# Assignment 2 MC & TD Prediction Report

## 1 Experiment Requirements

- Programming language: python 3
- You should implement both first-visit and every-visit MC method and TD(0) to evaluate state value in small grid world.

## 2 Code & Analysis

### 2.1 Environment

Given that MC and TD methods focus on unknown MDP, here I split environment and prediction model.

First, initialize the gridWorld class with edge length and terminal states. State space, action space, reward, and how state changes with action, that is `self.act_trans`, are defined here.

```
def __init__(self, grid_len, terminal, gamma=1):
    self.grid_len = grid_len  # edge length of grid word
    self.states_num = grid_len ** 2  # number of states
    self.states = [i for i in range(self.states_num)]  # all states
    self.terminal = terminal  # all terminal states
    self.gamma = gamma  # discount factor
    self.reward = -1  # reward
    self.actions = ['n', 's', 'w', 'e']  # all actions
    # how action lead to state changing
    self.act_trans = {'n': -grid_len, 's': grid_len, 'w': -1, 'e': 1}
```

Then, function `getStates()` returns all states, function `getTerminal()` returns all terminal states, and function `getAction()` returns all possible actions.

We can use function `generateInitialState()`, which returns a initial state randomly chose from non-terminal states, to reset the environment.

```
def generateInitialState(self):
    init_state = random.choice(self.states)
    while init_state in self.terminal:
        init_state = random.choice(self.states)
    return init_state
```

Function `getNextState()` finds reward and next state according to current state and action. It also returns whether the next state is terminal state.

```
def getNextState(self, state, action):
    next_state = state + self.act_trans[action]
    # return to current state if next state is out of grid
    if next_state < 0 or next_state >= self.states_num or (
            next_state % self.grid_len != state % self.grid_len and
            next_state // self.grid_len != state // self.grid_len):
        next_state = state
    is_end = next_state in self.terminal
    return self.reward, next_state, is_end
```

Function `generateEpisode()` can generate an episode according to input policy. First, generate the initial state. Then, keep going according to policy and restore state, action, and reward until terminal state is reached. Finally, return the sequences of states, action, and reward.

```
def generateEpisode(self, policy):
    ep_states, ep_actions, ep_rewards = [], [], []
    cur_state = self.generateInitialState()
    is_end = False
    while not is_end:
        ep_states.append(cur_state)
        cur_action = random.choice(policy[cur_state])
        ep_actions.append(cur_action)
        cur_reward, cur_state, is_end = self.getNextState(cur_state, cur_action)
        ep_rewards.append(cur_reward)
    return ep_states, ep_actions, ep_rewards
```

## 2.2 Model free prediction

Because MC model and TD model have got a lot in common, I create a base class called `ModelFree`. Environment is essential to initialization while policy and discount factor gamma is selective. If not given, policy will be initialized to uniform random policy and discount factor will be initialized to 1.

```
def __init__(self, environment, policy=None, gamma=1):
    self.environment = environment
    self.states = environment.getStates()
    self.state_num = len(self.states)
    self.policy = policy
    self.gamma = gamma
    if not policy:
        self.policy = [self.environment.getActions() for i in range(self.state_num)]
        for state in environment.getTerminal():
            self.policy[state] = []
```

We can use function `setPolicy()` to set a new policy.

```
def setPolicy(self, policy):
    self.policy = policy
```

There are also some functions about output and visualization in class `ModelFree`. Given the weak link between those function and our theme, I only list their names and usage. The implement detail can be found in source code. Function `outputValue()` prints values of states in gridWord-style to command line. Function `plotValues()` can visualize the values of states via heatmap. Function `plotDelta()` can plot the figure how delta(the max difference between values of this update and last update) change with episodes.

## 2.3 First-visit & every-visit MC

Class `MC` is a derived class of `ModelFree`.

Monto-Carlo policy evaluation uses empirical average sample return. To avoid repeated calculation, I use function `computeEisodeReturn()` to compute return at each step of an episode.

```
def computeEpisodeReturn(self, ep_states, ep_reward):
    ep_len = len(ep_states)
    ep_returns = [0 for i in range(ep_len)]
    ep_returns[-1] = ep_reward[-1]
    for i in range(ep_len-2, -1, -1):
        ep_returns[i] = ep_reward[i] + self.gamma * ep_returns[i + 1]
    return ep_returns
```

