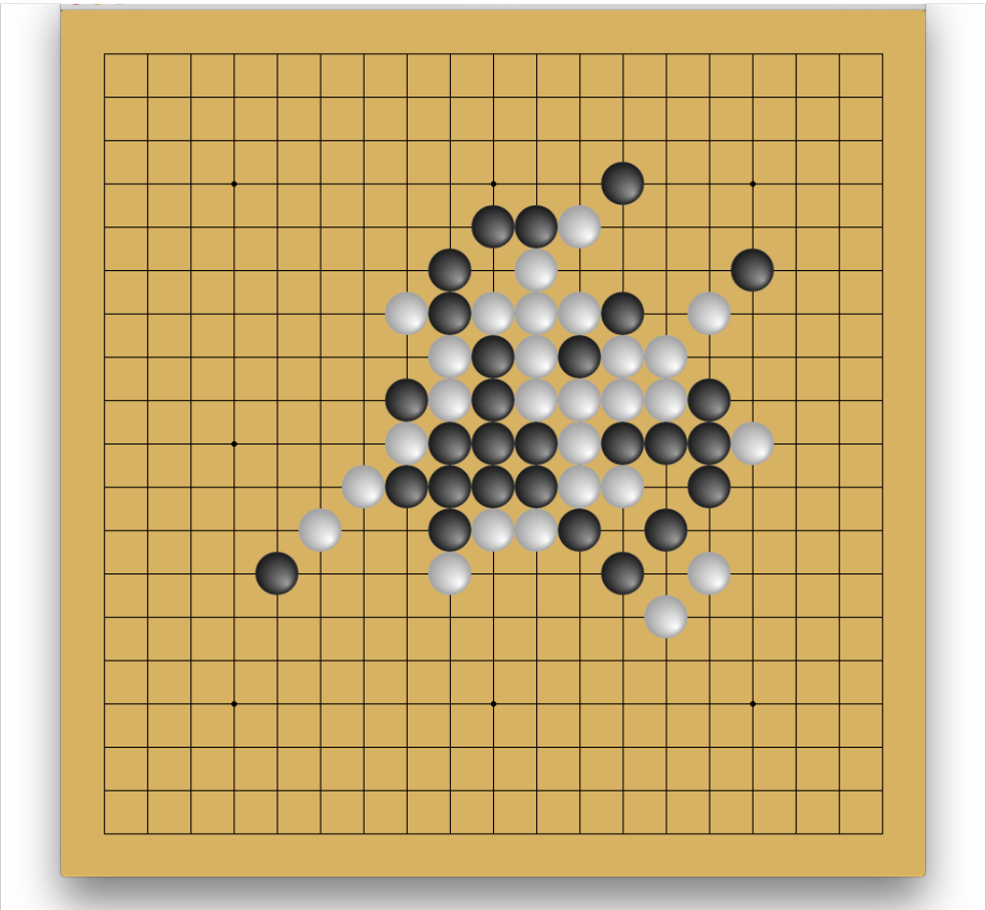


Capstone Project

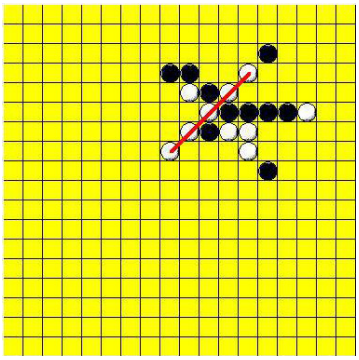
USING DRL TO TRAIN A GOMOKU AGENT

Tong Zou | Udacity MLND | 2/3/2017

I. Definition

PROJECT OVERVIEW

Gomoku is an abstract strategy board game. Also called Gobang or Five in a Row, it is traditionally played with Go pieces (black and white stones) on a go board with 19x19 or 15x15 intersections; This game is known in several countries under different names. Black plays first if white did not win in the previous game, and players alternate in placing a stone of their color on an empty intersection. The winner is the first player to get an unbroken row of five stones horizontally, vertically, or diagonally. This game is basically an extension to Tic-Tac-Toe. It's played on a much larger board and instead of connecting 3 stones together, we need to connect 5. The following is a picture of a winning position for white.



Even though Gomoku is a pretty simple game, it's non trivial. There is a world championship in different cities of the world every two years since 2009.

