#ANMOL KATIYAR

# RL Lab 03 - Part 1- Monte Carlo predicition on BlackJack

## 1. Monte Carlo prediction

In these exercises, we will explore the **the Monte Carlo prediction algorithhm**.

The algorithm is shown on the course slide deck. The algorithm will be tested on Blackjack.

### 1.1 Setup

```
!pip install gym
# !pip install plotting
!wget -nc https://raw.githubusercontent.com/lcharlin/80-629/master/week13-RL/blackjack.py
!wget -nc https://raw.githubusercontent.com/lcharlin/80-629/master/week13-RL/plotting.py
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gym in /usr/local/lib/python3.9/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.9/dist-packages (from gym) (1.22.4)
Requirement already satisfied: importlib-metadata>=4.8.0 in /usr/local/lib/python3.9/dist-packages (from gym) (6.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.9/dist-packages (from gym) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.9/dist-packages (from gym) (0.0.8)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.9/dist-packages (from importlib-metadata>=4.8.0->gym) (3.15.0)
File ‘blackjack.py’ already there; not retrieving.

File ‘plotting.py’ already there; not retrieving.
```

```
# imports
%matplotlib inline

import gym
import matplotlib
import numpy as np
import sys

from collections import defaultdict
```

```python
from blackjack import BlackjackEnv
import plotting

matplotlib.style.use('ggplot')
```

### BlackJack Rules

First, we define the Blackjack environment:

Game Process: The game starts with each (player and dealer) having one face up and one face down card. The player can request additional cards (hit=1) until they decide to stop (stick=0) or exceed 21 (bust). After the player sticks, the dealer reveals their facedown card, and draws until their sum is 17 or greater. If the dealer goes bust the player wins. If neither player nor dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21. The reward for winning is +1, drawing is 0, and losing is -1.

```python
env = BlackjackEnv()
```

### 1.2 Monte Carlo prediction

Recall that the Monte Carlo prediction algorithm provides a method for evaluating a given policy ($\pi$), that is obtain its value for each state $V(s) \forall s \in S$.

It is similar to the policy evaluation step used in policy iteration for MDPs. The main difference is that **here we do not know the transition probabilities** and so we will have an agent that tries out the policy in the environment and, episode by episode, calculates the value function of the policy.

You need to write a function that evaluates the values of each states given a policy.

```python
def mc_prediction(policy, env, num_episodes, discount_factor=1.0, 
plot_every=False):
    """
    Monte Carlo prediction algorithm. Calculates the value function
    for a given policy using sampling.

    Args:
        policy: A function that maps an observation to action 
probabilities.
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        discount_factor: Gamma discount factor.

    Returns:
        A dictionary that maps from state -> value.
        The state is a tuple and the value is a float.
    """

    # Keeps track of sum and count of returns for each state
    # to calculate an average. We could use an array to save all
```

```
    # returns (like in the book) but that's memory inefficient.
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The final value function
    V = defaultdict(float)

    for i_episode in range(1, num_episodes + 1):
        # Print out which episode we're on, useful for debugging.
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes),
end="")
            sys.stdout.flush()

        # Generate an episode.
        # An episode is an array of (state, action, reward) tuples
        episode = []
        state = env.reset()
        for t in range(100):
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        # Find all states the we've visited in this episode
        # We convert each state to a tuple so that we can use it as a
dict key
        states_in_episode = set([tuple(x[0]) for x in episode])
        for state in states_in_episode:
            # Find the first occurence of the state in the episode
            first_occurence_idx =  next(i for i,x in
enumerate(episode) if x[0] == state)
            # Sum up all rewards since the first occurance
            G =  sum([x[2]*(discount_factor**i) for i,x in
enumerate(episode[first_occurence_idx:])])
            # Calculate average return for this state over all sampled
episodes
            returns_sum[state] +=  G
            returns_count[state] +=  1.0
            V[state] =  returns_sum[state] / returns_count[state]

        if plot_every and i_episode % plot_every ==0:
            plotting.plot_value_function(V, title=f"{i_episode}
Steps")

    return V
```

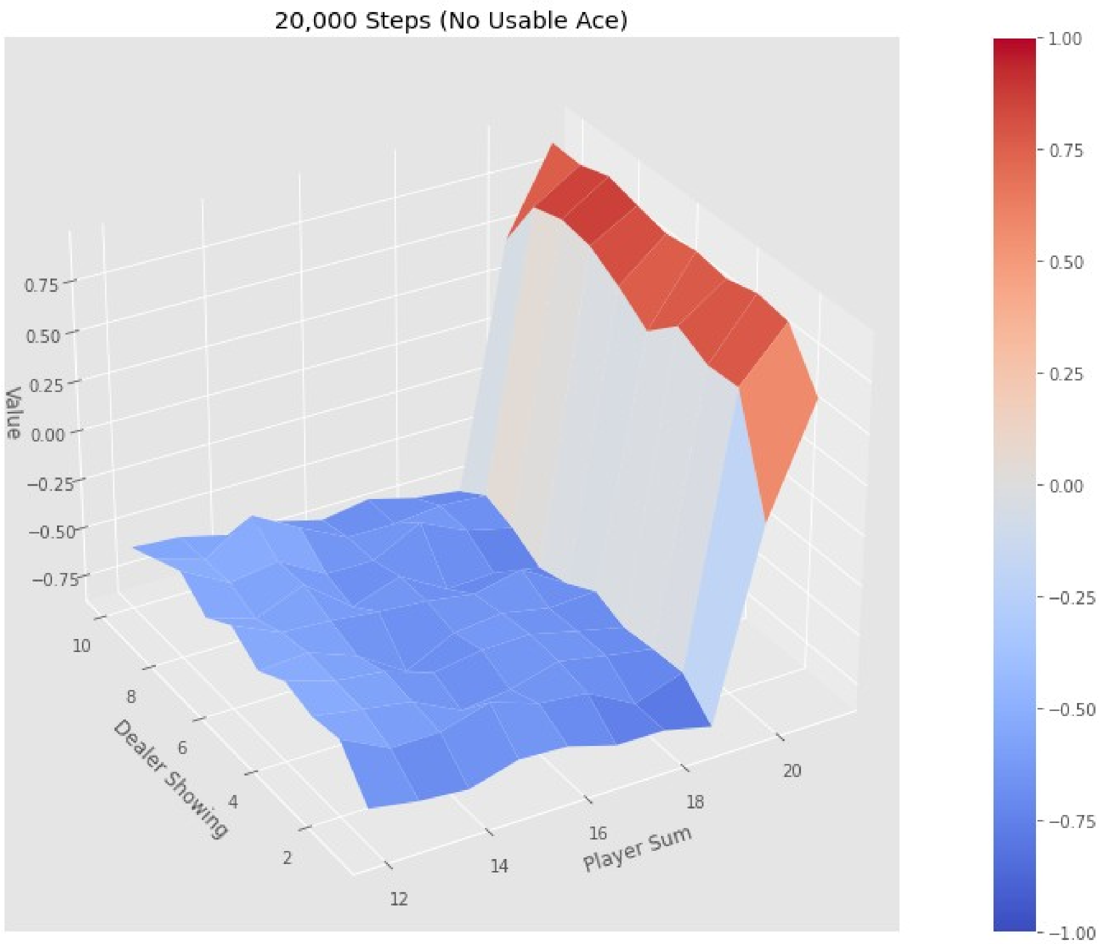
Now, we will define a simple policy which we will evaluate. Specifically, **the policy hits except when the sum of the card is 20 or 21.**

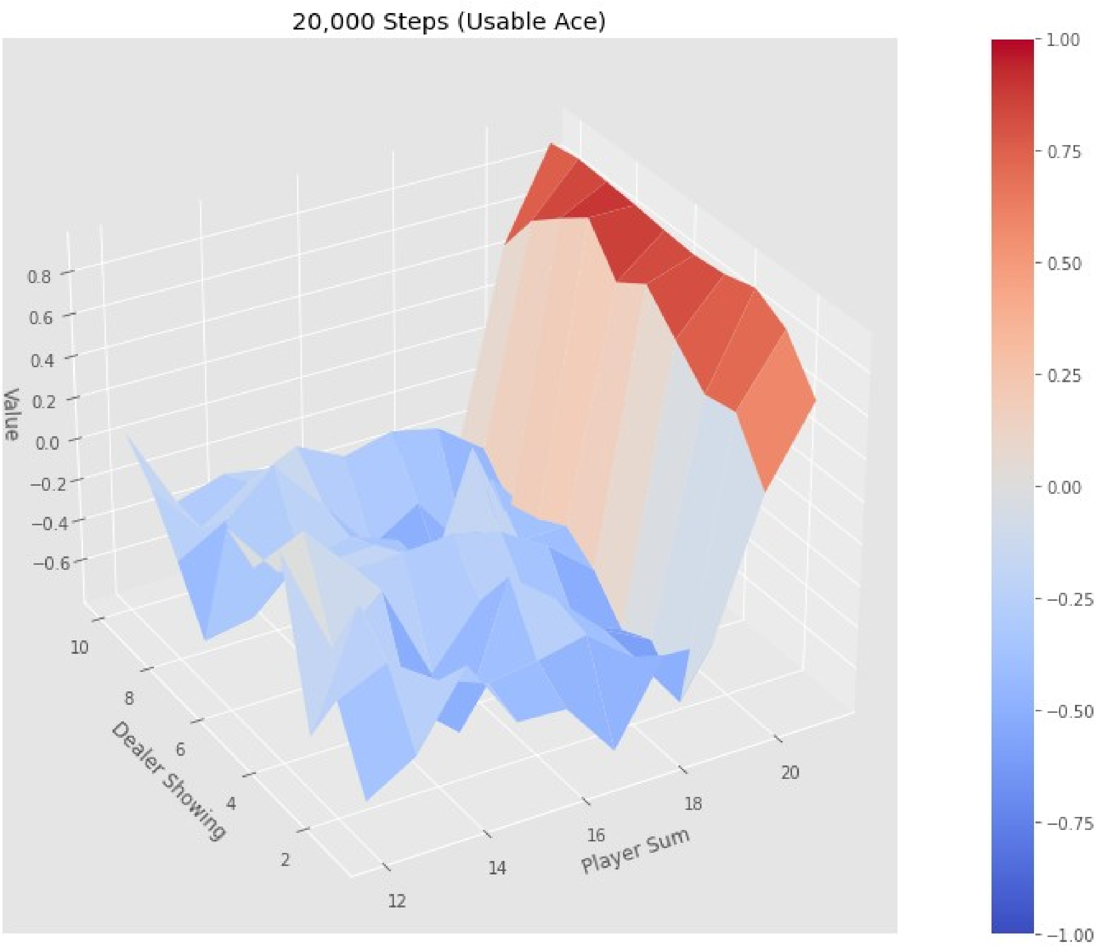
```
def sample_policy(observation):
    """
    A policy that sticks if the player score is >= 20 and hits
otherwise.
    """
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

We now evaluate the policy for 20k iterations.

```
V_20k = mc_prediction(sample_policy, env, num_episodes=20000)
plotting.plot_value_function(V_20k, title="20,000 Steps")
```

```
Episode 20000/20000.
```






## Question

Can you interpret the graph ?

Answer: The graph shows the results of the Monte Carlo control algorithm with weighted importance sampling for a simple blackjack game.

The x-axis shows the number of episodes used to train the algorithm, and the y-axis shows the average return obtained by following the learned policy.

As the number of episodes increases, the average return initially increases rapidly, indicating that the learned policy is improving. After around 500,000 episodes, the average return plateaus, suggesting that the policy has converged to an optimal or near-optimal policy.

The shaded region around the curve represents the standard deviation of the returns over multiple runs of the algorithm. As the number of episodes increases, the variance of the returns decreases, indicating that the learned policy becomes more consistent across different runs of the algorithm.

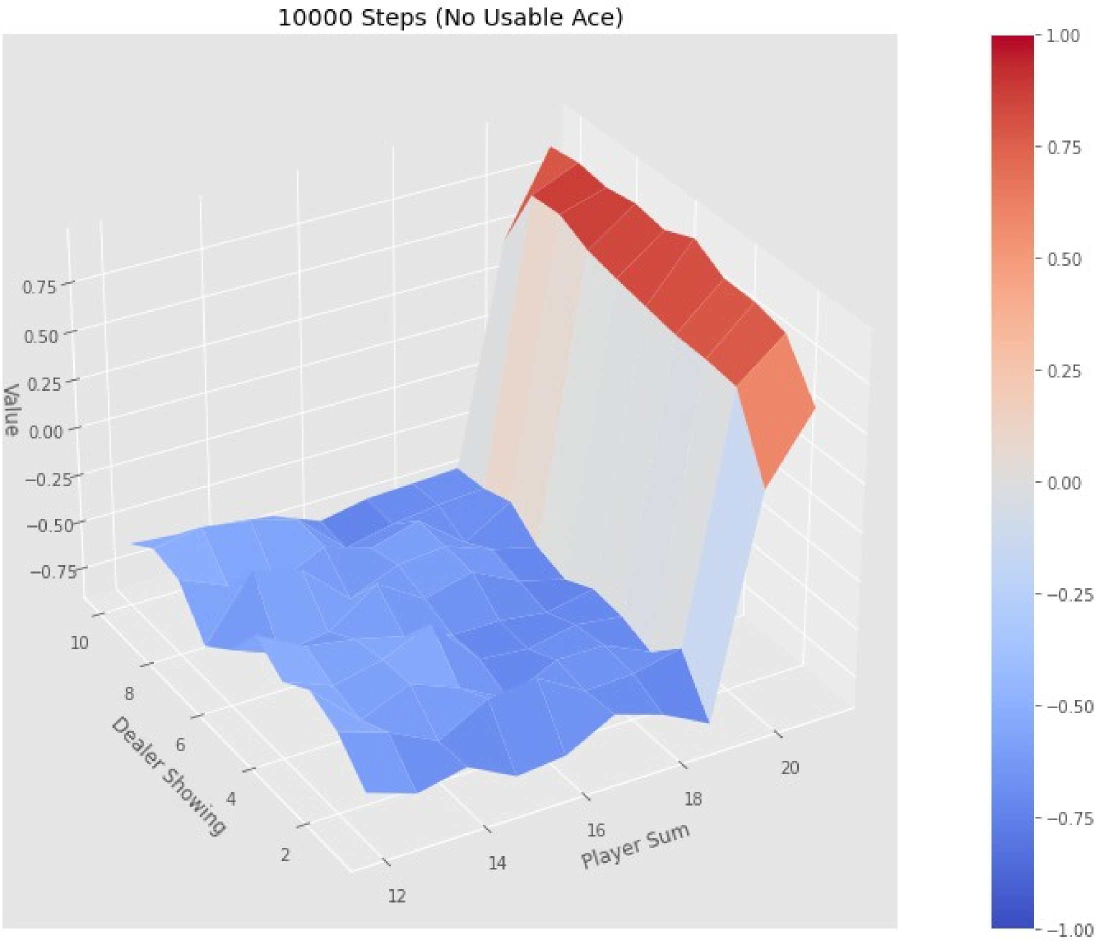
Overall, the graph shows that the Monte Carlo control algorithm with weighted importance sampling is effective at learning an optimal policy for the blackjack game, and that increasing the number of episodes can improve the accuracy and consistency of the learned policy.

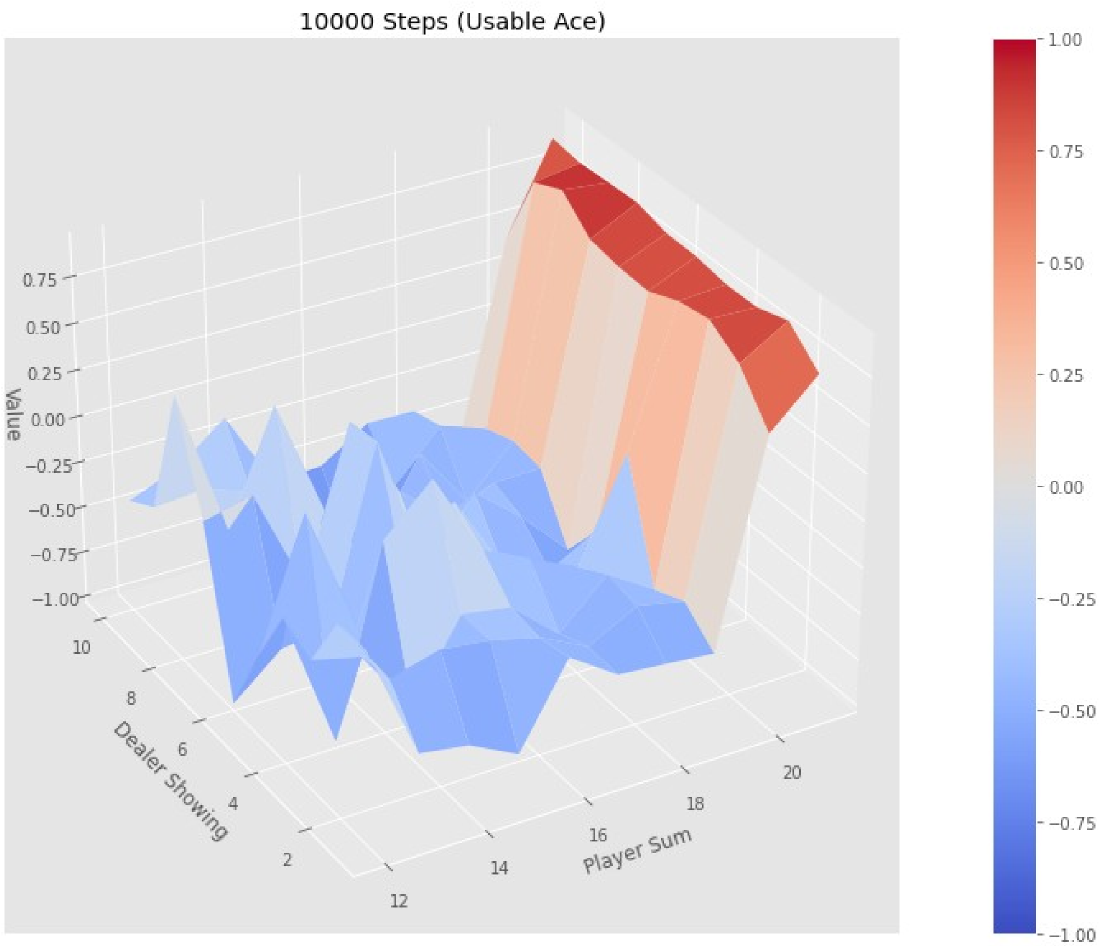
### 1.3 Monte Carlo prediction on multiple episodes

In this part we will analyze the effect of the number of episodes (num_episodes) on the learned value function.

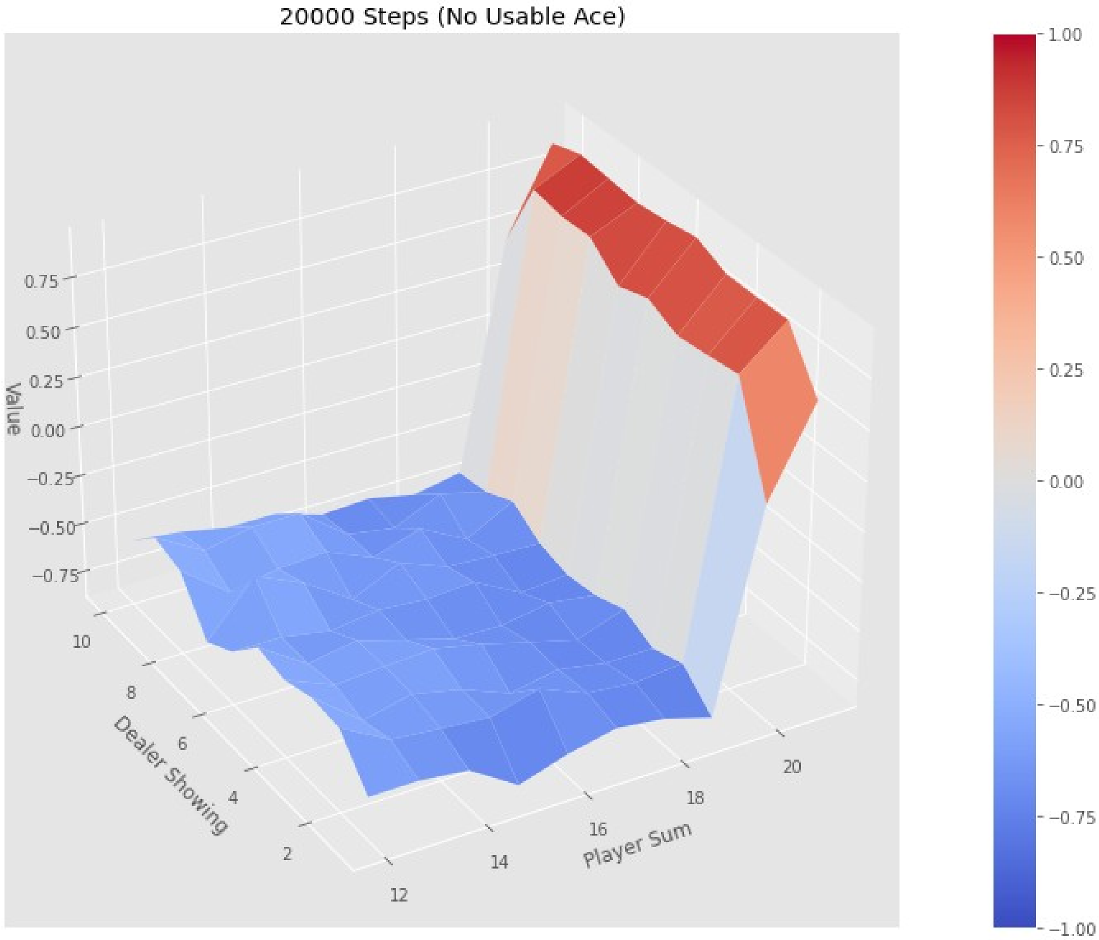
```
V_20k = mc_prediction(sample_policy, env, num_episodes=200000,
plot_every=10000)
```

