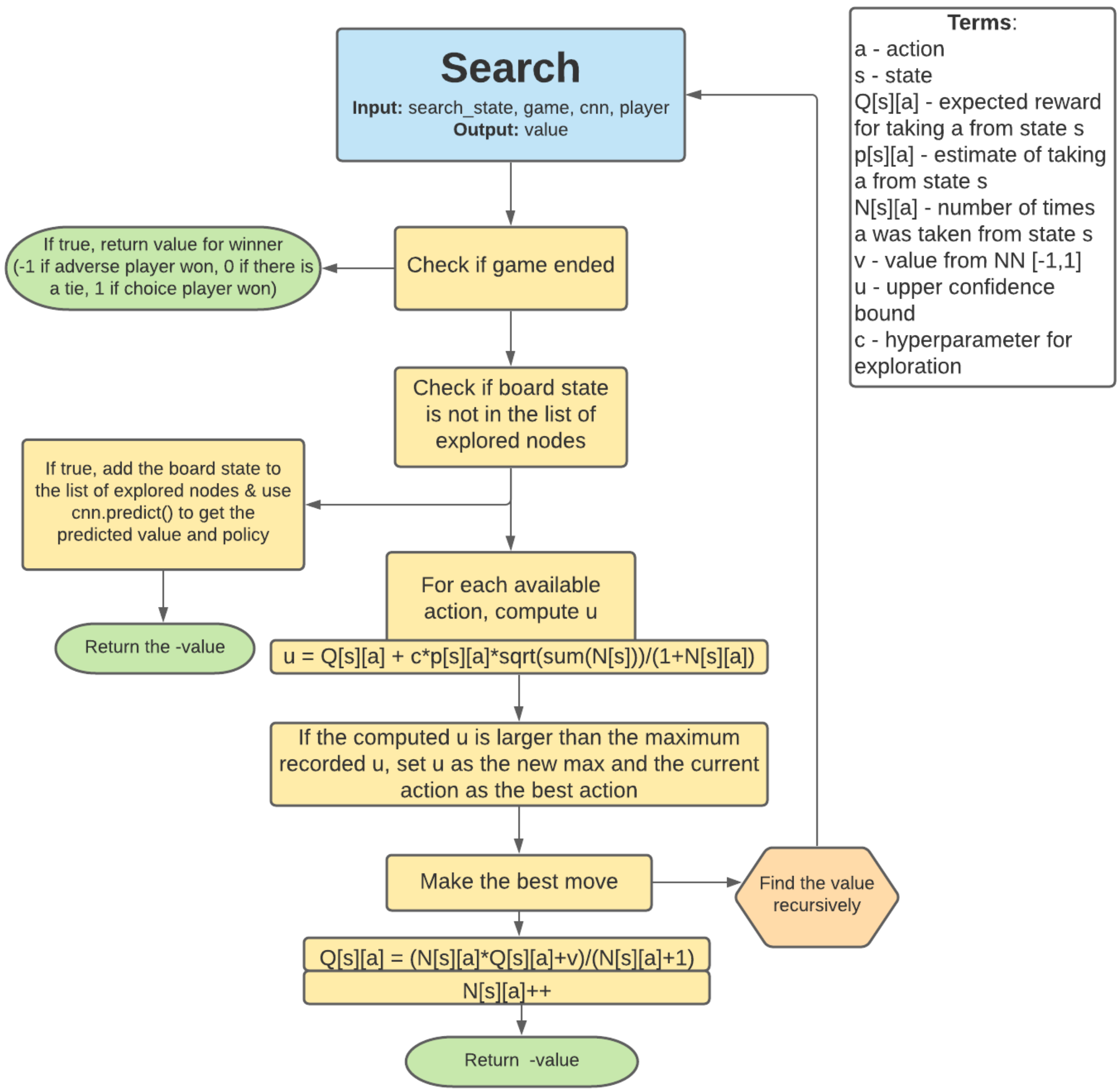


Figure 7: Monte Carlo Tree Search.

the CNN is used to predict the value and policy for the state, which are stored in the node. Then the negate of the predicted value is returned. If S already exists in the list of explored nodes, u (the upper confidence bound) is computed for each available action. If u is larger than the maximum stored value for u, then u is set as the new max and the current action is selected as the best action. The resulting best move is taken and the value for the next state is found recursively. The expected reward is then calculated and the negate of the recursively-found value is returned.