As in the area of using AI to play Gomoku, to quote Wikipedia[1]: "People have been applying the artificial intelligence technique on playing Gomoku for several decades. In 1994, L. V. Allis raised the algorithm of proof-number search (pn-search) and dependency-based search (db-search), and proved that when starting from an empty board, the first player has a winning strategy using these searching algorithms. In 2001, the winning strategy was also approved for Renju, a variation of Gomoku, when there was no limitation on the opening stage.

However, neither the theoretical values of all legal positions, nor the balanced rules used by the professional Gomoku players have been solved yet, so the topic of Gomoku artificial intelligence is still a great challenge for computer scientists, such as the problem on how to improve the Gomoku AIs to make it more strategic and competitive. Nowadays, most of the state-of-the-art Gomoku AIs are based on the alpha-beta pruning framework."

In December of 2013, a small company in London called DeepMind published their pioneering paper "Playing Atari with Deep Reinforcement Learning"[2] to arXiv. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased.

The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human! Later, DeepMind created AlphaGo[3], a Go playing AI. In October 2015, it became the first Computer Go program to beat a human professional Go player without handicaps on a full-sized 19×19 board. In March 2016, it beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps.

Inspired by the AphaGo's success, in this project, I try to use deep reinforcement learning methods to train a Gomoku agent. Due to time limitation and my own knowledge limitation, the policy that I find is not going to be optimal, I will do my best to see how intelligent I can make the agent behave, whether the agent can learn some basic rules of the game without any prior knowledge.

PROBLEM STATEMENT

There are quite a few different methods in Deep Reinforcement Learning, there are DQN (Deep Q-Network), Gradient Policy, etc. AlphaGo has both a policy network and a value network. In this project, I will just use a policy network and use the Policy Gradient method only, this is heavily influenced by Andrej Karpathy's blog post.[4]

I've setup a couple areas to explore in this project.

A. Can the Agent learn how to make valid moves? A valid move in Gomoku is placing a stone in any empty space.
B. Can the Agent learn how to beat a random opponent? A random opponent just plays valid moves randomly.
C. Can the Agent learn the basic strategies of the game? Some basic strategies would be:
    1. When we have 4 stones connected, and there is empty space to make a move, then make the move to win. (S1)
    2. When opponent has 4 stones connected, and there is empty space, try to block it. (S2)
    3. When opponent has 3 stones connected, and there are empty spaces on both ends, try to block it. (S3)
    4. When we have 3 stones connected, try to extend it. (S4)
    5. When we have 2 stones connected, try to extend it. (S5)
    6. When we have just 1 stone somewhere, try to extend it. (S6)

    These strategies will be used as the metric to evaluate my agent.
D. What's the difference in training the Agent against a fixed opponent than to self-play?

In summary, my goal is to make the Agent as smart as possible using the Policy Gradient method.

METRICS

To test how the agent well the agent has trained, I have setup a couple opponents for testing.

A. Random opponent. This will just play any random valid moves.
B. Naïve opponent. This is a very simple opponent which executes a fixed strategy. I have created four levels for this Agent:
    0. The level 0 naïve opponent will execute strategy S1, S2, S3, in that order, and if none found, uses random move.
    1. The level 1 naïve opponent will do what level 0 does, and then execute S4. If none found, uses random move.
    2. The level 2 will extend level 1 with strategy S5.
    3. Level 3 naïve opponent will also execute S6.
C. Finally, I also incorporated an intelligent AI opponent. [5]

The level 3 naïve opponent basically corresponds to a beginning human player.

To test how well the agent performs, I have used two metrics. One metric is called the `running_reward` and is defined in the Benchmark section.

After training the agent, to test how well it performs against a specific opponent, I simply let the two play for a fixed number of episodes such as 1000 and take the winning ratio. The default would be the level 3 naïve opponent.

II. Analysis

DATA EXPLORATION

Since this is a reinforcement learning project, there are no external datasets, but I do need to have an environment setup. After some searching around, I've decided to use OpenAI Gym as a framework for the environment. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms.[6] But unfortunately, there is no Gomoku environment. So as part of this project, I have created an OpenAI Gomoku environment. The environment will by default uses the random opponent.

In this environment, there are black player and white player. Black player always plays first. The agent can be the black or the white player. States are represented by a multidimensional numpy matrix. Below, n represents the dimension of the board. If we have a 9×9 board, then $n = 9$.

