CS546 Final Project

David SeWell – Apply MCTS + Q learning to Connect4 03/20/2020

Overview

For this project my goal is to take some of the main concepts from the AlphaGO Zero paper and apply them to a smaller domain. I chose the game of connect4 to be a testing ground to reach these ends. IN particular I wish to hone in on two aspects of the AlphaGO paper that were interesting and also manageable. That is Monte Carlo Tree Search (MCTS) and the use of neural networks to improve performance of MCTS. The way this is achieved in the paper is a bit nuanced so I decided to try a simple improvement of MCTS with a Neural Network to help understand the efficacy of the approach. So this project is broken up into two parts. Part 1 is the details and implementation of MCTS on a connect 4 environment provided by kaggle as part of a competition they provide. Then Part 2 is to run regression on a connect 4 data set that has data of the form (X,Y) where X is the board state and Y is the result as predicted by assuming perfect play. This can be done because connect 4 is a solved game.

MCTS

In this section I will give a very brief overview of MCTS. The name sort of implies what the algorithm is. The first part MC implies that it works by running monte carlo simulations. Monte carlo methods can sometimes be a surprisingly good way to give you empricial estimates of a statistic that you are interested in. The second part of the name TS implies that the method is doing some sort of tree search. So in a nutshell MCTS is a way to do tree search by running simulations. MCTS can deal well in situations with high branching factors and so has been applied well to environments like chess and Go.

There are 4 phases in MCTS. Selection, Expansion, Simulation and Backpropagation.

1. Selection - This is where starting from the root of the tree successive nodes are chosen based on the following UCT criterion.
   $Q(s,a) + c\sqrt{\frac{\ln(N_p)}{N_i}}$
   where $Q(s,a)$ is the value estimate at a given state action pair. Generally this is just $\frac{\sum(R_i)}{N_i}$ The average reward from playing the node i. Finally N_p is the count of the parent node and c is an exploration parameter that needs to be decided on beforehand. A common value used seems to be $\sqrt{2}$. So that is what I use in the experiments that follow.

2. Expansion - Following the selection phase the selected node will need to be expanded to further explore the tree. The only exception here is if you are at a leaf node in which case the simulation terminates after the selection step. Expansion generally works by adding all

of the children nodes of the currently selected node as branches. They are then initialized with values of 0 for both count and reward (N_i, R_i).

3. Simulation - This is where a simulation is run from the expanded node / nodes to estimate the reward from there. This is generally done by just allowing play to continue randomly from the node in question until play ends.

4. Backpropagation - In this step the nodes along the path that lead to the given outcome are updated. In zero sum games this usually means that whatever is getting propagated for one player then one minus that amount will get propagated for the other player. Depending on the structure of the game / reward.

OUTLINE OF MCTS IN PYTHON - THis is a sample of the code that I used in my actual implementation. As can be seen by reading through this code. It is actually pretty simple and can be adapted to many different environemnts.

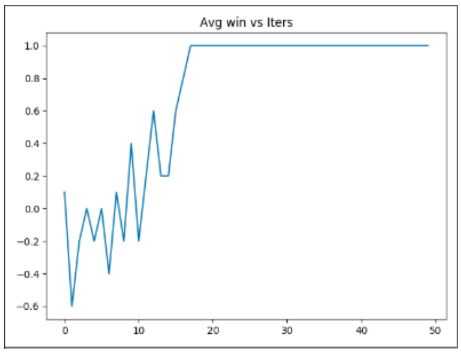
Listing 1: Sample Python code – MCTS outline

```
class MCTS:
    "Monte Carlo tree searcher basic"

    def __init__(self, c=np.sqrt(2)):
        self.Q = defaultdict(int)  # total reward of each node
        self.N = defaultdict(int)  # total visit count for each node
        self.children = dict()  # children of each node
        self.c = c

    def choose(self, node):
        "Choose the best successor of node. (Choose a move in the game)"
        def score(n):
            return self.Q[n] / self.N[n]  # average reward
        return max(self.children[node], key=score)

    def do_rollout(self, node):
        "Make the tree one layer better. (Train for one iteration.)"
        path = self._select(node)
        leaf = path[-1]
        self._expand(leaf)
        reward = self._simulate(leaf)
        self._backpropagate(path, reward)

    def _select(self, node):
        "Find an unexplored descendent of ‘node‘"
        path = []
        while True:
            path.append(node)
```

```
            if node not in self.children or not self.children[node]:
                # node is either unexplored or terminal
                return path
            unexplored = self.children[node] - self.children.keys()
            if unexplored:
                n = unexplored.pop()
                path.append(n)
                return path
            node = self._uct_select(node)  # descend a layer deeper

    def _expand(self, node):
        "Update the ‘children‘ dict with the children of ‘node‘"
        if node in self.children:
            return  # already expanded
        self.children[node] = node.find_children()

    def _simulate(self, node):
        "Returns the reward for a random simulation (to completion) of ‘↩
           node‘"
        while True:
            if node.is_terminal():
                reward = node.reward()
                return reward
            node = node.find_random_child()

    def _backpropagate(self, path, reward):
        "Send the reward back up to the ancestors of the leaf"
        for node in reversed(path):
            self.N[node] += 1
            self.Q[node] += reward
            reward = reward
    def _uct_select(self, node):
        "Select a child of node, balancing exploration & exploitation"
        # All children of node should already be expanded:
        assert all(n in self.children for n in self.children[node])
        log_N_vertex = math.log(self.N[node])
        def uct(n):
            "Upper confidence bound for trees"
            return self.Q[n] / self.N[n] + self.c* math.sqrt(
                log_N_vertex / self.N[n]
            )
        return max(self.children[node], key=uct)
```

