Alpha Go Zero Project
COL878: Reinforcement Learning

Sudeep Agarwal
Indian Institute of Technology, Delhi
2015CS50295
cs5150295@iitd.ac.in

Ankur Sharma
Indian Institute of Technology, Delhi
2015CS50278
cs5150278@iitd.ac.in

Navreet Kaur
Indian Institute of Technology, Delhi
2015TT10917
tt1150917@iitd.ac.in

I. Problem Statement

The following project implements the game of Go on a board size of 13x13 with the Chinese set of Rules and Komi value of 7.5. Details of these rules can be found here. Our implementation is based the AlphaGoZero paper [Silver et al., 2017] by DeepMind. As described in the paper, we use MCTS with a single neural network that gives policy and value predictions with a self-play based adversary strategy to learn how to play.

II. Implementation Details

A. Representation of State

Since the current 13x13 matrix of board position is not sufficient to determine a state, we use past 5 board positions of the current and the opposite player along with the color of the current player to determine the state. Therefore, the state is represented by a tensor of shape 11x13x13. The policy is represented by a vector of size 171x1 as there are 13x13 actions on the board, plus 2 for pass and resign actions.

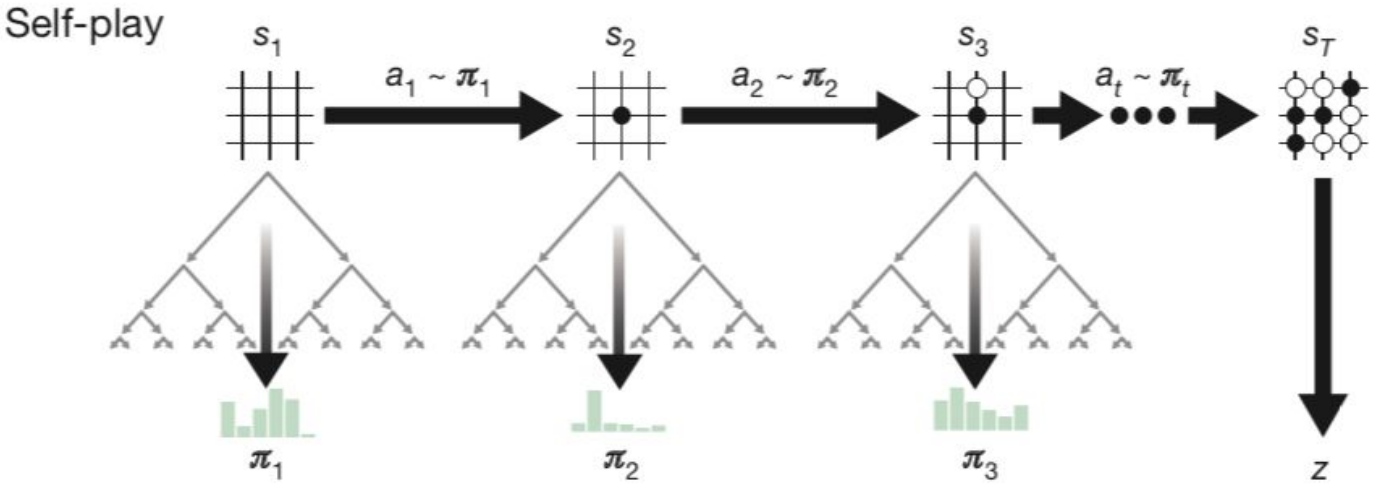
B. Episodes from MCTS

All the code has been implemented from scratch. The (x,y) coordinates of each stone on the board are used to represent board positions since no images are used. Using MCTS, the agent plays with itself until 100 simulations. MCTS can be viewed as a self-play algorithm that computes a vector of search probabilities for each state *s* recommending moves to play $\pi = \alpha_\theta(s)$, which is proportional to exponential visit count for each move, $\pi_a \propto N(s,a)^{1/\tau}$, where τ is the temperature parameter. Initially, the temperature is set to 1.0. After training for ten episodes, we set the temperature to 0.4

At each time step, an action is selected in MCTS according to

$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + U(s_t, a))$, where

$$U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$



We use the pachi-py simulator to model the dynamics of the environment. The agent can carry out multiple simulations starting from the same state in the copy of a simulator to propagate the value up the search path. This way more the number of simulations of MCTS, better are the decision choices.

C. Policy and Value Network

We tried several architecture tweaks like changing the order of layers, number of convolutions, types of pooling, use of regularizers like dropout and batch normalisation. We have reported the architecture which gave the best accuracy.