- $state[0]$ is a $n \times n$ square matrix that represents black player's stones. If there is a black stone at position (i, j), then $state[0][i][j] = 1$, else it's 0.
- $state[1]$ is a $n \times n$ square matrix that represents white player's stones. If there is a white stone at position (i, j), then $state[1][i][j] = 1$, else it's 0.

- $state[2]$ is a $n \times n$ square matrix that represents positions that has stone already. If there is a black or a white stone at position (i, j), then $state[2][i][j] = 1$, else it's 0.

The main function of this environment is ***step***. It takes in a single parameter ***action***, which is represented by a number from 0 to n^2. 0 to $n^2 - 1$ maps to a position on the board, and action n^2 represents a resignation move. So there are total $n^2 + 1$ possible actions. The function outputs a tuple: $(observation, reward, done, info)$.

- `observation(object)` is just the current state of the board.
- `reward(float)` is the reward for entering this state. In this environment, the agent will get a reward of 1 if it has won the game, and -1 if it has lost the game, and 0 otherwise.
- `done(boolean)` whether it's time to `reset` the environment again. This will indicate the game has finished.
- `info(dict)` is diagnostic information useful for debugging.

This is just an implementation of the classic "agent-environment loop". Each time step, the agent chooses an action, and the environment returns an observation and a reward.

When the environment sees an invalid action, i.e., out of range or a stone is already at the position, it will either raises an exception or marks the player as lost depending on a preset parameter.

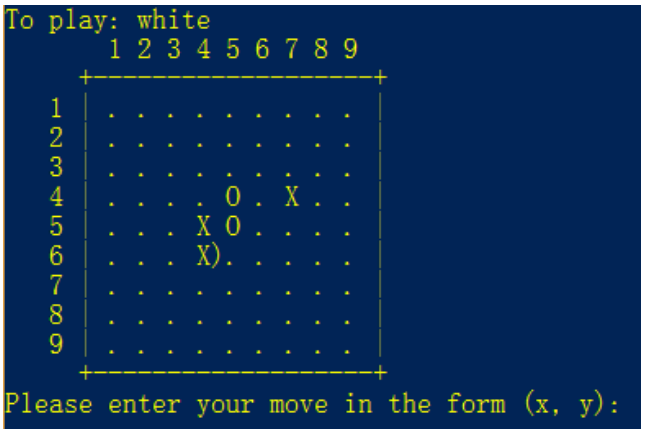
The environment also implements the render function, which will output an ASCII based rendering of the game board and the black and white stones. Black stone is represented by 'X' and white stone represented by 'O'.

I have also implemented some static helper functions for the environment to make programming an opponent easier.

To make the project more manageable, I have restricted to only the 9×9 board size. Even though the board is not too big, the number of possible states are still very big, there are a total 3^{81} possible states.

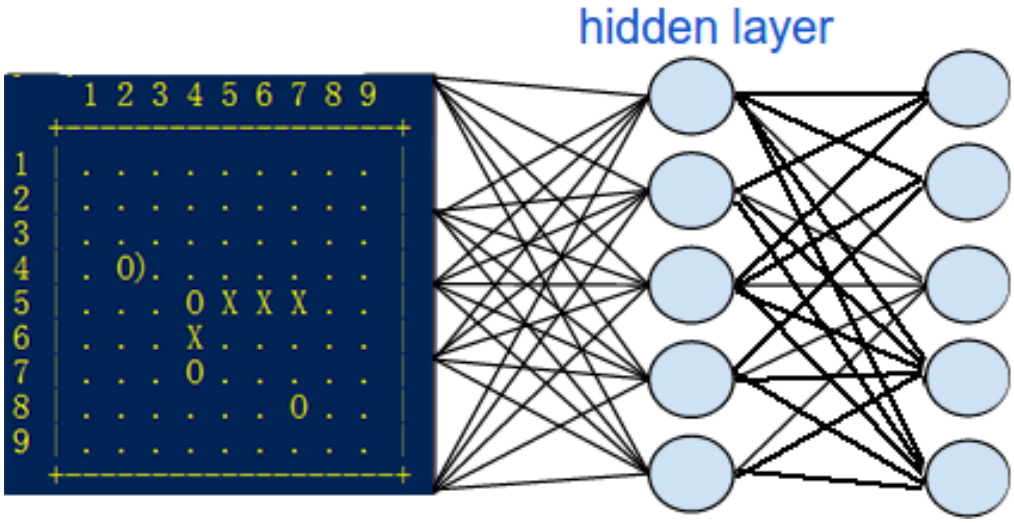
EXPLORATORY VISUALIZATION

Since we don't have any external data, I will show a typical board rendering.



