Project #3

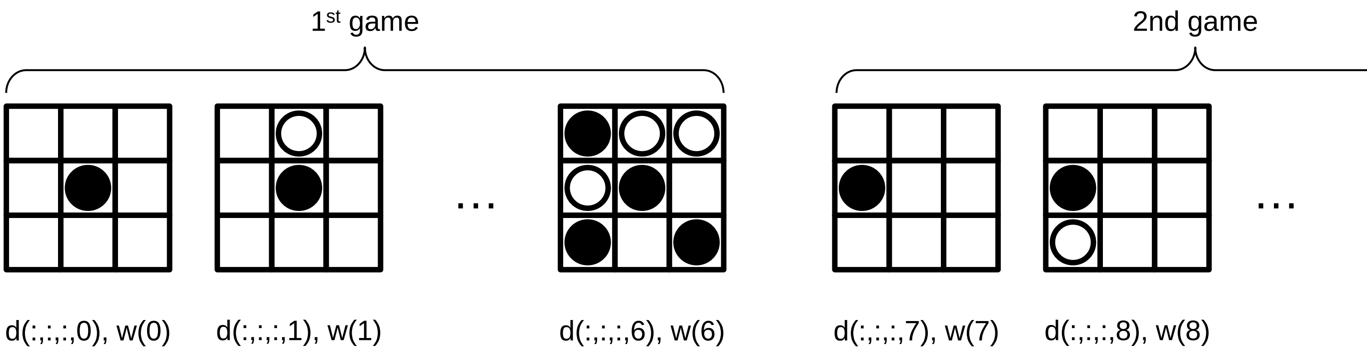
EE488C Special Topics in EE <Deep Learning & AlphaGo>, 2016 Fall
Information Theory & Machine Learning Lab., School of EE, KAIST

- Contents
  - Training a simple neural network playing tic-tac-toe
  - Mini AlphaGo

- Due date
  - 11:59pm on Wednesday December 28
  - Due to hard constraint on grading, the deadline will be NOT be extended.

- Experiment #1 – Training a simple neural network playing tic-tac-toe
  - For this, you need tictactoe.py, game.py and strategy.py, which are in project3.zip.
    - tictactoe.py: This trains value networks to play tic-tac-toe and test them. We don’t use a policy network for tic-tac-toe.
    - game.py: This specifies the game environments. There are 4 games defined in game.py, i.e., game1 is a simple version of go mentioned in class (page 20 of Lecture notes #24), game2 is tic-tac-toe, game3 is 9x9 Omok (5 in a row), and game4 is Othello. In tictactoe.py, we import game.py and we set ‘game=game2()’ to choose game 2, which is tic-tac-toe.
    - strategy.py: This contains the ‘next_move’ class, which determines the next move. Only the function ‘val’ in the class is needed for tic-tac-toe, which receives the current board state as its input and outputs the next action of the player by using a value network and 1-level tree search.
  - Run tictactoe.py. It will train and test value networks that can play tic-tac-toe.
  - Two value networks are generated after the code tictactoe.py fishes running. The first network is the first-generation value network that is trained to predict the winner using games played between two players who play purely randomly. The second network is the second-generation value network that is trained to predict the winner using games played between two players based on the previous-generation value network plus some randomness. Value networks in tic-tac-toe have 2 convolutional layers and 2 fully connected layers. The input layer takes the current board state as its input and the output layer produces 3 output values, i.e., 3 softmax values for ‘black win’, ‘white win’, and ‘tie’. The first convolutional layer is defined by ‘tf.nn.conv2d(state, weights1, strides=[1,1,1,1], padding=‘SAME’)’. ‘padding’ is set to be ‘SAME’, which means the output has the same size as the input by applying zero padding at the input.