The attributes of each node are updated by using the dictionary data structure; a node is indexed with a string value of the game state. After a leaf node is reached, each edge of the tree that was traversed is updated with a new v and Q, using backpropagation. If a terminal state is reached, the value for the game is propagated. This allows the N value at the root to better approximate the policy. The selfPlay method then uses the search tree, from the search method, to find a policy and value for each state. When a game ends, selfPlay sets the value for all states in that path to 1 if the player of choice wins, -1 if the adverse player wins, or 0 if there is a tie. The set of boards, value, and policies are returned to be used for CNN training in the policy iteration method.

The final result from the MCTS is the returned value and the final result from self-play is the set of game states, policies, and final value. Inside of the selfPlay method, the MCTS is performed a specified number of times. In the traditional AlphaZero program, the simulation is run 1,600 times. My current implementation runs search 10 times. Running the simulation a greater number of times would likely result in an improved accuracy in policies and values for each node in the tree. Since Connect Four usually ends in less moves than a game of Go, a small number of MCTS iterations should be sufficient to build a search tree.

2.4 Self-Play

Self-play is accomplished using the selfPlay method. This method takes an input of a Connect Four game, a CNN, and the player of choice. After initializing a list of boards, values, and policies. A while loop is used to check if the game has ended. If the game has not ended, MCTS is performed for a specified number of times (10 for my implementation).

Then the policy is found by using the board's current state as an index. An action is determined stochastically, by the policy, to encourage exploration. After the action is taken, the next state is found and the loop repeats until the game ends. Once over, a list of boards, policies, and the value for the game is returned.

The output of the selfPlay method is used in the policyIteration method as training data for the CNN. A specified number of iterations and episodes are used to run the self-play method and then train the CNN. After self-playing a game, multiple training tuples can be generated. My current implementation uses 50 iterations and 1 episode per iteration. It is difficult to fit the dimensions correctly to train on multiple training tuples at once but I think it is beneficial to train with multiple episodes at once because the CNN will be less likely to overfit to a specific game.

3 Integration and Results

Figure 8 shows an example of the trained AlphaZero program playing itself. The values for each action in the policy are shown first as a list. A list of actions is displayed under the policy and the action of choice is displayed next. All rows are indexed starting at zero, going from left to right, and all columns are indexed starting at zero, going from top to bottom. The initial policy for the empty board is not equal for every state because the network is trained. In addition, the action choice is stochastic, so the action with the highest probability is not always taken.

From analyzing the gameplay, the CNN generally prefers choosing the 5th column (4th index) over any other column. As shown in the game states, this behavior is useful for blocking itself so that is why it most likely continues to have a high probability for choosing the 5th column for actions. Player -1 did miss a chance to win earlier in the game but still managed to win at the end.

Figure 9 shows an example game where I played my AlphaZero implementation. I was player 1 and AlphaZero was player -1. While playing, the program was very different than a human player because it takes less than a second to respond. At the beginning of the game, AlphaZero was able to get three in a row and had a chance to win before I blocked it. The strategy for my implementation of AlphaZero seems to be getting a diagonal connect

Figure 8: Self-play example.