ALGORITHMS AND TECHNIQUES

The algorithm that I will use in this project is called Policy Gradient. In a reinforcement learning problem, there are different approaches to the problem. Most algorithms will try to learn either the value function $V(s)$, the state-action function $Q(s,a)$ or the policy function $\pi(a|s)$. In Policy Gradient method, we try to learn the policy directly. The policy function is represented by a neural network:



By training this neural net, we try to get a good approximation to the optimal policy. Policy gradient methods rely upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. For a more thorough discussion of Policy Gradient, see [4].

Below is a list of the hyper parameters used in this project and how they affect the training of the agent:

1. `H`: The policy network is a fully connected neural net with a single hidden layer. This parameter specifies how many hidden neurons there are. I started at 200 but

changed it to 500. With 500 hidden neurons, I think the neural should have enough power and complexity to recognize the different patterns.

2. `batch_size`: This specifies how many episodes to run before we calculate the gradients. I want to batch up the gradient calculation to make the calculation more accurate. I set this to 10 in this project.
3. `update_per_batch`: This specifies how many batches of calculation before we start a weight update. This is also set to 10. I have tweaked this a bit but due to time constraint, not able to tell which is a good value.
4. `learning_rate`: learning rate basically specifies how much we update the weights each time. I have set this to 10^{-4}, it will take a bit longer to train but it should be small enough to not miss the optimum.
5. `gamma(`γ`)`: This is the discount factor for reward. This stays at 0.99 during the entire development.
6. `decay_rate:` This is the decay factor for RMSProp leaky sum of grad^2. [7]

Here are the steps of the algorithm:

1. Initialize the weights randomly by the Xavier initialization algorithm.
2. For each step of action, we save the input, and the hidden states, we also save the gradient for the output. The gradient is basically the action that we took minus the calculated probabilities. When the episode is finished, if the reward is positive, then we will use the positive discounted reward to multiply our gradient. If the reward is negative, we will use the negative discounted reward to multiply our gradient. So actions that lead to win will get positively promoted and actions that lead to lose will be suppressed.
3. When we reach the `batch_size` of episodes, we calculate the batch of gradients.
4. When we reach the `update_per_batch` of batches, we perform the weight update.
5. And we just repeat the whole process again.

For the activation function on the hidden layer, I used Rectified Linear Unit, and haven't got a chance to test other activation functions. For the output layer, I just used a softmax function.

I will talk about how we decide when to stop training in later sections.

BENCHMARK

For benchmark purpose, I maintain a couple variables in the algorithm above.

1. `reward_sum`: For each batch of episodes, this will simply add up all the rewards. If our `batch_size` is 10, then the maximum possible value for `reward_sum` is 10 and the minimum will be -10. It will be reset to 0.
2. `running_reward:` This is like the mean rewards accumulated so far. For the initial batch, it's just equal to the `reward_sum`. After each batch, it's recalculated by the formula:

```
running_reward=0.99*running_reward+0.01*reward_sum
```

This basically discounts the old `reward_sums` continually, and when new `reward_sum` comes in, it takes more precedence. Here is a more detailed look at how this is calculated. Suppose our episode batches has rewards $r_0, r_1, r_2, \dots r_n$, where $r_n$is the latest batch, then our `running_reward` would be: $0.01(r_n + 0.99r_{n-1} + 0.99^2 r_{n-2} + 0.99^3 r_{n-3} + \cdots + 0.99^n r_0)$. We can see that this benchmark is a good indication for how well our agent is doing against the opponent. If we are in a 10 batch update, then a `running_reward` of around 5 would mean the agent is consistently scoring a 6 or above in each batch of episodes, which would mean beating the opponent at $8:2, 9:1$ *or* $10:0$.

After some experimenting, I've settled on the value 4 as a good indication that the agent is consistently winning against the opponent.

When training the agent using self-play, I also set the threshold to copy the model at 4. i.e., whenever the current agent has a `running_reward` of 4 or above, I use the current model for the opponent and reset the `running_reward`.

III. Methodology

DATA PREPROCESSING

Since I don't have external data, there is not much preprocess done. But I do preprocess the state before feeding it into the policy network, since it just takes a vector. I basically use the black position matrix and subtract the white position matrix, and then convert the whole thing into a single vector. So in this vector, a 1 indicates a black stone, a -1 indicates a white stone and 0 means empty.