Experiment 1: ConnectX

To test MCTS on connect4 I chose to use the connectx environment provided by kaggle. This environment is structured in a similar fashion to how OpenAI constructs their Gym environment. Which I recommend looking into if interested. The environment provides any easy api where the goal is to allow an algorithm to easily interact with it. The board they provide is flexible in terms of size and number of pieces in a row necessary to win a game but I use the standard size of a 6x7 board and 4 in a row as a win condition. The environment provides a built in agent to play against called "negamax" which is apparently built using alpha-beta pruning. I chose a C (exploration parameter) value of $\sqrt{2}$ as seems to be the suggested value to start with without additional information. I run the algorithm from the perspective of player 1. I decided to run the experiment for 500 games performing 5 rollouts every action performed by the agent. So the average rollouts per experiment would come out to 500*5*(average game length) which although is unknown is capped by the max possible moves per games which is 42 / 2 = 21. So the number of rollouts performed is something less than 500*5*21 = 52,500. To me this seemed like an adequate amount of rollouts to learn decent play. The graph of the results are below. Every 10 games i plotted the average result from those games. The graph is quite 'jagged' and I think this comes from the fact that "negamax" is deterministic. This means that an incremental improvement will result in a sort of step-wise improvement in outcomes. Likewise for an incorrect adjustment in the model. In either case at around 20 iterations (200 games) the MCTS model has successfully found a winning strategy and since the "negamax" agent is deterministic once this is found then the model will always win.



Adding Neural Networks

In the ALphaGo Zero paper by google deepmind, MCTS is employed with great success to beat all previous known attempts at the game of Go. MCTS was used in a much different way however than the basic version that I previously described. The two main differences (there are more) are in the select step and the simulation step. The select step changes to the following $Q(s,a) + c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ Where $P(s,a)$ is the probability given from the policy network and $N(s,b)$ are still just counts. The simulate state changes to using a Value Network. The value at a state then becomes a simple lookup by feeding the current node of interest through a network to evalue the value of the current state. This value is then backpropagated up the tree as

before. The training for this network can be done in different ways however AlphaGo Zero does not do any Regression or Supervised learning from pro games. This is extremely impressive and something worth trying to understand in further detail however in this project I decided to go the Supervised Learning route to learn the value network. As mentioned previously connect 4 is a solved game meaning that every possible state of the game has a known optimal move. Fortunately a data base of optimal solutions is readily available. University of California Irvine (UCI) has a nice collection of datasets for machine learning. One data set is called 'Connect-4 Data Set' and it contains a data set of board states for the game theoretic value from the first players perspective.

Experiment 2: Training the value network

The data set contains 67557 entries. Each entry contains 42 features one for each place on the board and 1 label. The features are string and are either 'b' for blank, 'x' for player 1 or 'o' for player 2. The label is 'win','loss' or 'draw'. The first step in this experiment was to get the board features to conform to the same format as is used in the kaggle environment and is therefore the one the MCTS expects. This required some parsing and manipulation as the ordering of the board was different from environment to environment. The end result is to have an array of features that consists of 0 for blank 1 for player 1 and 2 for player 2. The label was converted to -1,0,1 for loss,draw and win. The data is now ready for training.