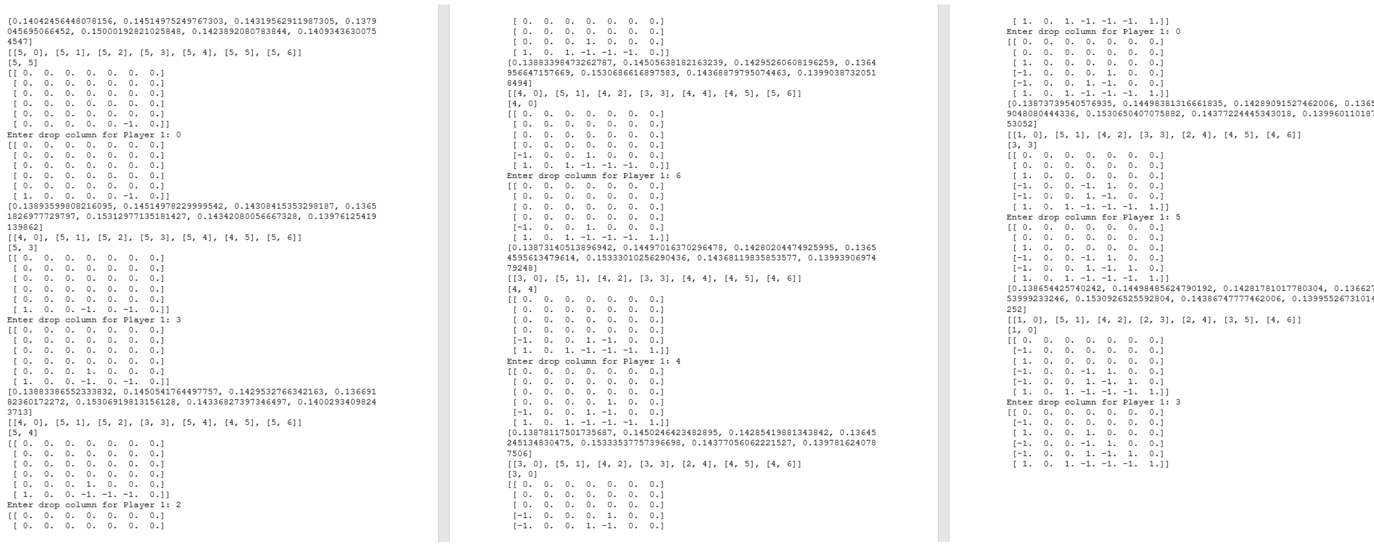


Figure 9: Human vs. AlphaZero example.

four. While AlphaZero did not win this game, its plays were pretty consistent with what I would expect from a human player.

For a small CNN and game state, I think it blocked the other players' moves well. From its current training data, it seems to favor the 5th column when choosing an action. It could be further improved by increasing training time or increasing the number of conv2d layers in the CNN.

4 Future Work

To further improve my implementation of AlphaZero for Connect Four, the CNN architecture can be expanded upon, a longer training time or GPUs can be used, the exploration hyperparameter can be optimized, and deterministic vs. stochastic action choices can be explored. By increasing the CNNs size to 20-35 conv2d blocks, I think the trained network's issue with selecting the fifth column can be mitigated. It is likely that this problem is due to a network that is not deep enough to extract useful information. In addition, an increase in the amount of self-play training data and time for training will allow the network to have more intuition when choosing a move. Lastly, the effects of hyperparameters and iteration numbers can be explored to optimize the performance of my AlphaZero implementation.

5 Conclusion

AlphaZero was implemented to play the game Connect Four. All classes and methods were tested individually and after their integration. From the games of self-play and human-play, my implementation of AlphaZero blocks moves well but it does not always see where it could win. Future work in expanding the CNN architecture and increasing training time will likely improve the performance of the program. By learning, implementing, and testing AlphaZero, I have realized that it is very difficult to know what results to expect from the program. When using reinforcement learning to train a decision-making system, the structure of the CNN can be logically explained but the policy for each action may not be entirely understood. This leads to a need for future work in training and optimization.

A AlphaZero Instructions

A.1 Connect Four Game

All instructions are implemented in the code in Figure 17.

1. Create a new game (use n=6 and m=7 for traditional Connect Four board size:

new_game = connect_four_game(n,m)

2. Play a game with first_player = -1 or 1:

new_game.play_game(first_player)

A.2 CNN

1. Create a new Connect Four Game:

new_game = connect_four_game(n,m)

2. Create a CNN object:

network = CNN(new_game)

3. Create a network:

network.ReturnModelFunctionalMultiHead()

4. To predict a policy and value:

Alpha Zero for Connect Four

Game Representation

Players: -1, 1

Empty spaces: 0

In [1]:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import matplotlib.pyplot as plt
import random
import collections
```

In [2]:

```
class connect_four_game():
    def __init__(self,n,m):
        self.board_state = np.zeros((n,m)) # empty board of n rows and m columns
        self.action_space = []
        self.n = n
        self.m = m
        # NN input
        self.channel_0 = np.zeros((1,n,m))
        self.channel_1 = np.zeros((1,n,m))
        self.channel_2 = np.zeros((1,n,m))
    def clear_board(self):
        self.board_state = np.zeros((self.n,self.m))
        self.channel_0 = np.zeros((1,self.n,self.m))
        self.channel_1 = np.zeros((1,self.n,self.m))
        self.channel_2 = np.zeros((1,self.n,self.m))
    def find_actions(self):
        self.action_space = []
        for col in range(self.m):
            for row in reversed(range(self.n)):
                if self.board_state[row,col] == 0:
                    self.action_space.append([row,col])
                    break
    def make_move(self,player,drop_col):
        self.find_actions()
        for action in self.action_space:
            if action[1] == drop_col:
                self.board_state[action[0],action[1]] = player
                self.update_state(action[0],action[1],player)
                return 1
        else:
            return -1

    def check_winner(self,player):
        # four ways to win: four horizontal, four vertical, four diagonal left, four diagonal right
        non_zero_count = 0
        for col in range(self.m):
            for row in range(self.n):
                if self.board_state[row,col] != 0:
                    non_zero_count += 1
                if self.board_state[row,col] == player:
                    if row <= self.m-4: # enough room to connect four vertical
                        # check for four in a row verticle
                        if self.board_state[row+1,col] == player\
                        and self.board_state[row+2,col] == player\
                        and self.board_state[row+3,col] == player:
                            return 1
                        if col >= self.m-4:
                            # check for four in a row left diagonal down
                            if self.board_state[row+1,col-1] == player\
                            and self.board_state[row+2,col-2] == player\
                            and self.board_state[row+3,col-3] == player:
                                return 1
                    if row >= self.m-4:
                        if col <= self.m-4: # enough room to connect four horizontal
                            # check for four in a row right diagonal up
                            if self.board_state[row-1,col+1] == player\
                            and self.board_state[row-2,col+2] == player\
                            and self.board_state[row-3,col+3] == player:
                                return 1
                        if col >= self.m-4:
                            # check for four in a row left diagonal up
                            if self.board_state[row-1,col-1] == player\
                            and self.board_state[row-2,col-2] == player\
                            and self.board_state[row-3,col-3] == player:
                                return 1
                    if col <= self.m-4: # enough room to connect four horizontal
                        # check for four in a row horizontal
                        if self.board_state[row,col+1] == player\
                        and self.board_state[row,col+2] == player\
                        and self.board_state[row,col+3] == player:
                            return 1
                        if row <= self.m-4: # enough room to connect four vertical
                            # check for four in a row right diagonal down
                            if self.board_state[row+1,col+1] == player\
                            and self.board_state[row+2,col+2] == player\
                            and self.board_state[row+3,col+3] == player:
                                return 1
        if non_zero_count == self.m*self.m:
            return 0
        return -1

    def update_state(self,row,col,player):
        if self.board_state[row,col] == 1:
            self.channel_0[0,row,col] = 1
        elif self.board_state[row,col] == -1:
            self.channel_1[0,row,col] = 1
        if player == 1:
            self.channel_2 = np.ones((1,self.n,self.m))
        elif player == -1:
            self.channel_2 = np.ones((1,self.n,self.m))*-1
```

```python
    def play_game(self,first_player):
        self.clear_board()
        player = first_player
        game_over = -1
        while game_over == -1:
            drop_col = input("Enter drop column for Player " + str(player) + ": ")
            move_success = self.make_move(player,int(drop_col))
            while move_success == -1:
                drop_col = input("Enter drop column for Player " + str(player) + ": ")
                move_success = self.make_move(player,int(drop_col))
            self.print_board()
            game_over = self.check_winner(player)
            if game_over == 1:
                print("Game over player " + str(player) + " won!")
            elif game_over == 0:
                print("Game over no one won")
            if player == -1:
                player = 1
            else:
                player = -1



    def print_board(self):
        print(self.board_state)
```

Convolutional NN

Input: state of the board: image of player 1 locations, image of player 2 locations, image of current player value (1 or -1)

- input nodes: for an 8 step history, 2 players, and color feature, there will be a nxmx17 stack input

Output: continuous value of the board state for the current player, policy probability vector

- output nodes: m possible starting positions to move and add another layer to normalize results into probability distribution

conv block -> 35 ResNets -> dense layer (fully connected) -> policy and value head (adjust as needed)

7x7 conv, 64, /2 -> pool,/2 -> 3x3 conv, 64 (x6) -> 3x3 conv, 128,/2 -> 3x3 conv, 128 (28)

see https://www.biostat.wisc.edu/~craven/cs760/lectures/AlphaZero.pdf for more info

CNN example: https://adventuresinmachinelearning.com/introduction-resnet-tensorflow-2/

In [3]:
```python
# build the network
# Multi-head model
class CNN():
    def __init__(self,game):
        self.n = game.n # rows
        self.m = game.m # columns
        game.find_actions()
        self.actions = game.action_space
        self.state = np.concatenate((game.channel_0,game.channel_1,game.channel_2),axis=0)
        self.pi = np.zeros((len(game.action_space),1))
        self.value = 0
        self.model = 0
        self.prob = 0