One interesting problem is how I should preprocess the input if the color of the player is different. Since I trained the agent always as the black player, when I do self-play, I need to use the same model for the white player also. One way to think about this is that since Gomoku is kind of a symmetric game, if a position is good for the black player, then it should be good for the white player, so I shouldn't need to do anything extra. And it kind of works ok. But after some testing, it turns out to be better to reverse the data, i.e., use white position matric to subtract the black position matrix. This turns out to be better.

IMPLEMENTATION

The implementation is quite a bit more work than I imagined. The first hurdle was to implement the Gomoku environment. Even though it's not a difficult task, but to make the environment bug free and well tested took some effort and some bugs were quite subtle. For example, the environment checks whether the agent won or lost after the agent move. Then it will generate the opponent move and pass back the state. But it didn't

check for whether the opponent has won. The checking of whether a board is in a finished state could be quite expensive if we check all possible rows, columns, diagonals. I finally settled on a regular expression based solution.

Due to these early bugs in the code, the initial data that was run were quite wasted.

I also was too ambitious at the beginning, and implemented an AI opponent to train the agent. This turns out to be a pretty bad idea, since the AI opponent does a tree search, and the performance is quite slow. It takes around a second for the AI to make a move, this makes training extremely slow. Also, the agent never wins against the AI, so basically every action is a bad action, and after a whole night's training, things didn't improve at all. I decided to try something more naïve to start with. The random opponent was actually quite easy to beat, and it doesn't really mean much if my agent can only beat the random opponent. So I used some common well known strategies to code up a naïve agent, and with different levels of search for moves. This turns out to be a quite good opponent to train with, since it's really fast and has reasonable difficulty.

I also tried to train the agent to by self-play, and this run into a problem at the beginning. My agent was able to beat the initial model, but then would be having a `running_award` of around 0 and never improve. This really baffled me and not sure where the bug was. Only until much later that I realized when I tried to copy the new model into the opponent, I used the `copy()` method on a dictionary, and this didn't make a copy of the numpy matrix inside the dictionary, so essentially the model is being shared between the agent and the opponent.

Implementing the neural network and gradient calculation is also not trouble free. At one time, I tried to use Torch and Tensorflow instead of coding my own network. But finally decided to stick with my own network, this way, I can learn more about how neural network works.

REFINEMENT

The code base is continually improving. The following is a list of what I have done to improve the program and the algorithm, roughly in a chronological order.

- Added human play to test out how good is the agent.
- Added the naïve opponent to train the agent against.
- Changed how often the gradient is calculated. At the beginning, it's calculated for every episode. And since each episode the reward is either 1 or -1, there is no averaging of the win episodes and lose episodes, the calculation was not accurate enough. Later I have settled on a batch size of 10, i.e., calculate the gradient for every 10 episodes.
- Added an update batch size, how many batches to calculate before we update the weights in the network. I varied a bit between 5 and 10, I haven't got the time to see the difference, but settled it to be 10.

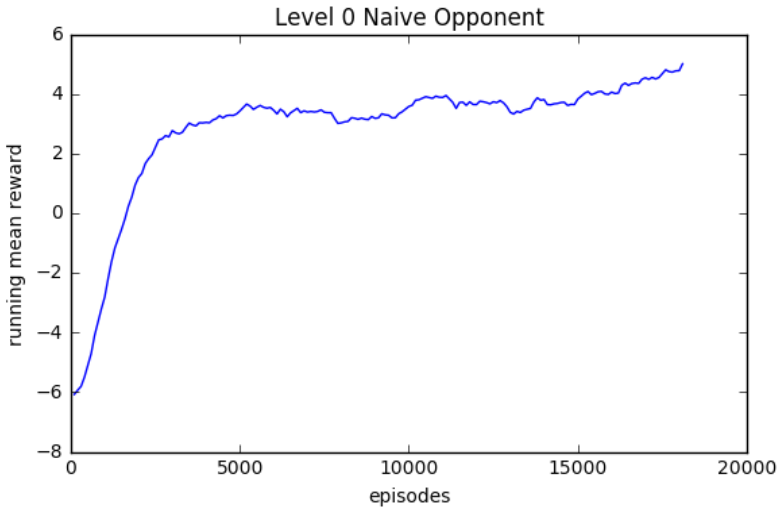
- The number of hidden neurons at the beginning is 200. The speed of training was faster, but the agent wasn't able to converge to a solution given the training metric. I changed to 500 and stayed with this for the rest of the project.
- The learning rate was tested between 10^{-3} and 10^{-4}. It trains a lot faster when the learning rate is 10^{-3}, and it's quite fast for the agent to beat the random opponent. But the agent also run into a lot of dead neurons after an extended training session, i.e., the output layer outputs 0 probabilities, and the agent essentially became random again. So I decided to keep 10^{-4} and the agent trains a lot better for extended time.
- After beating the Level 3 naïve opponent, I tried again to train by self-play and see if I can get a better result. But after some extended training, even though the agent seems improving by beating it's own model, but it lost to the level 3 naïve opponent again, and took quite a long time to get back it's abilities after using the level 3 as opponent again. So having a good opponent is definitely very important for training the agent, and pure self-play is hard to reach a global optimum without extra techniques.