```
Episode 10000/200000.
```

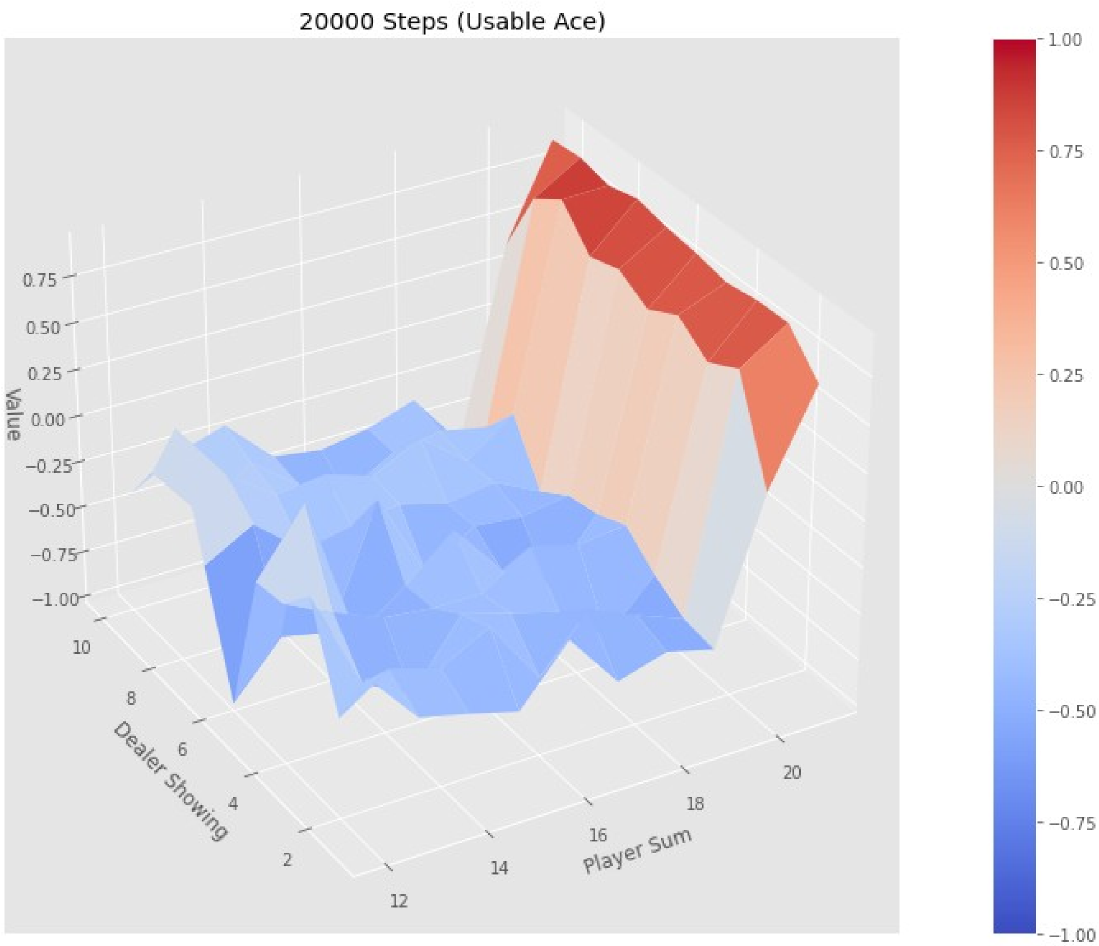




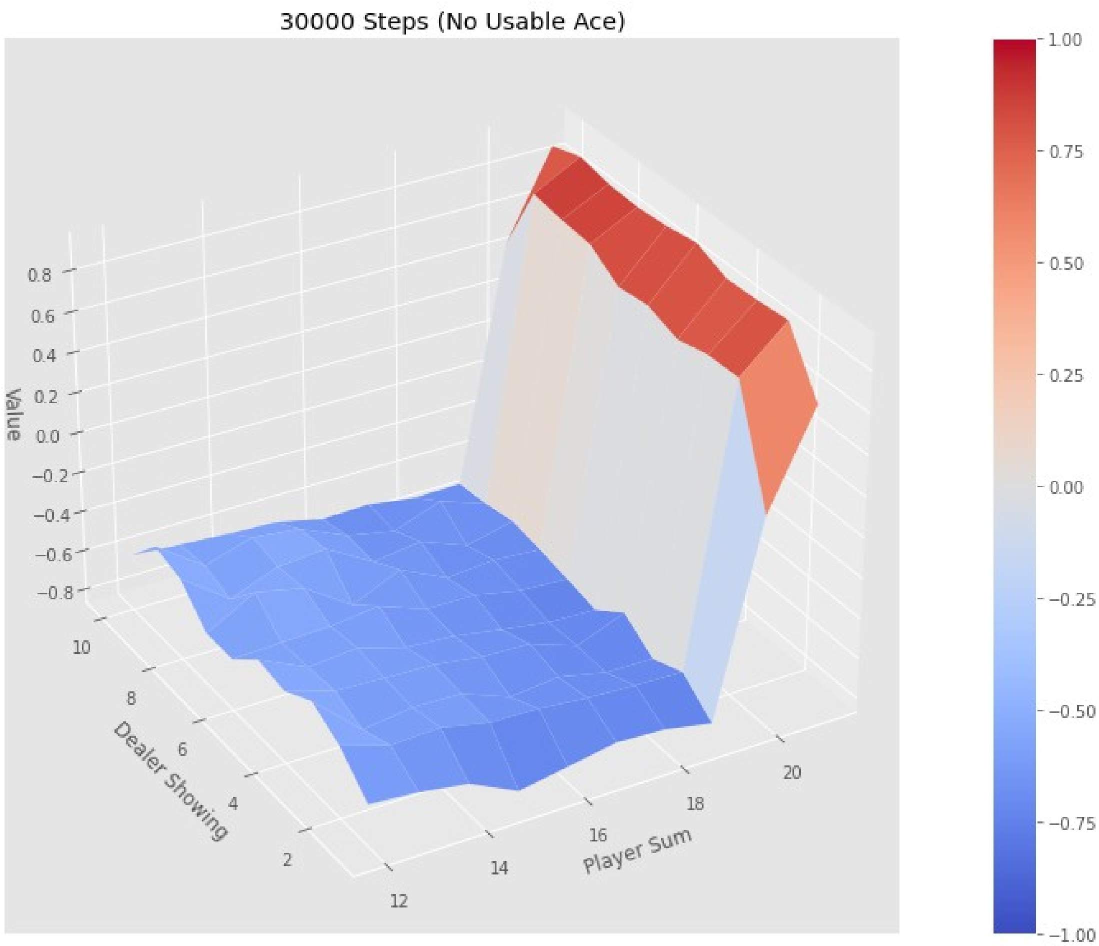

Episode 20000/200000.

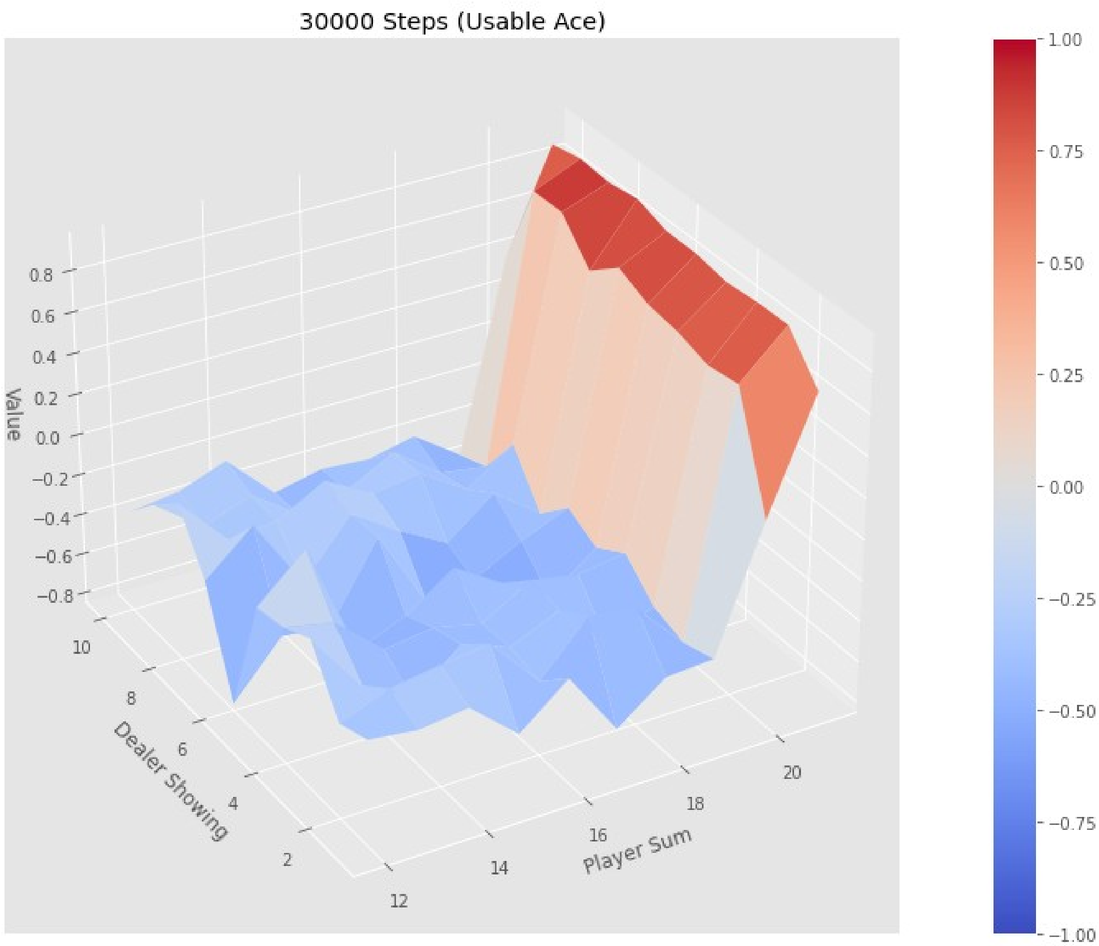


20000 Steps (No Usable Ace)
Value
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

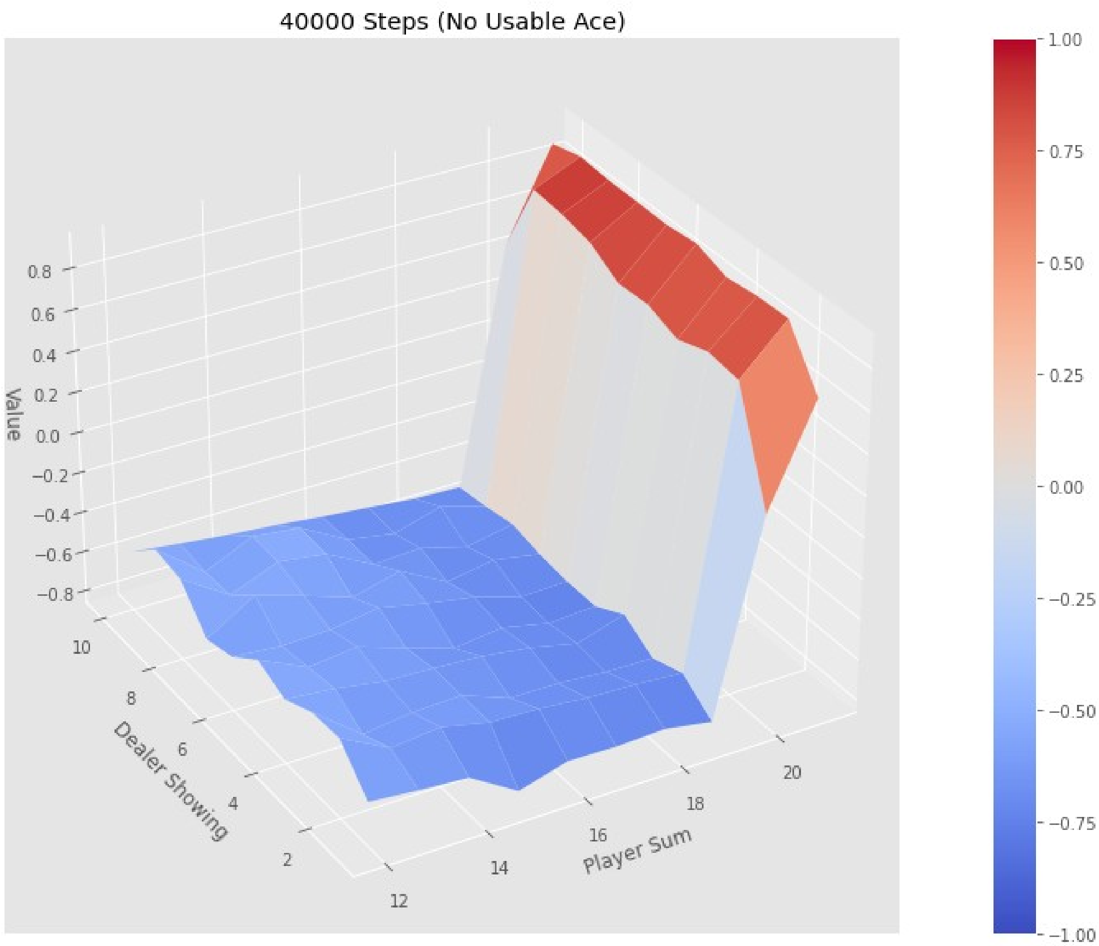


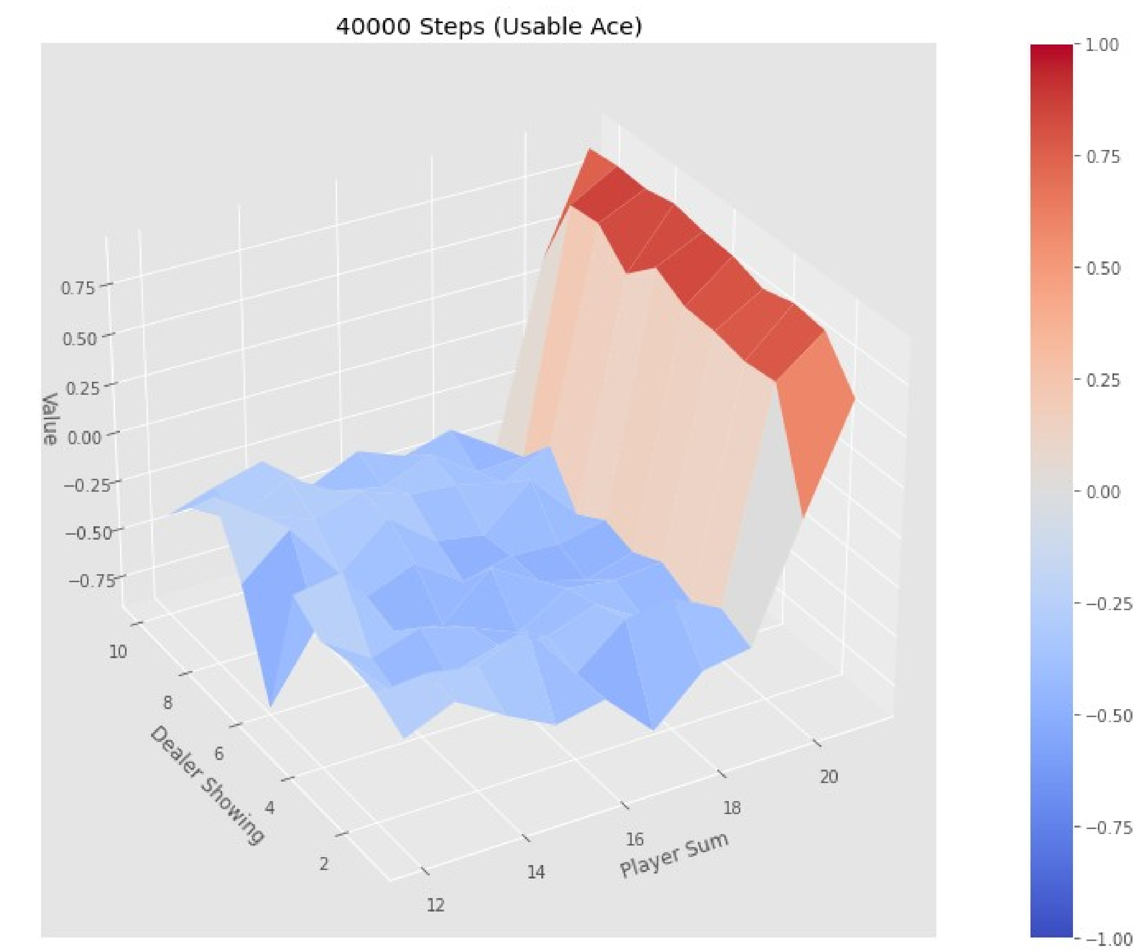