The most important function `_MCPolicyEval()`, used in both first-visit MC and every-visit MC, need two inputs, number of episode `episode_num` and `visit_method`, either 'first' or 'every'. First, initialize all the states and counters of states to zero. Then, use function of environment to get a whole episode and compute its returns. Traverse states in this episode and check if the state is visited for the first time or visit method is 'every'. If so, increase the count of the state, update value by incremental mean $\frac{G_t - V(S_t)}{N(S_t)}$, and update `delta`, the max difference between values of this update and last update. Finally, loop the second step for `episode_num` times.

```
def _MCPolicyEval(self, episode_num, visit_method):
    values = {state: 0 for state in self.states}     # initialize value for each state
    state_cnt = {state: 0 for state in self.states}  # initialize count for each state
    k, deltas = 0, []

    while k < episode_num:
        delta = 0
        ep_states, ep_actions, ep_reward = self.environment.generateEpisode(self.policy)
        ep_returns = self.computeEpisodeReturn(ep_states, ep_reward)
        # record whether the state is visited in this episode
        first_appear = {state: True for state in self.states}
        for i, state in enumerate(ep_states):
            # if state isn't visited in this episode before or use every-visit, update state
            if first_appear[state] or visit_method == 'every':
                state_cnt[state] += 1
                old_value = values[state]
                values[state] += (ep_returns[i] - old_value) / state_cnt[state]
                first_appear[state] = False
                delta = max(delta, abs(values[state] - old_value))
        deltas.append(delta)
        k += 1
```

Then, in function `firstVisitMCpolicyEval()` and `everyVisitMCPolicyEval()`, we only need to call function `_MCPolicyEval()` with different visit method.

```
def firstVisitMCPolicyEval(self, episode_num):
    self._MCPolicyEval(episode_num, 'first')

def everyVisitMCPolicyEval(self, episode_num):
    self._MCPolicyEval(episode_num, 'every')
```

## 2.4 TD(0)

Class `TD` is also a derived class of `ModelFree`.

The key function `TD0PolicyEval()` is listed as follows. Since TD updates value toward not actual return $G_t$ but estimated return $R_{t+1} + \gamma V(S_{t+1})$, we don't need to wait for the end of whole episode but can update value as soon as environment reaches next step.

First, generate the initial state. Then, choose action according to policy, get reward and next state, update value via $\alpha[R+\gamma V(S^{'})-V(S)]$, and loop until this episode ends. Finally, loop above steps for `episode_num` times.

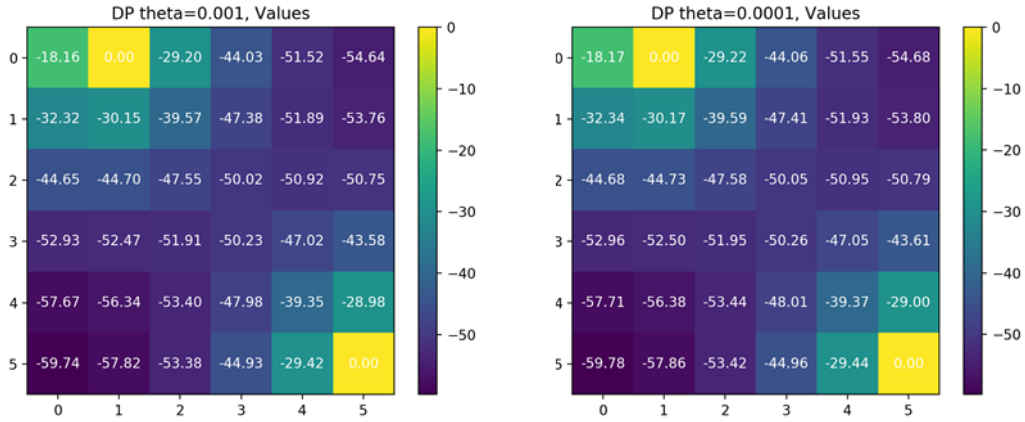
```
def TD0PolicyEval(self, episode_num):
    values = {state: 0 for state in self.states}
    deltas = []
    alpha, k = 0.05, 0

    while k < episode_num:
        delta, is_end = 0, False
        cur_state = self.environment.generateInitialState()
        while not is_end:
            cur_action = random.choice(self.policy[cur_state])
            cur_reward, next_state, is_end = self.environment.getNextState(cur_state, cur_action)
            old_value = values[cur_state]
            values[cur_state] += alpha * (cur_reward + self.gamma * values[next_state] - old_value)
            delta = max(delta, abs(values[cur_state] - old_value))
            cur_state = next_state
        deltas.append(delta)
        k += 1
```

# 3 Results & Analysis

Theoretically speaking, we can evaluation any policy. Here, take uniform random policy as example.

Other problem is that I don't have a real value function for uniform random policy. DP method I used in last assignment does give a good approximation of true value function. I will use the it as a approximation of true value.