To give a brief, this network combines the role of policy and value network into a single architecture. It consists of a convolutional layer followed by 10 residual blocks of 2D convolutional layers with batch-norm and ReLu activation. The output of this is fed to two heads - policy head and value head. While the policy head consists of a convolutional layer followed by a fully connected layer that outputs a distribution over legal actions, the value head consists of a convolutional layer followed by two fully connected layers and tanh activation. All the convolutional layers are followed by batch-norm layers. The output of the value head is a positive or negative scalar value depending on whether the black or white player wins.

A detailed summary of the architecture is given below:

```
AlphaGoNNet(
  (conv): Conv2d(17, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (res_layers): Sequential(
    (0): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (3): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (4): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (5): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (6): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (7): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (8): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (9): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  (p_conv): Conv2d(256, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (p_bn): BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (p_fc): Linear(in_features=338, out_features=171, bias=True)
  (p_log_softmax): LogSoftmax()
  (v_conv): Conv2d(256, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (v_bn): BatchNorm2d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (v_fc1): Linear(in_features=169, out_features=256, bias=True)
  (v_fc2): Linear(in_features=256, out_features=1, bias=True)
  (tanh): Tanh()
)
```

The loss function used for optimisation consists summation of (i) the *Value Loss*, a MSE loss between value predicted by network and the reward at the end of the episode, and (ii) the *Policy Loss*, a cross-entropy loss between the policy predicted by network and that given by MCTS. Stochastic Gradient Descent with momentum is used for training. L2-regularisation was used for all the parameters of the network.

We rotate the board by 90, 180, 270 anticlockwise and flip it horizontally and vertically for data augmentation. Since these board positions are equivalent, this ensures robustness in our model and allows us to train for these states as well without having seen these states before.

D. Parameters

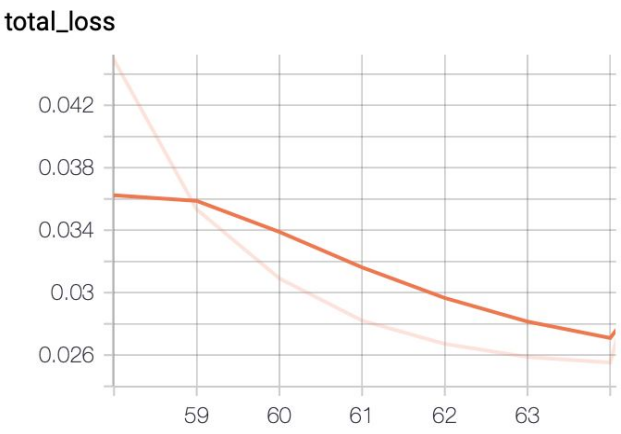
The following table gives a detailed list of all the parameters used in the project. These are specified in the *config.py* file.

	Parameter Name	**Parameter Value**
number of actions	NUM_ACTIONS	171
board size	BOARD_SIZE	13
number of time steps used for specifying current state	TIMESTEPS	5
number of features for specifying the state, this is equal to twice the number of time steps plus 1 for the color of current player	NUM_FEATURES	11
number of simulations in MCTS	NUM_SIMULATIONS	100
komi value	KOMI_VALUE	7.5
color of starting time	PLAYER_COLOR	Colour.BLACK.__str__()
Constant determining the level of exploration	CPUCT	1.0

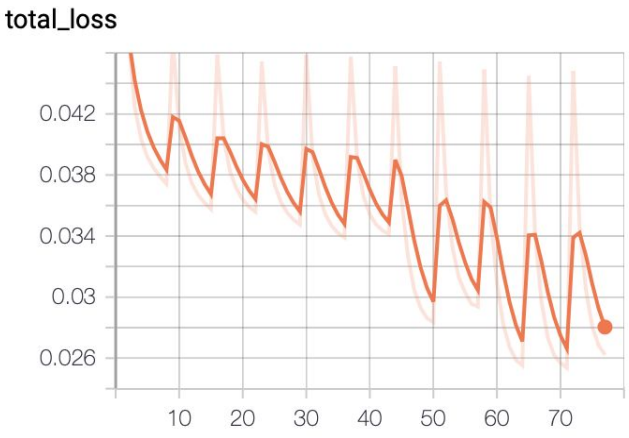
learning rate for SGD	EPSILON	1e-8
the index denoting pass action	PASS_ACTION	169
number of episodes for training	NUM_EPISODES	100000
number of residual block layers	NUM_LAYERS	10
number of epochs	NUM_EPOCHS	10
cut off for episode length	NUM_MOVES	400
batch size for training	BATCH_SIZE	128
the checkpoint to be loaded for prediction	CHECKPOINT_COUNTER	1

