Shi Xingyue

Dr. Rao Yanghui

Artificial Intelligence

27 December 2018

Final Project: Othello Using Reinforcement Learning

[Abstract]

In this final project, I implemented Othello game again, but using different methods. I read Ree et al.'s paper and tried out two TD methods, namely Q-learning and SARSA on my own [3]. I also learnt about the advanced algorithm, AlphaGo Zero by studying Silver et al.'s paper [2]. At the end of my report, I compared all the five algorithms I have studied on this semester, namely Minimax, Q-learning, SARSA, Genetic Algorithm and AlphaGo Zero.

1 Methods

1.1 Algorithms Comparison

1.1.1 Alpha-beta Pruned Minimax

This is the algorithm I used in Lab 6 to implement Othello. Detailed explanation of algorithm and implementation are included in that corresponding report so I would skip introducing it here.

1.1.2 Q-Learning

Q-learning is an on-policy TD method. It gradually updates the action value function to generate an ideal Q table to guide the agent to play games (detailed steps are listed in Flowchart and Pseudo-codes section). In this experiment, what I have implemented is the algorithm combining neural networks together with Q-learning, which is called as "DQN". The Q-learning network tries to find a function $f(s, a)$

which is approximate to the action value function $Q(s, a)$. We can then use $f(s, a)$ as $Q(s, a)$ in actual use, so that we no longer need to keep record of a huge Q table.

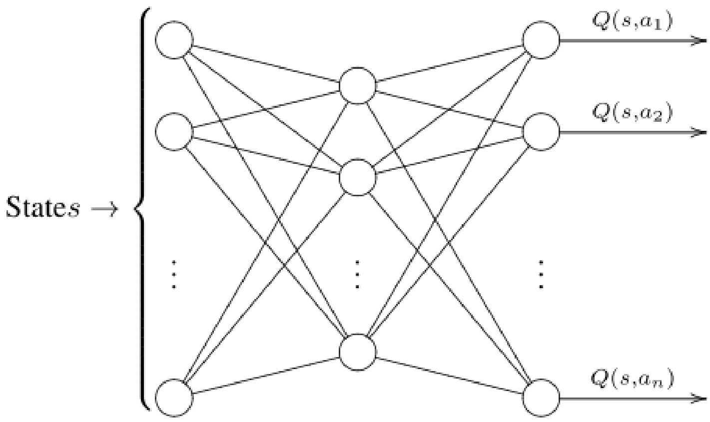


Figure 1 Q-learning network. This network tries to approximatethe values of each possible action in the states given as inputs [3].

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r_t + \gamma \max_a \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t))$$

Expression 1 This expression shows how to assign Q-value in Q-learning algorithm, where $0 < \alpha < 1$ *is the learning rate.*

1.1.3 SARSA

SARSA is an off-policy TD method. I think the implementation of SARSA is easier than Q-learning, because its action value function does not have "max". Most part of it is identical to Q-learning except for action value function.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r_t + \gamma\hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t))$$

Expression 2 This expression shows how to assign Q-value in SARSA, which is quite similar to that in Q-learning.

1.1.4 AlphaGo Zero

AlphaGo Zero is a well-developed algorithm to train agent, which takes advantage of several basic methods.

1.2 Flowchart and Pseudo-codes

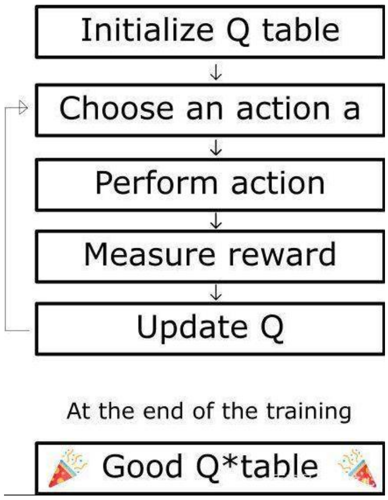


Figure 2 The flowchart representing the overview of Q-learning algorithm [4]. SARSA algorithm is identical to Q-learning except for that SARSA uses a different action value function.

Algorithm 1 SARSA

Initialise Q arbitrarily, $Q(terminal, \cdot) = 0$
repeat
    Initialize s
    Choose a ϵ-greedily
    repeat
        Take action a, observe r, s'
        Choose a' ϵ-greedily
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$
        $s \leftarrow s', a \leftarrow a'$
    until s is terminal
until convergence

List 1 Pseudo-code of SARSA

Algorithm 2 Q-learning

Initialise Q arbitrarily, $Q(terminal, \cdot) = 0$
repeat
    Initialize s
    repeat
        Choose a ϵ-greedily
        Take action a, observe r, s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$
        $s \leftarrow s'$
    until s is terminal
until convergence

List 2 Pseudo-code of Q-learning

1.3 Key Codes with Comments

The overview of my codes is shown in Figure 3. Class Game, class Board and class Human are not the points of this project so I simply presented their member functions' declaration, but no implementation. I will show details of class Agent and class NN.

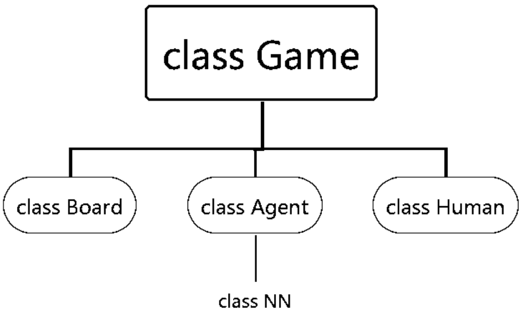


Figure 3 The classes call tree view. You can have an overview of the entire project from this view.

```
class Game:
    def __init__(self)
    def addPlayer(self, player, log_move_history = True)
    def getScore(self)
    def run(self, show_board = False)
```

List 3 Class Game

```
class Board(object):
    BLACK = 1
    WHITE = -1
    def __init__(self)
    def getScore(self)
    def getState(self)
    def isOnBoard(self, x, y)
    #先判断某一步是否合法
    #若合法，更新棋盘
    #若非法，不更新
    def updateBoard(self, tile, row, col)
    #沿着8种可能的方向判断某一步棋是否合法（是否会引发reversion）
    #返回值为造成的reversion的数量
    def isValidMove(self, tile, xstart, ystart)
    def printBoard(self)
```

List 4 Class Board

```python
class Human:
    def play(self, place_func, board_state, me, _)
```

List 5 Class Human

```python
class Agent:
    def __init__(self, q_lr, discount_factor, net_lr = 0.01)
    def play(self, place_func, board_state, me, log_history = True)
    def updateWeights(self, final_score)
```

List 6 Class Agent

```python
class NN:
        #初始化神经网络
        #参数layer_dims指定网络的层数和每一层的结点数量，显然8*8的Othllo输入层应有64个结点
        #参数learning_rate,在反向传播算法计算校正值delta的时候要用到
        def __init__(self, layer_dims, learning_rate)
        #将训练得到的weights输出到文件
        def save(self, filename)
        #读weights文件
        def load(self, filename)
        #将矩阵转换成对应的向量
        def mkVec(self, vector1D, add_bias = True)
        #输出某一状态s下的Q向量Q(s)
        def getOutput(self, input_vector)
        #反向传播
        def backProp(self, sample, target)
```

List 7 Class NN. The most difficult and significant function is “backprop”, whose implementation is commented detailed in the following List.

```
#反向传播
def backProp(self, sample, target):
    # Propagate forwards to get the network's layers' outputs
    outputs = [sample]
    for i in range(len(self.layers)):
        outputs.append(activation(self.layers[i].dot(np.vstack((outputs[i], 1)))))

    # These will still need to be multiplied by the output from the previous layer
    # e.g. layer_deltas[0]*outputs[-2]
    layer_deltas = np.empty(len(outputs) - 1, object)

    # 输出层
    layer_deltas[-1] = (target - outputs[-1]) * dactivation(outputs[-1])


    # i == current layer; Walk backwards from second to last layer (Hence
    # start at -2, because len-1 is the last element) Also recall that
    # range "end" is exclusive.
    for i in range(len(layer_deltas) - 2, -1, -1):
        # Need to do i+1 because outputs[0] == the input sample, and i+1 is
        # the ith layer's output
        layer_derivative = dactivation(outputs[i+1])

        # 校正值delta
        layer_deltas[i] = layer_derivative * (self.layers[i+1].T.dot(layer_deltas[i + 1])[:-1])

    for i in range(len(self.layers)):
        # Because outputs[0] == input sample, layer[i] input == outputs[i]
        # This is delta_weights
        self.layers[i] += self.learning_rate * np.c_[outputs[i].T, 1] * layer_deltas[i]

    return outputs[-1]
```

List 8 Function backProp. The activation functions used in it is tanh (some other work tried sigmoid instead).

2 Results and Analysis

2.1 Result Examples

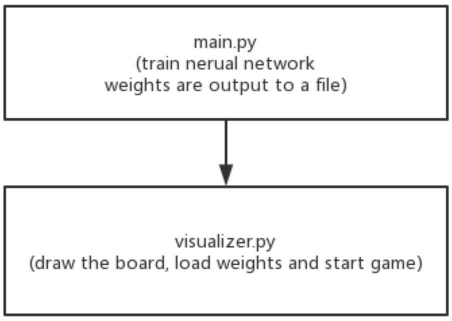


Figure 4 Program "main.py" should be run at the first place to train neural network (get weights), then we can run "visualizer.py" to load these weights and start an Othello game.

Probability of exploration	Initialized to 0.1
Learning rate of Q-learning/SARSA	1
Learning rate of neural network	0.01
Discount factor	1

Table 1 Parameter settings of reinforcement learning algorithms and neural networks. I basically used the same settings as those in Ree et al.'s experiments [2].

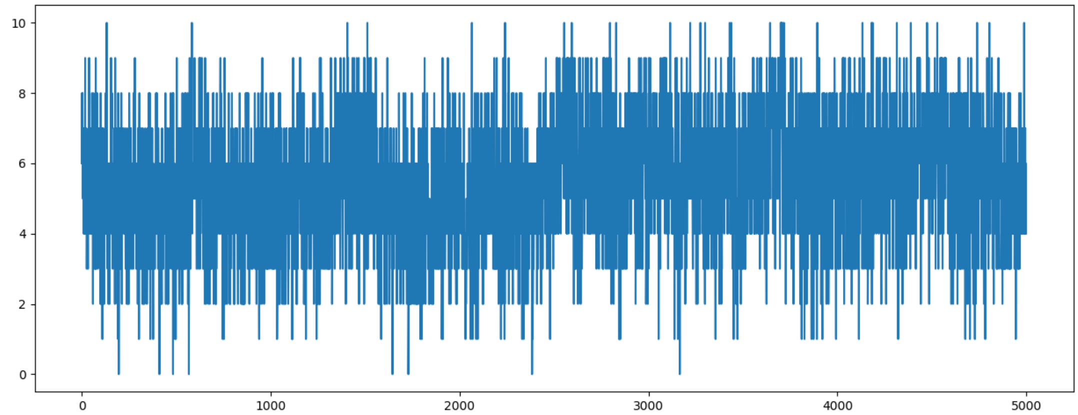


Figure 5 The ratio on winning throughout neural network training process (Q-learning). I ran 50,000 epochs with 10 matches each. Though the ratio fluctuates sharply.

I took the first 3 rounds as running results example.

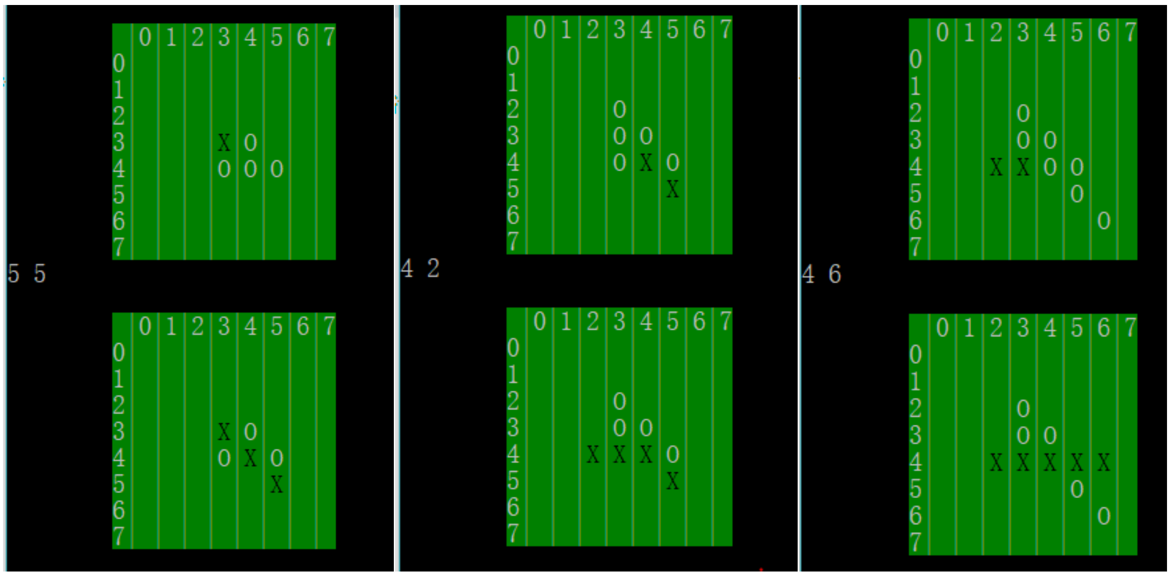


Figure 6 The first 3 rounds of Othello game. Agent plays white.

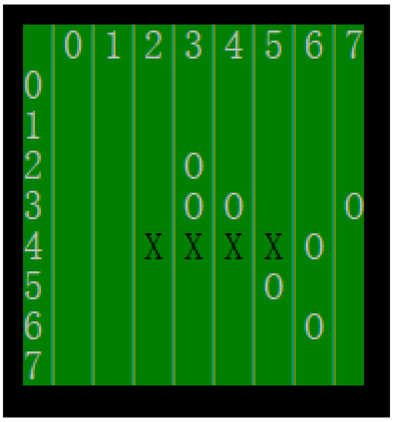


Figure 7 After the first 3 rounds. It looks like the agent masters the technique of occupying diagonal line and edges much better than my previous agent using Minimax!

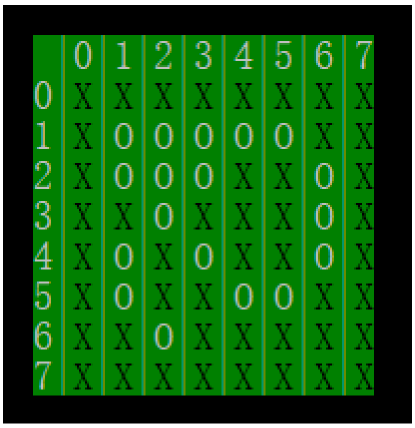
2.2 Evaluation Indices

I got my two agents respectively using Minimax (MODIFIED VERSION! I improved it by attempts listed in the next section) and Q-learning have a match.

It turned out that the modified Minimax agent did much better job! Possible reason for the unideal performance of Q-learning agent are:

- Insufficient training epochs.
- Neural network design. I used an linear function to approximate Q function here. Maybe other models could have better result.

By the way, I regretted badly that I did not increase search depth earlier…because the modified Minimax agent might have saved our rank 😥



3 Reflection

My job done during this project can be divided in to two parts, I firstly tried to improve my previous Minimax Othello and then implemented another version using DQN algorithms.

When trying to improve my Minimax Othello agents. I had attempts as below:

- Give "Corners" higher weights in evaluation function.

  When having match with other teams. I found that my agent failed to occupy corners quickly, so I modified the weight of "Corners" in evaluation function to higher value.

- Increase search depth.

  The initial search depth of my Iterative Depth Searching is 2, which may be too small. I changed it to 4.

After implementing DQN, as described in this report, I still had some further work I did not have sufficient time to finish:

- My Q-learning agent should learn from better master! In this lab, my agent learnt from a random player. Although it seems that the agent has master Othello pretty well, Ree's paper puts that learning from smarter player results in better performance.
- The performance of my Q-learning agent is unclear. I have only let it have match with my Minimax agent. Maybe it should have more matches with all kinds of agents.

Up to now, I have studied on 5 algorithms possible to implement Othello agent. Here is my understanding of them.

Minimax	**Q-learning (DQN)**	**SARSA**	**Genetic Algorithm**	**AlphaGo Zero**
Relatively easy to implement but have many opportunities to	Q-learning is not very difficult to understand and	Similar to Q-learning. Q-learning can get trapped in	I didn't try this algorithm by my own. According to	Ha the boss of these 5 algorithms! It takes advantage

improve! I think the process to design an ideal evaluation function is the most interesting part of this lab.	bring about. It is the construction of neural network brings me main workload this time. Actually, I do not think network is necessary here to save storage space because Othello don't have so much game information.	Maximization Bias, while SARSA not because it doesn't use "max" when updating.	my teammate, this algorithm takes considerable time to train if lacking reasonable variation policy. Performance of this algorithm implemented by my teammate doesn't seem acceptable.	of deep learning and reinforcement learning. I guess it is unfair to do rank using AlphaGo Zero agents…

Table 2 5 algorithms that can be used to implement Othello agent.

Work Cited

[1] Haykin, S. (2009). *Neural Networks and Learning Machines, Third Edition.* Pearson Education.

[2] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354.

[3] Van Der Ree, M., & Wiering, M. (2013, April). Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play. In *ADPRL* (pp. 108-115).

[4] https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe

[5] http://mi.eng.cam.ac.uk/~mg436/LectureSlides/MLSALT7/L3.pdf

[6] https://zhuanlan.zhihu.com/p/21421729