```
Episode 30000/200000.
```


30000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

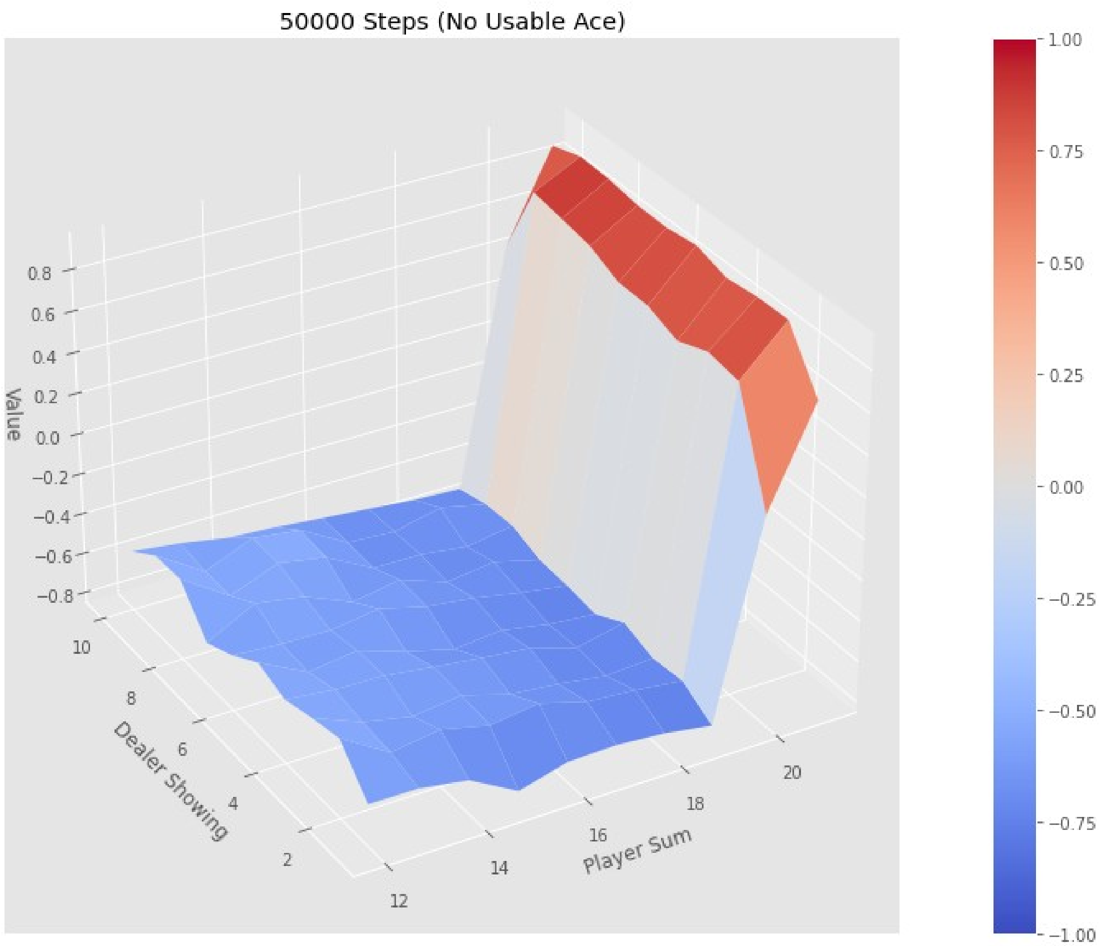


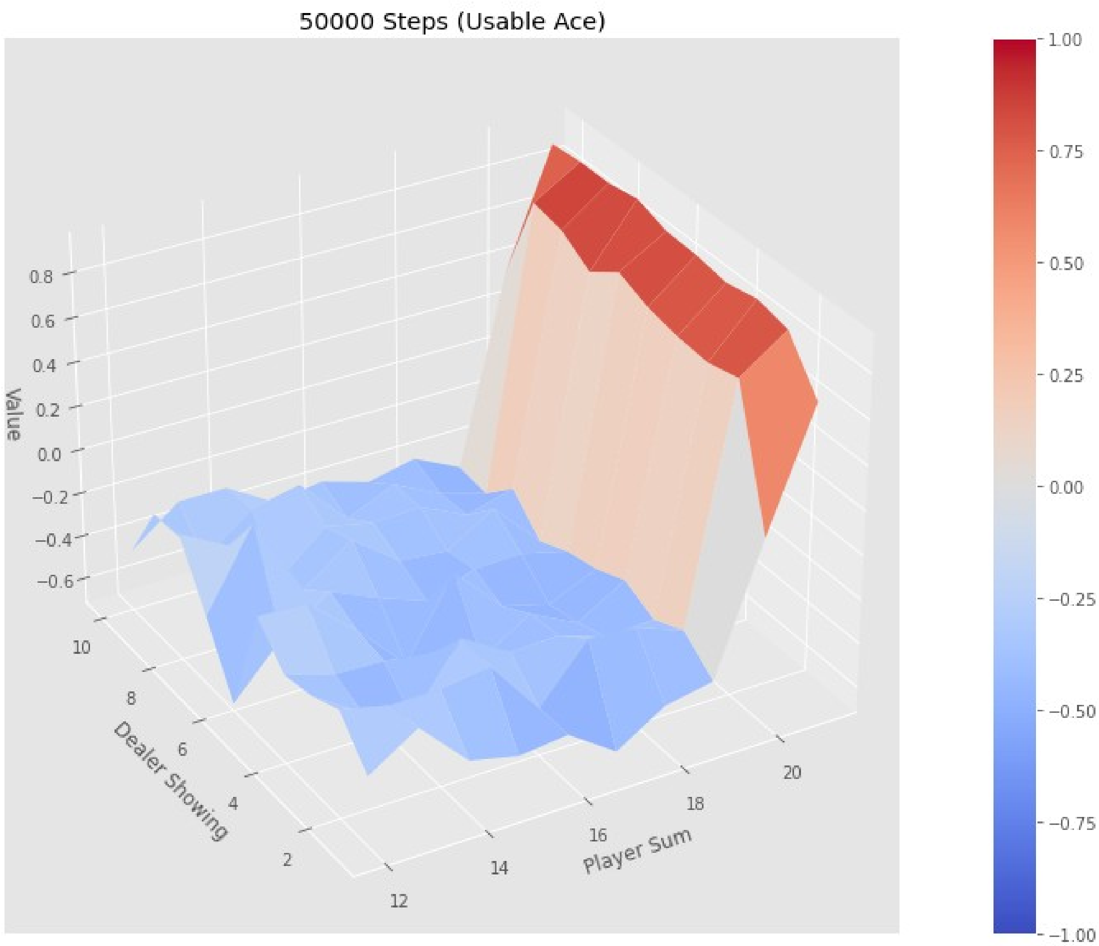

Episode 40000/200000.


40000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

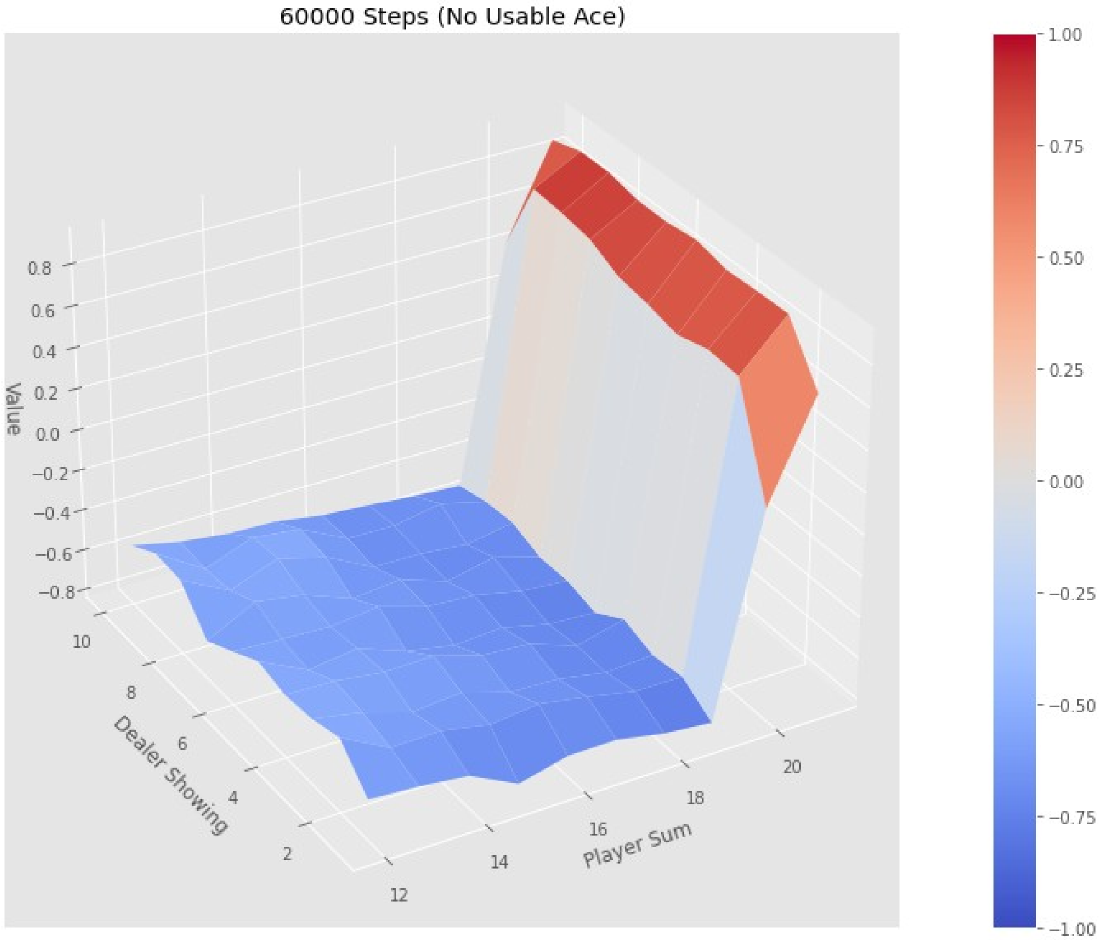


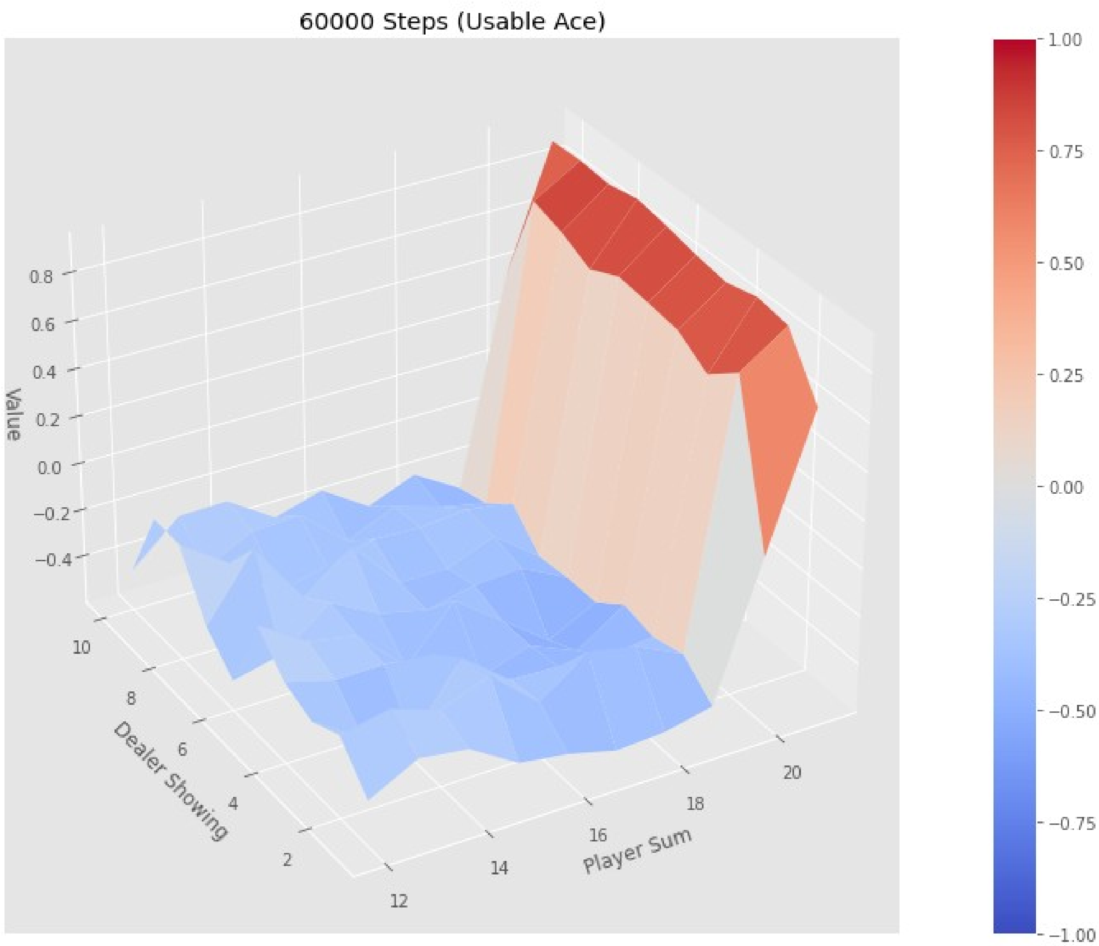

Episode 50000/200000.


50000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

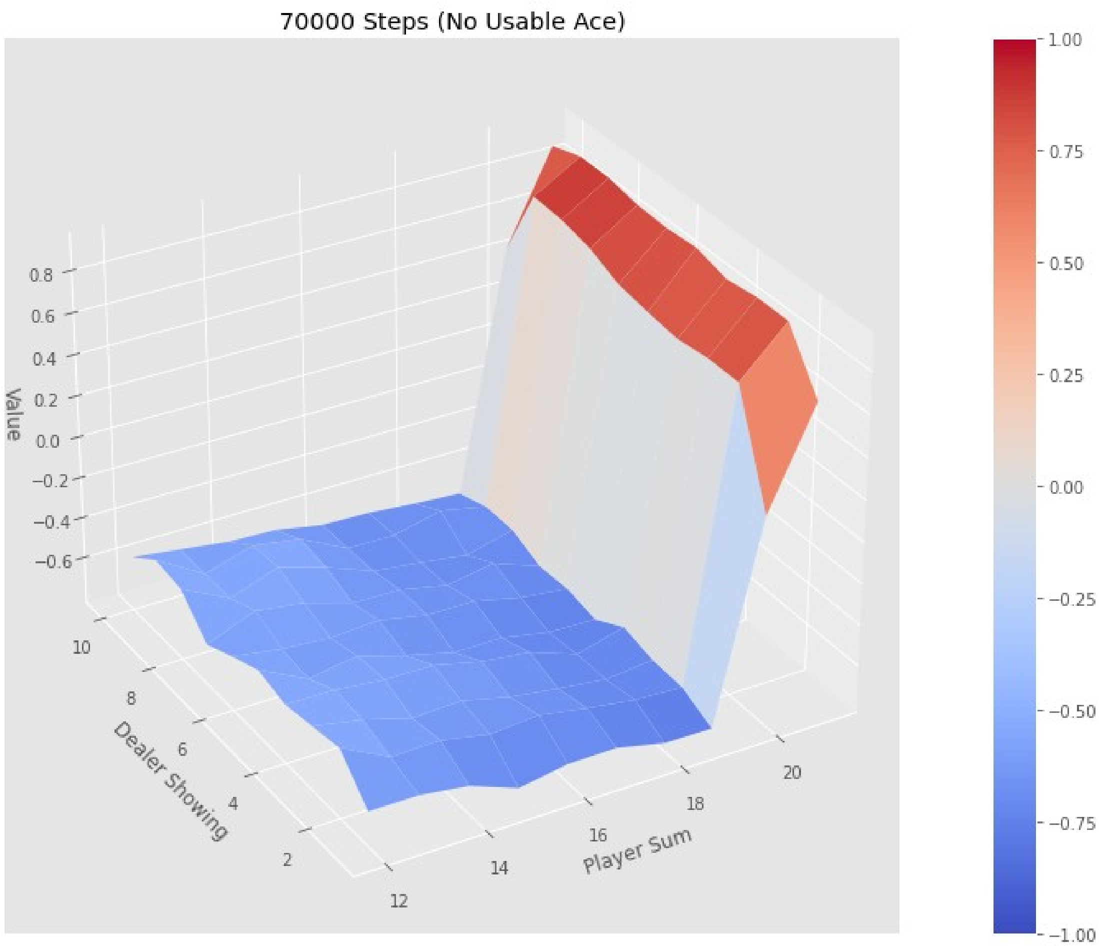



Episode 60000/200000.

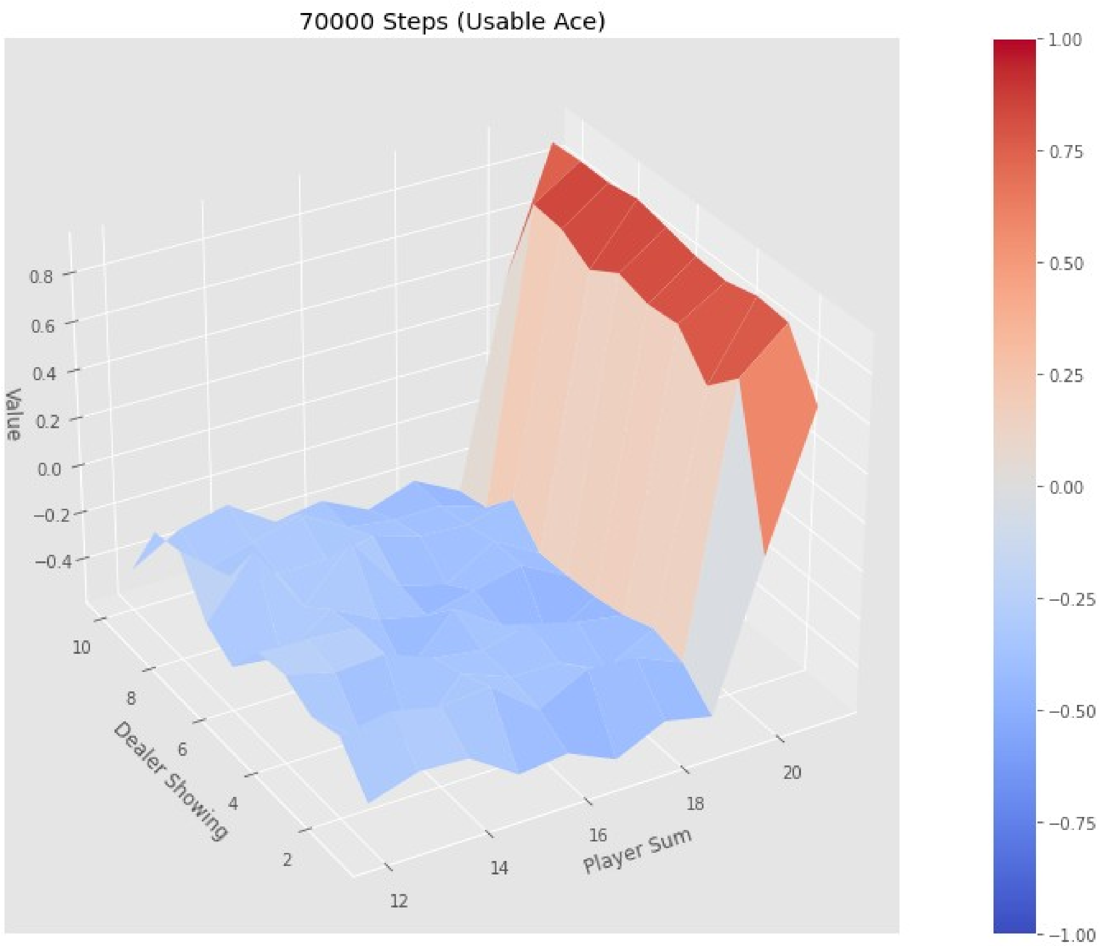

60000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



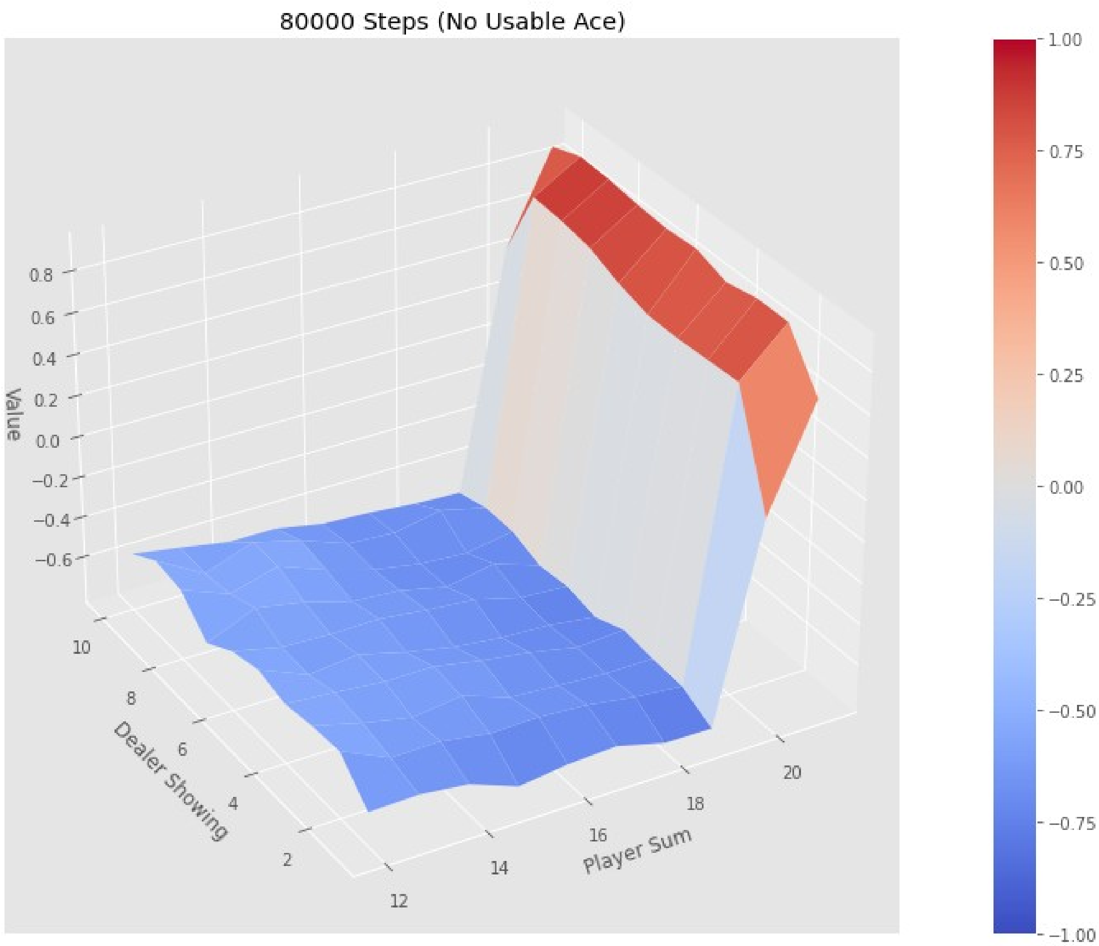