III. Observations

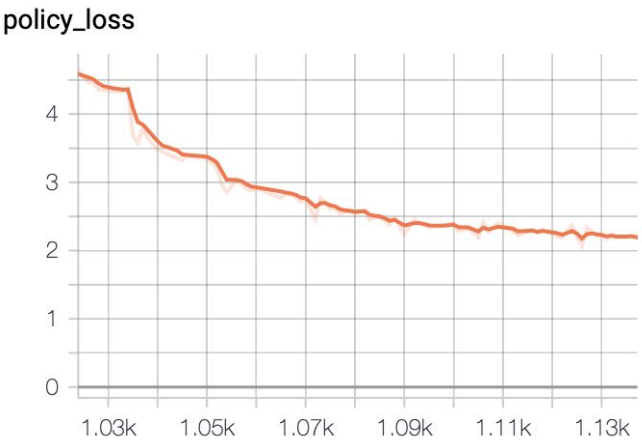
The following graphs show the variation of loss as one episode progresses and with increasing number of episodes.
It can be noticed that although the total value loss fluctuates and does not continuously decrease, the overall loss and total policy loss continuously decrease. The fluctuations in total value loss are not of big concern since the scalar value of loss itself is very small, of the order 10e-3.



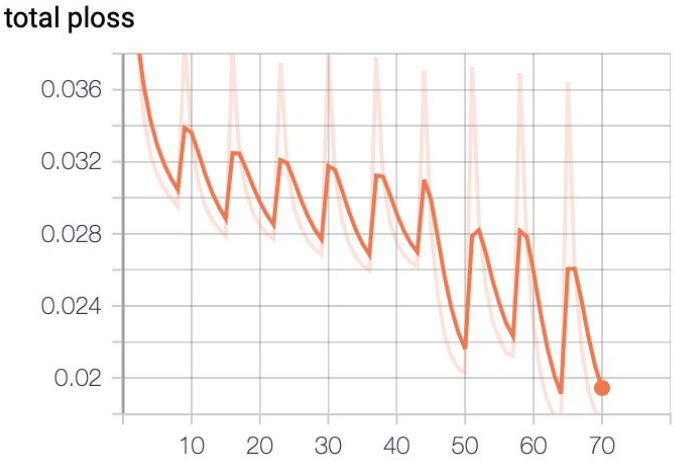
Total Loss v/s length of episode



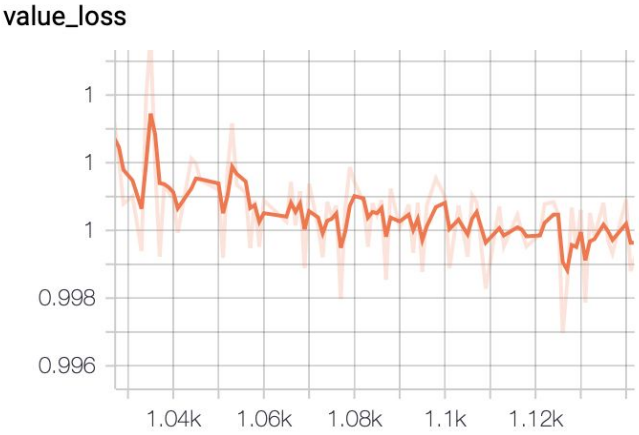
Total Loss v/s number of episodes



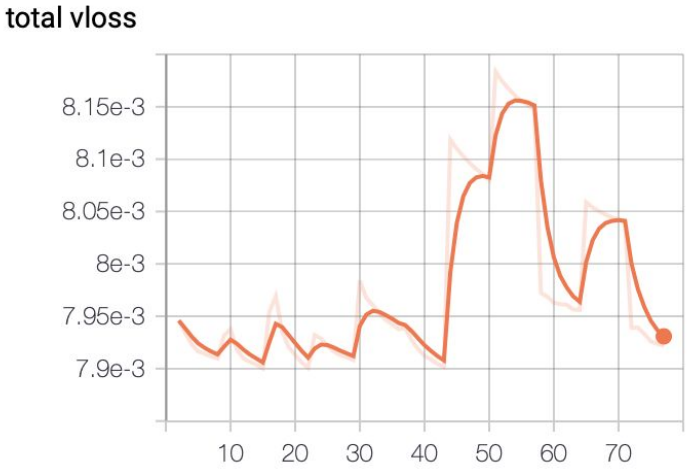
Policy Loss v/s length of episode



Policy Loss v/s number of episodes



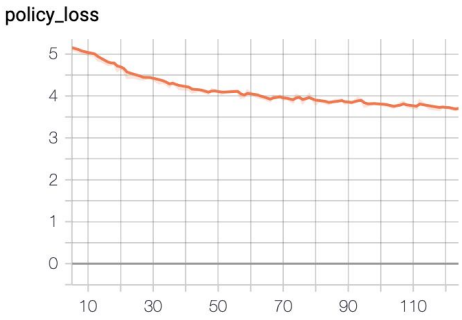
Value Loss v/s length of episode

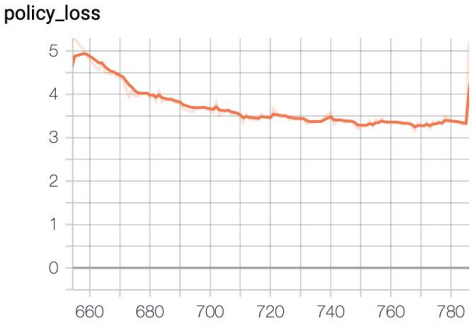


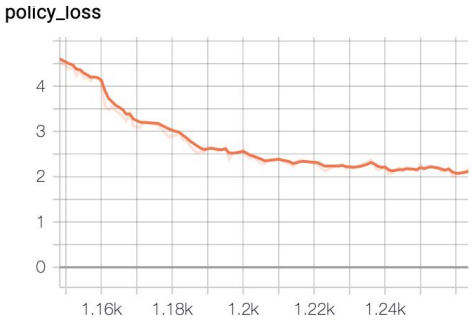
Value Loss v/s number of episodes

Below are shown some figures that show how policy and value loss decrease within an episode as training progresses. We can notice that the rate of decrease of loss increases with more number of episodes whereas the same decreases for the value loss. The value loss remains in the range 0.990-1.00 during the whole training process.

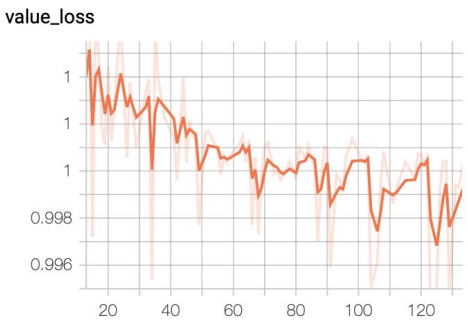
Decrease of Policy Loss within an episode as the training progresses

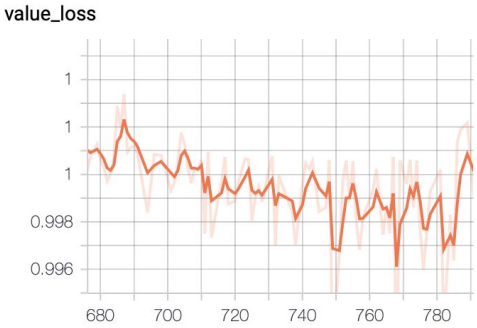


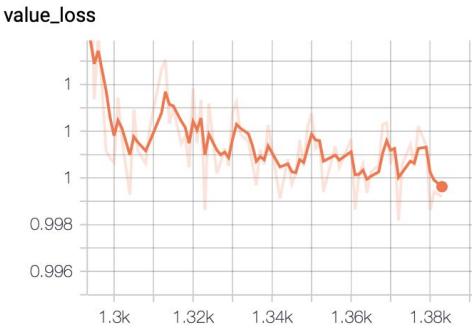




Decrease of Value Loss within an episode as the training progresses







IV. Challenges Faced

- Infinite Episode Length
- Passing State to Functions (issues with copying)
- Determining Legal Moves
- Issues in Running MCTS
- Large Training and Prediction Time