To train the network I decided to use a simple single hidden layer using pytorch the ML library from facebook. I split the data with 80 $\%$ being used for training and the rest being used for testing. The training set was created by random sampling the data without replacement. They hyperparameters and specs for the model were as follows

1. Data Input size = 42
2. Hidden layer size = 100
3. Output layer size = 1
4. learning rate = 0.01
5. Loss used = MSE
6. Momentum = 0.9 - It seemed to work better with momentum
7. Optimize = SGD with backprop

The following plot displays the result of training. I ran the model for 10k iterations recording both training and test error. I suppose not too surprisingly the test error did not diverge too much from the training. Its not surprising because the training samples are from the true distribution

Lets look at some actual prediction on different boards to see if the model is doing reasonable things. The first two images below are positions that were labeled as winning and seemed to be correctly identified as winning by the model. These position seem to make sense as player

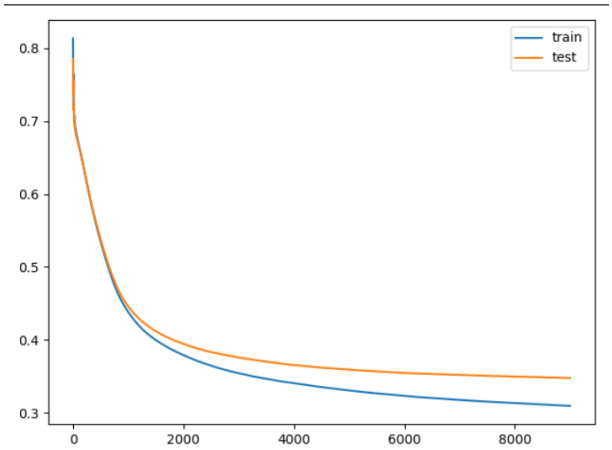


Figure 1: Train / Test loss

1 has more control of the center which I assume is a good thing although I am no connect4 expert.

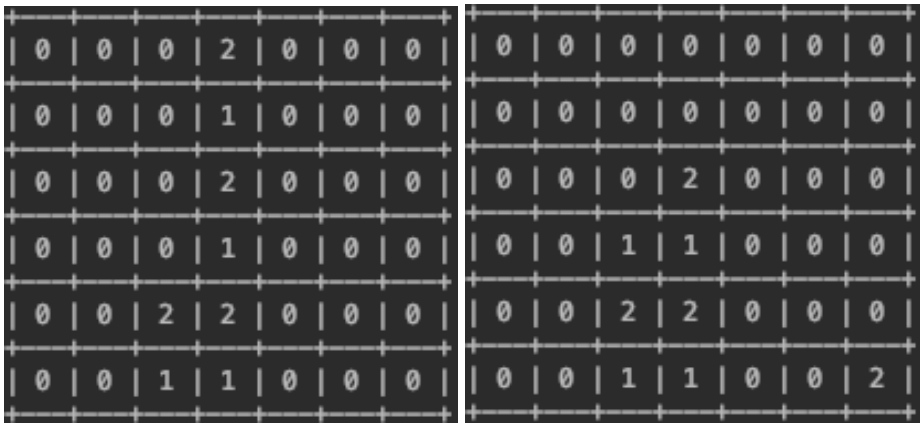


Figure 2: Left Figure: (predicted value = 1.223, actual=1) Right Figure: (predicted value = 1.25, actual =1

Here are two board positions that were correctly identified as losing for player 1

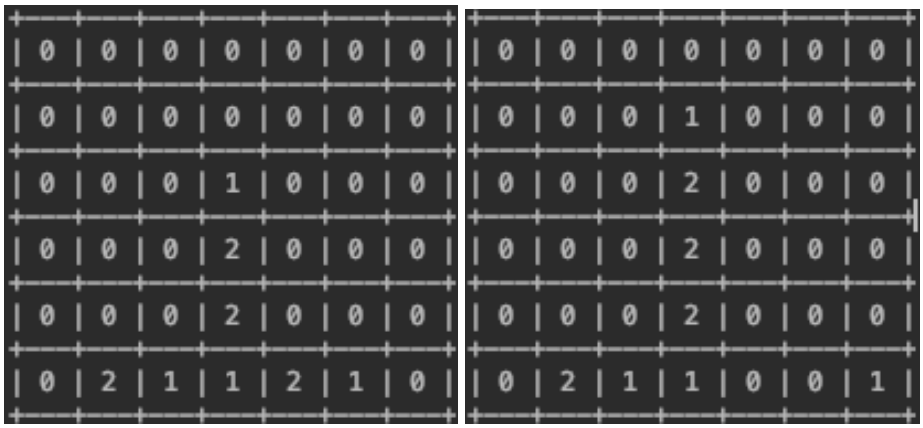


Figure 3: Left Figure: (predicted value = -0.021, actual=0) Right Figure: (predicted value = -0.233, actual =0

And alas Neural networks are not perfect.One thing that stood out in finding some board positions that were off by a lot was that when player 1 had a lot of pieces in the center the network would say it was a good position when sometimes it was not. For example the image on the left of Figure 4. Then others that were incorrect were perhaps just too close to call. The image on the left of Figure 4 to me is not obvious that player 2 has a clear edge so perhaps it is close to a coin flip as the network suggests. I counted the number of examples that had an error > 0.9 as these would probably be true errors not just getting a coin flip wrong. I found that on the full data set (training + test) this totalled 87. This is great news as these are the type of examples that could really throw off our MCTS.