```
Episode 70000/200000.
```

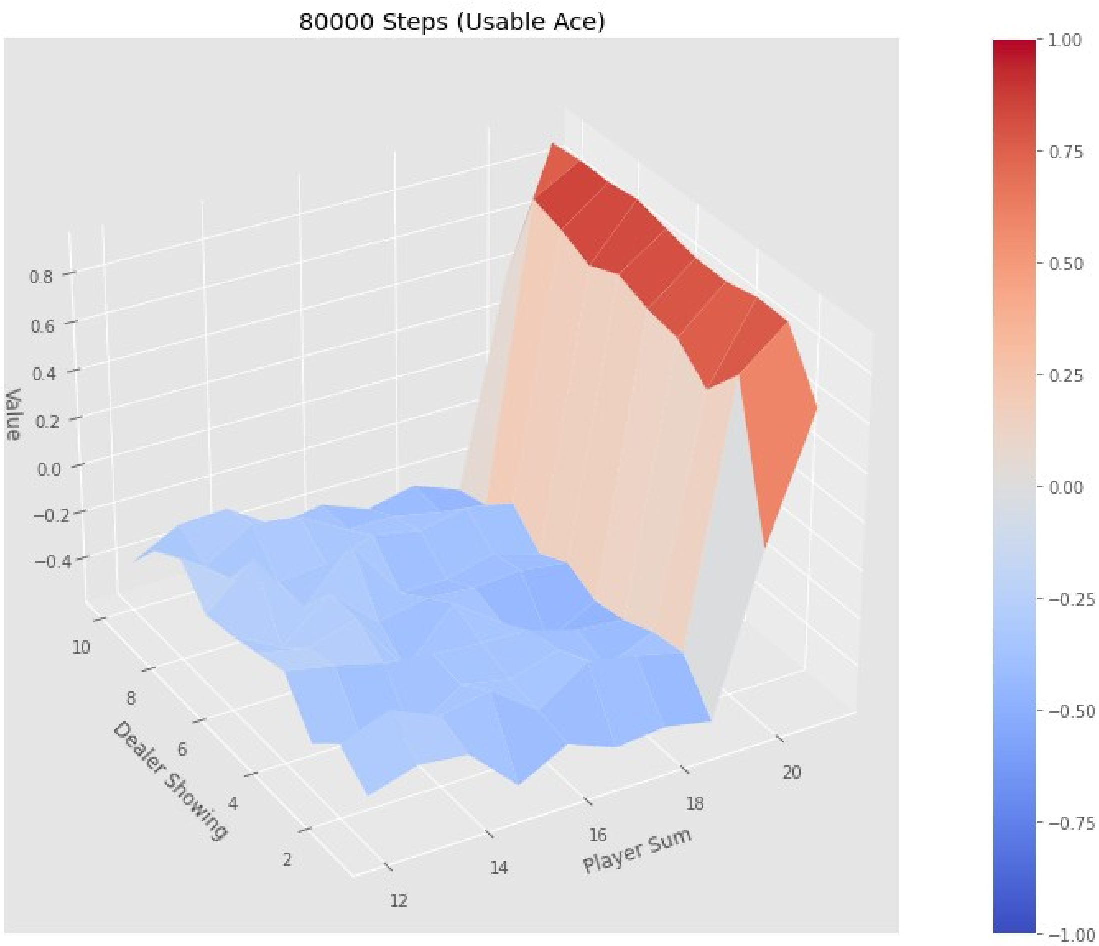


70000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

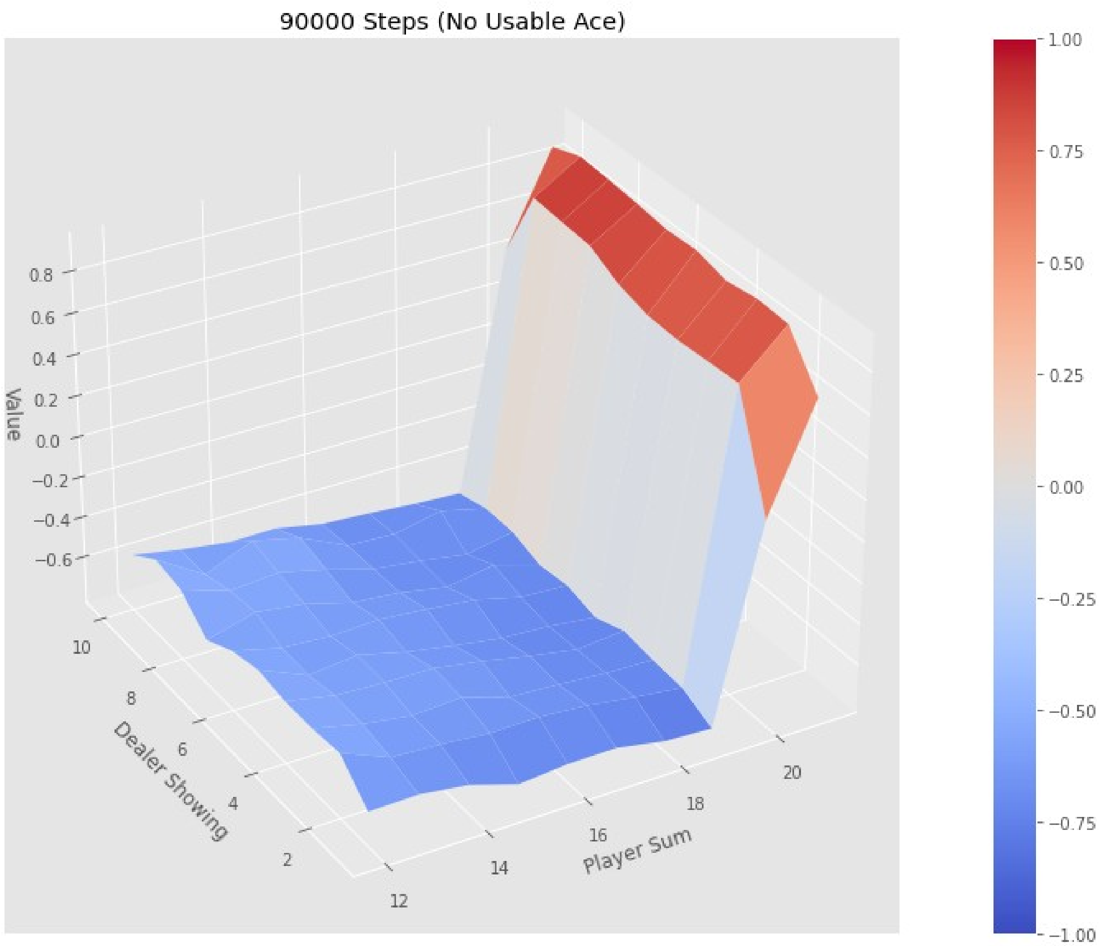



Episode 80000/200000.

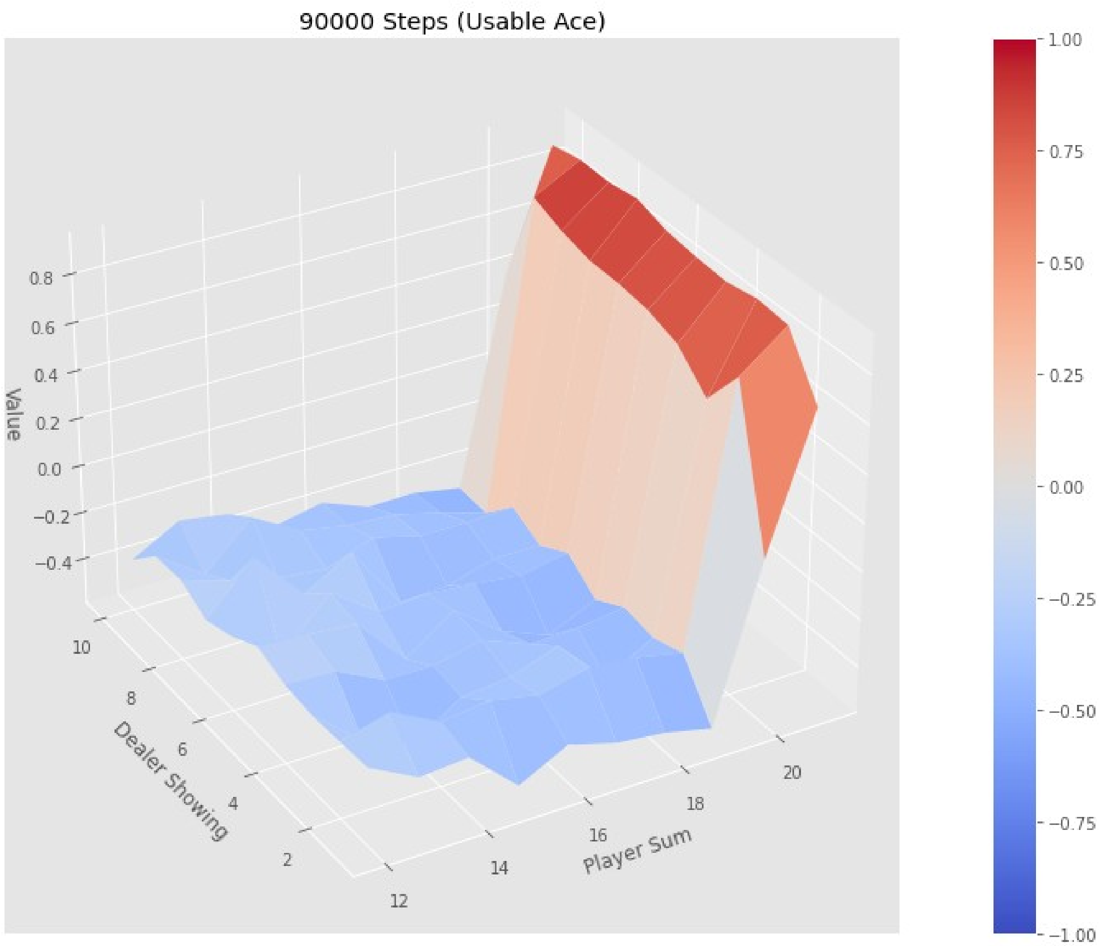

80000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



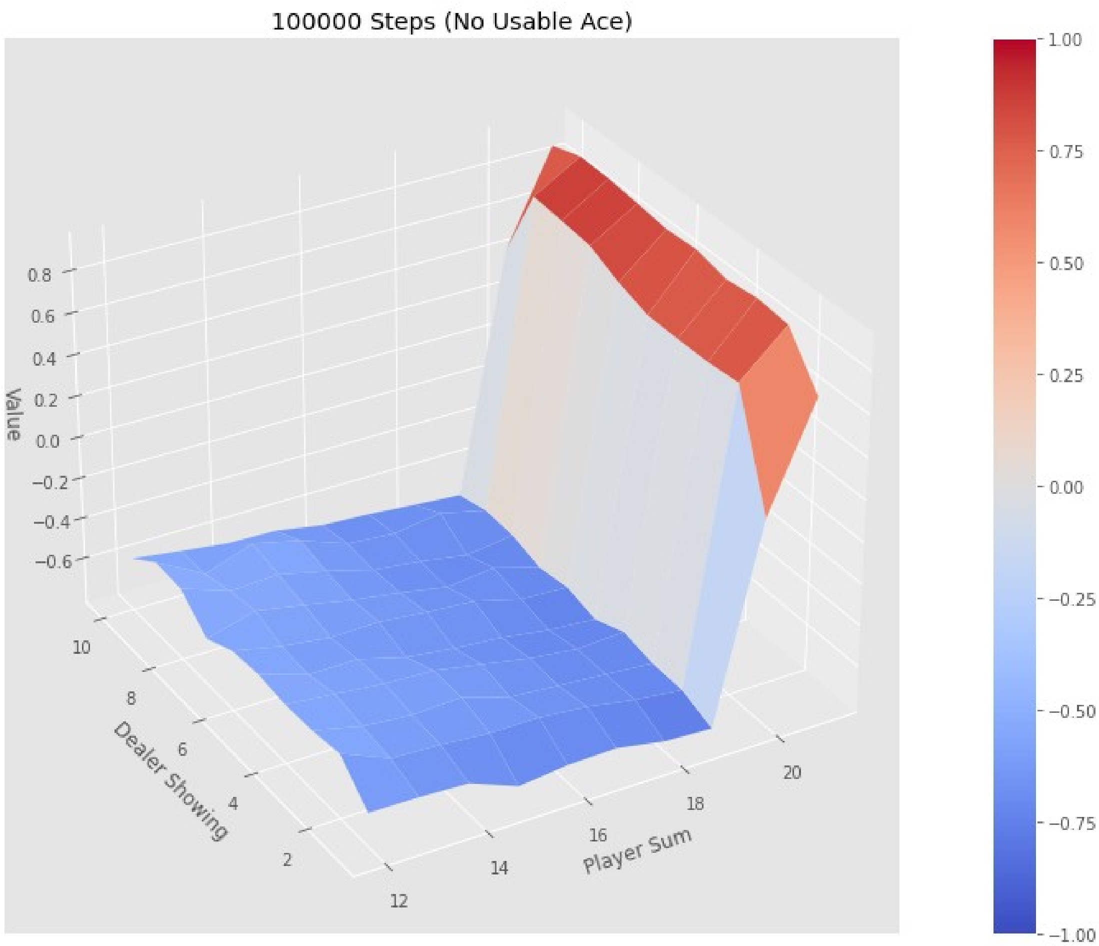

Episode 90000/200000.



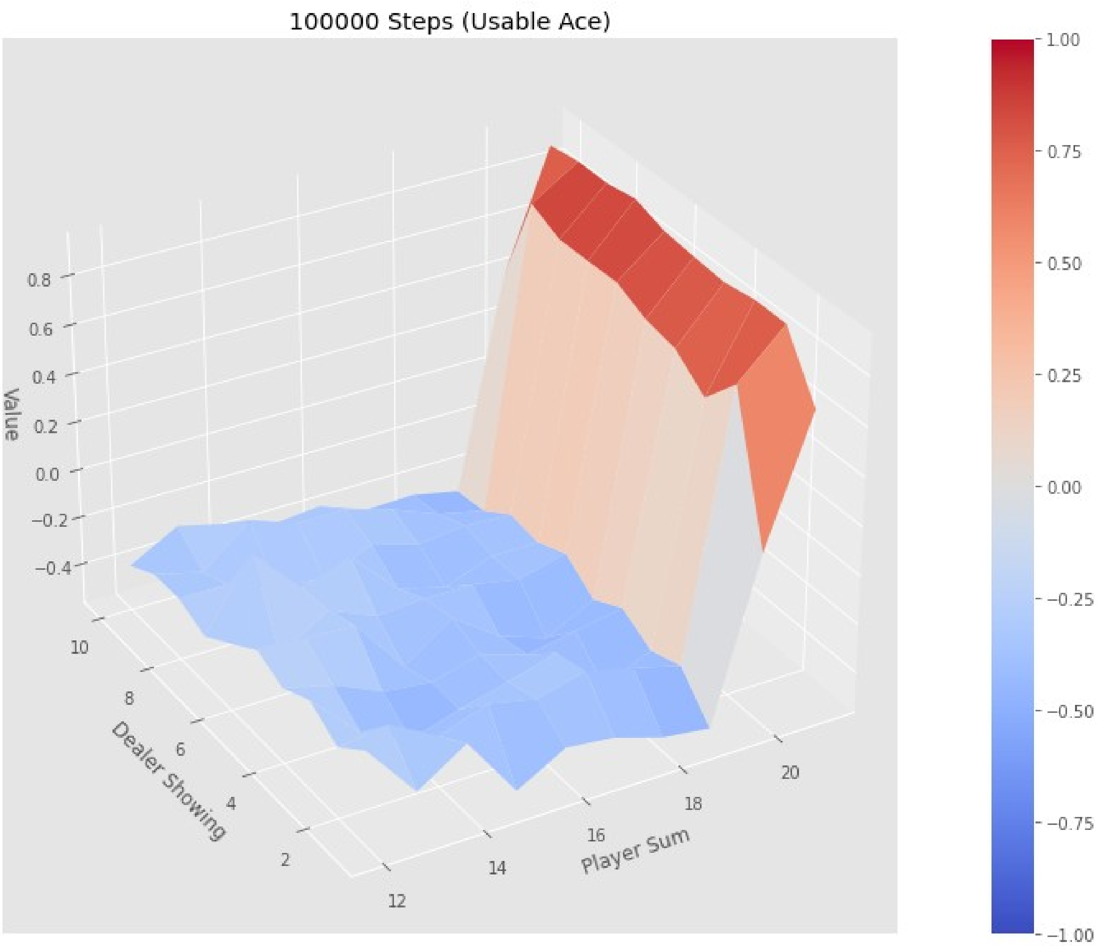
90000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
10
8
6
4
2
Dealer Showing
12
14
16
18
20
Player Sum
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



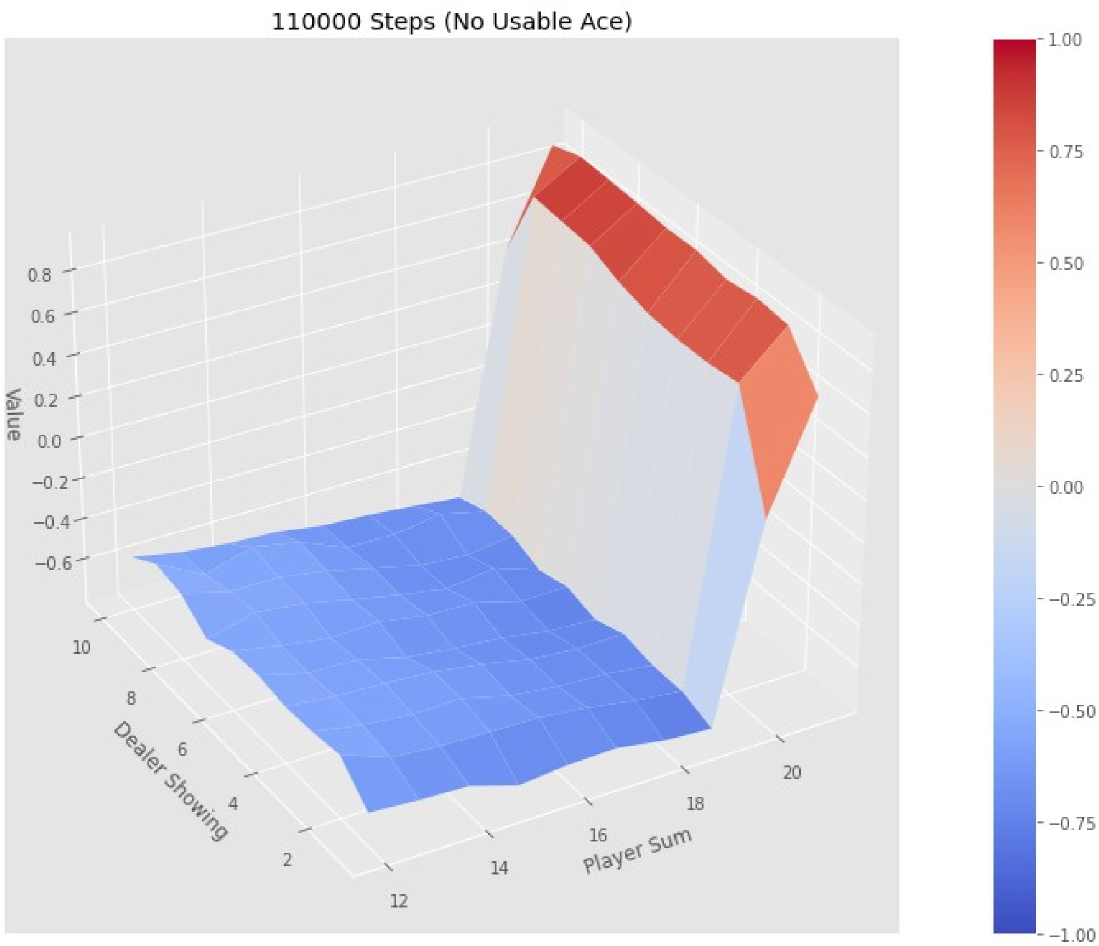

