Project #3

EE488B Special Topics in EE <Deep Learning & AlphaGo>, 2017 Fall
Information Theory & Machine Learning Lab., School of EE, KAIST

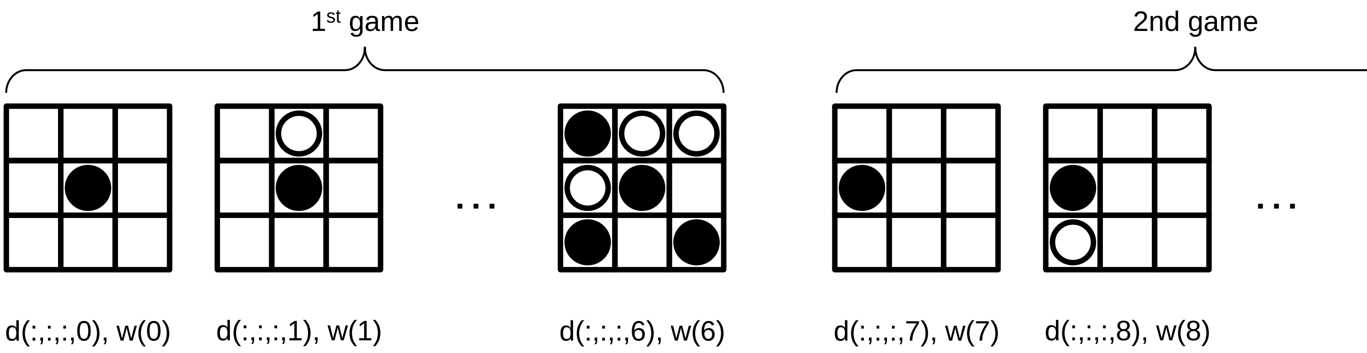
- Contents
  - Training a simple neural network playing tic-tac-toe
  - Mini AlphaGo Zero

- Due date
  - **11:59pm on Wednesday December 20 (extended from December 17)**
  - But, you can submit anytime before the due date.

- Background
  - For project 3, you will train and test simple neural networks that paly tic-tac-toe and Go (with smaller board size).
  - For project 3, only value networks are used. A greedy policy can be found by evaluating each position for the next move using the value network. This corresponds to a tree search with depth = 1 and width = maximum. In boardgame.py, a tree search with depth up to 3 is implemented, but for project 3, we will only assume depth = 1.
  - Although we do not use any policy network, the greedy policy using the value network and single-depth tree search can be improved as we go through training iterations. This aspect is similar to the policy improvement mechanism in AlphaGo Zero.
  - As in AlphaGo Zero, we start from random policy without using any game records. You will be able to see policy induced by value network and tree search improves as we go through training iterations.

- Exercise #1 – Training a simple neural network playing tic-tac-toe
  - For this, you need tictactoe_train.py and boardgame.py, which are in project3.zip.
    - tictactoe_train.py: This trains value networks to play tic-tac-toe and test them.
    - boardgame.py: This implements the game environments. There are 4 games defined in boardgame.py, i.e., game1 is Go (with slight modification to the rules), game2 is tic-tac-toe, game3 is 9x9 Omok (5 in a row), and game4 is Othello.
    - In tictactoe_train.py, we import boardgame.py and we set ‘game=game2()’ to choose game 2, which is tic-tac-toe.
  - Run tictactoe_train.py. It will train and test two value networks (generation 0 and generation 1) that can play tic-tac-toe.
    - The first network is the first-generation (generation 0) 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 (generation 1) 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=5,000' games are played in the first training phase. The first and the third input arguments of the function 'play_games' specify the neural networks for the first (black) and the second (white) players, respectively. For generation=0, they are specified as '[]' as seen above since we don't have any neural networks initially and we will use pure random plays. The first and second player's randomness is specified by 'r1' and 'r2', respectively. They are both vectors of length 'n_train'. If the k-th element in r1 is 1, then it means the first player's randomness is 1 for the k-th game, i.e., purely random actions. If it is 0, then it means purely greedy action. '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 shown below and the first 7 entries in 'w' contains information that indicates '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 planes. The 3 inputs for each position is encoded similarly as in AlphaGo Zero, but simpler, i.e., $[X_t, Y_t, C]$. For example, for the k-th board state
        - d(i,j,0,k)=1 means presence of the current player's stone in position (i,j)
        - d(i,j,1,k)=1 means presence of the opponent's stone in position (i,j)
        - d(i,j,2,k)=1 means the current player is black, 0 means white
    - '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. Note that this step is not similar to what’s done in AlphaGo Zero. 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, we split the dataset into mini-batches with size ‘size_minibatch’ each and train neural networks. Random shuffling is performed for each epoch. ‘sess.run(optimizer, feed_dict=feed_dict)’ with ‘feed_dict = {S: d[indices, :, :, :], W: w[indices]}’ is performed to train the neural network, where a random subset of ‘d’ goes to the input of the neural network ‘S’ and the corresponding ‘w’ becomes the labels ‘W’. Instead of ‘tanh’ in the value network part of AlphaGo Zero, we have 3 softmax outputs corresponding to win, loss, tie, i.e., our value network is trained to predict who is likely to win (or tie) when a board state is given. This part is different from AlphaGo or AlphaGo Zero.
- 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 (i.e., ‘r2’ is an all-one vector). ‘P’ plays black, i.e., it plays first, with no randomness (i.e., r1 is an all-zero vector). ‘n_test’ is the number of games played for testing.
- 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.
- For generating training data in the second training phase, the following values for ‘r1’ and ‘r2’ are used.
    - As mentioned before, ‘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. Such randomization can be useful for generating diverse game records. Note that this randomization strategy is different from that in AlphaGo or AlphaGo Zero.
    - 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 will get something like the following when you run tictactoe_train.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_train.py
Generating training data for generation 0
Data augmentation
Start training
Epoch:   0      Iteration:     99       Loss:    0.90735
Epoch:   0      Iteration:    199       Loss:    0.78354
                                       ⋮
Evaluating generation 0 neural network against random policy
 net plays black: win=0.9890, loss=0.0000, tie=0.0110
 net plays white: win=0.9000, loss=0.0340, tie=0.0660
                                       ⋮
Evaluating generation 1 neural network against random policy
 net plays black: win=0.9910, loss=0.0000, tie=0.0090
 net plays white: win=0.8970, loss=0.0000, tie=0.1030
```

- 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 2,000 games against a player that plays pure randomly.
- After each training phase, the parameters of the trained value networks are saved as a checkpoint file. In tictactoe_train.py, the parameters of the first and second generation value networks are saved as ‘tictactoe_gen0.ckpt.*’ and ‘tictactoe_gen1.ckpt.*’, respectively. We can later load the parameters by using the restore function.
- To play games interactively with a trained network, do the following:

```
>> python tictactoe_interactive.py ./tictactoe_gen1.ckpt
```

    - ‘tictactoe_gen1.ckpt’ is the check point file containing the value network trained in the second training phase.
    - The program ‘tictactoe_interactive.py’ defines the neural network structure in the same way as done in ‘tictactoe_train.py’ and loads the checkpoint file ‘tictactoe_gen1.ckpt.*’ and starts 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.
    - If you do ‘game.play_interactive([],0,[],0)’, then two humans can play the game. If you do ‘game.play_interactive(P,0,P,0)’, then the neural network P plays against itself.
- 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.

- Exercise #2 – Mini AlphaGo Zero
  - For this, we need the following files:
    - boardgame.py: This specifies the game environments as before.
    - go_train.py: trains value networks
    - go_interactive.py: plays interactive games between a human and a neural network
    - go_test_players.py: plays games between two neural networks
  - Some simplifications to Go rules
    - Board size is smaller, i.e., 5x5, to reduce training time. Arbitrary board size is supported by boardgame.py, but training will take longer if you increase the board size.
    - A player cannot reduce its own single-size territory unless doing so can prevent the opponent from capturing the player’s stone
    - Voluntary passing is not allowed, unless no move is possible
    - Game is played until no more plays are possible for both players. Therefore, all effectively dead stones (사석) need to be captured, all neutral spaces (공배) need to be filled, and all territory whose size is bigger than one needs to be reduced until there are only single-size territories. This simplifies counting the size of territory at the end of a game.
    - Rules are still not simple, but you don’t even need to know them to train mini AlphaGo Zero because deep learning will automatically “learn” the rules without explicitly given the domain knowledge.
  - tictactoe_train.py and go_train.py are very similar. Main differences are:
    - ‘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_train.py since the game of Go is more complicated.
  - To play interactive games between a human and trained value or policy networks, do the following:

```
>> python go_interactive.py ./go_gen1.ckpt
```

  - To play games between two neural networks ‘./go_gen0.ckpt’ and ‘./go_gen1.ckpt’, do the following:

```
>> python go_test_players.py
```

    - It will play the following four games and show result for each game.
      - ‘./go_gen0.ckpt’ (black) vs. ‘./go_gen0.ckpt’ (white)
      - ‘./go_gen0.ckpt’ (black) vs. ‘./go_gen1.ckpt’ (white)
      - ‘./go_gen1.ckpt’ (black) vs. ‘./go_gen1.ckpt’ (white)
      - ‘./go_gen1.ckpt’ (black) vs. ‘./go_gen0.ckpt’ (white)

- Task #1 (Tic-tac-toe – 10 points)
  - Train a value network to play tic-tac-toe so that your neural network never loses against any opponent. You need to make sure your neural network never loses whether it plays black or white. In tic-tac-toe, the result will always be a tie if both players play perfectly.
  - Do not change the neural network architecture in tictactoe_train.py since it should already be complex enough to play tic-tac-toe perfectly.
  - In your report, include the following:
    - Explanations on how you trained the network.
    - Source code (name it as ‘project3_task1.py’) that loads the checkpoint file containing your trained neural network and verifies that your neural network never loses against any opponent whether it plays black or white.
    - Explanations on how you designed your code.
    - Submit your report to the collection box outside N1-618 before the deadline. If you cannot come, then you can instead upload your report to your home directory of your computer in N5. In that case, the file name should be project3.* (doc, pdf, etc.)
  - In addition to submitting your report, submit your files electronically as follows:
    - Place your source code and checkpoint files in your home directory of your computer in N5. File names should be ‘project3_task1.py’ and ‘project3_task1.ckpt.*’. If file names are incorrect, they will not be collected.

- Task #2 (Mini AlphaGo Zero – 5 points)
  - Train 5 neural networks such that the first is trained based on purely random plays, the second is trained based on game records played by the first neural network, the third is trained based on game records played by the second neural network and so on. You may want to gradually increase ‘n_train’ for each generation so that later generation neural networks are more powerful.
  - Modify ‘go_test_players.py’ so that it plays 25 round-robin tournaments among the 5 networks, i.e.,
    - gen0 (black) vs. gen0 (white)
    - gen0 (black) vs. gen1 (white)
    - …
    - gen4 (black) vs. gen4 (white)
  - Note that each network plays black 5 times and white 5 times and it even plays against itself.
  - Do not change the neural network architecture in go_train.py since it should already be complex enough.
  - In your report, include the following:
    - Explanations on how you trained the networks.
    - Source code (name it as ‘project3_task2.py’) that loads 5 checkpoint files containing your trained neural networks, plays 25 round-robin tournaments, and shows the game results.
    - Explanations on how you designed your code.
    - Submit your report to the collection box outside N1-618 before the deadline. If you cannot come, then you can instead upload your report to your home directory of your computer in N5. In that case, the file name should be project3.* (doc, pdf, etc.) This single file should contain your report for both tasks 1 and 2.
  - In addition to submitting your report, submit your files electronically as follows:
    - Place your source code and checkpoint files in your home directory of your computer in N5. File names should be ‘project3_task2.py’, ‘project3_task2_gen0.ckpt.*’, ‘project3_task2_gen1.ckpt.*’, …, ‘project3_task2_gen4.ckpt.*’. If file names are incorrect, they will not be collected.
  - Computers in Haedong lounge will not be available from 4pm to 6pm on Saturday December 18 due to an event at School of EE. Computers in LG hall and N5 can be used during the time.