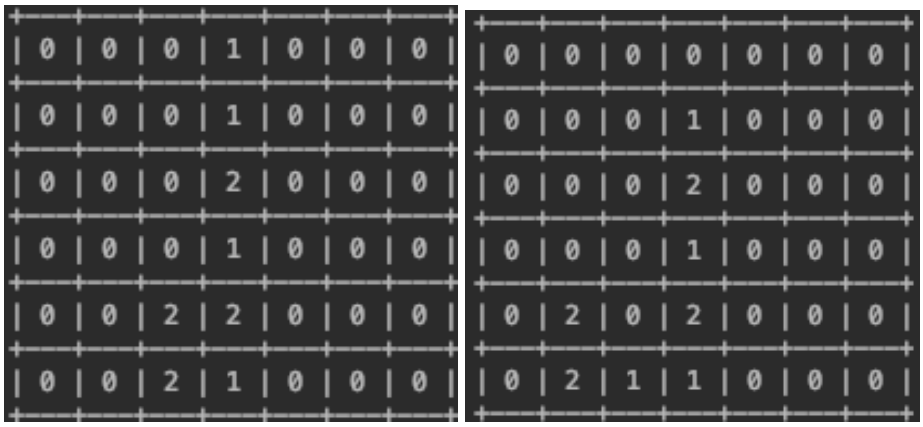


Figure 4: Left Figure: (predicted value = 0.693, actual=0) Right Figure: (predicted value = 0.524, actual =0

Experiment 3 MCTS + NN

The goal of this experiment is to run MCTS with all the same setting as in experiment 1 but using the NN to improve simulation in the MCTS algorithm and observe the differences. In the vanilla version of MCTS the agent would just act randomly during the simulation phase of rollout. This provides an opportunity for improvement. Gelly,Silver [1] wrote a paper explaining a few different ways to approach this. One that they suggested was an $\epsilon - greedy$ type approach with a guided network. That is with probability ϵ the agent will act randomly and the rest of the time will take a greedy action $w.r.t$ the value network. That is the agent will take action $\underset{a}{max} Q(s,a)$ with probability $1-\epsilon$. The value at $Q(s,a)$ is given by the value network by looking at the value of next states given by the action under consideration. So this was really a fairly simple change code wise. The change to the $_simulate method is shown below. The neural net (NN) gets passed as a parameter to the default_policy_s\epsilon$

Listing 2: Original simulate method

```
1 def _simulate(self, node):
2     while True:
3         if node.is_terminal():
4             reward = node.reward()
```

```
            return reward
        node = node.find_random_child()
```

Listing 3: New simulate method

```
def _simulate(self, node):
        "Returns the reward for a random simulation (to completion) of ‘node‘"
        invert_reward = False
        while True:
            if node.is_terminal():
                reward = node.reward()
                return reward
            if node in self.children:
                node = node.default_policy_search(self.NN,self.children[node])
            else:
                node = node.default_policy_search(self.NN,None)

            invert_reward = not invert_reward

def default_policy_search(self,NN,children):
        if np.random.random() < self.epsilon:
            return self.find_random_child()
        else:
            if not children:
                children = self.find_children()
            best_reward = -100
            best_node = None
            for i,child in enumerate(children):
                board = np.array(child.state[0].observation.board)
                x = torch.from_numpy(board).type(dtype=torch.float)
                reward = NN(x).detach().numpy()[0]
                if reward > best_reward:
                    best_node = child
            return best_node
```

Conclusion

The goal of this project was to learn and successfully implement some of the components of AlgphaGo Zero. I implemented two pieces MCTS and the use of a NN to guide search. AlphaGo zero does not use any information outside of self play and so they used a network in a much

different way than I did. I successfully showed however that it was fairly simple to plug a network into the search component of MCTS for improved results. There are a number of possible ways to iterate on what I have done. First you could incorporate a network into the select portion of the algorithm. This was alluded to earlier with a modification of UCT. You could try and train the value network through self play as is done in AlphaGo zero. I would like to have taken the time to run my algorithm against a theoretical optimal bot or at least allow my algorithm to use self play and play as player 1 and player 2 instead of from player 1.Either way my takeaway from these series of experiments is that MCTS is a very flexible algorithm that seems like it can probably be applied to a number of different domains. MCTS can be infeasible in some large domains like the game of Go but some clever use of Networks to prune search can drastically cut down on computational resources.

References

1. [1] - Gelly,Silver - Combining UCT
2. [2] - Silver Mastering the game of go without human knowledge
3. [3] - Kaggle ConnectX Competition
4. [4] - UCI Connect 4 data set
5. [5] - Github codebase for this porject