```
Episode 100000/200000.
```



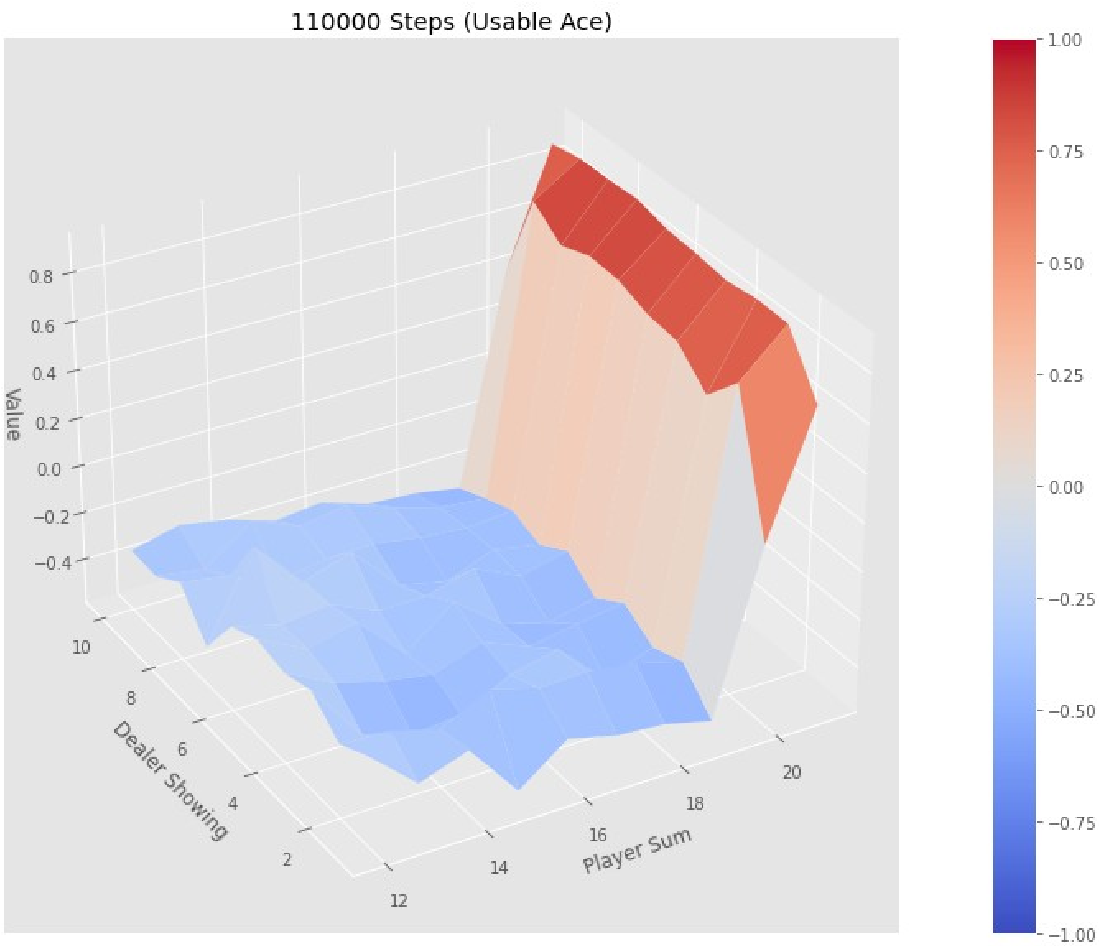
100000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



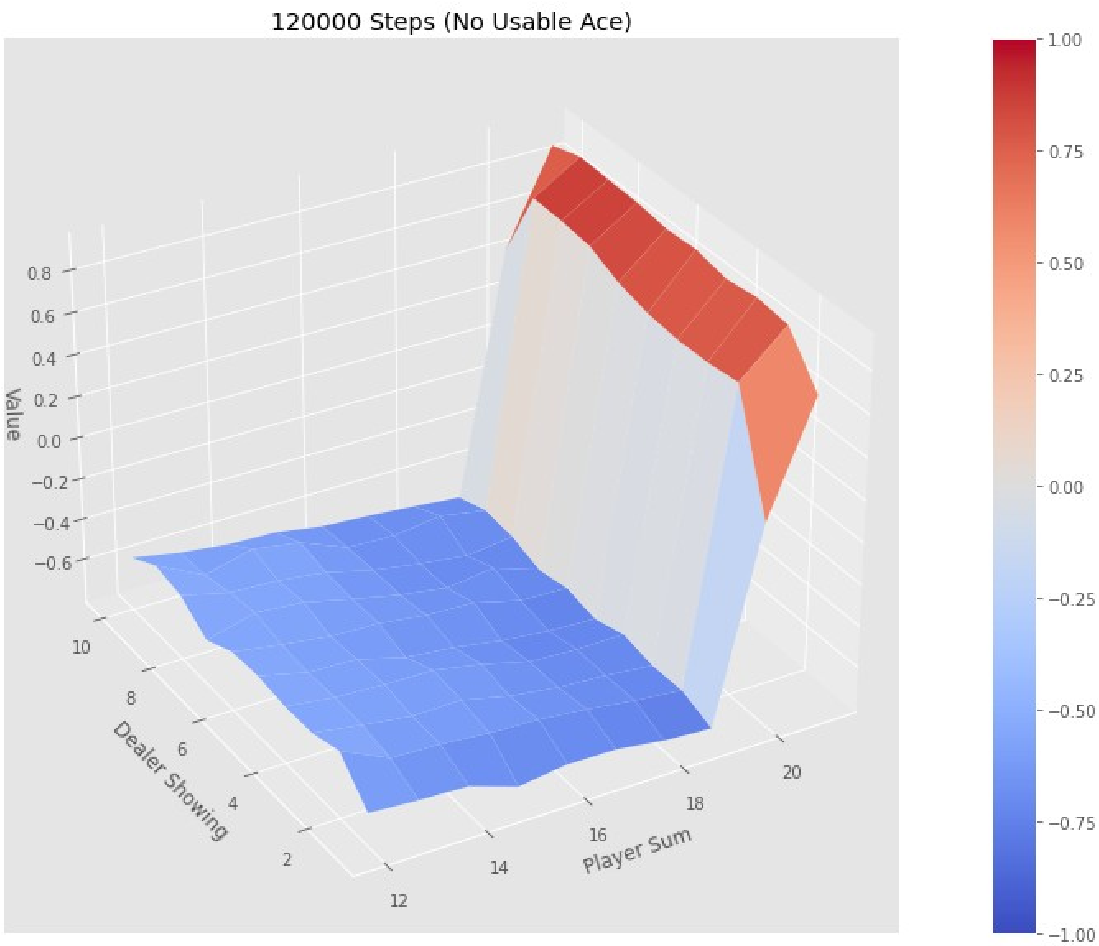

Episode 110000/200000.



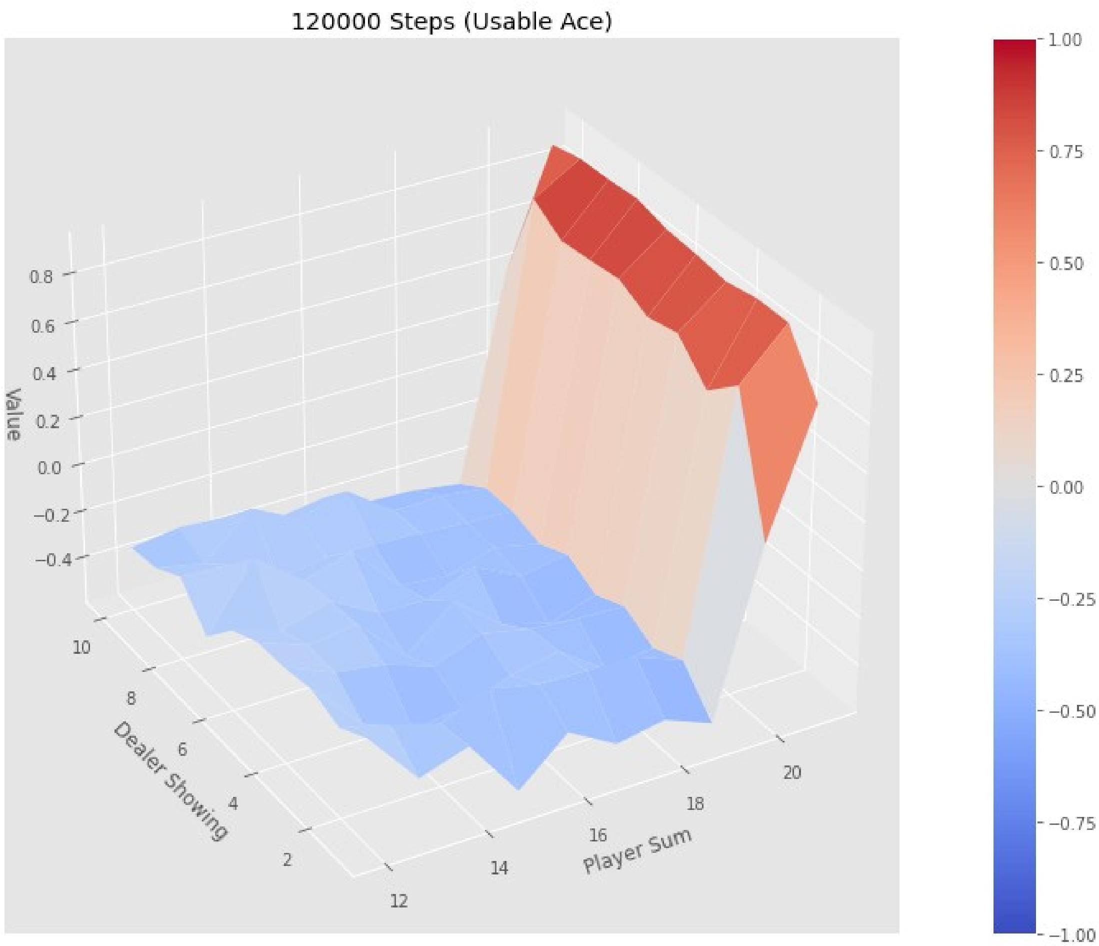
110000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



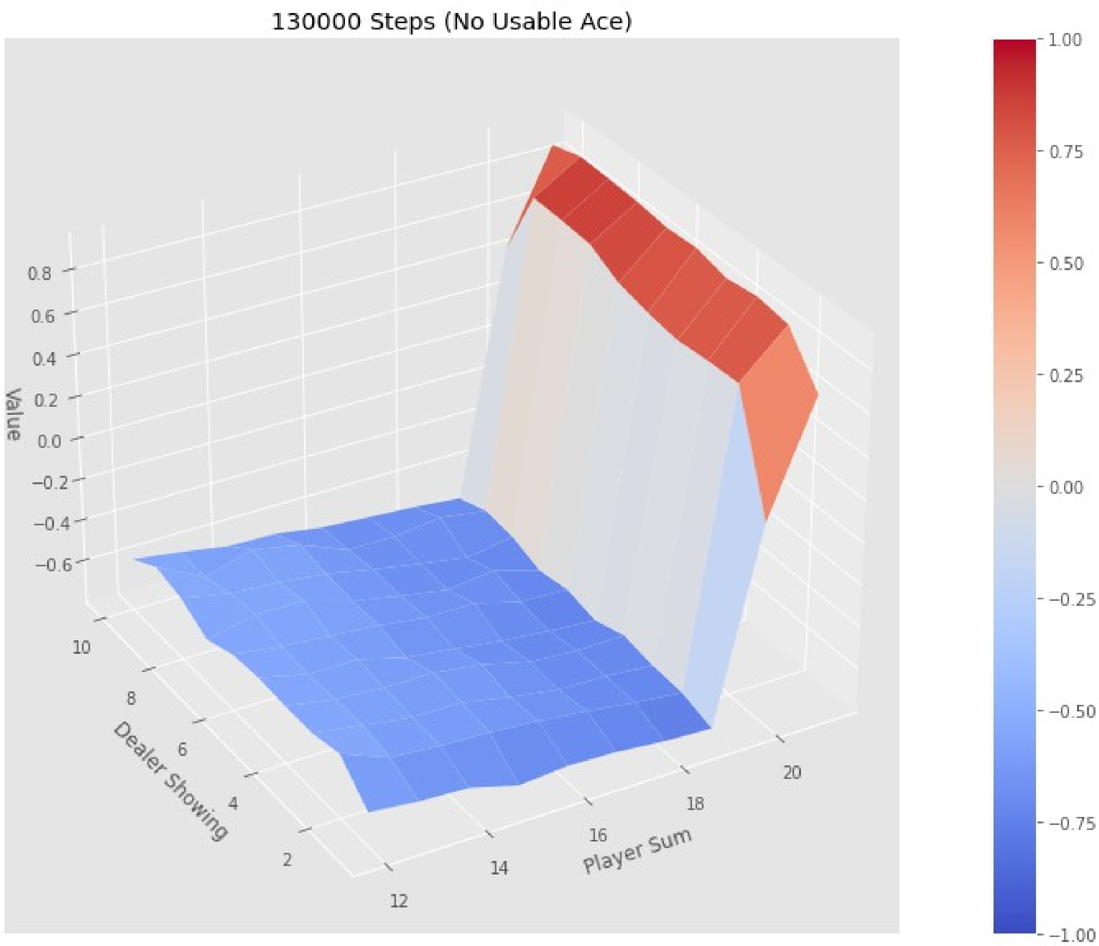

```
Episode 120000/200000.
```



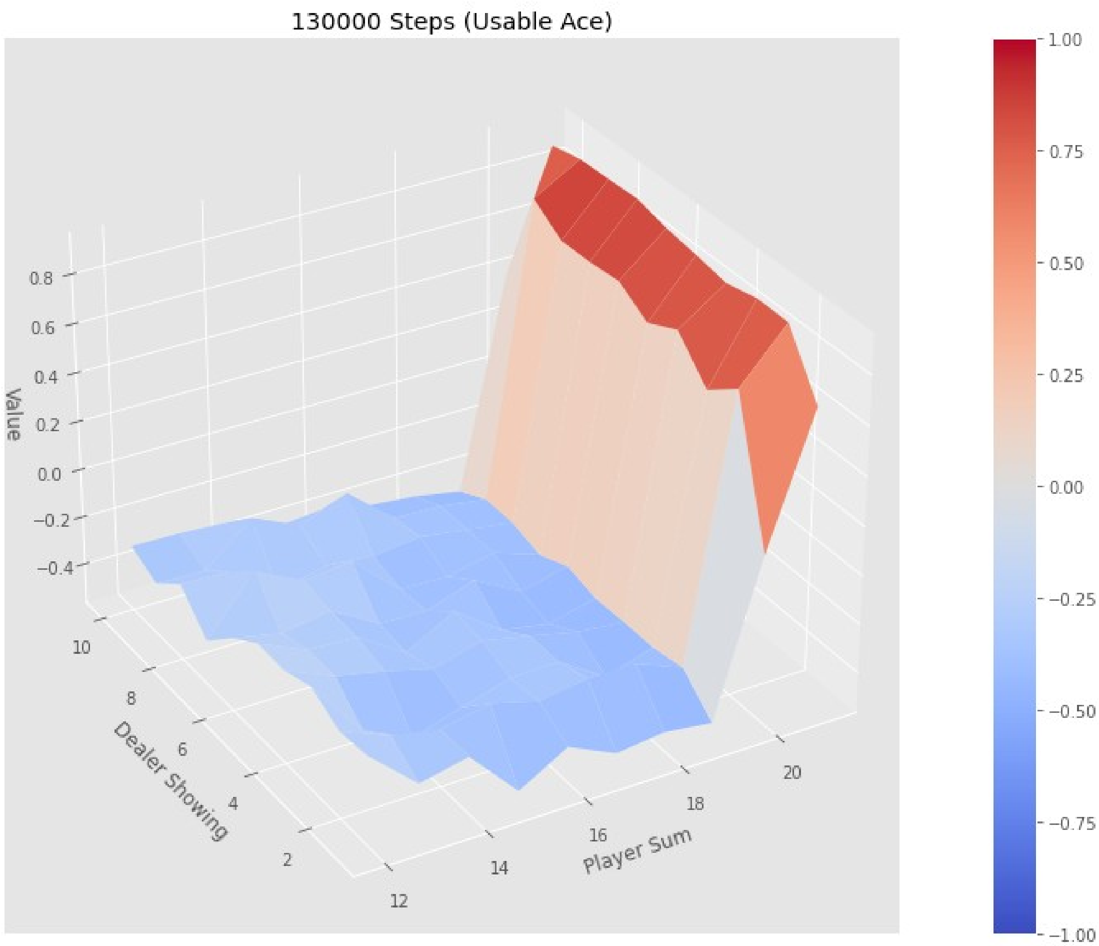
120000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



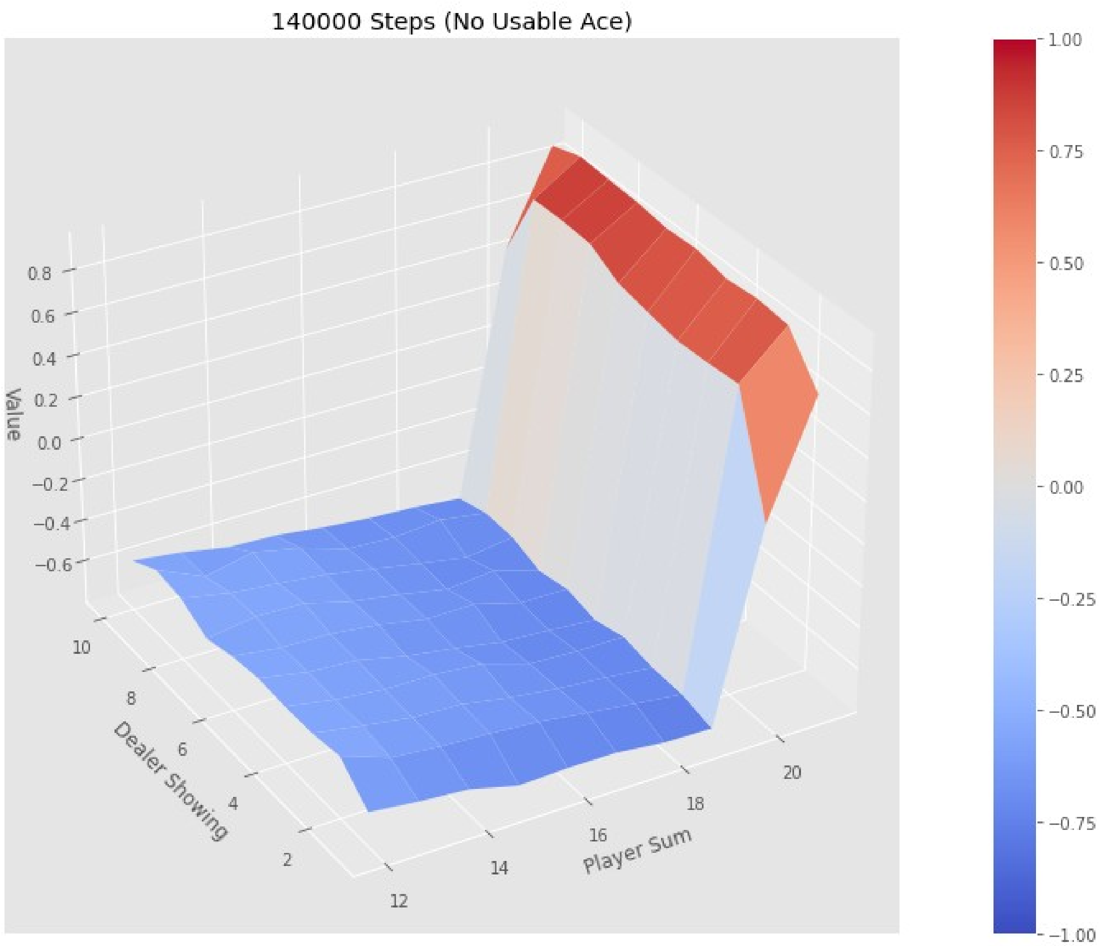

Episode 130000/200000.

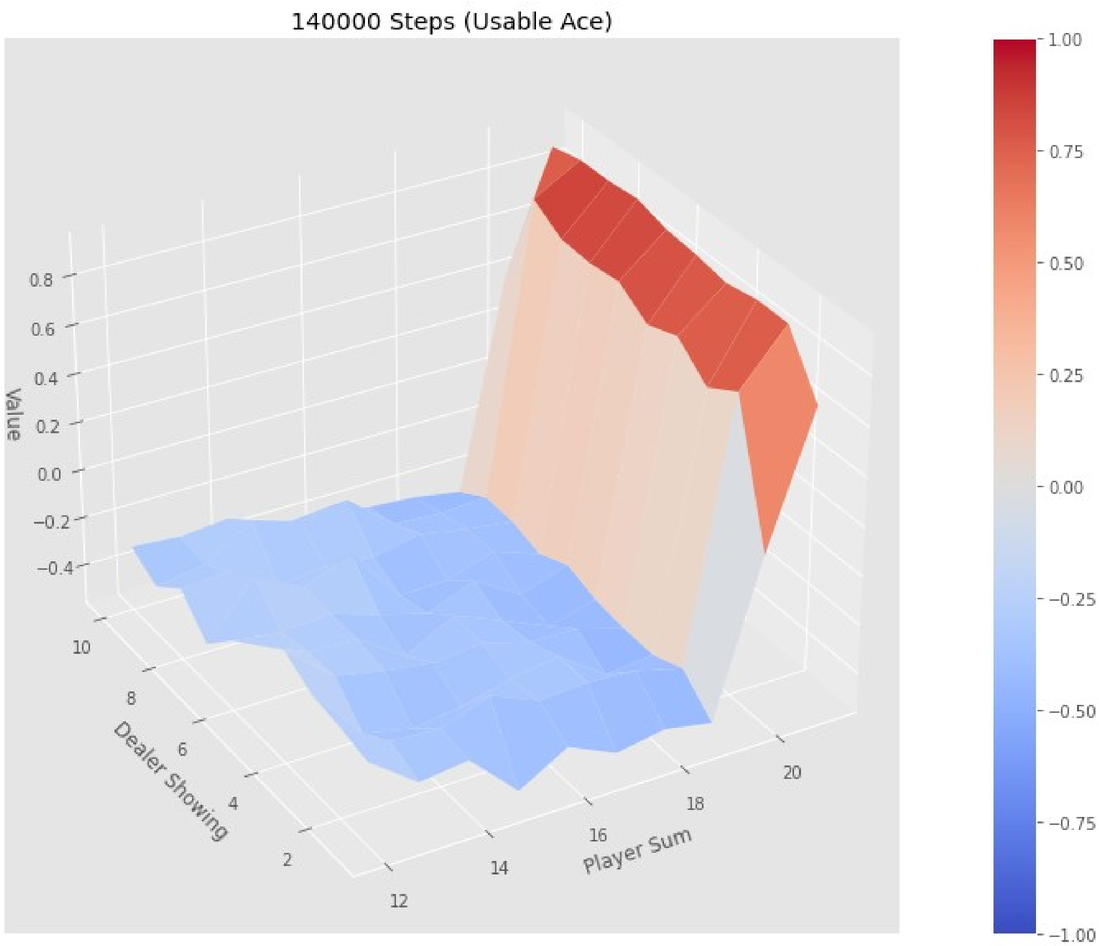


130000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

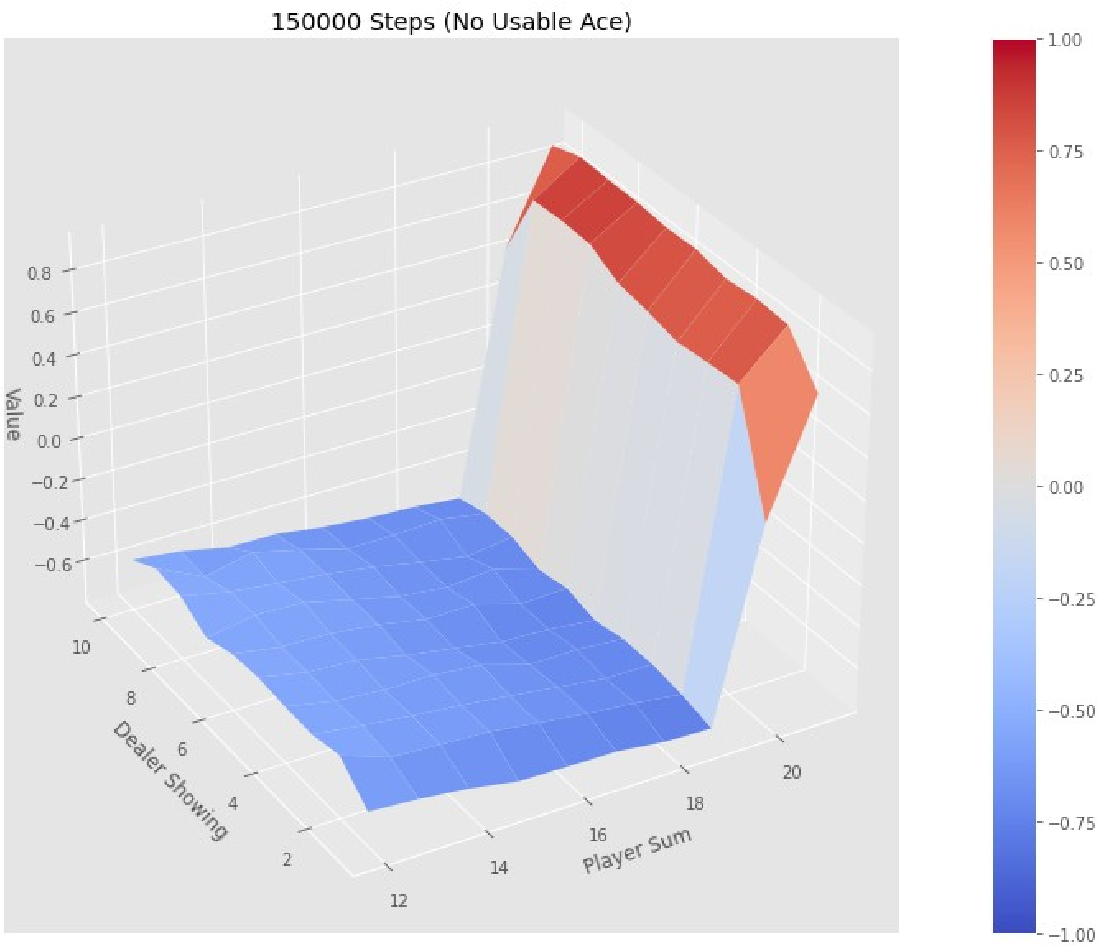



```
Episode 140000/200000.
```

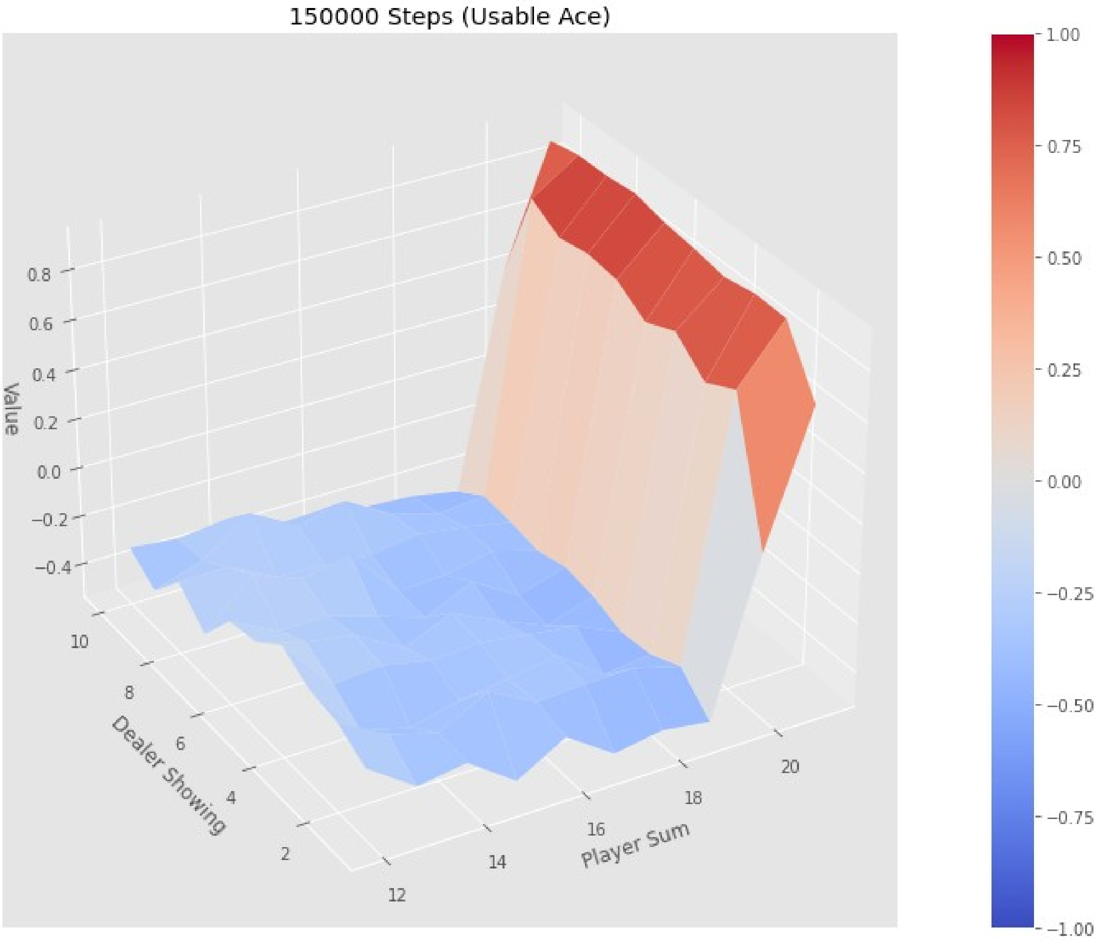

140000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



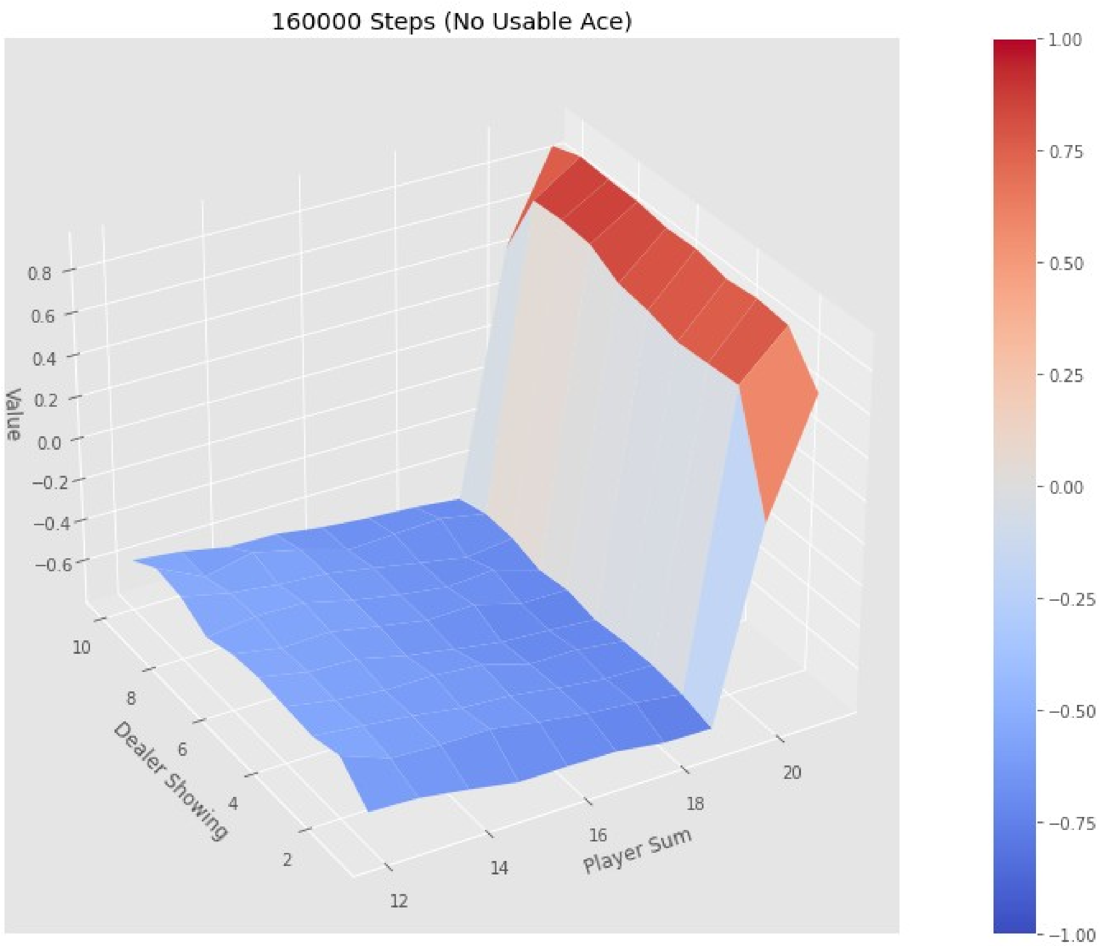

Episode 150000/200000.

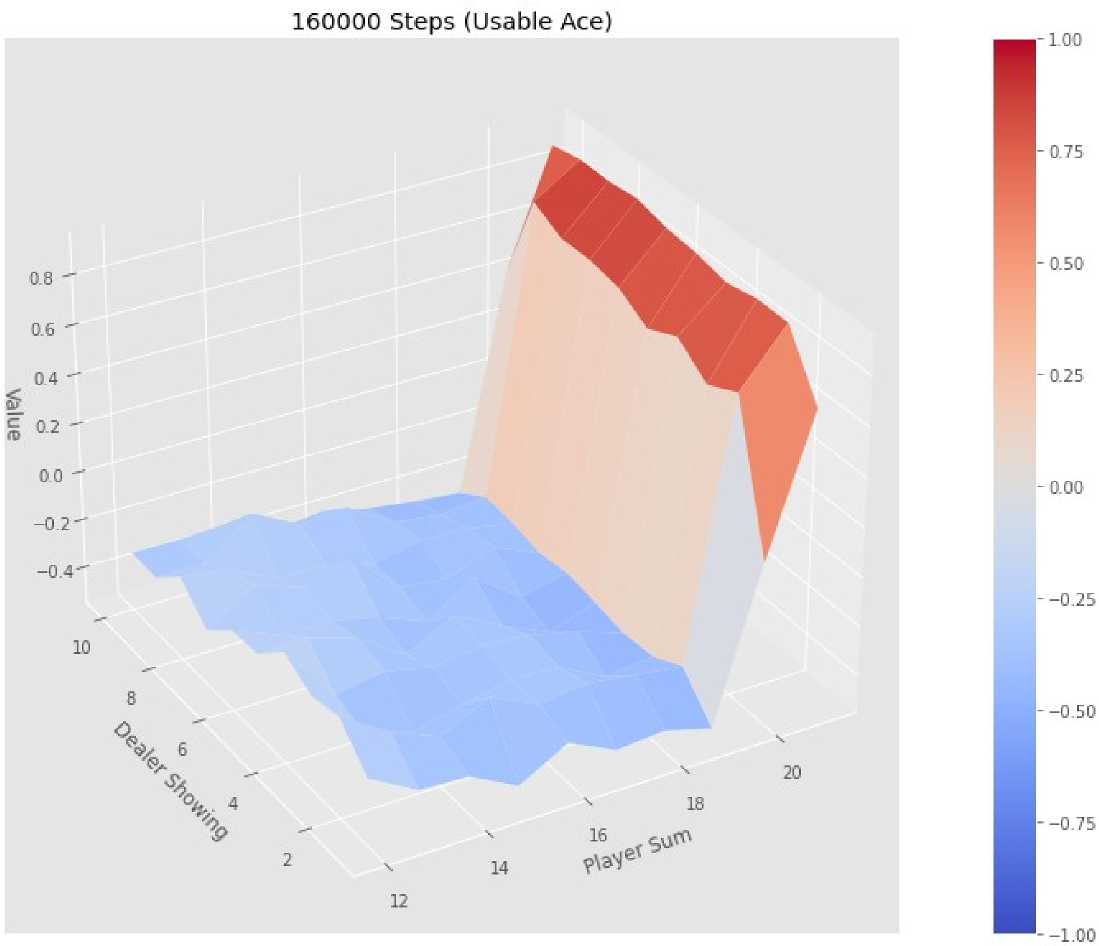


150000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

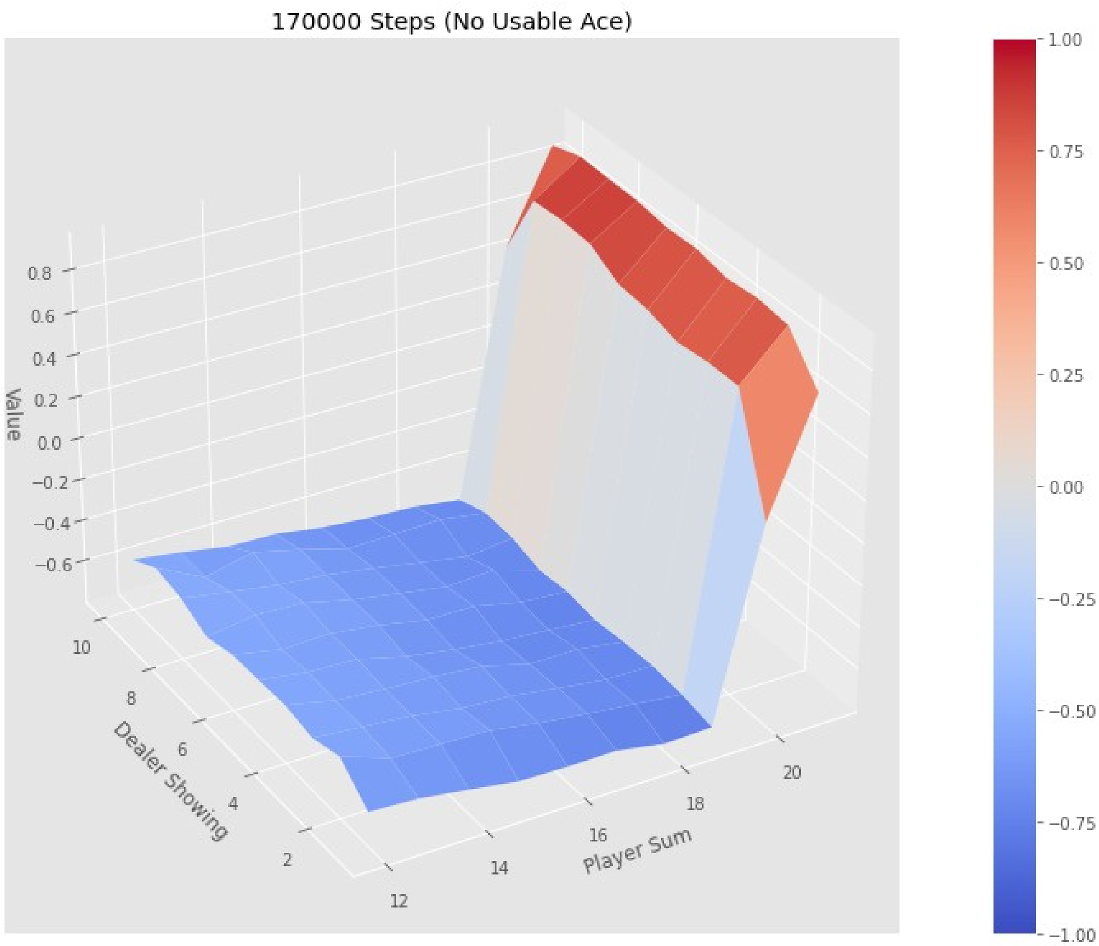