IV. Results

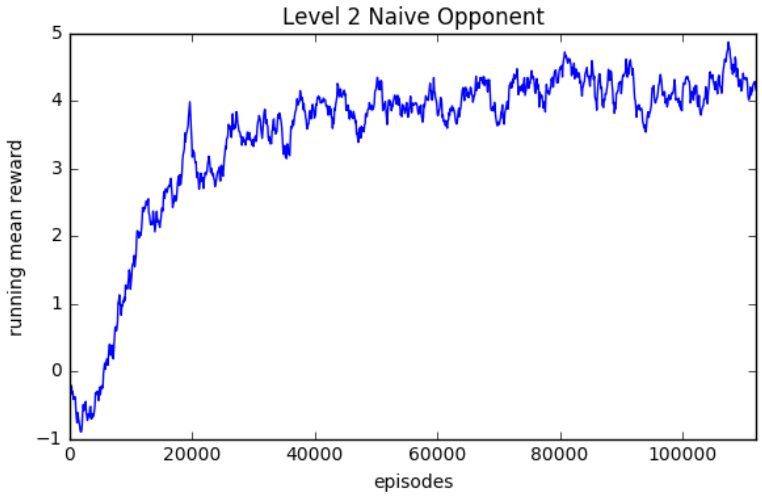
MODEL EVALUATION AND VALIDATION

I first tried to train the agent against the random opponent, but the agent is not limited to valid moves. I was hoping the agent could learn how to make valid moves. But after a whole night's training, the agent is has not learnt the valid moves. I think one reason is that since the entire game could last maximum of 81 moves, the agent has to learn how to make a valid move on every step of the game, one step wrong would lose the game, and this sounds like will take really long to do. So I decided to restrict to valid moves for the agent only.

I then trained the agent against the level 0 naïve opponent, and it's relatively easy to achieve a running mean reward of 5.0, and it only took around 18000 episodes.

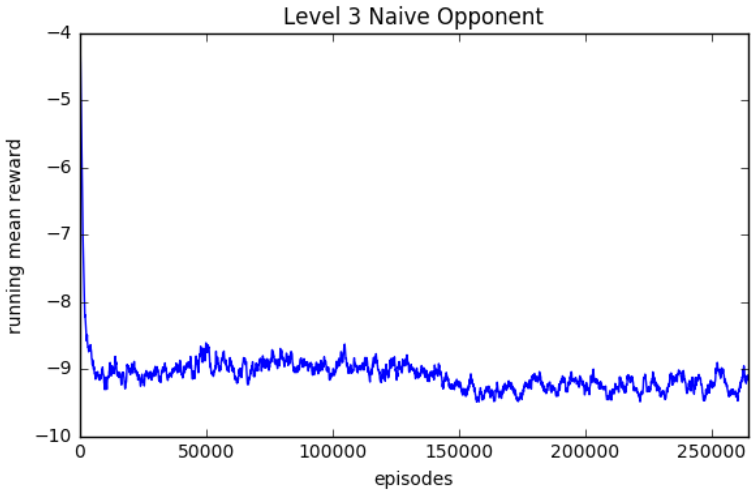


I skipped the level 1 naïve opponent and directly went for the level 2 naïve opponent. This opponent will actively extend any connected 2 or more stones if there exists. Here is the training plot for that:



From the plot, it's quite easy to reach a running mean reward of 4.0, but I was trying to reach 5.0, and it kind of fluctuates and never quite get there.

