ENGG*6600-01 ST: Reinforcement Learning

Programming Assignment Report

Submitted by Debanjan Mitra

The Easy21 rules: -

First let's recall the rules of Easy21's game. A player (me) is playing against a dealer. An unlimited deck of cards (from 1 to 10 and red or black coloured) is available: red cards account for negative numbers whereas black cards account for positive ones.

At the beginning of the game, the dealer picks a black card: he starts off with a number between 1 and 10. Then, it's my turn to pick cards until I lost or decide to stop picking cards (this action is called "stick"). I lose if the sum of my cards is smaller than 1 and greater than 21. When I decide to stick, it is dealer's turn to pick. The rules are the same for him.

However, the dealer's policy is known: when his sum is smaller than 17, he continues to pick but if his sum is greater or equal to 17, he sticks (i.e. the game finishes). When the dealer sticks, his sum and mine will be compared: the winner is the one with the largest sum.

1) Implementation of Easy21 Environment

The environment of the Easy21 game is implemented using the following concept: -

1) First, we are initializing the Easy21 class with min and max values of both the Player and the Dealer

```
    def __init__(self):
        self.minCardValue, self.maxCardValue = 1, 10
        self.dealerUpperBound = 17
        self.gameLowerBound, self.gameUpperBound = 0, 21
```

2) We are defining the action space which returns a value of 0 for Stick and 1 for Hit.

```
    def actionSpace(self):
        return (0, 1)
```

3) We are initializing the game by generating random values between 0 and 21, 1 and 10.

```
    def initGame(self):
        return (np.random.randint(self.minCardValue, self.maxCar
dValue+1),
               np.random.randint(self.minCardValue, self.maxCard
Value+1))
```

4)We are defining the deck based on the following

a) Each draw from the deck results in a value between 1 and 10 (uniformly distributed) with a colour of red (probability 1/3) or black (probability 2/3).

b) The game is played with an infinite deck of cards (i.e. cards are sampled with replacement).

c) There are no aces or picture (face) cards in this game.

```
    def draw(self):
        value = np.random.randint(self.minCardValue, self.maxCar
dValue+1)

        if np.random.random() <= 1/3:
            return -value
        else:
            return value
```

5) A function named "step" is defined, which takes three inputs (first one is playerValue which is between 1 and 21, second one is dealerValue which is between 1 and 10, and an action a (stick or hit), and then it returns the sample of next state (which may be terminal if the game is finished) and a reward.

a) At the start of the game both the player and the dealer draw one black card (fully observed)
b) Each turn the player may either stick or hit
c) If the player hits, then she draws another card from the deck
d) If the player sticks, she receives no further cards
e) The values of the player's cards are added (black cards) or subtracted (red cards)
f) If the player's sum exceeds 21, or becomes less than 1, then she "goes bust" and loses the game (reward -1)

```
def step(self, playerValue, dealerValue, action):

  assert action in [0, 1], "Expection action in [0, 1] but got %
i"%action

  if action == 0:

    playerValue += self.draw()

    # check if player busted
    if not (self.gameLowerBound < playerValue <= self.gameUpperB
ound):
        reward = -1
        terminated = True

    else:
        reward = 0
        terminated = False

  elif action == 1:
      terminated = True

      while self.gameLowerBound < dealerValue < self.dealerUppe
rBound:
            dealerValue += self.draw()

      # check if dealer busted // playerValue greater than deal
erValue
      if not (self.gameLowerBound < dealerValue <= self.gameUpp
erBound) \
               or playerValue > dealerValue:
               reward = 1

      elif playerValue == dealerValue:
               reward = 0

      elif playerValue < dealerValue:
               reward = -1

     return playerValue, dealerValue, reward, terminated
```

This environment will be used for model-free reinforcement learning, hence the MDP transition matrix should not be explicitly represented. The dealer's actions should be treated as a component of the environment; for example, calling step with a stick action will play out the dealer's cards and return the ultimate prize and terminal state.

2) Monte Carlo Control

Monte Carlo simulation is a type of model-free prediction in which there are no pre-existing Markov Decision Processes (MDP) for transitions/rewards. MDPs formally define a situation in which the state has complete control over the environment. MC approaches use k episodes of experience to learn directly. A game is a game in Easy21. The fundamental concept is that the value of a state/action is equal to the empirical mean of the total discounted return.