- In the first training phase, i.e., generation=0, ‘[d,w]=game.play_games([], [], r1, [], [], r2, n_train, nargout=2)’ is used to produce game records between two players playing randomly. ‘n_train=10,000’ games are played in the first training phase. The first and the forth input arguments of the function ‘play_games’ specify the value networks for the first (black) and the second (white) players, respectively. But, they are specified as ‘[]’ as seen above since we don’t have any value networks initially and we will use pure random plays. The second and fifth input arguments specify the policy networks for the first and the second players, respectively. Since we don’t use policy networks in tic-tac-toe, they are also set to ‘[]’. The first and second player’s randomness is specified by ‘r1’ and ‘r2’, respectively. They are both vectors of length ‘n_train’. If an element in r1 is 1, then it means the first player’s randomness is 1 for the corresponding game, i.e., purely random actions. ‘nargout=2’ means the number of outputs of this function is 2. Later we will explain the case when an element of ‘r1’ or ‘r2’ is not equal to 1.
    - For example, a game may look like the following. We assume black always plays first. The game ends when the winner is decided or the board is full. In the example below, black wins the first game after 7 moves. These 7 board states are marked as ‘black win’. These data are returned in ‘d’ and ‘w’ when you do ‘[d,w]=game.play_games([], [], r1, [], [], r2, n_train, nargout=2)’, i.e., the first 7 entries in ‘d’ contains the first 7 board states below and the first 7 entries in ‘w’ contains information that ‘black win’. The next game data is stored in subsequent entries in ‘d’ and ‘w’ as shown below.



    - In ‘[d,w]=game.play_games([], [], r1, [], [], r2, n_train, nargout=2)’, ‘d’ is a 4-dimensional matrix of size 3*3*3*n, where ‘n’ is the number of board states, which is about 7~8 times the number of games played since there are usually about 7~8 moves on average per game. For each board state, ‘d’ is 3*3*3 since the board size is 3*3 and for each position there are 3 states, i.e., empty, black or white. The state for each position is encoded as a one-hot vector. For example, for the k-th board state
        - d(0,0,0,k)=1, d(0,0,1,k)=0, d(0,0,2,k)=0 means board position (0,0) is black
        - d(0,0,0,k)=0, d(0,0,1,k)=1, d(0,0,2,k)=0 means board position (0,0) is white
        - d(0,0,0,k)=0, d(0,0,1,k)=0, d(0,0,2,k)=1 means board position (0,0) is empty
    - It is possible to use a denser coding for representing the board positions, e.g., d(0,0,0,k)=1 means black, d(0,0,0,k)=0 means empty, and d(0,0,0,k)=-1 means white. Then, the dimension of ‘d’ can be reduced to 3*3*1*n from 3*3*3*n. However, using the one-hot vector strategy used in AlphaGo, training can become easier since separate weights are assigned for separate board state (black, white, empty).
    - ‘w’ is a vector of length ‘n’, which contains the winner information for each state, i.e., 0 means a tie, 1 means black won, and 2 means white won. In the example above, w(0)=w(1)=…=w(6)=1, which means black won the first game. Note that the value of ‘w’ is the same for all board states belonging to the same game.

- After ‘d’ and ‘w’ are generated, ‘[d,w]=data_augmentation(d,w,[])’ is performed. Since the game result is invariant to rotations and reflections, we can perform such operations to enlarge the dataset without explicitly playing more games. After data augmentation, the sizes of ‘d’ and ‘w’ become 8 times larger, i.e., the size of ‘d’ becomes 3*3*3*(8*n) and the size of ‘w’ becomes 8*n. The last input argument of data_augmentation should be empty for tic-tac-toe. After data augmentation, shuffling is done to randomize the order.
- After data augmentation and shuffling, we split the dataset into mini-batches with size ‘size_minibatch’ each and train neural networks. ‘sess.run(optimizer, feed_dict=feed_dict)’ is performed to train the neural network. ‘d’ is the input of the neural network and ‘w’ is the desired output, i.e., the value network is trained to produce who is likely to win (or tie) when a board state is given. During training, the following will happen.
    - Random moves that are neutral will tend to be diluted and will not contribute much to the value network’s parameters.
    - However, some random moves that are either advantageous or disadvantageous will contribute to the value network’s parameters and the value network will learn that some moves are good and some moves are bad.
    - As a result, the resulting value network will be able to play better than a player who plays purely randomly.
- Then, ‘game.play_games(P, [], r1, [], [], r2, n_test, nargout=1)’ is performed to evaluate the trained value network ‘P’ by making it play against a player that plays randomly (‘r2’ is an all-one vector). ‘P’ plays black, i.e., it plays first, with no randomness (r1 is an all-zero vector). ‘n_test’ games are played. Then, ‘game.play_games([], [], r1, P, [], r2, n_test, nargout=1)’ is performed to evaluate the network when it plays white against a player that plays randomly, where r1 is now an all-zero vector and r2 is an all-one vector.

