TRAINING A DEEP NEURAL NETWORK FOR COMPUTER GO PLAYER FOR NEXT-MOVE PREDICTION

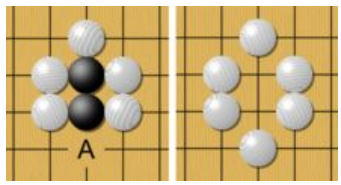
ABSTRACT

The type of the Go game is defined as alternating Markov Games. Examples of such games include chess, checkers, othello, and backgammon. There is a state space S where a current player can play, an legal action A on any state $s \subset S$, and a transition function $S^{successor} = (s, a, \xi)$ for a selecting action a in state s with a random ξ input of stone. The complexity of Go game requires ultimate human intuitive judgment rather than think through limited possible alternatives. For computers, Go's next-move prediction problem is very challenging. Its patterns are complex to analyze and evaluate, making it nearly impossible to approach by general brute-force evaluating. Recent research has proved that Go problem can be studied by computer using Deep Convolutional Neural Network. This method can train data into possibility for each specific situation and give out the highest possible move made by human players.

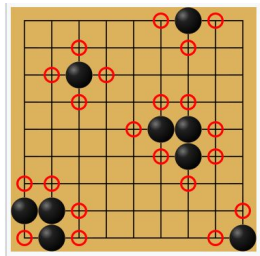
In this paper, we show the work of using Deep Convolutional Neural Network(DCNN) based move predictions by comparing and contrasting various training models and data pre-processing approaches from well-known publications. The work is inspired by Facebook Darkforest Go project and the research paper from Tian and Zhu. The Monte Carlo Tree Search (MCTS) is excluded as being off the research area.

1 INTRODUCTION

Go is a strategic board game where two players aim to surround more territory than opponent. It has a set of rules are all involved with liberties, describing how to capture pieces, what is forbidden and judgement for some typical situation. Our model is trained with board situations, so we introduce the terminologies and rules of Go as listed below. Please also note that we consider the traditional Chinese rule set only (relative to Japanese or AGA rule set).



Basic Rule: if white plays position A, blackstone group loses all its liberty and is captured by white; Suicide rule: A player cannot place a stone such that it or its group immediately has no liberty unless it will cause some capture and get some liberty right after.

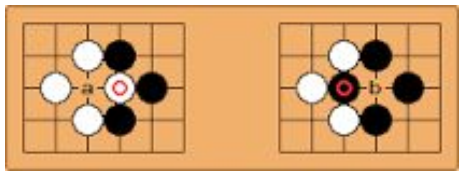


Five groups and their liberties (marked by circles).

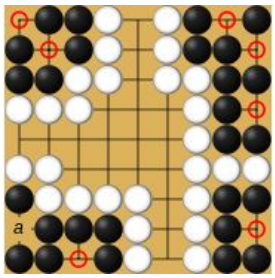
Liberty: A liberty is a vacant point that is immediately adjacent to a stone in a cardinal; meaning orthogonal, direction, or is connected through a continuous strings of same-colored stones to such a point. A stone, chain, or a group must have at least one liberty to survive. A group that has two or more separate internal liberties (defined as 'eyes') is impossible to be captured. We calculate the liberties of a placed stone by finding all possible liberty positions for each player's territory and check whether they are occupied by the opponent.

Territory (Eyes): Eyes are internal liberties of a group of stones that, like external liberties, preventing the group's capture. The only way to kill a group

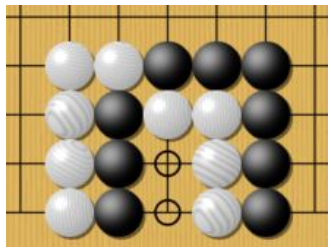
with an eye is to fill all of its liberties before calling *atari[1]*. The presence or absence of eyes in a group determine life or death of that group. A group with no eyes, or only one eye, will die unless its owner can develop them. Conversely, a group with two or more eyes will live. An opponent cannot capture such a group because it is impossible to remove all liberties of the group by playing one stone, without violating the suicide rule.



Ko Rule: Players are not allowed to make move to return the game to its previous situation: for example, black plays position A then white plays position B.



Life and Death Rule: those circles are called eyes; if a group with two or more eyes is alive, otherwise it is dead and it is captured by white: black groups on the top are alive, but black groups at the bottom are dead (for bottom left situation, if white plays position a, then black has only one eye and the whole group is dead).



Seki Rule: in this case, no players want to place stone in two circles because if so, opponent can capture stones by playing the other circle: so that no players in this situation will receive points.

Ladder Move: Ladder is a basic sequence of moves in which an attacker pursues a group in atari in a zig-zag pattern across the board. If there are no intervening stones, the group will hit the edge of the board and be captured.

Building an evaluation and prediction system upon integrating all the rules requires non-trivial modeling expertise. Every single move can have effect on the next probability of prediction. Humans players and computer have strategic difference: human experts rely on pattern recognition to decide next move whereas computer has to go through many possible moves to find that particular position at the moment which may not be as far-sighted as human player. Paper from Yuandong Tian and Yan Zhu has success to extract features in a granular systematic way that separates all attributes of a placed stone. With the advancement of DCNN and abundant sample data, training a policy network model becomes doable.

Thus, our goal is to train numerous amount of games on a 19-by-19 board using our Deep Convolutional Neural Network model in order to predict next several moves under given situation. We will build our own DCNN model, train it with newly purchased data from GoGoD 2015 and compare the result with other publications.

2 Prior work

Take-away from Papers Facebook Darkforest Go and Maddison's paper "Deep Go"

After analyzing two models from these two papers by comparison and contrast, we find several interesting points that need to be mentioned and could be used in our own experiments:

1. Training model structure:
   Darkforest paper uses 12-layer full convolutional network whereas Maddison's paper has only 6 layers. However, their results are close. Darkforest has 25 feature planes but Maddison's paper has more: 36 feature planes including liberties after the move, captures after the move, ladder move, etc. Darkforest paper has accuracy that drops when adding too much layers. This justifies Maddison's paper that they find excessive layers cannot break the learning barrier when a model has more than 12 Covn layers: even though the model gets more layers, the improvement stalls, and this maybe due to information loss.
2. Using nonlinearity function `ReLU` instead of `Tanh`:
   We find both articles use ReLU method. The definition of ReLU is $f(x) = max(0, x)$ where $x = Weight*a + b$. There are basically two benefits: The first one is to induce sparsity when $x \leq 0$. If ReLU exists in a layer, it creates more sparse results for the hidden layer. The second one is that ReLU causes less gradient vanishing problem. Since when $x > 0$, gradient has a constant value whereas sigmoid and tanh will have the gradient smaller when x increases.
3. Using kernel size 5x5 or 3x3:
   Kernel size has to be odd since we need to have 1 kernel in the middle in order to not blur the graph. Then we find 5 x 5 and 3 x 3 is the most useful kernel size to train Go because: on the one hand, the practice on data shows that low level features within 5 x 5 pixels are representative and we take a close look by using smaller kernel in that situation; on the other hand, if we try to use 1 layer with 7 x 7 filter size as first layer, we had better to use a 2-layer stack with 5 x 5 kernel or a 3-layer stack with 3 x 3 kernel because the later ones use fewer parameters and make the possibility closer to locality.
4. Using of `SoftMax` and `ClassNllCriterion` for next move prediction:
   Softmax classifier can transform hidden units into probability scores of the class labels the model provides. We are able to transform pattern recognition problem into a two dimensional grid classification problems. Since the eventual goal is to figure out the likelihood of next move.
5. Not using Max pooling:
   We tried max pooling function for every three layers among those 10 consecutive layers in Facebook DarkForest model source code. Our purpose is to discretize the input and keep the most responsive interest points for further layers to analyze. But it turns out to result in lower accuracy. This might be caused: a good move may not be such responsive unless it's an obvious move; max pooling will prevent some irregular but good moves to show up in the model.

3 METHOD

We model an input of 19-by-19 board as of current situation, where the data is read from sgf file into multiple feature planes.

A sample sgf file from GoGoD is shown on the right. Each step is saved as a single command

```
(;GM[1]FF[4]
CA[UTF-8]
PW[Sakai Hideyuki]
WR[9d]
PB[Hane Naoki]
BR[9d]
SZ[19]
EV[36th Gosei Final]
RO[Game 3]
US[GoGoD95]
;B[qd];W[dd];B[pq];W[dp];B[oc];W[po];B[qo];W[qn]
;B[qp];W[pm];B[nq];W[qe];B[pe];W[qf];B[rd];W[pf]
;B[ql];W[oe];B[pl];W[ol];B[om];W[ok];B[nm];W[qj]
;B[rn];W[gq];B[cf];W[fc];B[bd];W[ch];B[dh];W[di]
;B[no];W[sm];B[pp];W[nc];B[nb];W[ob];B[pb];W[od]
;B[pc];W[mb];B[oa];W[mc];B[gd];W[gf];B[gc];W[fd]
;B[ge];W[ff];B[hf];W[hg];B[if];W[ig];B[id];W[jf]
...
```

separated by ';'. For example, B[qd] in line 1 represents a new black stone that is put on position qd. Our preprocessing procedure will translate it into [17,4] and put an 1 on corresponding position. SGF file also contains critical information about the players. The sample sgf shows a 9d player Sakai Hideyuki held white stone (WR[9D]) in this game and his opponent was another 9d (BR[9D]) player named Hane Naoki.

We purpose to try out the data modeling by using *27* feature planes, as which is devised based upon the original paper by Tian and Zhu's. Eventually, We use different amount of layers in various DCNN designs by comparing many Go Game models permuted with different number of feature planes. By analyzing results, we then tweak some parameters in the model so as to sharpen the predict accuracy of our final model. This trial and error approach will relate to statistical methodology such as classification and regression analysis in part 3 of this report.

3.1 FEATURE CHANNELS

We populate each situation after a change on the board as an input or an output for the training. In order to train for each rules of the game, we extract all features from a current board situation. There are many 'current board situations', think of it as multiple frames in a forward-speed Go game video. We map all situations into one single game. As been said, a current board situation contains many underlying information that has been played out by two players according to the rules. Thus, we extract essential features, each feature corresponding to one feature plane. The next page has a table that illustrates the planes:

Rules / Terms	Information	Number of Planes
Liberties	4 binary layers per player, write 1 on the first layer if there is one liberty, write 1 on the second layer if there are two liberties, etc. If there is none, write 0.	8 (4 per player)
Rank	write 1 for the current rank plane.	9
History moves	how long the move has occurred. (0, 1) $HistoryMove = e^{(-t*0.1)}$.	2 (1 per player)
Position Mask	1 real layer of position: this layer records all stones' position using formula exp(-0.5*l) where l is the distance between stone's position and center of board. The calculate formula is different from Facebook's since we want to make sure each value on the board is represents by some real number larger than 0 and smaller than 1.	1
Current/Opponent/Empty	All binary layers. One for current player and one for opponent. Each layer indicates the region of one player. The third layer shows empty position remain on board.	3
Future Legal Move	1 binary layer of legal position for next move: considering both empty position and k-o rule (k-o rule prevents player to put stone on captured position unless the step could cause another capture).	1
Captured stones	2 binary layer of capture stone: 1 for current player and 1 for opponent. Each layer records position of stones which has been captured.	2
Ladder Move	1 binary layer of ladder move: this layer will be filled by 1 if last move (made by opponent) is a ladder move.	1

We have 27 feature planes overall to map each 'current situation'. Below is the code to parse these information from sgf file into a computable tensor. A quick SGF file specification intro can be found here at: http://www.red-bean.com/sgf/user_guide/index.html#what

A sample SGF parse and save Lua snippet is shown:

```
-- store the whole game from a sgf file
BOARD_DIM = 19
local file = io.open(path..sgfFile, 'r')
local content = file:read('*all')
local game = torch.Tensor(
    contentSplitter(content), 47, BOARD_DIM, BOARD_DIM
    ):fill(0)

-- to get the BR's dan rank
function sgfLoader:get_ranks()
     if string.match(self.sgf[1].BR, "%dd") ~= nil then
          return self.sgf[1].BR
```

3.2 NEURAL NETWORK ARCHITECTURE

The training process is to extract features from an input and predict a next move based on existing features. To get as accurate prediction as we could. We trying to improve the result accuracy in the following two aspects.

1. Training Model:

   Our model is based on Facebook's DarkForest model, which has 12 convolutional layers, each followed by a ReLU non-linear layer and a SpatialBatchNormalization layer. In order to save time, we use 7 Convolutional layers instead of 12 in most tests, but there are also tests with 12 layers but less game datasets.

   We choose to use convolutional layer with zero stride to ensure that for each 19-by-19 input feature plane, we will get an 19-by-19 output. We don't use pooling layers for the same reason. Some paper we read use linear layer after convolutional layer. We try 2 different networks with linear layer: one with 1 linear layer and another with 2 linear layer including 1000 hidden units. Both of them cannot converge during the first 10 epochs.

2. Dataset:

   Several experiments have been put in practice with various parameter tweaks.

   1. The amount of data used for training:

      We extract 300 games among all games in 2013 from the GoGoD and choose 40 steps randomly to generate training data in the beginning. Then we increase number of games used for training from 300 to 600, 1200, 2400 and 4200.

   2. Weights of history layer:

      Originally, we used last 5 step's position as history layer. We changed formula into *exp(h*(-0.1))* where *h* is the number of turn since stone is put. This formula comes from Facebook's paper and has two advantages:

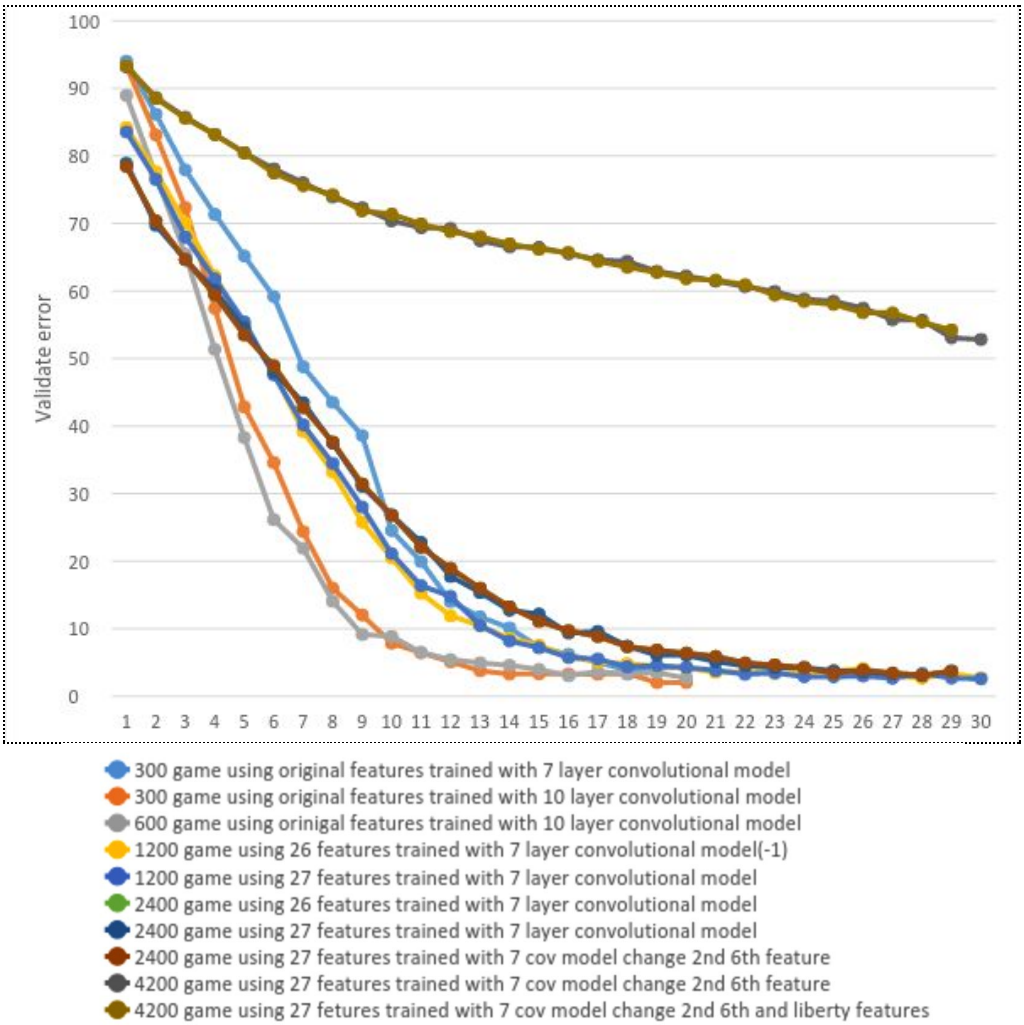
      - 1) Recent moves weights more in the final move decision.

- 2) Allows more history steps to be taken into consideration.

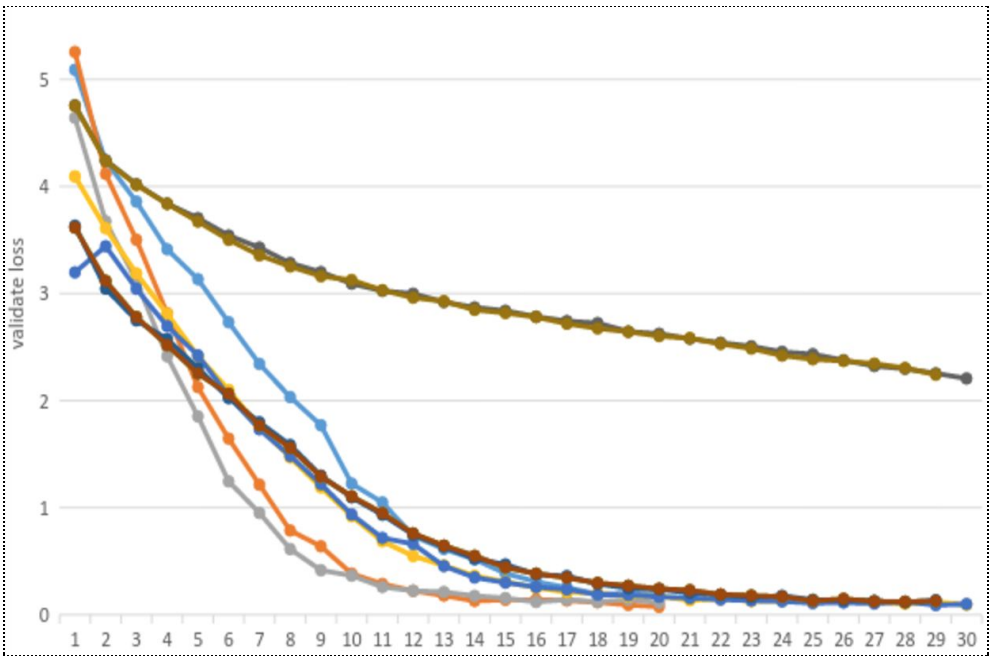
3. Layer marking for the rank of a player:
   Two approaches to mark rank layers. One is to fill corresponding layer with 1, the other is to fill corresponding layer with -1 if player's rank is less than 9 and only fill out 1 at the *'9th'* layer if a player ranks 9d. This is because that 9d rank is professional for Go player and it worth larger weight. The idea also come from Facebook's paper.
4. Add position mask using formula *exp(-0.5*l)* where l is the distance between stone's position and center of board.
5. Multiply -1 for opponent's layer to distinguish current player and its opponent. During the first few experiment, we generate both feature equally. Using layer 1 and layer 2 as example, these two layers contain information of all current player and opponent's stone's position, assuming that current player has stone on position [yx], we will then put an 1 on layer 1's corresponding position. And we do the same for opponent's stone. After several experiments, we decide to use -1 rather than 1 to represents opponent's position, which distinguish it from current player, the purpose for such change is to let model learn difference between current player and opponent.

Exp Indx	Number of Layers	Number Of Conv Layers	#Input feature planes	Specifications	Accuracy	#Training Games	Epoch
f1	20	7	26		0.135	300	30
f2	23	10	26		0.1208	300	30
f3	23	10	26		0.1516	600	30
f9	20	7	26	Using -1 to fill in the rank layer instead of 1. Conv(27>92,5*5,1,1,2,2) - ReLU - Norm - Conv(92>384,3*3,1,1,2,2) - ReLU - Norm - (Conv(384>384,3*3,1,1,1,1) - ReLU - Normalization)*3 - View(361)	0.221	1200	50
f4	23	10	27	Optimized history and position layers.	0.2258	1200	
f10	20	7	27			1800	30
f11	30	7	26	Using 26 layer feature plane structure to reconcile if 27 feature does perform better.	0.1927	2400	30
f5	20	7	27	Similar to above, but with different feature planes and different training games	0.2508	2400	30
f6	20	7	27	Changed feature of opponent's position plane and history.	0.295	2400	30
f7	20	7	27	0.0001 Learning rate + Changed feature of opponent's position plane and history	0.31	4200	30
f8	20	7	27	0.0001 Learning rate + Changed feature of opponent's position plane, history, and liberty.	0.3025	4200	30

Graph 1. Validate Error vs Experiments



Graph 2. Validate Loss vs Experiments



3.3 TRAINING

Training begins by teaching a ConvNet to simply predict where an Go expert player would do his or her next move. Ideally, each board position and its corresponding `{numberOfPred}` subsequent moves will be move by an iterator to form a training pair. (In our experiment, we use 1, which predicts one single next move.) Additional pair(s) can be constructed by doing plan rotation and reflection so as to create more granular training data, each board situation can have up to 8 different transformed situations. ReLU non-linearity and backpropagation are used (automatically in our Torch 7 program) to maximize the probability of selecting the given action.

In training, we first started off using local Mac OSX dual core with slow training 12+ hours per epoch for our first trial, 300 games training data and 7 convolutional layers. The time decrements to 30 minutes on HPC per epoch. As one of the analytical results listed in the third part of the report, increase in training dataset results in longer training time. 600 training datasets results in 1+ hour per epoch. We pick up on Amazon AWS as well as Mercer's HPC, which reduces to 12 minutes on average per each epoch for the same training set and number of games, and a training set with 4200 games results in a 29-minute-per-epoch training speed.

The code below is a sample command that uses the same logic from homework 3 to pass in values of various parameters for easy runs. Learning rate was firstly set to 0.001 for easy testing, then divided by 10 as convergence tends to stop dropping.

```
opts-27.lua:
     th train.lua
          -net orignal_darkforest  -batchsize 10
          -nThreads 3              -LR 0.001
          -nEpochs 30              -numberOfPred 1
          -batchsize 128
```

Each training has an output log file includes the training and validation accuracy on each epoch as well as system time consumption. Many well-known Go game models are also tried with minimal changes in our input data.

We use following code file for our experiments.

- `rdataset-4200-c26-1.lua`: the code is used to generate training dataset. To save time, we run 7 jobs at the same time on hpc, each process 600 games. Test data is generated using same code with slight change in the number of game.
- `test-4200-c26.lua`: the code is used to calculate accurate for each model.
- `connect.lua`: the code is used to connect 7 separate generated dataset together.
- The following codes is based on traffic-sign classify code in assignment 3.
- `main-4200-c26.lua`: the code is used to run training process
- `opts-27.lua`: the code is used to decides running epoch and learning rate.
- `network-27.lua`: the code is used to store network architecture.

Simply use `th rdataset-4200-c26-1.lua` files to generate dataset, make sure the code has the right path and all games under the datasets directory are renamed with sequential numbers start with 1. Note that this file only processing 600 games.
To train model, use command:

```
qlua main-4200-c26.lua.
```

3.4 EXCLUDING MCTS

One thing that needs to be pointed out is the neglection of MCTS in this project. We acknowledge that by all means MCTS; a powerful value and fast-policy network that helps to rollout the game all the way to the end to see who wins, will definitively improve the final result of our trained model. However, its field is not aligned with the Deep Learning subject.

4 CONCLUSIONS AND FUTURE WORK

According to all experiments that have been illustrated with tables and graphs in section 2.2, we conclude several factors which could affect model's performance.

i) Number of the training data: the increase amount of training data improves model's performance significantly.
ii) Way to mark rank layers. Separate normal player from professional player increase model's performance.
iii) Position mask. Models using position mask feature have better performance than those without.
iv) Increasing conflict between current players and opponent. Change way to mark board information layer and history layer improve the performance. However. changing way to mark liberty layer will decrease performance.

Our result is not as good as Facebook's Darkforest model. Although many factors may exist, one of the main reasons is the number of data used for training. Since our dataset is generated by our preprocessing code, each time we modify features, we need to regenerate our training dataset. Unfortunately, our code for preprocessing is not optimized enough and it takes around 1 hour to process 100 source files.

Our best training results is trained with a 7 convolutional layer model, using 4,200 games from GoGoD, for 24 hours while Facebook's result has 144,748 games, trained with 12 convolutional layer model and takes 2 weeks to finish the training process with 50 epochs.

Further work:

1. Resume a previously trained model by logging the model and last trained epoch number so as to have on-call inspection and save more time if continuous training may show some promising result.
2. Try new feature planes. Prevalent patterns of Go Game have been broadly mastered and professional players seems very sensitive to these patterns and they would have counterplan for each pattern either. Since our prediction process is pattern recognize

process, using more pattern mask on board will help the outcome of a model have better performance.

3. The current program is limited with few corner cases. Special situations such as handicap is not implemented, yet it is logically easy to think of an approach such as initializing a new board with already assigned values for all 27 feature planes. We can certainly implement a parser or local 'image pixel' identifier specifically for some well-known strategies in each ranked game.
4. Of course, the other part of the very fundamental neural network as described in the original paper - the value network, which can help reduce the hundreds of possibility of moves down to a handful of more promising moves. While the policy network evaluate the breadth of the network, the value network can be devised as some representative depth level of the 'possibility tree'. By leveraging MCTS; instead of brute-force, we can localize a specific depth of the game, then find the more-likely-to-win path and continue the training with our policy network. This will not only enable faster training but also the utter purpose of the Go Game - to win a game.

5. Reference and Resource

1. Training data set: GoGoD [URL: http://gogodonline.co.uk/]
2. Yuandong Tian, Yan Zhu. Better Computer Go Player with Neural Network and Long-term Prediction. arXiv:1511.06410 [cs.LG] 2016.
3. Christopher Clark, Amos Storkey. Teaching Deep Convolutional Neural Networks to Play Go. arXiv:1412.3409 [cs.AI] 2015.
4. Move Evaluation in Go Using Deep Convolutional Neural Networks, Chris J Maddison University of Toronto cmaddis@cs.toronto.edu A. Huang, I. Sutskever, D. Silver.
5. Rules of Go, Wikipedia [URL: https://en.wikipedia.org/wiki/Rules_of_go]
6. Facebook Research / Darkforest Go, Github [URL: https://github.com/facebookresearch/darkforestGo]