DeepGo

Deepanshu Jindal, Arpan Mangal, Mankaran Singh

2016CS10312, 2016CS10321, 2016CS50391

1. Introduction

We try to replicate AlphaZero for the game of Go on board size of 13x13. We use Monte Carlo Tree Search based self play for generating examples using which we train a neural network for predicting policy and value for a given state. We describe the optimization strategies we use for improving performance of game simulation using MCTS, as well as a lot of other tricks discovered while experimentation.

2. Code Description

Our code for the Alpha Go Zero bot is divided into multiple files. The code for main player is in AlphaGoZero_1. Rest of our code is in utils_1 folder. It has the following files:

- **goenv.py :** This file contains the code for the simulator which we have used for generating training data for the neural network. We used a different simulator rather than using the one provided by TA's because the TA simulator didn't had a copy function(initially). The simulator we used can be found here. We have made some changes to this simulator to make it consistent with the chinese rules.
- **fnet.py :** This file contains the code for neural network which is trained using self play. It's detailed description and architecture is discussed in Section 3.
- **montecarlo.py :** This file contains the code for Monte Carlo Tree Search. It's detailed description is discussed in Section 4.

2.1. Libraries used

The following python libraries are required for training and running our bot:

- Torch
- Pachi_py
- tqdm
- joblib

Apart from these libraries, some standard libraries used are: Numpy, Datetime, sys, copy, traceback, time, os, random, pickle, gc, six and math.

3. Neural Architecture

Our neural architecture consists of four major modules:

1. **Convolutional Block**: Consisting of one CNN Layer followed by BatchNorm
2. **Residual Block**: Comprises of two CNN blocks followed by a skip connection from input
3. **Value Head**: CNN Layer followed by two fully connected layers - outputs a scalar indicating value of the given state
4. **Policy head**: CNN Layer followed by one fully connected layer which gives softmax probabilities over action space

The model takes in as input a 17x13x13 array - where 16 layers are the past 16 frames and the last layer is all 0's or 1's depending on which player has to play.

The neural pipeline looks like this:

1. A CNN block expands input form 17 layers to 256 layers
2. A series of Residual Blocks (we use 10 for our experiments) transform this 256x13x13 array
3. The resulting array is fed to Policy Head and Value Head which output the policy and value for the state.

```
AlphaNeural(
  (conv1): ConvBlock(
    (c1): Conv2d(17, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (b1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  {
   for i from 0 to 10
      (ResBlock i): ResidualBlock(
        (conv1): ConvBlock(
          (c1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (b1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
        (conv2): ConvBlock(
          (c1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (b1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
  }
  (policy_head): PolicyHead(
    (conv): ConvBlock(
      (c1): Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
      (b1): BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (fc): Linear(in_features=338, out_features=170, bias=True)
  )
  (value_head): ValueHead(
    (conv): ConvBlock(
      (c1): Conv2d(256, 1, kernel_size=(1, 1), stride=(1, 1))
      (b1): BatchNorm2d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (fc1): Linear(in_features=169, out_features=256, bias=True)
    (fc2): Linear(in_features=256, out_features=1, bias=True)
  )
)
```

This neural network is trained using the examples generated from MCTS. We use Cross Entropy loss for policy and Mean Squared Error loss for value. The network are trained by backpropagating on the sum of two losses. We maintain a running buffer of 100k states form which we sample a batch to train our model. For a given batch of examples we train our network for few epochs only in order to ensure that we do not overfit on the given batch.

4. Monte Carlo Tree Search

MCTS is used to generate a series of self-play games which uses the F-network to select the moves and perform simulations, and in turn generating optimal policy and value estimates which are used to train the network.

The game is played by performing a number of simulations from each move. For each move it repeatedly goes from the root node, to a previously unevaluated node (leaf), using the Q(s,a) + U(s,a) value for action selection (as specified in the paper).

4.1. Implementation

We implemented the tree in the form of having multiple dictionaries for storing things like Q(s,a) values, P(s,a) values, N(s,a) counts, valid moves for a node and so on. When a new node (leaf) is encountered, it is passed through the network to get value and policy estimate, and store it in corresponding dictionary. When encountering this node next time we just index into these dictionaries.

At every move, it performs a series of simulation, then computes the policy based on the N(s,a) counts, and takes the next move. It finally generates a sequence of moves at the end of the game, which acts as a batch. Each element will be the state at that move, the policy learnt, and the value of that state determined using the final reward.

Each MCTS call generates one such batch, having a sequence of these moves. To keep the generated game reasonable, moves were capped at appropriate values (like 450 moves) at different stages of training.

4.2. Training

We made multiple such MCTS calls to get a set of games. These games were stored in a FIFO buffer. After generating a fixed number of games, multiple random batches were selected from this set of positions and the network was updated using this. When new games were generated, the old positions were discarded based on the FIFO error.

4.3. Experiments and Tricks

1. To increase the training data, we performed data augmentation, by taking rotations and flips of each state in the training data buffer. This gave a 8 times increase in the amount of data available.

2. To fully utilize the GPU and machine compute power, multiple games were generated in parallel using the library 'joblib'.

3. Small insights like not taking the pass move when the board is essentially empty or if you are losing, helped nudge the policy towards the right direction.

4. To cut down memory cost, we need to clear the dictionaries time to time. Selecting and deleting nodes from dictionaries was found to be very slow, so we adopted a strategy of randomly clearing the whole dictionaries at some arbitrary stage in the game. This was equivalent to random initializations on the game path.

5. Next, we observed the bot stops exploring after a certain time, and keeps on just exploring one path to infinite depth. So we basically capped the depth to 60, and if it reaches 60, we just return the value of that state.

6. To encourage exploring, if N(s,a) / N(s) value for any (s,a) value exceeds 60%, it means it is taking that move too greedily, so we enforce it do more exploration by choosing other moves

5. Hyperparameters Used

5.1. Neural Network Hyperparameters

- Number of Residual Blocks: 10
- Optimizer: Adam
  - Learning Rate: 0.05
  - Weight Decay: 10^{-4}
- Mini Batch Size for training: 256
- Number of Batches: 16384 * 8 / 256

5.2. MCTS parameters

- Running Buffer size: 150,000
- Number of MCTS Simulations: 200
- cpuct: 1.5
- Temperature parameter: 30 moves
- #Games generated in parallel: 10
- Avg. time per game generation: 200 secs