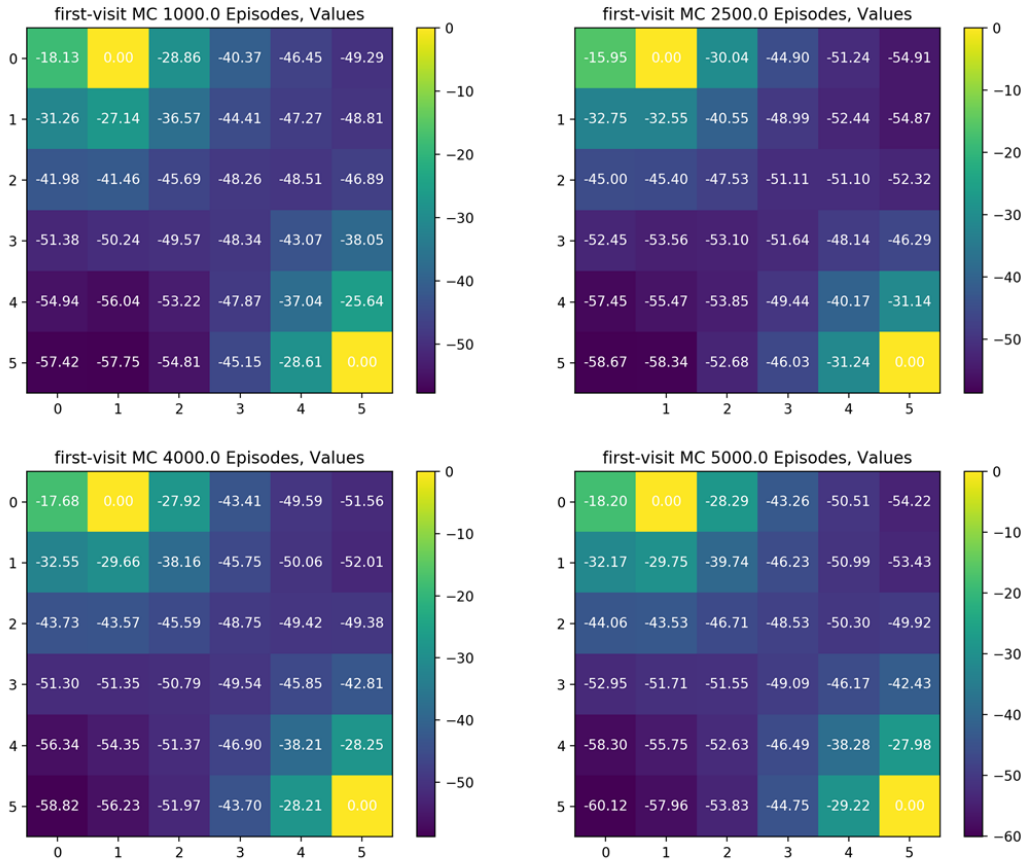



Actually, later I find uniform random policy is hard to evaluate and the results I get and convergence may vary from training to training. I try optimal policy I get in DP method and it turns out its training, whose details are shown in part 4, is much easier.

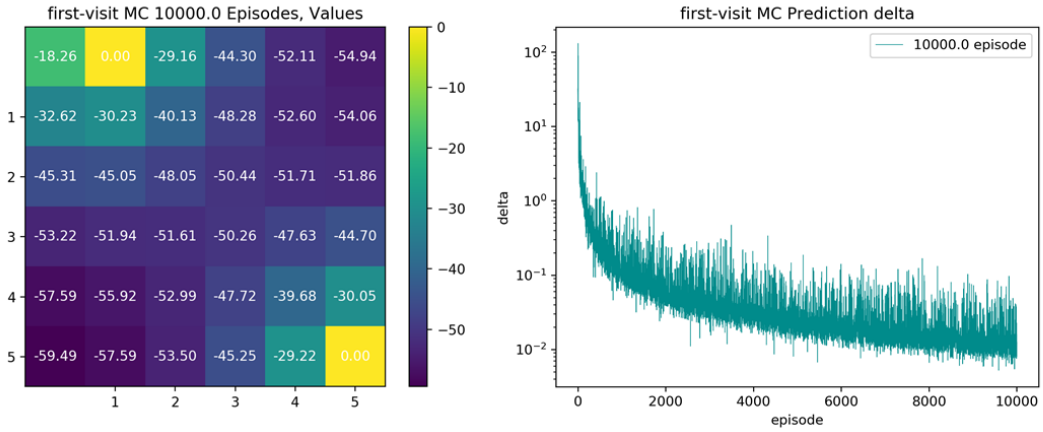
Given that if we don't know any information about policy, it's a good idea to start with uniform random policy, I still take uniform random policy as example here. Its results are listed as follows.

## 3.1 First-visit MC

It's hard to tell when First-visit MC method converges. In between 1000 to 5000 episodes, values may fluctuate around the real values. After that, values become more stable but still vibrate. Because I use $\frac{G_t - V(S_t)}{N(S_t)}$ to update $V(S_t)$, as the number of episode increases, $N(S_t)$ increases and $\frac{G_t - V(S_t)}{N(S_t)}$ becomes small, therefore, values become more stable.