- In the second training phase, i.e., generation=1, '[d,w]=game.play_games(P, [], r1, P, [], r2, n_train, nargout=2)' is performed. It collects game data between two players, where both are using the previous-generation value network 'P'. A total of 'n_train=25,000' games are played in the second training. Then, data augmentation is performed as before and training the second-generation value network is performed. Finally, the performance of the trained network is evaluated against a player who plays randomly.
    - 'r1' and 'r2' control randomness in the moves by the two players. If there is no randomness, i.e., if both r1 and r2 are zero, then the game plays will become deterministic and only one game record can be obtained when we do '[d,w]=game.play_games(P, [], r1, P, [], r2, n_train, nargout=2)'. Then the data in '[d,w]' is almost useless as a training set due to lack of diversity, i.e., 'd' and 'w' will contain 'n_train' copies of the same data. 'r1' and 'r2' are real vectors of length 'n_train'. In the second training phase, each entry in 'r1' and 'r2' is randomly independently generated as follows. With probability 50%, it is generated as a random real number between 0 and 1 (uniform distribution between 0 and 1), which tells the function 'play_games' to randomize a move with the specified probability. With probability 50%, it is generated as a random integer '-m' between -1 and –mt (uniform distribution), where mt=floor(game.nx*game.ny/2) is roughly the maximum number of moves per player, which tells the function 'play_games' to randomize the first 'm' moves. Therefore, in the second training phase, for 50% of the games 'play_games' will randomize a move with probability between 0 and 1 and for the other 50% of the games 'play_games' will randomize the first 'm' moves, where m is random between 1 and mt.
    - Unlike the first training phase, these games are not entirely randomly played. There will be many smart moves since we are using the first-generation value network to generate training examples. The second-generation value network will be able to learn from those smart moves in addition to random moves. As a result, the resulting second-generation value network is likely to be able to play better than the first-generation value network.
- If you perform more training phases, i.e., if you increase the maximum number of training phases to run, then you may be able to get even stronger value networks. However, the performance tends to saturate after about 3 or 4 phases and the performance may even become worse. To do better, you may want to change the randomness in 'r1' and 'r2' or come up with a better strategy for generation=2,3,….
- You may want to change some training parameters such as learning rate, mini-batch size, maximum epochs, and momentum for better performance.
- When you run tictactoe.py, it will take about 5~10 minutes on a GTX 960 GPU. Server computers have GTX 960 GPU's.

- You will get something like the following when you run tictactoe.py. Since there are many sources of randomness, you will get a different result every time you run it. For example, ‘r1’ and ‘r2’ are random and weights are initialized randomly for neural networks.

```
>> python tictactoe.py
Generation:  0
Data augmentation step ...
Done!
Epoch:  0       | Iteration:  50        | Loss:  1.07223
Epoch:  0       | Iteration:  100       | Loss:  1.03869
Epoch:  0       | Iteration:  150       | Loss:  0.923943
                                     ⋮
Evaluating generation  0 neural network
 net plays black: win= 0.9892   lose= 0.0       tie= 0.0108
 net plays white: win= 0.9017   lose= 0.0213    tie= 0.077
                                     ⋮
Evaluating generation  1 neural network
 net plays black: win= 0.9894   lose= 0.0       tie= 0.0106
 net plays white: win= 0.89832  lose= 0.0       tie= 0.10168
```