- Difficulty in Hyperparameter Tuning due to large training time. On changing parameters like momentum and number of
- After training our model on 5 episodes with 10 simulations of MCTS, we found out that the random policy was still performing better than the trained model. We made the following changes to deal with this problem:
    - We increased the number of simulations from 20 to 100. This allows our agent to make optimal choices by providing it good training examples to learn from.
    - To decrease the prediction time within boundary range of 5 seconds, we decreased the number of timesteps which are being fed to the network while ensuring a complex enough network with an adequate capacity to learn.

V. Usage Instructions

Our source code consists of the following files inside the `utils_6` folder :

- policyValueNet.py : Architecture and training procedure for the neural network
- MCTS.py : Gives policy for a given state after running MCTS for 100 simulations.
- selfPlay.py : Starts training the model after loading the latest checkpoint
- utils.py : Contains helper functions to be used for MCTS & Policy-Value Network
- config.py : Parameters required for the configuration
- enums.py : Enumeration class for BLACK & WHITE colours

To install the dependencies of this project, run:
`pip install -r requirements.txt`

To download the model in `model_6` folder:
`sh download_model_6.sh`

To start the train (best checkpoint will be automatically loaded):
`sh train.sh`

To start playing with `AlphaGoPlayer_6.py` against a random/fixed agent, execute:
`python tournament.py`

To visualise the decrement in the loss of policy & value (inside the `utils_6` folder):
`tensorboard logdirs='./logs' --port=6006`