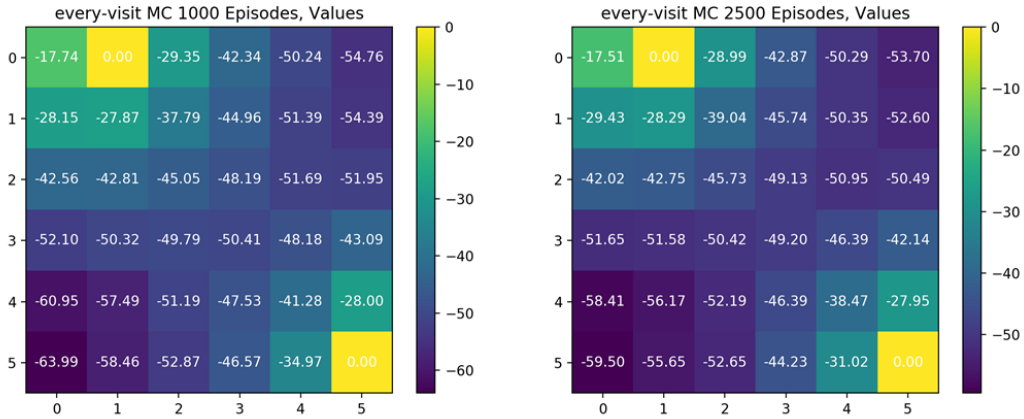

Here, I also plot a picture of value function and delta I got at 10000 episodes. It's very close to what I get via DP policy evaluation. As for delta, it decreases as number of episode increases, despite of vibration. Although its decrement may mainly comes from $1/N(S_t)$, to some extend, it can still act as as a sign of convergence. From this picture, we can observe that delta decrease more smoothly after 5000 episode, that's why values become more stable after that.



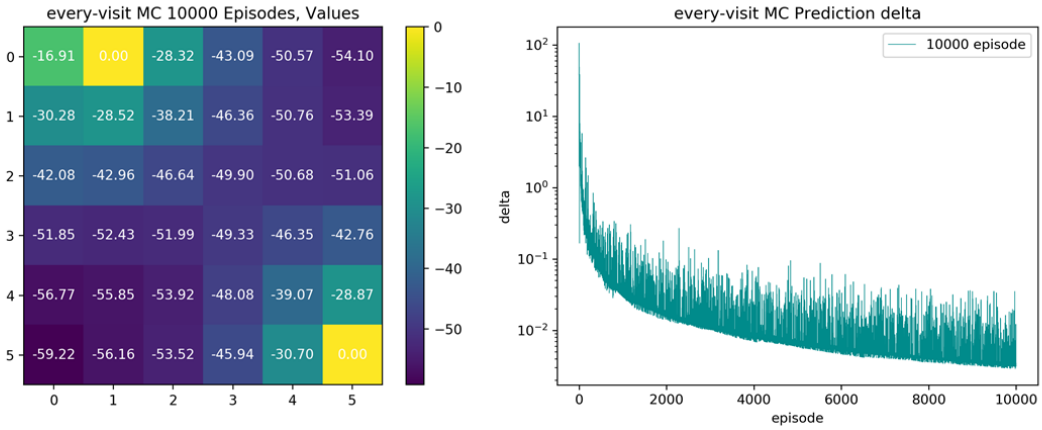

## 3.2 Every-visit MC

The result of every-visit MC is similar to first-visit MC.

In between 1000 to 4000 episodes, values may fluctuate around the real values. After that, values become more stable but still vibrate.



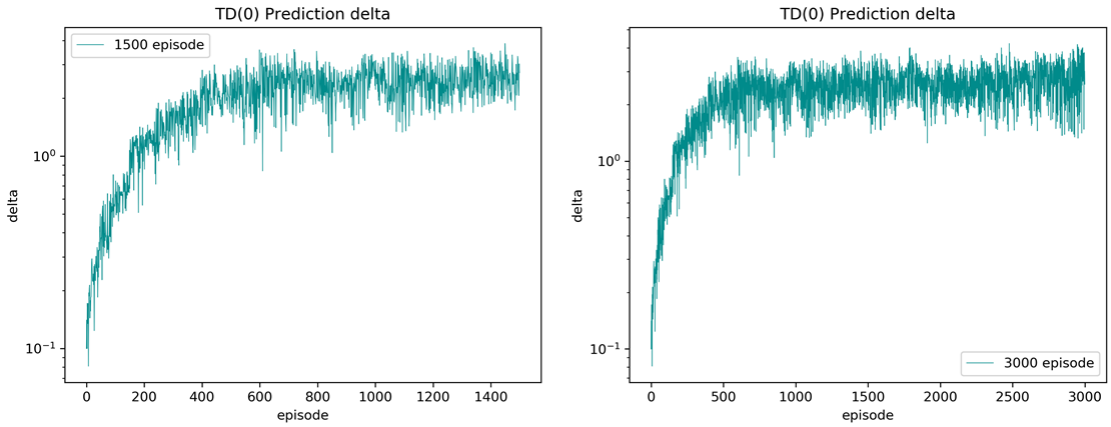

This tendency is also shown in picture of delta, delta decrease more smoothly after 4000 episode when values become more stable.