- You can see the first-generation value network (generation 0) is already winning most of the time against a player who plays purely randomly.
- The second-generation value network (generation 1) performs better than the first-generation value network. In the above example, it did not lose any of the 25,000 games against a player that plays pure randomly. In this case, the second-generation value network may be playing perfectly, i.e., never loses. In tic-tac-toe, the result will always be a tie if both players play perfectly.
- After each training phase, the parameters of the trained value networks are saved in a file by using the save function in the Saver class. In tictactoe.py, the parameters of the first and second generation value networks are saved in the files ‘tictactoe_generation_0.ckpt’ and ‘tictactoe_generation_1.ckpt’, respectively. We can later load the parameters in python by using the restore function.
- To play interactive games with a value network, do the following:

```
>> python tictactoe_interactive.py ./tictactoe_generation_1.ckpt
```

  - ‘tictactoe_generation_1.ckpt’ is a check point file containing two value networks trained in the second training phase, one playing black and the other playing white.
  - This program ‘tictactoe_interactive.py’ defines the neural network structure in the same way as done in ‘tictactoe.py’ and loads the check point file ‘tictactoe_generation_1.ckpt’ and performs an interactive session to play between a human and a computer.
  - In the last line of ‘tictactoe_interactive.py’, you can find ‘game.play_interactive(P,[],0,[],[],0)’, which starts an interactive game session between a value network ‘P’ and a human player.
- Deep learning can give us a neural network that can play tic-tac-toe perfectly within minutes of training using a cheap GPU without any explicit instruction about how to play the game. This is a lot more efficient than inventing an algorithm and writing software codes line by line by a human. Furthermore, as seen in AlphaGo, even the most complicated algorithms invented by humans and extensive software codes written by humans are no match for deep neural networks.
- For tic-tac-toe, you don’t need to submit anything. But, you may want to experiment with the code ‘tictactoe.py’ since the codes for mini AlphaGo are based on it and it is easier to understand.

- Experiment #2 – Mini AlphaGo
  - For this, we need go_train_value.py, go_train_SL_policy.py, go_train_RL_policy.py, game.py, and strategy.py from project3.zip.
    - go_train_value.py: This trains value networks by supervised learning.
    - go_train_SL_policy.py: This trains policy networks by supervised learning
    - go_train_RL_policy.py: This trains policy networks by reinforcement learning
    - game.py: This specifies the game environments.
    - strategy.py: This contains the ‘next_move’ class, which determines the next move. The function ‘val’ produces the next move by using value networks and full tree search with a maximum depth specified by ‘nlevels’, the function ‘pol’ produces the next move by using policy networks and a full tree search with depth=1. The function ‘val_and_pol’ is a function that you need to complete. In the function, you may use only value networks, only policy networks, or both value and policy networks. Currently, the functions ‘val’ and ‘pol’ use full tree search. You may want to implement MCTS in your function ‘val_and_pol’. We will explain more detail in Task #1.
  - First, we explain how mini AlphaGo works and how its neural networks are trained. Refer to page 20 in Lecture notes #24 for details on simplified Go rules for mini AlphaGo.
  - There are some differences between tictactoe.py that trains value networks for tic-tac-toe and go_*.py that trains value and policy networks for mini AlphaGo. In mini AlphaGo, we do the following.
    - ‘game=game1()’ is used to specify that the game is ‘simple go’ with board size 5*5.
    - The neural network now has 3 convolutional layers and 2 fully connected layers, i.e., one more convolutional layer than the one in tictactoe.py since the game of Go is more complicated.
    - In tictactoe.py, we trained a single value network in each training phase. In tic-tac-toe, you can see who the next player is by simply counting the number of stones. But, in the game of Go, you need a separate input that can tell us who the next player is since you cannot figure out the next player by simply looking at the current board state because stones can be captured. In AlphaGo, the current player’s color is given as an extra input to the value network. Instead of providing such an extra input, we simply train two separate value networks in each training phase in mini AlphaGo – one playing black (value_network0) and the other playing white (value_network1). This way, we don’t need to make any architectural changes for the neural network. For policy networks, we also train two separate networks in each training phase, one playing black (policy_network0) and the other playing white (policy_network1). In AlphaGo, there’s only one policy network since its inputs and outputs are from the current player’s perspective. In our case, we simply use two separate networks to make things more explicit.
    - All these networks in mini AlphaGo, i.e., value and policy networks playing black and white, take the current board state as input. The output of a value network has 3 softmax units as in ‘tictactoe.py’. The output of a policy network has 25 softmax units, one per board position.