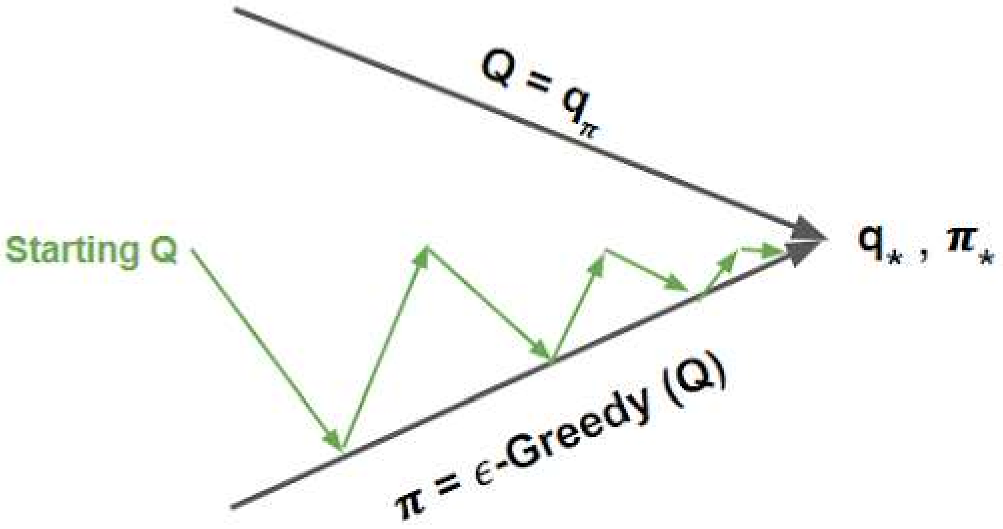
We aim to learn the action-value function Q(s,a) from experience episodes under policy, pi. To accomplish this, we will approximate and determine our best policy using Monte Carlo estimation in control. The objective is to analyse and improve our policy on a regular basis in order to improve our Q estimation.

The MCPE pseudo-code is:

```
for a big number of steps:
   Run one episode and store the states you've visited and the rewards you got.
   for every time-step t:
      Increment counter: N(s) ← N(s) + 1
      Increment total return: G_s(s) ← G_s(s) + Gt
      Value is estimated by mean return V(s) = G_s(s)/N(s)
```

The value function will get close to the real value function as N(s) → ∞

We have action = argmaxa in A(Q(s,a)) with an exploration threshold given by epsilon, which means we choose an action at random with some probability epsilon. We run through a complete episode with MC and then update our Q using the outcome reward. Our policy has been updated as a result of this change. We finally converge to our ideal policy and value function by altering our policy and evaluating after every episode.



For this, we'll employ e-greedy exploration, which is one of the most fundamental concepts for ensuring continuous exploration and is defined as follows:

Policy(a | st) = { probability 1 – e to choose the greedy action (the best we know, maxQ(st,a)),

```
                probability e choose random action
        }
```

Now that we have our policy, let's look at the MCPC pseudo-code. By sampling episodes and examining the environment, we can enhance our value function.

```
For a big number of steps:
 Run one episode and store the states you've visited and the rewards you got.
 for every time-step t:
     Increment counter: N(St) ← N(St) + 1
     error ← (Gt − Q(St, At))
     Q(St, At) ← Q(St, At) + αt * error
```

For this exercise:

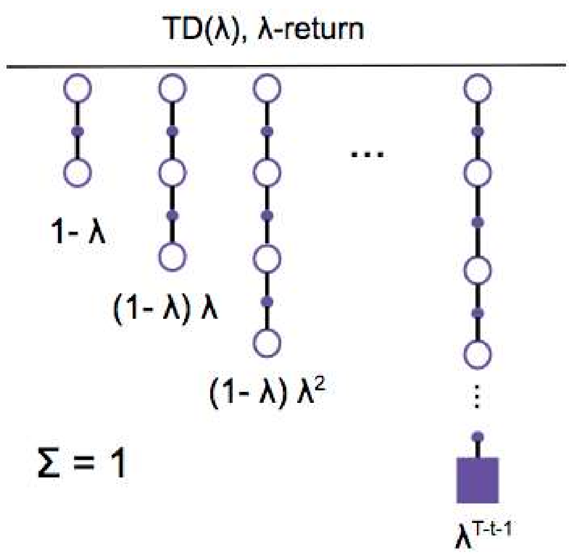
e-greedy exploration strategy with e = N0/(N0 + N(st)), where N0 = 100 is a hyper-parameter, N(s) is the number of times that state s has been visited, and N(st) is the number of times that the state s has been visited.

Initialise the value function to zero.

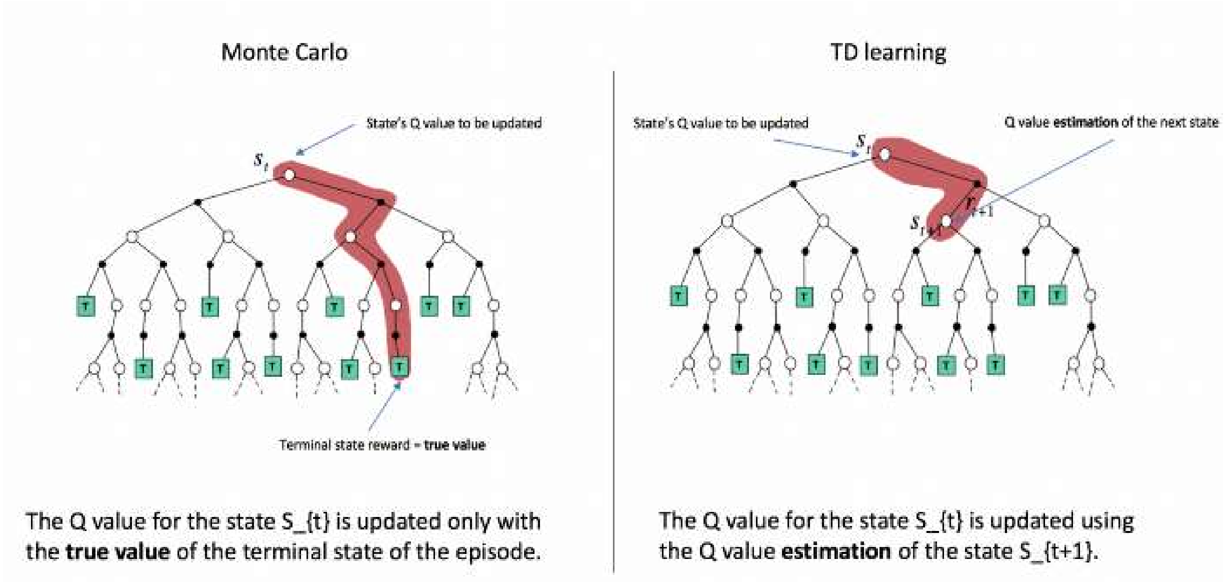
Use a time-varying scalar step-size of αt = 1/N(st, at).

Sarsa λ TD(λ) Control

Temporal difference learning is a model-free prediction method that bootstraps estimates of the action-value function Q and updates them at each time step. To acquire the cumulative results from a Monte Carlo simulation, you have to wait until the episode is finished. Gt then calculates the value of a state by averaging the returns and updates Q. TD() adjusts values based on bootstrapped estimations, so it doesn't have to wait for the final result to make changes. The averaging of n-backups and weights is called TD().



As a result, the Q value is updated on the fly, and SARSA begins learning from partial episodes rather than waiting until the end of the sampled episode, as MCC does (the Figure below illustrates this learning procedure).


Monte Carlo
State's Q value to be updated
s_t
T
Terminal state reward = **true value**
TD learning
State's Q value to be updated
Q value **estimation** of the next state
s_t
r_{t+1}
s_{t+1}

The Q value for the state S_{t} is updated only with the **true value** of the terminal state of the episode.

The Q value for the state S_{t} is updated using the Q value **estimation** of the state S_{t+1}.

Sarsa λ is a TD control method that leverages the utilisation of an eligibility trace to estimate a state's value. The amount of credit/balance you should assign to each state/action is determined by an eligibility trace. The fundamental notion here is that when we give an update, we want to focus more on the recent actions that have brought us to this point. This is the same as stating we want to make a forecast about the future based on more recent data. We update our prediction whenever fresh information becomes available.

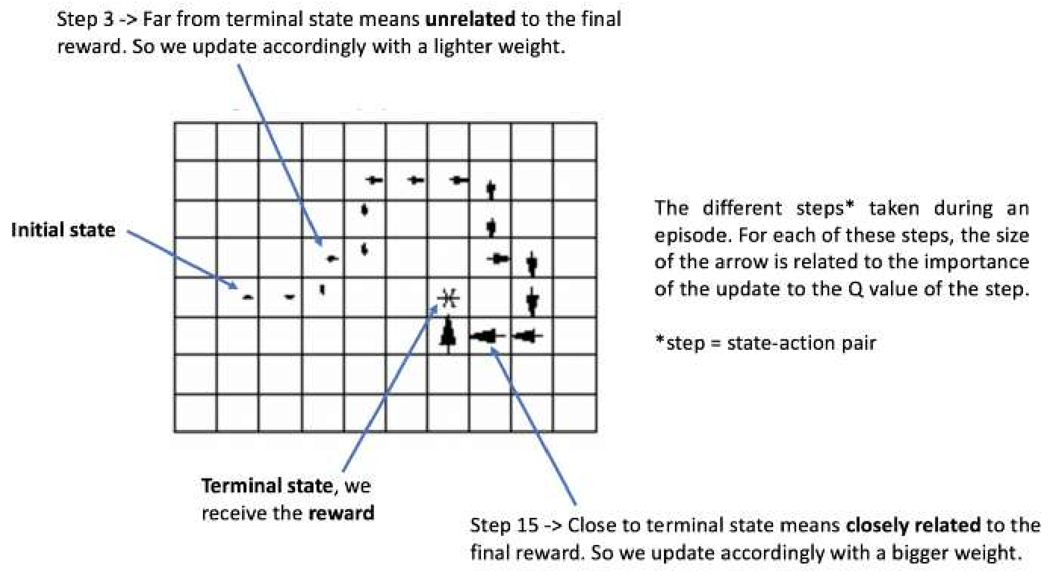
$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E_t(s,a)$$

The Pseudo-Code of Sarsa (λ) TD Learning is as follows: -

```
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all $s, a$
Repeat (for each episode):
    Initialize $s$, $a$
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all $s, a$:
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        $s \leftarrow s'$; $a \leftarrow a'$
    until $s$ is terminal
```

Eligibility Traces (ET): With ET, we want to weight the Q value updates of every state-action combination based on how frequently AND how long ago they were seen at a certain point in the episode.

The trace factor, often known as the λ parameter, determines how quickly the trace fades away. The λ parameter ranges from 0 to 1, with 0 indicating faster fading and 1 indicating slower fading. The Grid world example in Figure below illustrates this principle.


Step 3 -> Far from terminal state means **unrelated** to the final reward. So we update accordingly with a lighter weight.
Initial state
The different steps* taken during an episode. For each of these steps, the size of the arrow is related to the importance of the update to the Q value of the step.
*step = state-action pair
Terminal state, we receive the **reward**
Step 15 -> Close to terminal state means **closely related** to the final reward. So we update accordingly with a bigger weight.

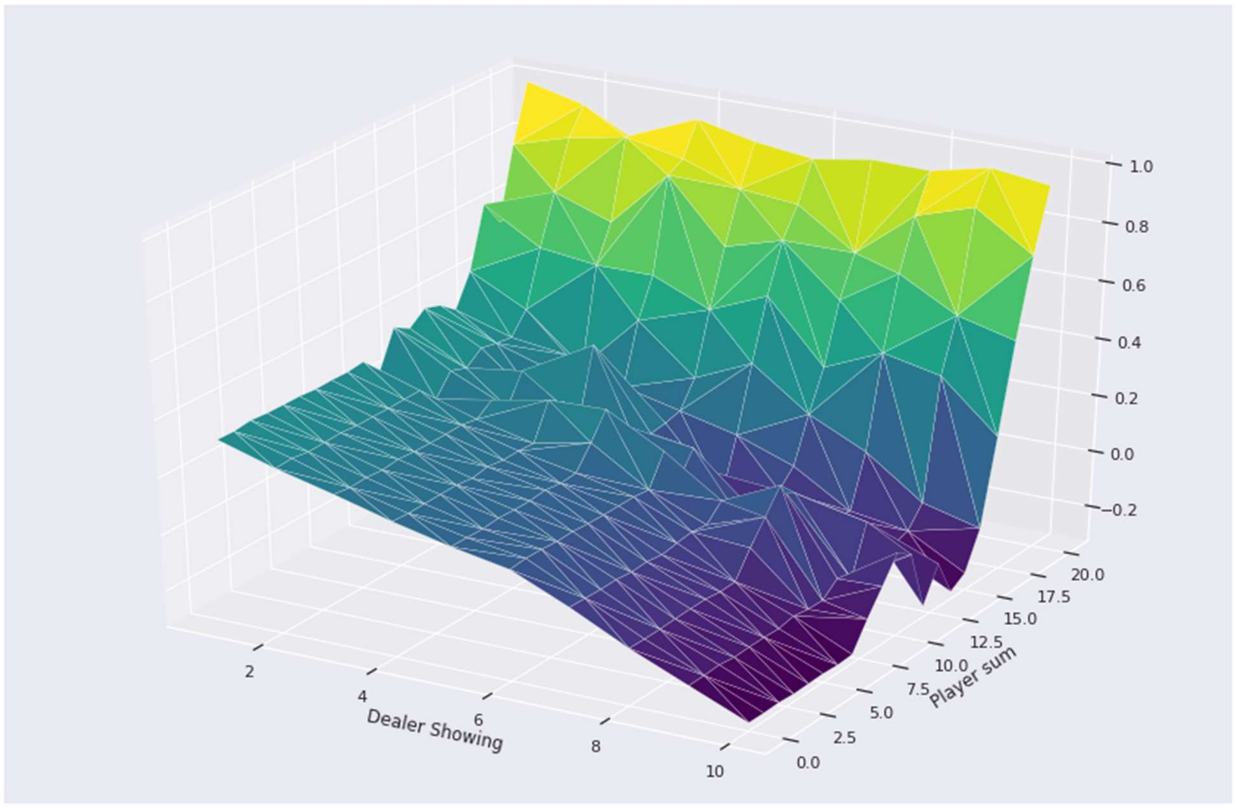
The eligibility trace vector E is used to accomplish this mathematically (s,a). Each state-action pair's decay factor is stored in E(s,a). The eligibility trace is applied to our TD error, dt, which is the difference between our previous and current estimates of Q.

Instead of choosing between TD learning and Monte-Carlo sampling, SARSA(λ) uses the trace factor λ, which scales from 0 to 1, to mix the two concepts. If λ =1, we return to the original MCC strategy (as discussed in section 2), and if λ = 0, we perform a one-step TD learning update. It's our responsibility to locate and select the optimum λ to allow the agent to learn the optimal policy as quickly as possible without compromising its learning.

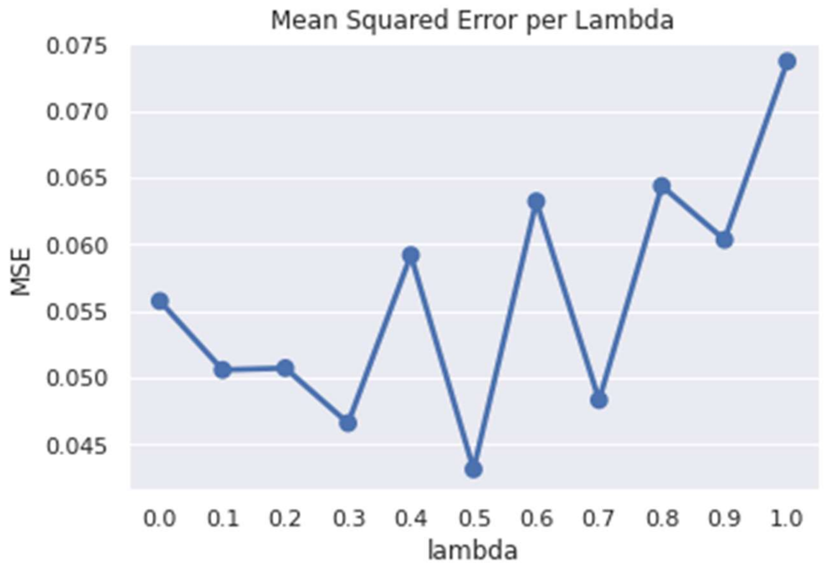
Due to its capacity to learn from incomplete episodes, an agent learning with SARSA does not need to sample as many episodes as a Monte-Carlo agent. This is particularly beneficial when episodes are really long and sampling too many of them becomes computationally prohibitive.

Result and Discussion

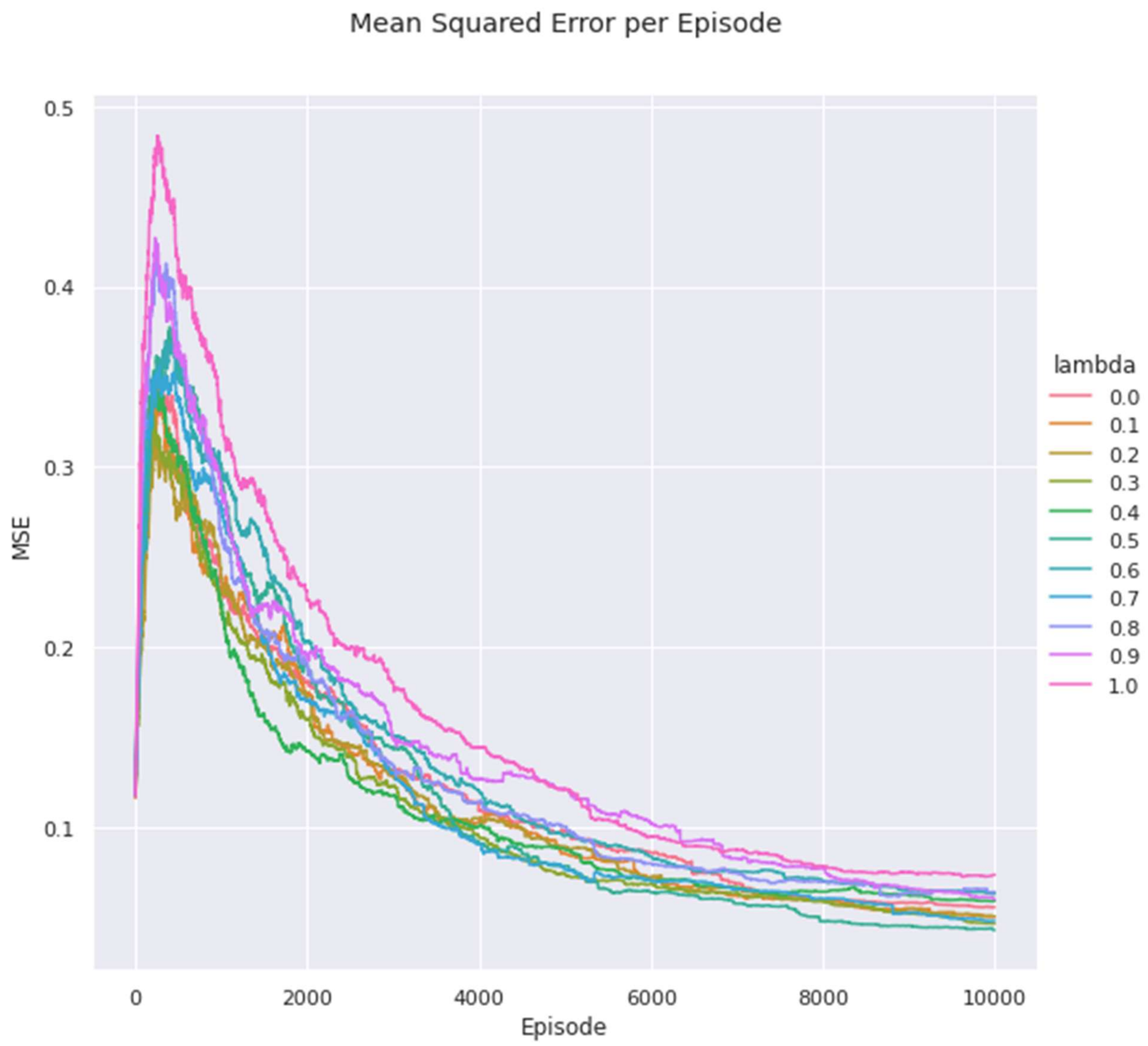
The game's 10 million episodes were analysed to produce the following Value function: The optimal Q value function becomes smoother as the number of samples increases. It's worth noting that states with a player sum close to 21 have a value of about 1. Furthermore, as the value of the dealer card rises, so does the value of the states. While the particular trend is obscured by noise and difficult to discern, it appears that when player sum equals 11, there is a local maximum. This is illustrated by the fact that if the player's total is 11, he or she cannot go bust in the next draw.



The graph below presents changes of mean-squared error for different values of lambda: 0, 0.1, 0.2, ..., 1. For each value, 1000 episodes have been evaluated. We see that we get the lowest MSE at λ = 0.5 and the highest is obtained at λ = 1.0



The MSE curves between ideal Q value and computed Q value, as well as the number of sampled episodes and different values, are depicted in the picture below. In comparison to the standard MCC approach (=1), the TD learning strategy (=0) requires fewer episodes to converge to the ideal Q value function.



References

[1] David Silver's Introduction to Reinforcement Learning

[2] https://pierrealexsw.github.io/2018/09/21/Easy21/

[3] https://towardsdatascience.com/playing-cards-with-reinforcement-learning-1-3-c2dbabcf1df0

[4] https://github.com/hereismari/easy21/blob/master/easy21.ipynb