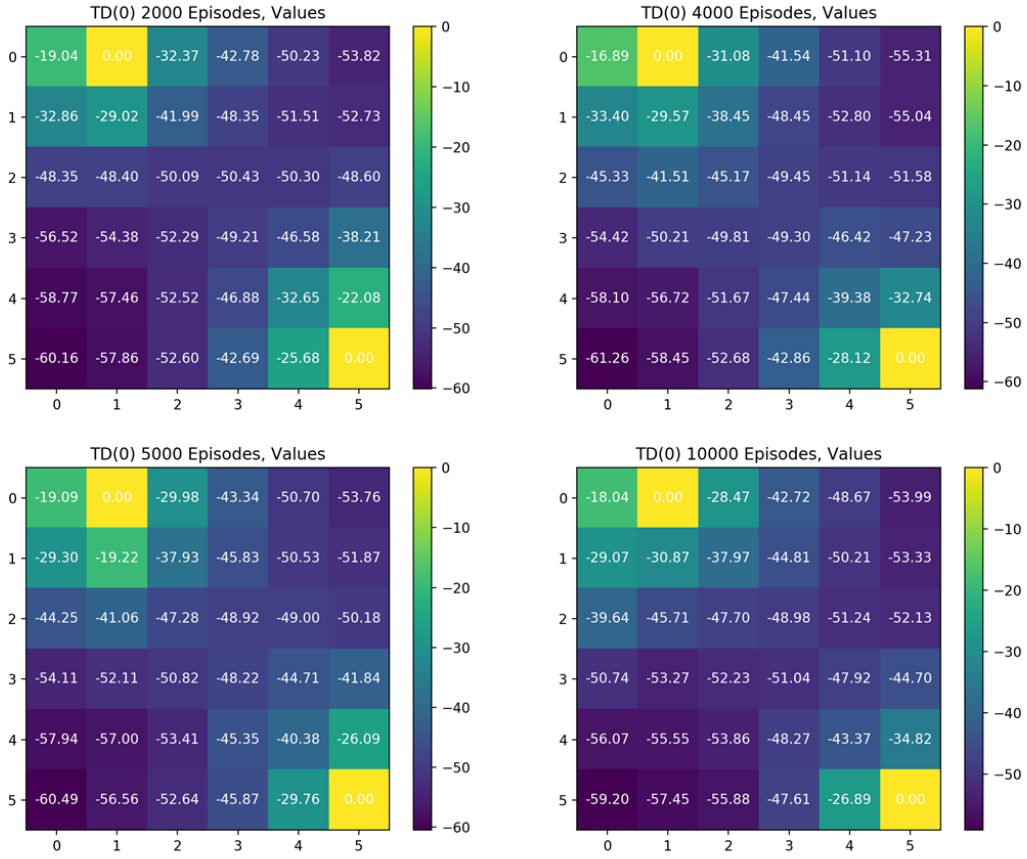

## 3.3 TD(0)

As shown in illustration, delta increases and then become stable after 1200 episodes when parameter $\alpha$ equals to 0.1. The reason why this tendency is different from that in MC method is that TD(0) method uses $\alpha[R+\gamma V(S^{'})-V(S)]$ to update value while MC method uses $\frac{G_t-V(S_t)}{N(S_t)}$ . As the number of episode increases, $\frac{G_t-V(S_t)}{N(S_t)}$ decreases and gradually approaches zero while $\alpha[R+\gamma V(S^{'})-V(S)]$ will be affected by a fixed parameter $\alpha$ and approaches some number related to $\alpha$.

Actually, parameter $\alpha$ does affect a lot and I will talk about it later in part 4.



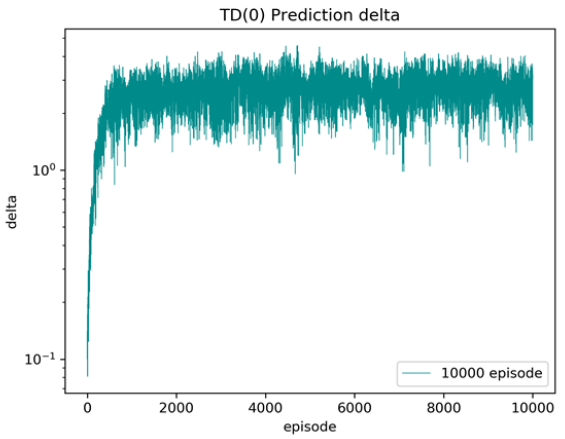

Despite that TD(0) method converges faster, its vibration is larger than MC method, as illustrations show. This largely results from the fact that $\alpha$ is too large to fit the number of episodes.