- Now, we show how to train 3 types of networks for mini AlphaGo, value network, SL policy network, and RL policy network.
- Training step 1: Training the value network by supervised learning (go_train_value.py)
  - First, we train value networks by supervised learning. This part is similar to what we did in 'tictactoe.py'. We first generate initial training data played between two players that play purely randomly and then train two value networks, one for playing black and the other for playing white. The following line in 'go_train_value.py' lets two players play against each other, where both players take purely random actions.

```
[d1,w1,wp1,d2,w2,wp2] = game.play_games([], [], r_all, [], [], r_all, n_train, nargout = 6)
```

  - The first three input arguments of the function 'play_games' are the value network, the policy network, and randomness of the first player (black). The next three parameters are the value network, the policy network, and randomness of the second player (white). 'r_all' is an all-one vector, which means actions are purely random. 'n_train' is the number of games to play and 'nargout' is the number of output arguments this function should produce.
  - Two sets of training data are obtained when you run 'play_games', i.e., one for training the value network for black and the other for training the value network for white.
  - The first three output arguments of 'play_games' are the board states 'd1', winner information 'w1', and estimated probabilities of winning 'wp1' when the last action was taken by black, which will be later used for training the value network for black. The next three output arguments of 'play_games' are the board states 'd2', winner information 'w2', and estimated probabilities of winning 'wp2' when the last action was taken by white, which will be used later for training the value network for white. 'wp1' and 'wp2' are estimated winning probabilities estimated by value networks. But since we are not using value networks for generating the training data in training phase 1, they are all-zero vectors. They are used for printing out estimated probabilities of winning in the function 'play_interactive' if value networks are used.
  - In the next training phase, we generate the training data by letting the two value networks obtained in the first training phase – 'V1' playing black and 'V2' playing white (see box below) - play against each other many times. Since the function 'val' is using a simple tree search with greedy action selection, the games played between the two value networks will always be the same, which is not good for generating the training data for training the next version of value networks. Therefore, we randomize some actions as done in 'tictactoe.py', i.e., for 50% of the games 'play_games' will randomize a move with probability between 0 and 1 and for the other 50% of the games 'play_games' will randomize the first 'm' moves, where m is a uniform random number between 1 and mt=floor(5*5/2)=12.

```
[d1, w1, wp1, d2, w2, wp2] = game.play_games(V1,[],r1,V2,[],r2,n_train,nargout = 6)
```

- Training step 2: Training the policy network by supervised learning (go_train_SL_policy.py)
  - This part is for training policy networks using supervised learning. In AlphaGo, SL policy network was trained using game records of human experts. Since we don’t have such data for 5x5 Go, we will use the value networks trained in training step 1 to train the SL policy networks as mentioned in class.
  - If we let two value networks play games against each other, then all the games will be played the same way since the function ‘val’ takes greedy actions. Therefore, we cannot use such game records for training policy networks due to lack of diversity. We randomize some actions as done before, i.e., we do the following to obtain game records.

```
[d1,w1,wp1,pos1,d2,w2,wp2,pos2] = game.play_games(V1,[],r1,V2,[],r2,n_train,nargout=8)
```