    # Functional Mult-head model
    def ReturnModelFunctionalMultiHead(self):
        Inputshape = (3, self.n, self.m)
        output_dim_value = 1
        output_dim_pi = len(self.actions)

        model_inputs = tf.keras.Input(shape=Inputshape)

        # Pass input layer by layer, construct model from input to output
        x = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=(7,7),
                strides=(1, 1),
                padding='same',
                activation='relu')(model_inputs) # can also use relu for activation
        x = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=(7,7),
                strides=(1, 1),
                padding='same',
                activation='relu')(x) # can also use relu for activation

        x = tf.keras.layers.AveragePooling2D(
                pool_size=(2, 2),
                padding='valid')(x)

        x = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=(1,1),
                strides=(1, 1),
                padding='valid',
                activation='relu')(x)
        x = tf.keras.layers.Conv2D(
                filters=64,
                kernel_size=(1,1),
                strides=(1, 1),
                padding='valid',
                activation='relu')(x)

        x = tf.keras.layers.Flatten()(x)
        x = tf.keras.layers.Dense(
                units=120,
                activation='relu')(x)
        x = tf.keras.layers.Dense(
                units=84,
                activation='relu')(x)
```

```python
        # Multi
        self.value = tf.keras.layers.Dense(
                units=output_dim_value,
                activation='tanh',
                name="head_value")(x)
        self.pi = tf.keras.layers.Dense(
                units=output_dim_pi,
                activation='softmax',
                name="head_pi")(x)
        self.model = tf.keras.Model(model_inputs, [self.value, self.pi]) # list of outputs
        self.model.compile(
        optimizer='sgd',
        loss=[
            tf.keras.losses.MeanSquaredError(), # two different loss functions as an example (do this f
            tf.keras.losses.CategoricalCrossentropy() # Could also use two of the same loss function
            ],
            loss_weights=[1.0, 1.0], # determines which head is valued more
        )
        self.model.summary()
        return self.model

    def fit(self,state,value,pi):
        model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath='./saved_model/multi/',
        save_weights_only=True,
        monitor='val_loss',
        mode='max',
        save_best_only=True)
        curr_state = np.concatenate(state, axis=0)
        value_state = np.array([np.concatenate(value, axis=0) for i in range(curr_state.shape[0])])
        pi_state = np.concatenate(pi, axis=0)
        print(curr_state.shape)
        print(value_state.shape)
        print(pi_state.shape)
        self.model.fit(
            x = curr_state,
            #x=state, # states
            y=[value_state, pi_state], # make it a list of values (value list w same length as x)
            validation_split = 0.2,
            batch_size=64,
            epochs=20,
            callbacks=[model_checkpoint_callback]
            )
        self.model.load_weights('./saved_model/multi/')

    def predict(self,pred_state):
        #Y_head_1_train = np.copy(Y_train)
        #Y_head_2_train = np.copy(Y_train)

        #Y_head_1_test = np.copy(Y_test)
        #Y_head_2_test = np.copy(Y_test)
        pred_state = pred_state.reshape([-1,3,self.n,self.m])
        self.value = self.model.predict(pred_state)[0]
        self.pi = [float(i[0]) for i in zip(*self.model.predict(pred_state)[1])]
        return self.value, self.pi

    def accuracy(self,test_state,true_value,true_pi):
        pred_state = test_state.reshape([-1,3,self.n,self.m])
        predv_head_1 = self.model.predict(pred_state)[0]
        truev = true_value
        total_acc = 0
        acc = 0
        for i in range(len(predv_head_1)):
            if predv_head_1[i] == true_value[i]:
                acc += 1
        total_acc += acc/len(predv_head_1)
        print("acc is : {}".format(acc/len(predv_head_1)))

        predv_head_2 = [float(i[0]) for i in zip(*self.model.predict(pred_state)[1])] # cross-entropy i
        truev = true_pi
        acc = 0
        for i in range(len(predv_head_2)):
            if predv_head_2[i] == true_pi[i]:
                acc += 1
        total_acc += acc/len(predv_head_2)
        print("acc is : {}".format(acc/len(predv_head_2)))
        return total_acc
```

In [4]:

```python
new_game = connect_four_game(6,7)
network = CNN(new_game)
network.ReturnModelFunctionalMultiHead()
network.state.shape
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 3, 6, 7)]    0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 3, 6, 64)     22016       input_1[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 3, 6, 64)     200768      conv2d[0][0]
__________________________________________________________________________________________________
average_pooling2d (AveragePooli (None, 1, 3, 64)     0           conv2d_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 1, 3, 64)     4160        average_pooling2d[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 1, 3, 64)     4160        conv2d_2[0][0]
__________________________________________________________________________________________________
flatten (Flatten)               (None, 192)          0           conv2d_3[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 120)          23160       flatten[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 84)           10164       dense[0][0]
__________________________________________________________________________________________________
head_value (Dense)              (None, 1)            85          dense_1[0][0]
__________________________________________________________________________________________________
head_pi (Dense)                 (None, 7)            595         dense_1[0][0]
==================================================================================================
Total params: 265,108
Trainable params: 265,108
Non-trainable params: 0
__________________________________________________________________________________________________

(3, 6, 7)
```

```
In [173]: new_game.play_game(-1)
```

```
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Game over player -1 won!
```

```
In [174]: np.concatenate((new_game.channel_0,new_game.channel_1,new_game.channel_2),axis=0)
```

```
array([[[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1.,  0.,  0.,  0.,  0.,  0.]],

       [[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1.,  0.,  0.,  0.,  0.,  0.,  0.]],

       [[-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1., -1., -1.]]])
```

```
In [179]: network.predict(np.concatenate((new_game.channel_0,new_game.channel_1,new_game.channel_2),axis=0))
```

```
(array([[0.]], dtype=float32),
 [0.1428571492433548,
  0.1428571492433548,
  0.1428571492433548,
  0.1428571492433548,
  0.1428571492433548,
  0.1428571492433548,
  0.1428571492433548])
```

MCTS

Self Play

Create a training set For each move, store:

- the game state
- the search probabilities (MCTS)
- the winner (1 for win, -1 for loss)

```
In [5]: new_game = connect_four_game(6,7)
        network = CNN(new_game)
        network.ReturnModelFunctionalMultiHead()
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_2 (InputLayer)            [(None, 3, 6, 7)]    0
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 3, 6, 64)     22016       input_2[0][0]
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 3, 6, 64)     200768      conv2d_4[0][0]
__________________________________________________________________________________________________
average_pooling2d_1 (AveragePoo (None, 1, 3, 64)     0           conv2d_5[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 1, 3, 64)     4160        average_pooling2d_1[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 1, 3, 64)     4160        conv2d_6[0][0]
__________________________________________________________________________________________________
flatten_1 (Flatten)             (None, 192)          0           conv2d_7[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 120)          23160       flatten_1[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)                 (None, 84)           10164       dense_2[0][0]
__________________________________________________________________________________________________
head_value (Dense)              (None, 1)            85          dense_3[0][0]
__________________________________________________________________________________________________
head_pi (Dense)                 (None, 7)            595         dense_3[0][0]
==================================================================================================
Total params: 265,108
Trainable params: 265,108
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
<tensorflow.python.keras.engine.training.Model at 0x18be82ff208>
```

```
In [6]:
class MCTS():
    def __init__(self):
        self.explored = []
        self.Q = collections.defaultdict(dict) # Expected reward for taking the action from the current
        self.p = {} # Initial estimate of taking an action from the current state according to the poli
        self.N = collections.defaultdict(dict) # The number of times the action was taken from the curr
        self.v = 0 # value from the neural network v E [-1,1]
        self.u = 0 # upper confidence bound
        self.c = 1 # hyperparameter for degree of exploration
        self.full_state = []

    def search(self,state,game,cnn,player):
        player_choice = game.check_winner(player)
        player_adverse = game.check_winner(int(player)*-1)
        if player_choice == 1:
            return player_choice # choice player won: 1
        elif player_adverse == 1:
            return -player_adverse # adverse player won: -1
        elif player_choice == 0:
            return 0 # tie: 0