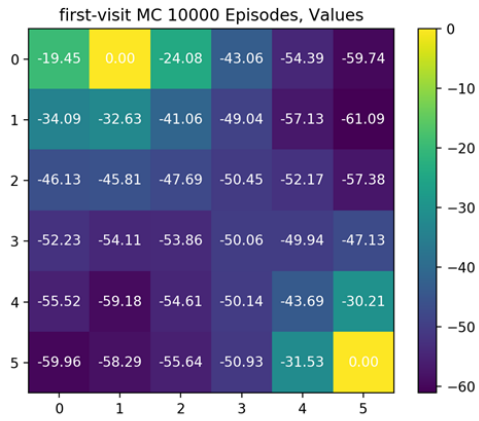


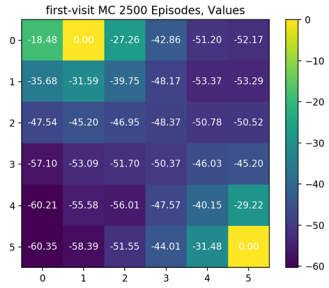

# 4 More experiments & Analysis

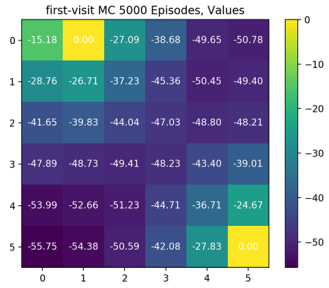
## 4.1 Use incremental mean alpha in MC

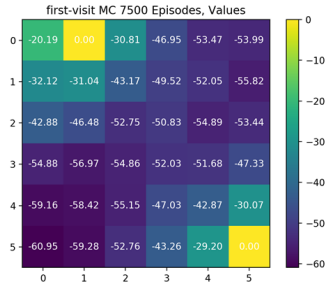
At class, we talked about updating $V(S_t)$ via $\alpha[G_t - V(S_t)]$. Here, I take first-visit MC as example and set alpha equals to 0.01. The picture of delta is similar to that in TD(0). Although it seems to converge fast in the first picture, values does fluctuate wildly and I don't think it performs well.



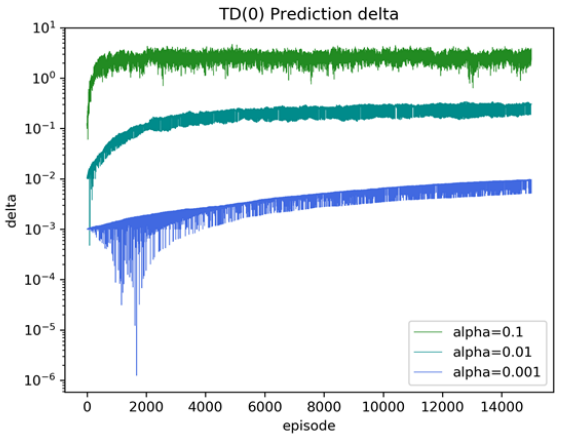



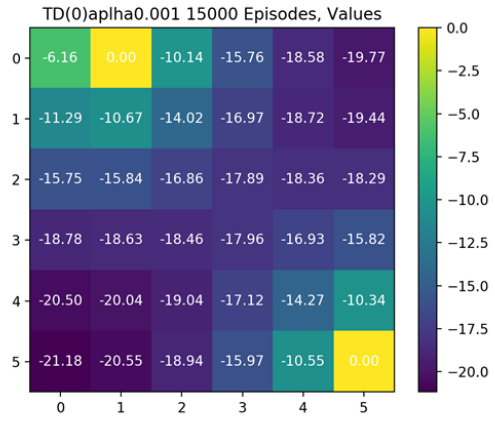

The performance of every-visit MC is similar to first-visit MC.

Therefore, we can reach the conclusion that using $\frac{G_t - V(S_t)}{N(S_t)}$ maybe better than $\alpha[G_t - V(S_t)]$

## 4.2 Alpha in TD(0)

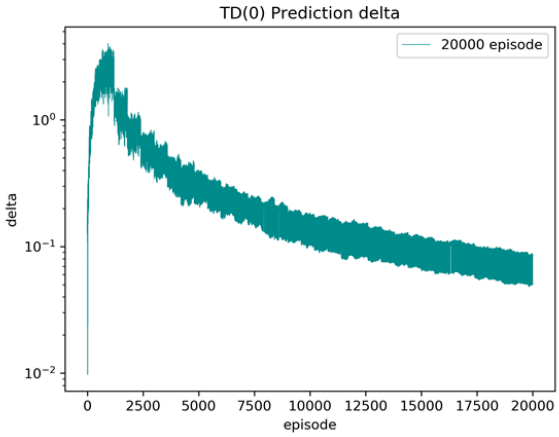
TD(0) method is sensitive to alpha. As shown in illustration, converging speed and stable delta dropped obviously with alpha decreasing. When alpha equals 0.001, it don't converge even at 20000 episodes. With alpha decreasing, the vibration of values also diminishes.