    - ‘r1’ and ‘r2’ are chosen to randomize some actions as before. ‘V1’ and ‘V2’ are value networks obtained in training step 1 playing black and white, respectively.
    - ‘d1’, ‘w1’, ‘wp1’, ‘d2’, ‘w2’, ‘wp2’ mean the same things as before.
    - We have two new outputs, namely ‘pos1’ and ‘pos2’. The function ‘play_games’ now returns these two additional outputs because ‘nargout’ is set to 8 as shown above. ‘pos1’ is the next action by white given the board state ‘d1’ and ‘pos2’ is the next action by black given the board state ‘d2’. Remember that ‘d1’ is the board state after black’s move and ‘d2’ is the board state after white’s move. Therefore, ‘d1’ and ‘pos1’ can be used to train an SL policy network for white since ‘d1’ is the board state after black’s move and ‘pos1’ is the white’s next move. Similarly, ‘d2’ and ‘pos2’ can be used to train an SL policy network for black. The very first move by black can be trained by using the first entry in ‘d1’ for each game since ‘d1’ does not return an empty board state and the first entry in ‘d1’ already contains the first move by black. Likewise, the last entry in ‘d1’ for each game is not used for training since it does not contain the next move.
    - Since there are many random actions in the resulting game records, they are not actually good for training SL policy networks due to low quality of games. Since we don’t have other choice, we use this simple training strategy for the initial training SL policy networks. But, at least this is better than training SL policy networks using pure random plays since there’s nothing to learn from pure random plays.
    - Since the quality of games are not so good due to randomness introduced by ‘r1’ and ‘r2’, we only use the winning games to train the SL policy networks, i.e., we use the game records where black is the winner for training SL policy network for black and likewise we use the game records where white is the winner for training SL policy network for white. Selecting only the winning games is done as follows:

```
temp_indices = [idx for idx in indices if w[i][idx] == 2-i]
```

    - Once you train SL policy networks, you may want to train the next generation SL policy networks by using games played between the previous SL policy networks. Since the outputs of an SL policy network are estimated probabilities for the next move, we can take probabilistic actions based on the probabilities and as a result many different game records can be produced.
  - The neural network structure for the policy network specified in ‘def policy_network(state, nx, ny)’ is different from that of the value network specified in ‘def value_network(state, nx, ny)’. Namely, there are no fully connected layers in the policy network to preserve the topology of the Go board and there is an additional convolutional layer with a 1x1x70 convolutional filter. Since the convolutional filter is 1x1 horizontally and vertically, it is simply an inner product of length 70 in the third dimension. Furthermore, the bias is a 5x5 matrix instead of a scalar for the last 1x1x70 convolutional layer. In AlphaGo’s last convolutional layer with a 1x1x192 convolutional filter, the bias is also a 19x19 matrix. This adds more degrees of freedom.

- Training step 3: Training policy networks by reinforcement learning (go_train_RL_policy.py)
  - In this step, we use reinforcement learning to improve the SL policy networks obtained in training step 2. We will keep a pool of policy networks, randomly sample policy networks from that pool, and generate game data to train the RL policy network. Then, we add the updated RL policy network to the pool and repeat the process many times.
  - As mentioned in training step 2, the quality of training examples used to train the SL policy networks are not very good. Therefore, to train the RL policy networks we will not only choose an opponent from the current pool of opponent policy networks but also use the value network as the opponent from time to time for improving the quality of the games. We can vary the fraction of the games where the value network is used as an opponent to see how it affects the outcome.
  - For generating the training data using a policy network as an opponent, we take actions randomly according to the output probabilities of the policy network, which is the default behavior of the function 'play_games', where 'policy_type' is set to 0 by default. If you want the policy network to take greedy actions, then set 'policy_type' to 1 when calling 'play_games'. For the RL policy network that is being optimized will also take actions randomly according to its output probabilities. Therefore, many different game records will be produced.
  - If a value network is used as an opponent, then we take greedy actions, which may affect the diversity of games due to its deterministic behavior. But, since the RL policy network that is being optimized will take actions randomly according to its output probabilities, still many different game records will be produced.
  - For the reinforcement baseline in REINFORCE algorithm, we will use the prediction from the value network as done in AlphaGo.
- To play interactive games between a human and trained value or policy networks, do the following:

```
>> python value_interactive.py ./value_black.ckpt ./value_white.ckpt
>> python policy_interactive.py ./policy_black.ckpt ./policy_white.ckpt
```