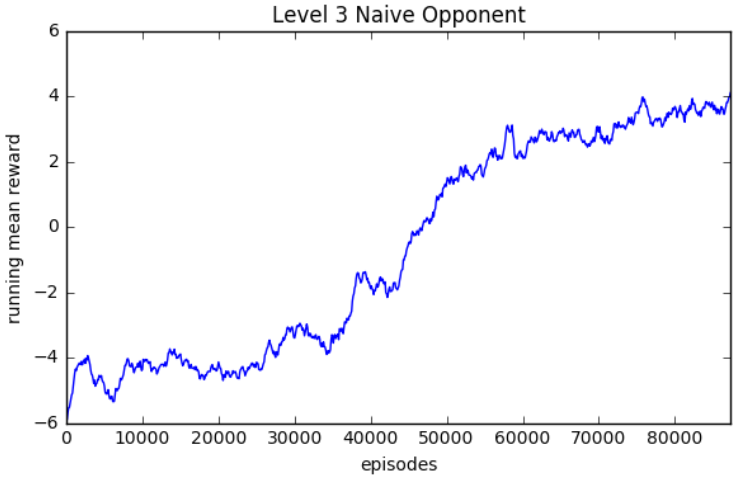
Next I tried to train the agent against the level 3 naïve opponent, which will actively extend any stone on the board, and here is the result:



After over a quarter million episodes, which took the entire night, the agent is not really improving at all. I am not sure if this needs more time to run or it's stuck in a local maximum and can't get out.

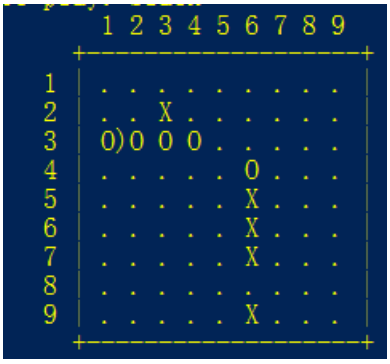
I decided to use self-play to train. After one million episodes, I decided to test it and see how good it is. It can only defeat the level 0 naïve opponent, and lost to level 1 and above. But I do notice something interesting, when I tried to play with it, the agent sort of acquired some 'common sense' about the game, for example, it knows that it should play the stone next to mine, and it sort of knows to connect the stones together, it just never quite able to connect 5 of them.

I then used this model as a basis and trained it against the level 3 naïve opponent, and here is the training plot:



After only 80000 episodes, the agent is able to defeat the level 3 opponent consistently. I think this is quite amazing. I decided to take a closer look at a typical game. I found out that the agent was winning the opponent by exploiting a hole in the opponent's algorithm.

Since the opponent will only block any 4 stones connected together or any open 3 stones connected together, the agent has learnt to create a situation like the following:



The 'X' is the agent and 'O' is the opponent. Notice that the agent created a hole at position (8, 6). Thus fooling the opponent not to block, and finally placing a stone at (8, 6) for the win.

During testing, the Agent plays black and has achieved a win ratio of 72.5%.

