Tic Tac Toe

Anthony Mayer Machine Learning final project

As someone who enjoys chess and is fascinated by machine learning, I wanted to learn how Google Deepmind's chess ai, Alpha Zero, worked. After reading their papers, I had an idea of how the algorithm worked, but didn't feel totally satisfied. I felt I needed to implement the algorithm to fully understand it. Computational resources being the main limiting factor, I decided on a generalised version of tic tac toe as the game. The simple nature of the game should limit the number of features the models need to extract, which in turn limit the depth of the networks. In my implementation of tic tac toe, the size of the grid and the number needed in a row to win can be tuned freely. The idea being that the complexity of the game can be turned up or down as desired.

How Alpha Zero works.

Alpha Zero uses a tree search and a neural network to decide on a move. The neural network outputs two distinct quantities. A policy vector that assigns a probability that an expert player would take an action, and a scalar value indicating which player is likely to win. Feeding a game state into a model and asking it to determine the winner is a huge ask. The model doesn't know what action the players will make, or who the players are. It makes more sense to ask the model who on average will win. This is where the power of a tree search comes into play. Instead of requiring the model to be correct, we require the model to be correct on average. Random forest is a classic example of an ensemble of weak predictions becoming strong. So alpha zero explores likely moves according to the policy vector, and propagates the value of the child states back up through the search tree.

My algorithm:

Alpha zero as presented in Deep Mind's papers requires a lot of time, or a lot of resources. I simplified wherever I could, but did not deviate from the core ideas of alpha zero. The first simplification was the policy vector. The action space of even simple games can be large. If tic tac toe grid is 6X6 the action space is a vector of length 36. Treating this as a classification problem requires a huge amount of data. Too much to generate on a normal computer. Instead we could just choose the child state with the largest value according to the value network. This would get rid of the need for a policy network entirely. The problem is over-fitting. We end up reinforcing behavior the

network already displays. Instead the policy network is fed an action and outputs the estimated probability that the action will produce descendent states with higher values than the current state. As we do tree search those statistics are propagated upwards through the tree. At the end of the game the data is pulled from the tree, and exactly calculates the percent of descendent states an action produces with higher values than the current state. Instead of a policy vector, the policy network just needs to produce a scalar quantity. Hopefully this decreases the amount of data required to train a policy function. The value network will be trained directly on the wins and losses of the previous generation. The policy network will encode the average behavior of the previous value function.

Training:

At each training step there are two players, each consisting of a policy network and a value network. One is the current best player, the other is a challenger. They play 200 hundred games against each other, with a tree search of 50 rollouts per move. The player that wins the most games is retained and the other is discarded. A new player is then trained on the action and value data of the previous generation.

Tree Search:

The key to this algorithm is the tree search. How exactly the player decides which descendent states to expand. The current game state is the root node. To decide which child state to visit we calculate a Q value. Q is equal to the average value of all descendents states plus the prior probability, calculated by the policy network, that the action will produce more valuable child states. The Q values for each child state are passed through softmax producing a probability vector P. The child node is selected with a random choice according to the probability vector P. This is to encourage exploration, and introduce enough randomness that a diverse set is games played by the best player and challenger player. This randomness can be turned off if we want to see a player make the best possible moves. We keep selecting new nodes in this manner until a leaf node is encountered then expand the search tree. When the new node is created its value is propagated upwards through the tree. Alpha zero calculates Q values differently, but I had to modify it due to doing less roll outs.

The value function and policy function are relatively simple convolutional networks. The value function returns a value on [-1,1]. The larger the value the better the state is for white. In the tree search white is considered a maximizing player, trying to increase the value function, and black a minimizing player trying to decrease the

value function. When the white player is doing tree search and wants to know what moves the black player may be interested in playing, the actions with low probability of increasing the value function are selected to explore. In this way the algorithm is similar to mini-max.

Challenges:

How much data and how large the neural network needs to be to learn relevant features is not clear. One solution may be to make a synthetic data set where the model tries to predict a one if two white pieces are touching, and a -1 if two black pieces are touching. The features needed are probably similar to the features needed to play the game. We could see exactly what kind of network and how much data is needed to solve that problem. Alpha zero uses a residual neural network. Implementing a relatively small residual network would probably be an improvement.

Tree search takes a lot of time. The games could be played in parallel. I ran out of time to figure out the best way to play games in parallel. This would be a massive improvement in both efficiency and the amount of data is produced.

Another problem is not enough value data. The action data from each generation was frequently over one hundred thousand. The value data was in the thousands. This was partially alleviated with data augmentation. The game is rotationally invariant, so there are three symmetry states for every game state. One for each 90 degree rotation. Another solution is to keep the value data from the previous few generations.

Improvements needed:

1. Parallelize game play to generate more data.
2. Implement residual networks.
3. Use Q values suggested by deep mind. I believe this requires a deeper tree Search. My Q values are a bit ad hoc.
4. Implement tree search in faster language like c++.