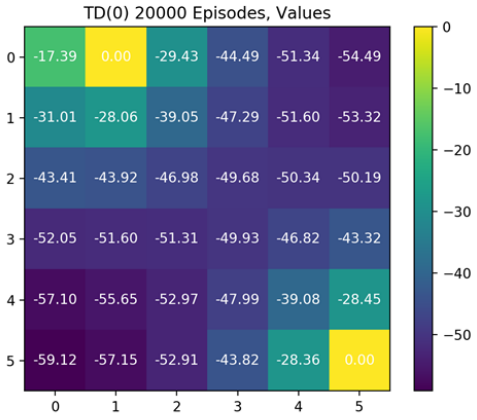


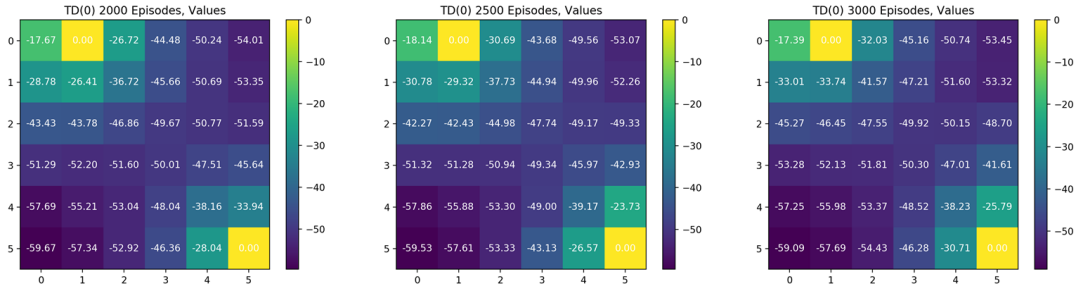

Alpha can be viewed as learning rate, showing how important the difference is. If it is large, model learns fast but may not converge or produce fluctuates greatly. If it is small, model learns slowly but has good convergence properties. Inspired by $1/N(S_t)$ used in MC method, alpha should be large if the number of episode is small and alpha should decrease as number of episode increases. So it's a good idea to diminish alpha as number of episode increases.

Here, I initialize alpha to 0.1 and diminish it to `60/total episodes` every 600 episodes after 1200 episode. As we mentioned in part 3, 1200 episode may close to convergence and diminishing alpha can reduce vibration.



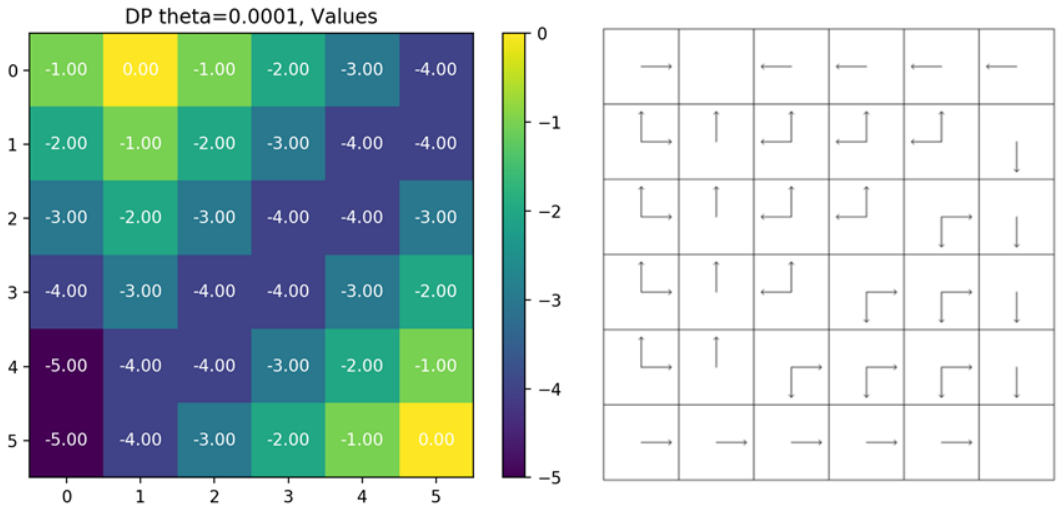

As shown in illustration, values are more stable when evaluated by dynamic alpha.



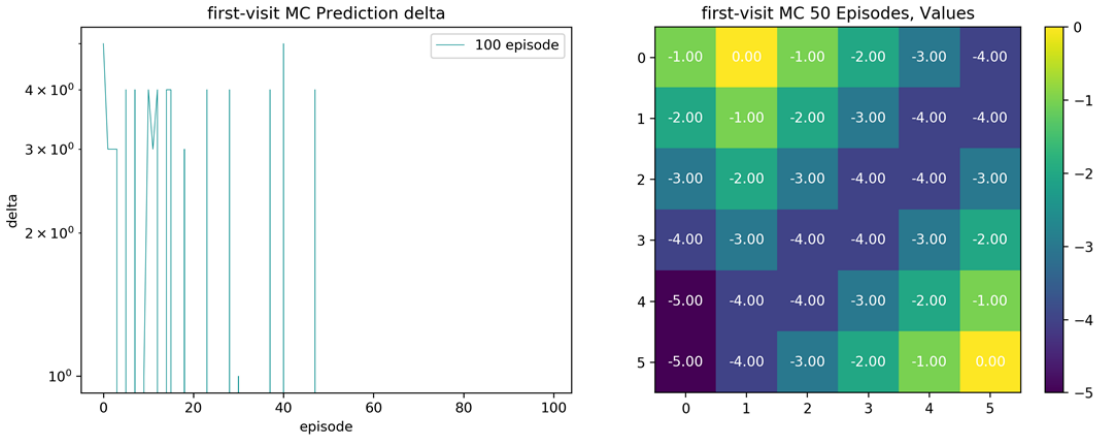

## 4.3 Evaluate optimal policy

I also evaluate other policy, like the optimal policy I got via DP method in last assignment. It's much easier to evaluate than uniform random policy.