JUSTIFICATION

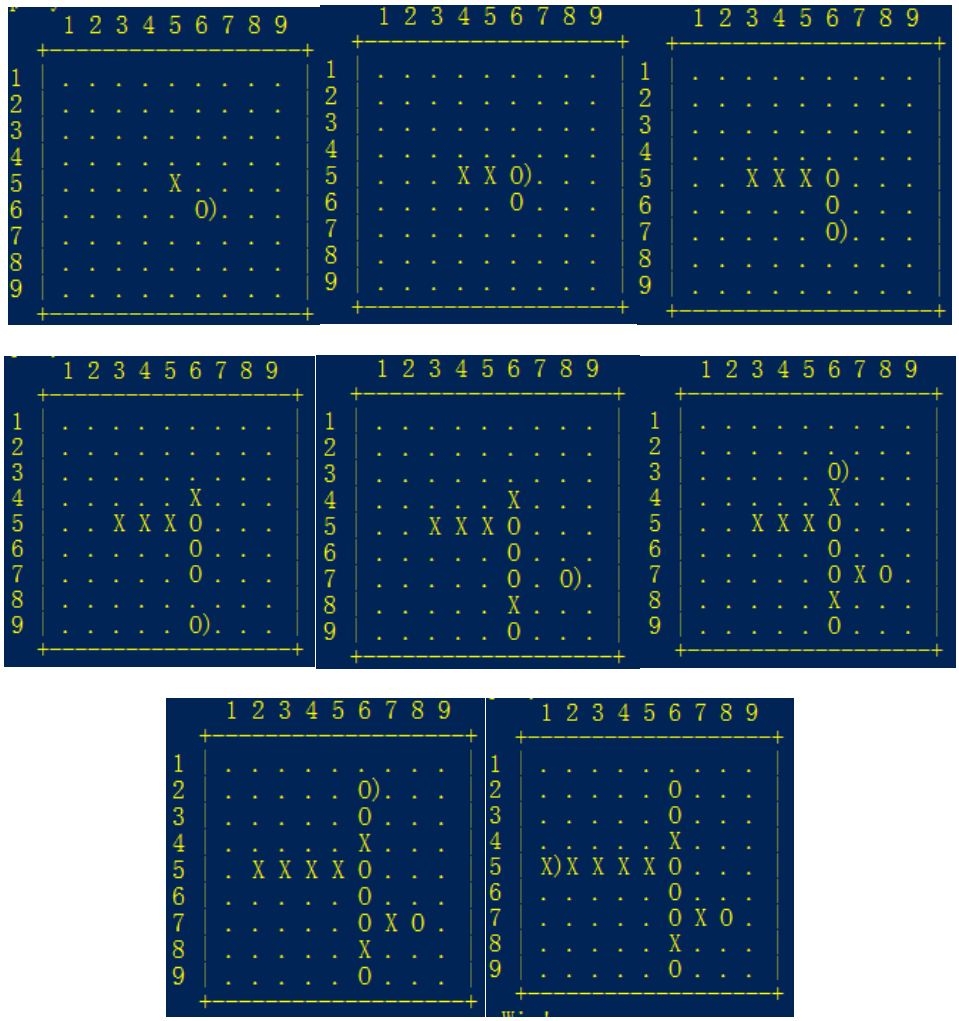
Using a neural network to train is not easy, especially debugging why the model is not converging. I believe if I make the opponent smarter, the agent should learn as well. I think by using self-play, it somehow positioned the model better for it to reach a higher maxima. Even though this agent is still not close to human playable, but it does acquire quite a lot of 'intelligence' in the game of Gomoku considering that nothing is programmed into it and everything is learned.

In summary, using policy gradient should be a viable method for training the Gomoku agent, even though considerable more tweaking is needed for it to become a better player. Maybe something like a value network needed to be added in.

V. Conclusion

FREE FORM VISUALIZATION

Here I show a game that I played with the agent. I try not to be too smart and make some stupid moves. (I am 'X' and agent is 'O')



It's clear that the agent is still quite naïve.

REFLECTION

When I was searching for a capstone project, I wanted to do something related to reinforcement learning, and it seems that playing board game is a perfect area for reinforcement learning. Also encouraged by the results of DeepMind's Atari playing agent and AlphaGo, I decided to work on this Gomoku agent. The whole process turns out a lot harder than I imagined, I highly under estimated the time it would require to tune and debug neural networks. But at the end, I think a somewhat satisfactory result is obtained, even though it's kind of far from my expectations initially. I think the hardest part is how to avoid the network to be in a local maximum, and not able to learn from there.

Even though there is a lot of work, but seeing the agent able to learn and make progress is also quite satisfying, I think I have learnt a lot about how neural network works internally now.

In brief summary, here is what I have done for the project:

- Implemented a Gomoku environment based on OpenAI Gym.
- Implemented a neural network that can perform Policy Gradient.
- Imported a Classic AI opponent.
- Implemented a naïve opponent for training.
- Implemented interactions with human and benchmark to test agent performance.

IMPROVEMENT

There are definitely a lot of ways I can think that should be able to improve the agent's performance.

- Use a better opponent, get rid of the loop holes that the agent was exploiting.
- Convert to a Tensorflow/Keras solution, and use GPU acceleration to speed up training.
- Use some other deep reinforcement learning methods, use policy gradient with actor-critic methods.
- Add a value network.
- Train a white player model separately from the default black model.

Bibliography

[1] https://en.wikipedia.org/wiki/Gomoku

[2] https://arxiv.org/abs/1312.5602

[3] http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[4] http://karpathy.github.io/2016/05/31/rl/

[5] https://github.com/Nero144/Gomoku

[6] https://gym.openai.com/

[7] http://sebastianruder.com/optimizing-gradient-descent/index.html#rmsprop