- Task #1
  - Design your own ‘val_and_pol’ function in ‘strategy.py’ to improve the performance of the original ‘val’ and ‘pol’ function. The ‘val’ and ‘pol’ functions in ‘strategy.py’ use only value and policy networks, respectively. You need to understand how ‘val’ and ‘pol’ functions work to write your own ‘val_and_pol’ function. You are not allowed to make any changes to ‘val’ and ‘pol’ functions in ‘strategy.py’. You are not allowed to change ‘game.py’.
  - This is optional, but you will get 5 points for trying each one of the following four options, i.e., you can get up to a total of 20 bonus points. When you train your own neural networks, you are not allowed to change the neural network structures (the number of layers and the number of neurons, etc.). Use the same neural network structures for value and policy networks in given codes such as ‘go_train_SL_policy.py’. You don’t need to do any of the following, but you will not get the bonus points.
    - Train your own value networks (one for playing black and the other for playing white)
    - Train your own SL policy networks (one for playing black and the other for playing white)
    - Train your own RL policy networks (one for playing black and the other for playing white)
    - Implement MCTS algorithm in ‘val_and_pol’ function
  - We will collect the ‘val_and_pol’ function and networks from each team, let them play against each other in a round-robin tournament. Grading will be based on your winning rate and you can get up to 40 points. If you choose to train your own value and/or policy networks, you must use them in your ‘val_and_pol’ function to get the 5 points mentioned above. You should not use both SL and RL policy networks in ‘val_and_pol’ even if you trained both of them. For example, you can use your value network and SL policy network in your ‘val_and_pol’ function or value and RL policy networks. If you choose to implement your own MCTS algorithm in ‘val_and_pol’ function, then the algorithm must be the sole function doing tree search in the function. Otherwise you will not get the 5 bonus points mentioned above. Name your ‘stratege.py’ that contains ‘val_and_pol’ function as ‘project3_strategy.py’ and save it in your home directory before the deadline.
  - If you choose to train your own value or policy networks, save them as ‘project3_value_black.ckpt’, ‘project3_value_white.ckpt’, ‘project3_policy_black.ckpt’, and ‘project3_policy_white.ckpt’ in your home directory before the deadline. Even if you trained both SL and RL policy networks, you can use only one of them in your ‘val_and_pol’ function. Therefore, submit only the ones that you are actually using in your function. If you choose not to train your own networks and if you are using some of the networks in project3.zip, you still need to save the files by renaming the value and/or policy networks you chose as ‘project3_value_black.ckpt’, ‘project3_value_white.ckpt’, ‘project3_policy_black.ckpt’, or ‘project3_policy_white.ckpt’ since we will use them to run your code.
  - If you work as a team, save your code and networks in the teammate’s home directory whose clicker ID is the smallest.
  - You also need to submit a report that explains how you designed your code (including MCTS algorithm if applicable) and how you trained your own neural networks if applicable. Save the report as ‘project3.docx’ or ‘project3.hwp’ in your home directory before the deadline.
  - Each move should not take more than 1 second measured on a server computer with GTX960 GPU. Since there can be unexpected delays caused by the OS, it is OK to exceed the time limit up to 3 times during each game. However, even in that case, it should not take more than 5 seconds per move. If over time, then we will replace your action by a random one. You can check how much time each move takes as done in ‘value_interactive.py’ or ‘policy_interactive.py’, which print out the elapsed time for each move by a computer.