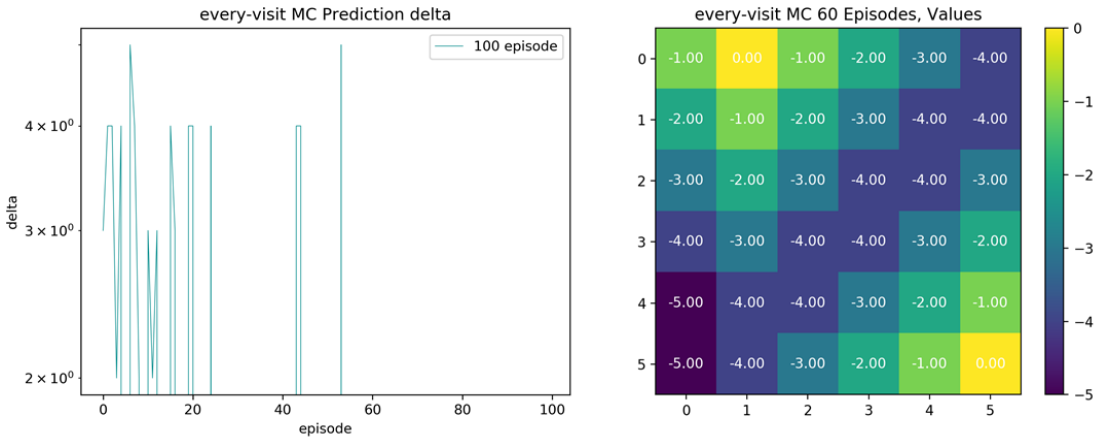
The optimal policy and values I got via DP are listed as follows.



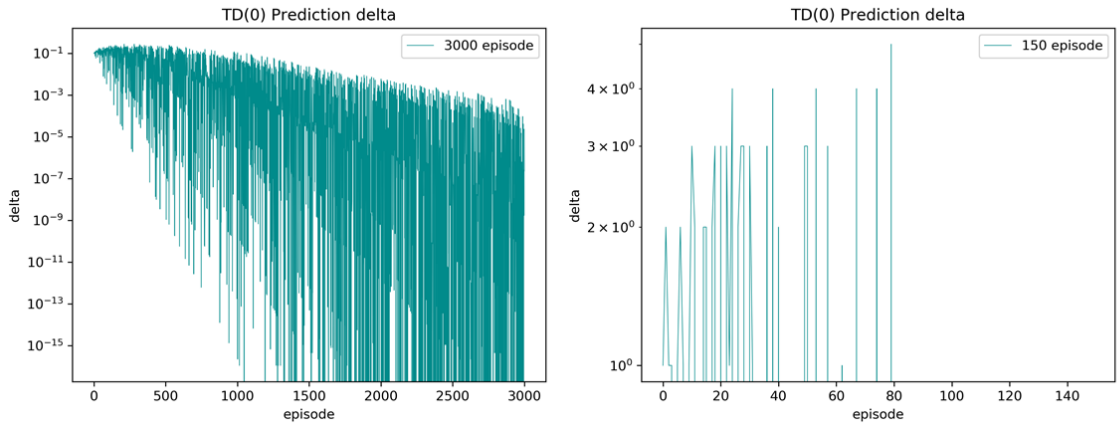

In this training, First-visited method converges to optimal values in between 40 and 50 episodes.




In this training, every-visited method converges to optimal values in between 50 and 60 episodes.




Converges speed of TD(0) method is sensitive to alpha. It takes 3000 episodes to optimal values when alpha is 0.1(left picture) while model with alpha equal 0.1 converges at 80 episodes.




Why I emphasize "in this training" is that the episodes I sampled are random and may be quite different from training to training, which may result in different converging speed.

## 4.4 Compare converging speed among three method

There are two reason why I don't compare those three method in above parts. First, when evaluating uniform random policy, it's hard to say when they converge. Second, they use different episodes which is random sampled and get different information, which make it hard to compare their performance in policy east to evaluate. It's better to compare batch MC and TD on same

finite experience.

## 4.4 Initial value sensitivity

As shown in slides at class, TD(0) is more sensitive to initial value, while MC not.

I wonder whether initial value contains initial terminal state value. If so, the result of MC and TD(0) will all change. If not, I don't think the result will change, so the sensitivity may come from converging speed. Given the reason I list in part 4.4, It's hard to compare converging speed under this setting.