        curr_state = game.board_state # get board configuration
        if str(curr_state) not in self.explored:
                self.explored.append(curr_state) # initialize the current state in the search tree
                self.full_state.append(np.concatenate((game.channel_0,game.channel_1,game.channel_2)))
                self.v,self.p[str(curr_state)] = cnn.predict(state) # set policy and value for state
                #game.find_actions()
                #valid_moves = [new_game.action_space[i][1] for i in range(len(new_game.action_space))]
                #self.p[str(curr_state)] = [self.p[str(curr_state)][i] for i in valid_moves]
                #if sum(self.p[str(curr_state)])>0:
                #    self.p[str(curr_state)] = self.p[str(curr_state)]/sum(self.p[str(curr_state)])
                #else:
                #    self.p[str(curr_state)] = self.p[str(curr_state)]+np.ones(len(self.p[str(curr_stat
                #    self.p[str(curr_state)]/=np.sum(self.p[str(curr_state)])
                return -self.v
        # determine the actions that maximizes the upper confidence bound, u
        u_max, best_action = -np.inf, -1
        game.find_actions()
        for action in game.action_space:
            action = action[1] # action is column choice
            self.u = self.Q[curr_state][action] + self.c*self.p[curr_state][action]*sqrt(sum\
            (self.N[curr_state]))/\
            (1+self.N[curr_state][action]) # upper confidence bound on the Q-values
            if self.u > u_max:
                u_max = self.u
                best_action = action
        action = best_action

        game.make_move(player,action)
        next_state = np.concatenate((new_game.channel_0,new_game.channel_1,new_game.channel_2))
        self.v = search(next_state,game,cnn)
        self.Q[curr_state][action] = (self.N[curr_state][action]*self.Q[curr_state][action]+self.v)/\
        (self.N[curr_state][action]+1)
        self.N[curr_state][action] += 1
        return -v

    def policyIteration(self,game,cnn,player):
        boards = []
        values = []
        pis = []
        for i in range(50): # iterations
            boards = []
            values = []
            pis = []
            for j in range(1): # episodes
                board_set,value_set,pi_set = self.selfPlay(game,cnn,player) # self play
                boards.append(board_set)
                values.append(value_set)
                pis.append(pi_set)
                game = connect_four_game(6,7)
            trained_cnn = cnn.fit(boards,values,pis)
        return trained_cnn

    def pitNNs(self,cnn_prev,cnn):
        wins_prev = 0
        wins_cnn = 0
        for i in range(10):
            game = connect_four_game(6,7)
            player = random.choice([-1,1])
            board_prev,v_prev,p_prev = self.selfPlay(game,cnn_prev,player)

    def win_rate(self,cnn):
        wins = 0
        N = 10
        for i in range(N):
            game = connect_four_game(6,7)
            player = random.choice([-1,1])
            board,v,p = self.selfPlay(game,cnn,player)
            if v[0] == 1:
                wins += 1
        return wins/N

    def selfPlay(self,game,cnn,player):
        boards = []
        values = []
        pis = []
        state = np.concatenate((game.channel_0,game.channel_1,game.channel_2),axis=0)
        curr_state = game.board_state # get board configuration
        game_over = -1
        first_player = player
```

```python
        while game_over == -1:
            for i in range(10):
                self.search(state,game,cnn,player)
            policy = self.p[str(curr_state)]
            #games.append([state,policy,None])
            boards.append(state)
            pis.append(policy)
            print(policy)
            game.find_actions()
            print(game.action_space)
            action_cols = [game.action_space[i][1]-1 for i in range(len(game.action_space))]
            policy_act = [policy[i] for i in action_cols]
            action = random.choices(game.action_space,weights=policy_act)
            action = action[0]
            print(action)
            game.make_move(player,action[1])
            print(game.board_state)
            state = np.concatenate((game.channel_0,game.channel_1,game.channel_2),axis=0)
            curr_state = game.board_state # get board configuration
            player_choice = game.check_winner(int(first_player))
            player_adverse = game.check_winner(int(first_player)*-1)
            if player_choice == 1:
                values.append(player_choice)
                return boards,values,pis # choice player won: 1
            elif player_adverse == 1:
                values.append(-player_adverse)
                return boards,values,pis # adverse player won: -1
            elif player_choice == 0:
                values.append(0)
                return boards,values,pis # tie: 0
            if player == -1:
                player = 1
            else:
                player = -1
```

In [7]:

```python
mcts = MCTS()
```

In [383]:

```python
ex_game = connect_four_game(6,7)
ex_game.play_game(-1)
```

```
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player 1: 1
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Enter drop column for Player -1: 0
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]
 [-1.  1.  0.  0.  0.  0.  0.]]
Game over player -1 won!
```

In [306]:

```python
mcts.search(np.concatenate((ex_game.channel_0,ex_game.channel_1,ex_game.channel_2),axis=0),new_game,net
```

```
array([[-0.00682186]], dtype=float32)
```

In [438]:

```python
new_game = connect_four_game(6,7)
state,value,pi = mcts.selfPlay(new_game,network,-1)
```

```
C:\Users\julia\Anaconda3\lib\site-packages\ipykernel_launcher.py:22: FutureWarning: elementwise comparison failed; returning scalar instead, but in the future will perform elementwise comparison


[0.14321313798427582, 0.14412978291511536, 0.14227421581745148, 0.14142997562885284, 0.14536611173582077, 0.14163018763065338, 0.14195653796195984]
[[5, 0], [5, 1], [5, 2], [5, 3], [5, 4], [5, 5], [5, 6]]
[5, 5]
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  0.]]
[0.14315803349018097, 0.14439022541046143, 0.14272059500217438, 0.13963463902247345, 0.14601497352123326, 0.14244329929351807, 0.14163815975189921]
[[5, 0], [5, 1], [5, 2], [5, 3], [5, 4], [4, 5], [5, 6]]
[4, 5]
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0. -1.  0.]]
[0.14085184037685394, 0.14446806907653809, 0.14155004918575287, 0.14122140407562256, 0.14863114058971405, 0.14205607771873474, 0.14
```

Figure 10: AlphaZero code

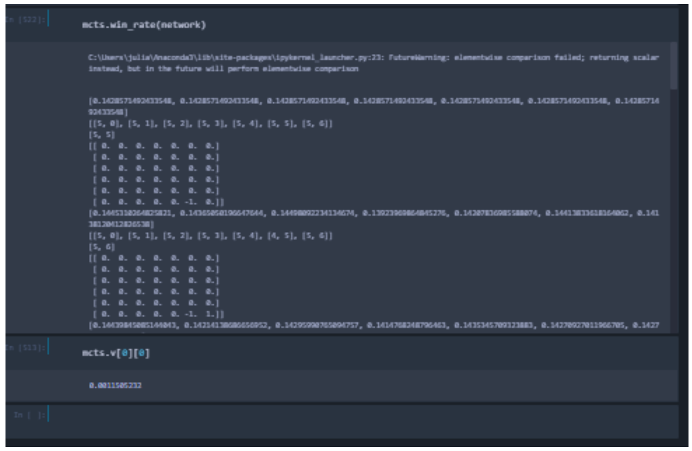


Figure 17: AlphaZero code.

network.predict(np.concatenate((new_game.channel_0, new_game.channel_1, new_game.channel_2) ,axis=0))

5. To train with a list of board states (np.concatenate((new_game.channel_0, new_game.channel_1, new_game.channel_2) ,axis=0)), a list of policies, and a list with the value for the game:

network.fit(boards,values,pis)

A.3 MCTS

1. Create a new Connect Four Game:

new_game = connect_four_game(n,m)

2. Create a CNN object:

network = CNN(new_game)

3. Create a network:

network.ReturnModelFunctionalMultiHead()

4. Create a MCTS object:

mcts = MCTS()

5. Play a game or create a state to search for and then input it in the search method to return a value for a MCTS:

state = np.concatenate((game.channel_0,game.channel_1, game.channel_2) ,axis=0)

mcts.search(state,new_game,network,player)

6. Create a new game object between runs or use new_game.clear_board().

7. Use a game object, network object, and an initial player value to run one self play game:

state,value,pi = mcts.selfPlay(new_game, network, first_player)

8. Perform policy iteration:

mcts.policyIteration(new_game,network,player)

9. Determine the trained CNN's win rate during self play:

mcts.win_rate(network)

10. Play against AlphaZero (put -1 or 1 for human_player input):

mcts.(new_game, network, human_player)

B Literature Cited

[1] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks