Deep Learning and Practice: 2048 TD

0756079 陳冠聞

May 30, 2019

1 Introduction

Reinforcement Learning is a branch of **Artificial Intelligence**. It allows machines and software agents to automatically determine the ideal behaviour within a specific context, in order to maximize its performance.

In this lab, I will implement reinforcement learning with **Temporal Difference** learning on the 2048 game. Furthermore, I will compare the difference between state value and after state value. Finally, I discuss additional strategy to gain performance.

2 TD Learning

2.1 Reinforcement learning

In Reinforcement learning setting, there are three major components: agent, environment and reward. The agent takes actions, the environment change and give reward to the agent, then the agent takes another action and so on. The goal of reinforcement learning is to find an optimal policy, which can maximize the expected sum of rewards.

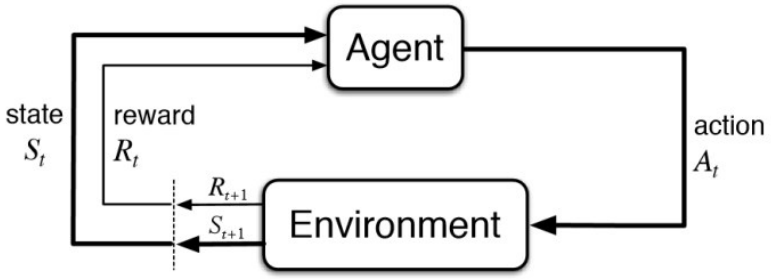

Figure 1: Reinforcement Learning

Most of the reinforcement learning problem formulated as an **Markov Decision Process** (MDP). A MDP is a tuple of $< S, A, P, R, \gamma >$, where S is the set of all states, A is the set of all action, P is the state transition probability, R is the reward function mapping the state to real value, and γ is the discount factor. The Markov property state that the future state is independent to the past given the current state

$$p(s_{t+1}|s_t) = p(s_{t+1}|s_1, ..., s_t)$$

2.2 TD(0)

We define **Value Function** $V(s)$ which is the long-term value of the state s. Once we learn the value function of all state, we learn the implicit policy by taking the action toward the state with max state value. If we have fully understanding of the environment, we can use $DynamicProgramming$ backup to solve the value function. Unfortunately, we don't know the environment in most of the interesting problems, or the dynamic programming is too slow when the number of state is too big.

Thus, **Model-Free** learning is a more popular approach, which learn directly from the episode without the knowledge of the environment. **Temporal Difference** learning is one of the model-free approach. In TD learning, we divide the long-term reward $V(s_t)$ into two parts, which are immediate reward R_{t+1} and discounted future reward $\gamma V(s_{t+1})$, and the learning target is to minimize the temporal difference between each adjacent timestep. That is, the TD target is

$$R_{t+1} + \gamma V(s_{t+1})$$

and we update the estimated value function by

$$V(s_t) \leftarrow V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

Comparing to another model-free approach **Monte-Carlo Learning**, which update the estimated value function by actual sum of discounted rewards G_t

$$V(s_t) \leftarrow V(s_t) + \alpha(G_t - V(s_t))$$

TD learning has lower variance, and is more efficient than MC learning on the problems that exploit Markov property.

2.3 Before State Value

In the 2048 game, after we slide the board, the board state will change based on the action, and randomly generate another tile which is also another state transition.

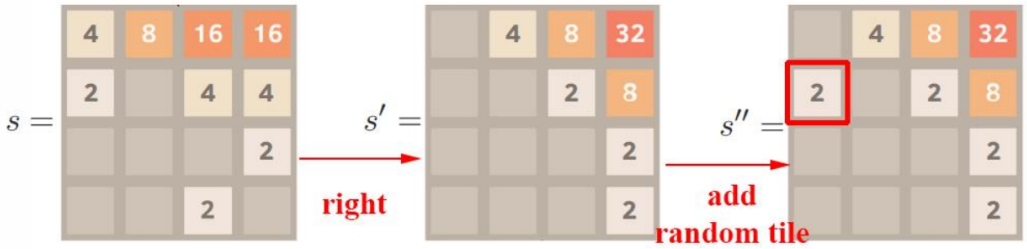

Figure 2: State Transition of 2048

Or it can be formulated as

$$s_t^{before} \xrightarrow{\text{action}} s_t^{after} \xrightarrow{\text{generate}} s_{t+1}^{before} \xrightarrow{\text{action}} \ldots \tag{1}$$

Thus, we have two candidate of the estimation of value function. On is the before state value, which estimate the long-term value of the board state before action. In this case, the next before state s_{t+1}^{before} is unknown when we want to choose the optimal action based on action reward and the estimated value of discounted future reward $\gamma V(s_{t+1}^{before})$. That is, it's not total model-free, we must know how the tiles are generated in 2048.

The learning of before state value can be described in following steps. First, we play game based on the policy that pick the action the has maximal expected action value. The action value of a state s and action a is

$$r + \sum_{s^{''} \in S} p(s^{''}|s^{'})V(s^{''})$$

where $s^{'}$ is the after state of taking action, and $s^{''}$ is the next before state. Second, we record the episode of the game with the form $(s, a, r, s^{'}, s^{''})$. Finally, we update the estimated before state value from the end of episode to the beginning by

$$V(s) \leftarrow V(s) + \alpha(r + V(s^{''}) - V(s))$$

2.4 After State Value

Another option of value function is the after state value, which estimate the long-term value of the board state after action (before randomly generate tile). In this case, the action value could be evaluated without the uncertainty because it's independent to the random generation of tiles, so it is total model-free since we don't have to know the transition probability. The learning of after state value can be described in following steps. First, we play game based on the policy that pick the action the has maximal expected action value. The action value of a state s and action a is

$$r + V(s^{'})$$

Second, we record the episode of the game with the form $(s, a, r, s^{'}, s^{''})$. Finally, we update the estimated before state value from the end of episode to the beginning by

$$V(s^{'}) \leftarrow V(s^{'}) + \alpha(r_{next} + V(s^{'}_{next}) - V(s^{'}))$$

where $s^{'}_{next}$ is the after state of $s^{''}$ by taking optimal action according current estimated value function, and r_{next} is the reward of that action.

3 Implementaion

I modify the code from https://github.com/lukewayne123/2048-Framework, which has implemented the game environment and the basic agent. The major modification is the features of N-tuples network. The original codes use two 4-tuples, which the corresponding indices are $[0, 1, 2, 3]$ and $[4, 5, 6, 7]$ respectively, and used for all possible board isomorphism. In my implementation, I use four 6-tuples, which the corresponding indices are $[0, 1, 2, 3, 4, 5]$, $[4, 5, 6, 7, 8, 9]$, $[0, 1, 2, 4, 5, 6]$ and $[4, 5, 6, 8, 9, 10]$ respectively, and used for all possible board isomorphism. I use I modify the weight agent, and implement the TD value update based on the above formula. Furthermore, I implement the after-state value TD learning using above formula.

```python
class weight_agent(agent):
    def init_weights(self):
        self.net += [weight(16**6)] # feature for line [0, 1, 2, 3, 4, 5] includes 16*16*16*16 possible
        self.net += [weight(16**6)] # feature for line [4, 5, 6, 7, 8, 9] includes 16*16*16*16 possible
        self.net += [weight(16**6)] # feature for line [0, 1, 2, 4, 5, 6] includes 16*16*16*16 possible
        self.net += [weight(16**6)] # feature for line [4, 5, 6, 8, 9, 10] includes 16*16*16*16 possible
        self.feature_idx = [[0, 1, 2, 3, 4, 5], [4, 5, 6, 7, 8, 9], [0, 1, 2, 4, 5, 6],[4, 5, 6, 8, 9, 10]]
        return

    def lineIndex(self, board_state):
        idxs = [0, 0, 0, 0]
        for f in range(4):
            for i in range(6):
                idxs[f] = idxs[f] * 16 + board_state[self.feature_idx[f][i]]
        return idxs

    def close_episode(self, ep, flag = ""):
        episode = ep[2:].copy()
        episode.reverse()
        for i in range(1, len(episode)-1, 2):
            before_state_next, _, _, _ = episode[i-1]
            after_state, move, reward, _ = episode[i]
            before_state, _, _, _ = episode[i+1]
            self.learn(before_state, move, reward, after_state, before_state_next)
```