Results:

The results were generally positive. There was definitely improvement right away.

The moves looked more intentional. A more objective measure is the amount of data generated in generation 2 is half that of generation 1. This is because the players learned to win. The hope would be that the players continually get stronger. Thats hard to say at this point because I ran out of time to train more than a few generations. I'm

very interested in seeing those results though. The 6th Generation seemed to develop some ability to defend. There are sample games at the end of this paper. Also by the 6th generation the best player was able to win several ways. Previous generations would seem to always go for the same three in a row.

Sources:

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. ArXiv e-prints

This youtube series was very helpful.

Channel: Skowster the Geek

Link: https://www.youtube.com/watch?v=NjeYgIbPMmg

Example of neural networks training.

```
Game 49 Done 12 1106
best player change
Train on 3330 samples, validate on 370 samples
Epoch 1/3
3330/3330 [==============================] - 1s 370us/sample - loss: 0.3251 - val_loss:
0.0570
Epoch 2/3
3330/3330 [==============================] - 0s 115us/sample - loss: 0.2865 - val_loss:
0.0268
Epoch 3/3
3330/3330 [==============================] - 0s 109us/sample - loss: 0.2545 - val_loss:
0.0269
Train on 78024 samples, validate on 8670 samples
Epoch 1/10
78024/78024 [==============================] - 14s 180us/sample - loss: 0.3148 - val_loss:
0.3599
Epoch 2/10
```

78024/78024 [==============================] - 13s 168us/sample - loss: 0.2695 - val_loss: 0.3480
Epoch 3/10
78024/78024 [==============================] - 13s 166us/sample - loss: 0.2577 - val_loss: 0.3474
Epoch 4/10
78024/78024 [==============================] - 14s 173us/sample - loss: 0.2539 - val_loss: 0.3353
Epoch 5/10
78024/78024 [==============================] - 13s 170us/sample - loss: 0.2520 - val_loss: 0.3357
Epoch 6/10
78024/78024 [==============================] - 13s 163us/sample - loss: 0.2460 - val_loss: 0.3302
Epoch 7/10
78024/78024 [==============================] - 13s 167us/sample - loss: 0.2435 - val_loss: 0.3282
Epoch 8/10
78024/78024 [==============================] - 13s 164us/sample - loss: 0.2431 - val_loss: 0.3281
Epoch 9/10
78024/78024 [==============================] - 13s 164us/sample - loss: 0.2443 - val_loss: 0.3276
Epoch 10/10
78024/78024 [==============================] - 13s 161us/sample - loss: 0.2389 - val_loss: 0.3254

Example games

Gen5 Best player playing white (white goes first)

```
[[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.]]
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]
```

```
 [ 0.  0.  0.  0.  0.]]


[[ 0.  0.  0.  0.  0.]
 [ 0. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.]]

[[ 0.  0.  0. -1.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.]]

[[ 0.  0.  0. -1.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  1.  1.  0.  0.]]

[[ 0.  0.  0. -1.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0. -1.  0.  0.]
 [ 1.  0.  0.  1.  0.]
 [ 0.  1.  1.  0.  0.]]



[[ 0.  0.  0. -1.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 0.  0. -1.  0.  0.]
```

```
 [ 1.  0.  0.  1.  0.]
 [ 1.  1.  1.  0.  0.]]
```

Gen5 Best player plays black.

```
[[0. 0. 0. 0. 0.]
 [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.  0.  0.  0.  0.]
 [ 0.  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.  1. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1. -1.  0.  0.]
 [ 0.  0.  0. -1.  0.]
 [ 1.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1. -1.  1.  0.]
 [ 0.  0.  0. -1.  0.]
 [ 1.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1. -1.  1.  0.]
 [ 0. -1.  0. -1.  0.]
 [ 1.  0.  0.  0.  0.]]
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1. -1.  1.  0.]
 [ 1. -1.  0. -1.  0.]
 [ 1.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  1. -1.  1.  0.]
 [ 1. -1. -1. -1.  0.]
 [ 1.  0.  0.  0.  0.]]
```

Gen6 Game 1.

```
[[0. 0. 0. 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.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.]
 [-1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0. -1.  1.  0.  0.]
 [-1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]

[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0. -1.  1.  0.  1.]
 [-1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  1.]
 [ 0. -1.  1.  0.  1.]
 [-1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

Gen6 game 3.

```
[[0. 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.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  0.]]

[[ 0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0. -1.  0.]]

[[ 0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0. -1.  0.]]

[[ 0.  1.  0.  0.  0.]
 [ 0.  1.  0.  0. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0. -1.  0.]]

[[ 0.  1.  0.  0.  0.]
 [ 0.  1.  0.  0. -1.]
 [ 0.  0.  0. -1.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0. -1.  0.]]
```

```
[[ 0.  1.  0.  0.  0.]
 [ 0.  1.  0.  0. -1.]
 [ 0.  1.  0. -1.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0. -1.  0.]]
```