Figure 3: Code Snippet for Agent (part 1)

```
    def compute_after_state(self, board_state, move):
        board_state = board(board_state)
        reward = move.apply(board_state)
        board_after_state = board(board_state)
        return board_after_state, reward

    def evaluate_state_action(self, board_state, op):
        move = action.slide(op)
        board_after_state, reward = self.compute_after_state(board_state, move)
        return reward + self.lineValue(board_after_state)

    def select_best_action(self, board_state):
        legal_ops = [op for op in range(4) if board(board_state).slide(op) != -1]
        if legal_ops:
            best_op = 0
            best_value = -1
            for op in legal_ops:
                value = self.evaluate_state_action(board_state, op)
                if value > best_value:
                    best_value = value
                    best_op = op
            return action.slide(best_op)
        else:
            return action()

    def learn(self, before_state, move, reward, after_state, before_state_next):
        move_next = self.select_best_action(before_state_next)
        after_state_next, reward_next = self.compute_after_state(before_state_next, move_next)
        reward_next = max(0, reward_next)
        TD_diff = reward_next + self.lineValue(after_state_next) - self.lineValue(after_state)
        self.updateLineValue(after_state, self.alpha*TD_diff/(4*8))
```

Figure 4: Code Snippet for Agent (part 2)

3.1 How the code work

In this lab, I use the random environment (standard 2048 game) provided by the framework mentioned above, and I modify the weight agent as the player to interact with the environment. Given a certain state, the player will evaluate the action value of each possible action using the sum of action reward and the value estimated of after state, and choose the best action based on the evaluation. In each game, every state and action token by player, the corresponding reward and the next state are all recorded. At the end of each game, the weights of the player will be updated using the episode record. For the training, I train the agent and save the weight and history of scores every 10k games.

```
start = 0 * 1000
total = 200 * 1000
interval = 10 * 1000

rand_environment = rndenv()
for begin in range(start, total, interval):
    end = begin + interval
    TD_player = player()
    if begin > 0:
        TD_player.load_weights("weights/%dk.pth"%(begin // 1000))
    history = train(TD_player, rand_environment, start=begin, total=end)
    if begin > 0:
        history_before = load_history("%dk"%(begin // 1000))
        history['score'] = history_before['score'] + history['score']
    TD_player.save_weights("weights/%dk.pth"%(end // 1000))
    save_history(history , "%dk"%(end // 1000))
    plot_hisotry(history, block=end//100, save=True)
```

Figure 5: Code Snippet for Training

4 Result

I train the weight agent for 135k games with the learning rate $\alpha = 0.025$, and it take about 3 days to complete. From the Figure 6 we can see that the score keep improving in the training.

After training 135k games, I run another 1,000 games to evaluate the max tiles the agent can achieve. As shown in Figure 7, the probability to reach 1,024 is about 93.3% . These two results indicate that the agent indeed learns how to play the 2048, and play better than most of normal players such as me.

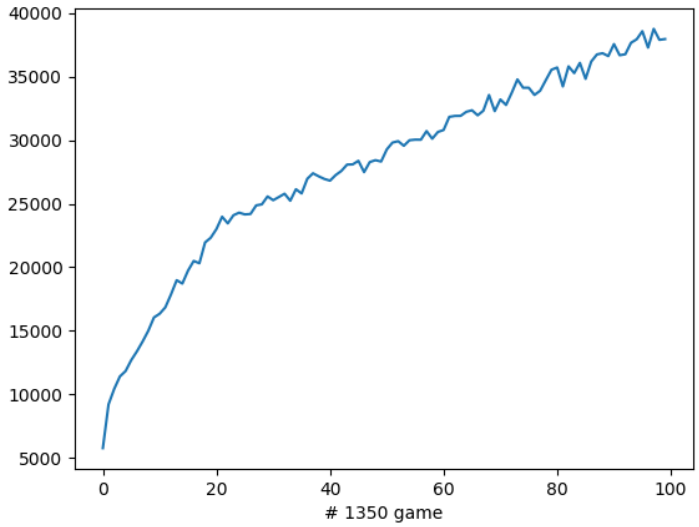

Figure 6: Score Trend

```
1000    avg = 39365, max = 114392, ops = 3019 (1540|114000)
        128     100.0%   (0.1%)
        256     99.9%    (1.5%)
        512     98.4%    (5.1%)
        1024    93.3%    (20.2%)
        2048    73.1%    (43.2%)
        4096    29.9%    (29.6%)
        8192    0.3%     (0.3%)
```

Figure 7: Result