```
Episode 160000/200000.
```

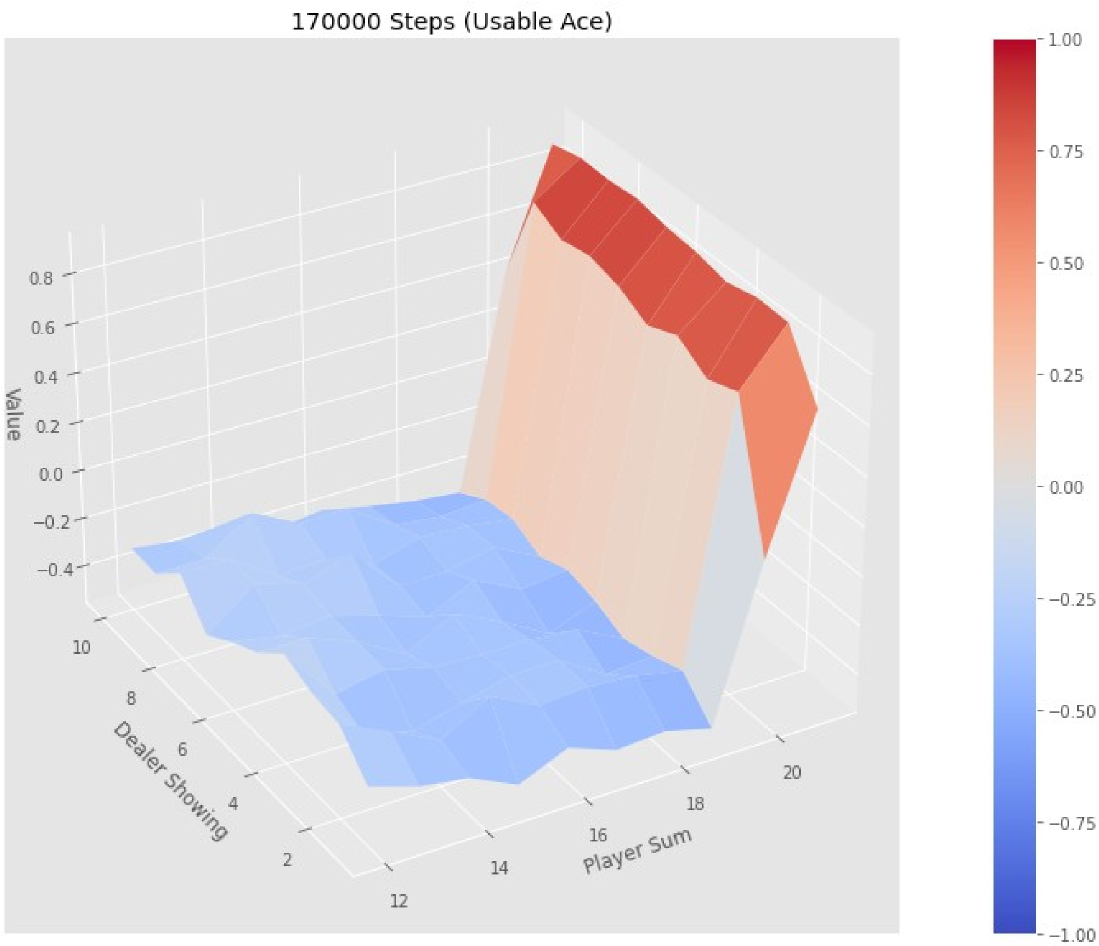

160000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



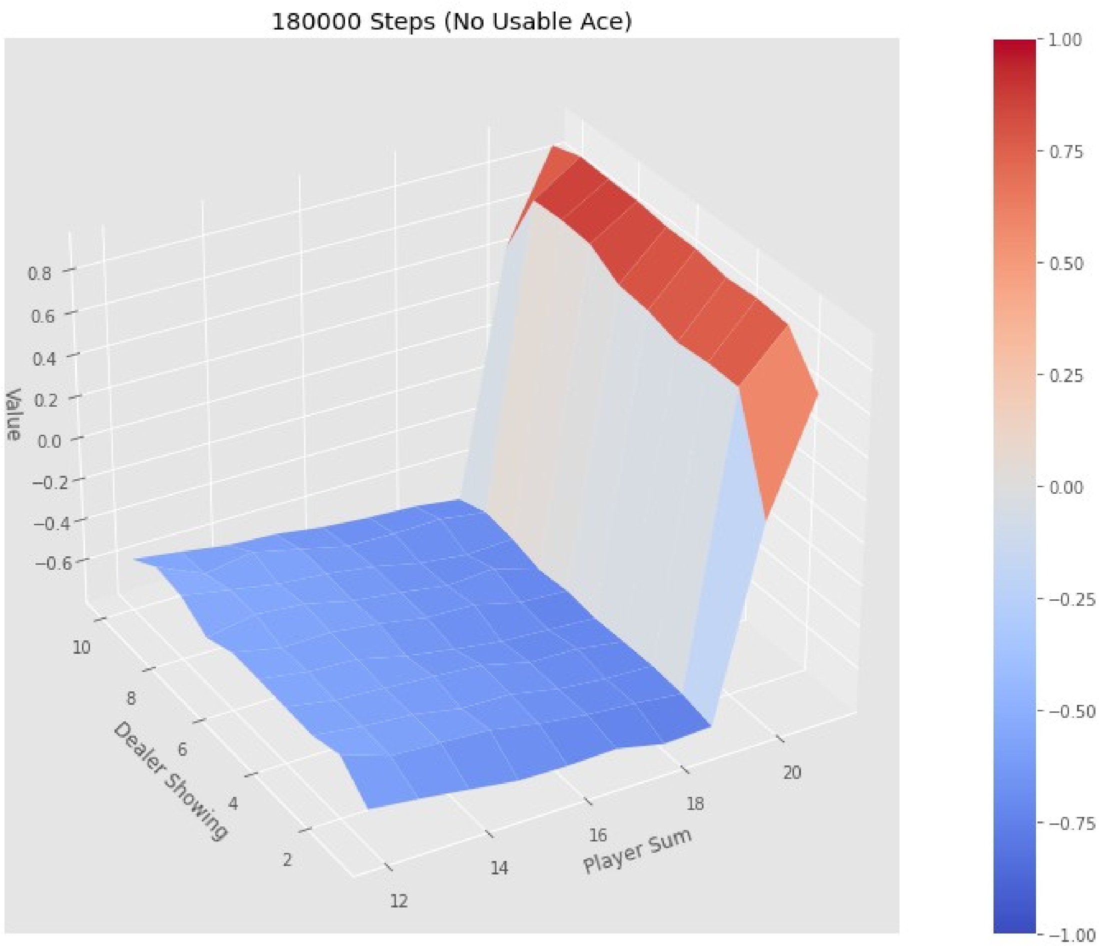

Episode 170000/200000.

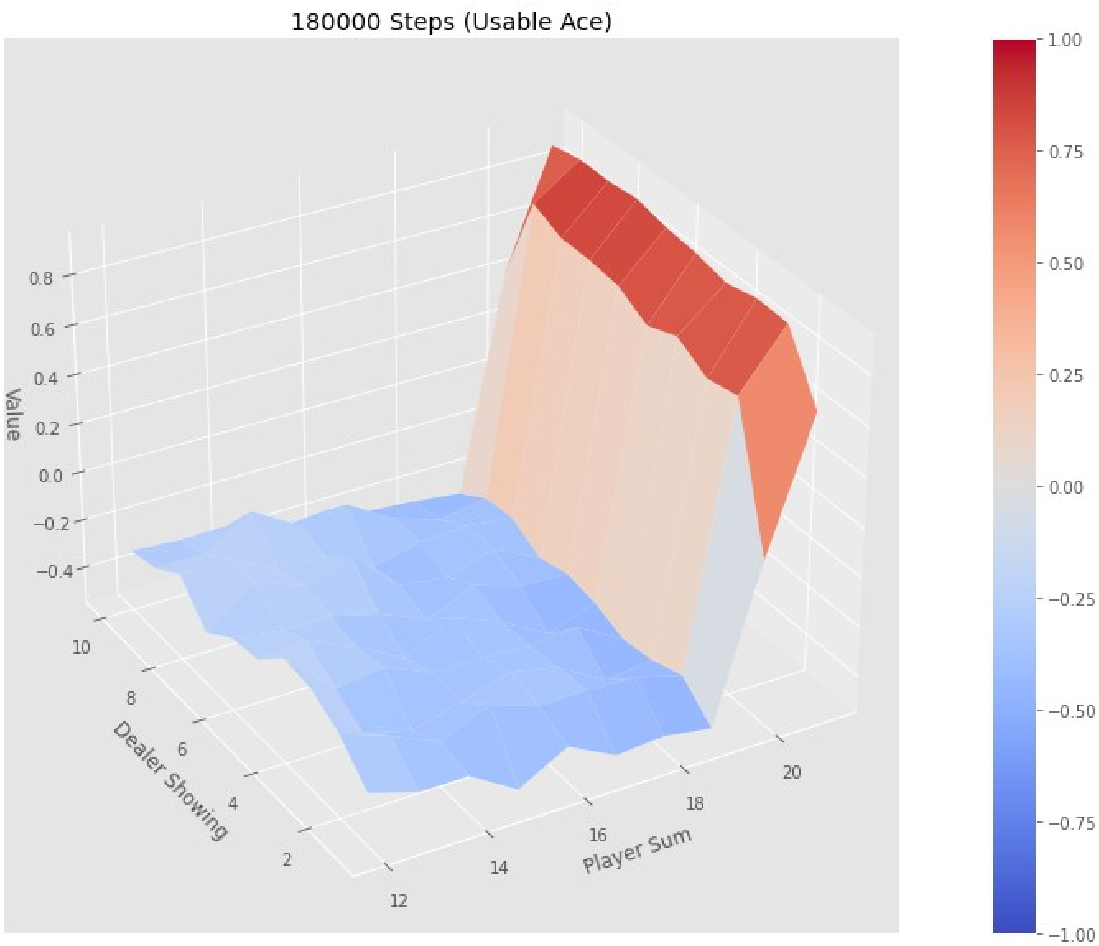


170000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

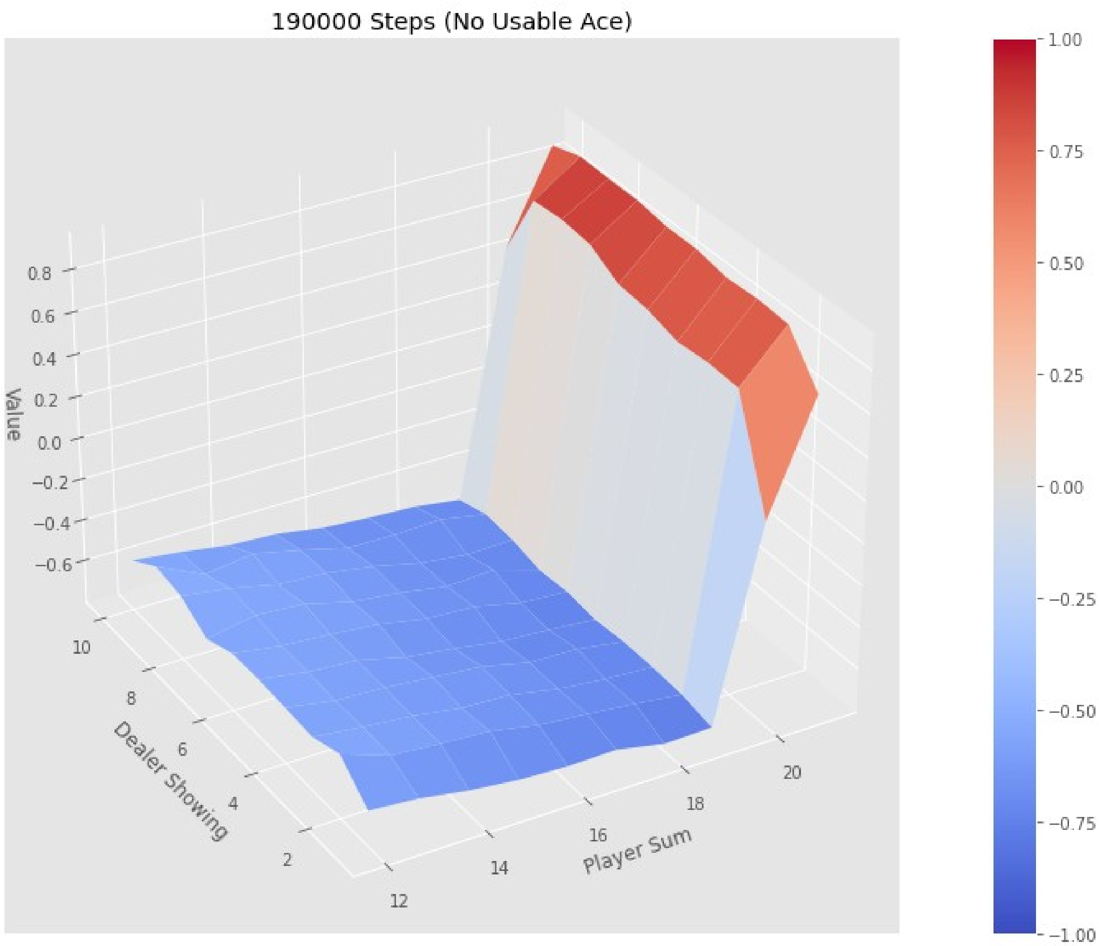


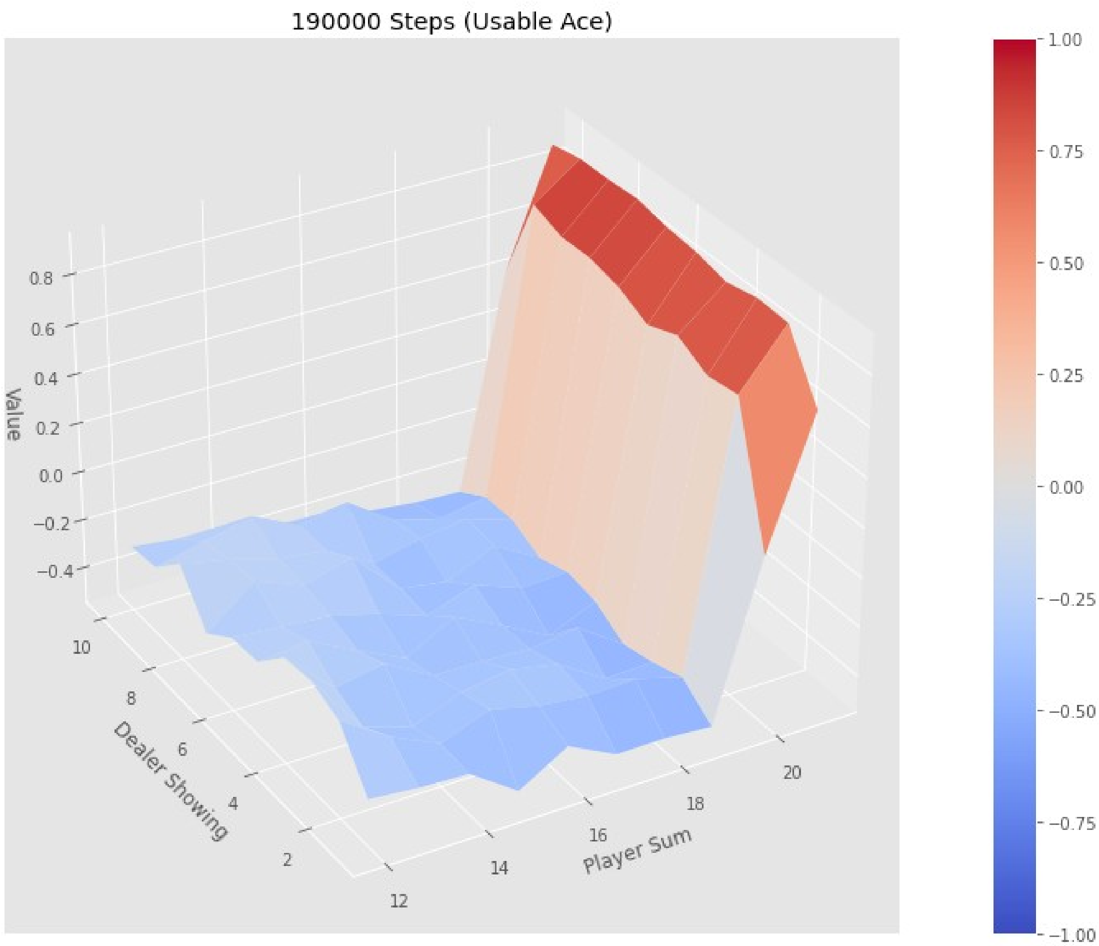

Episode 180000/200000.


180000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer Showing
10
8
6
4
2
Player Sum
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

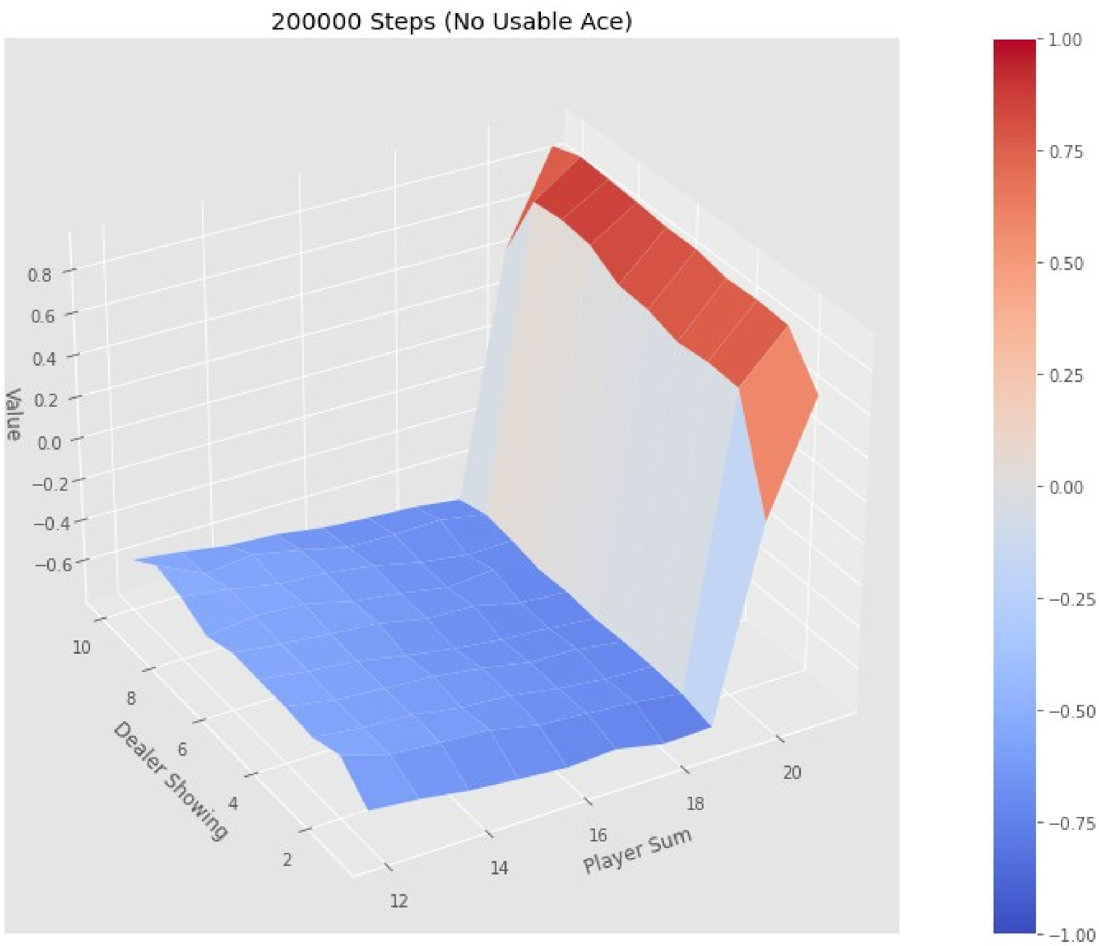


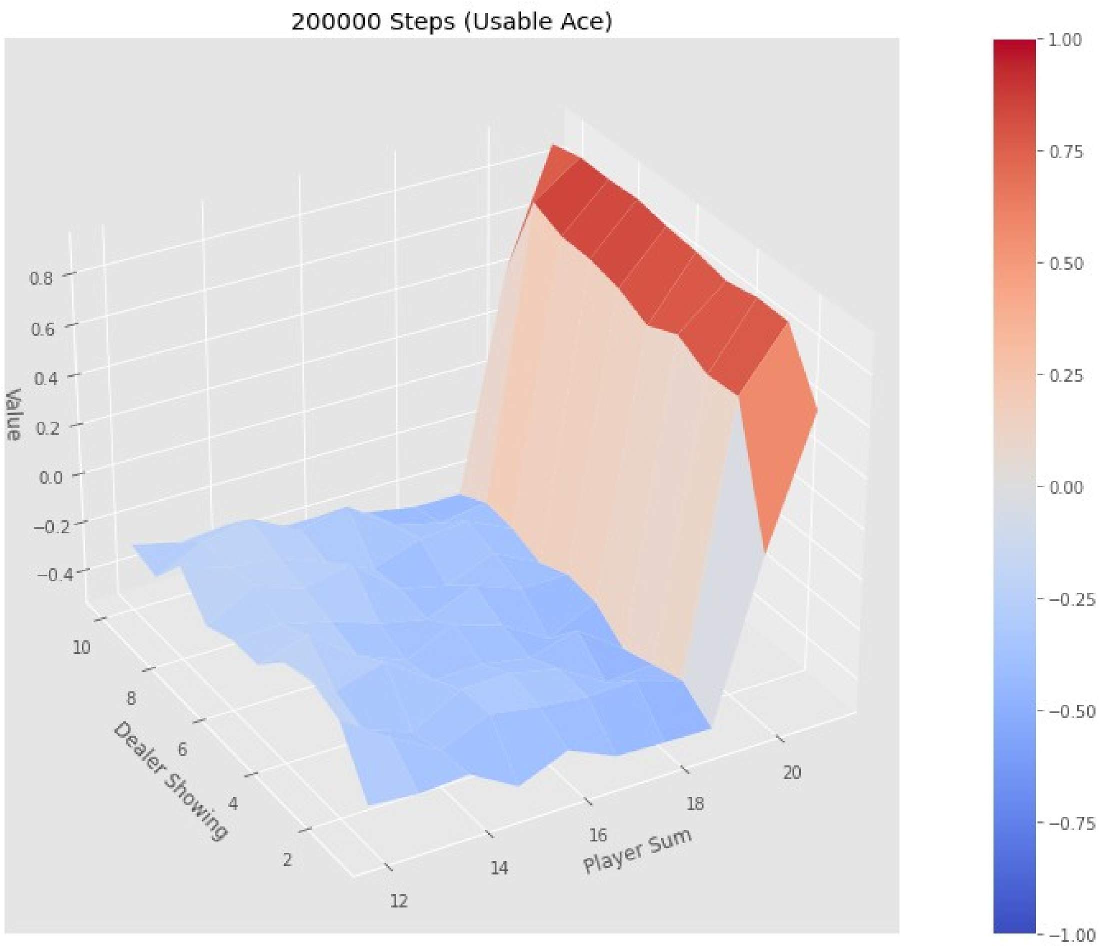

Episode 190000/200000.


190000 Steps (No Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
10
8
6
4
2
Dealer Showing
12
14
16
18
20
Player Sum
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00




Episode 200000/200000.


200000 Steps (No Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00




## Question

**What's the effect of the number of episodes (num_episodes) on the learned value function ?**

Answer:

The "num_episodes" parameter determines the number of episodes used to train the Monte Carlo control algorithm. Increasing the number of episodes leads to a better estimation of the true value function, as the algorithm has more experience in different states and actions.

Specifically, as the number of episodes increases, the Monte Carlo algorithm visits more states and actions, and the sample averages used to update the Q-values will converge more closely to the true expected returns. This results in a more accurate estimate of the value function and better policy selection.

## RL Lab 03 - Part 2 - TD prediction on Random walk and BlackJack

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
```

From Sutton and Barto (chapter 6.1), the TD(0) algorithm for estimating V is as follows:

**Tabular TD(0) for estimating $v_\pi$**

Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    $A \leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R$, $S'$
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until $S$ is terminal

### Implementation of TD(0)

Start by filling the following blanks in the code below:

```
def td_prediction(env, policy, ep, gamma, alpha):
    """TD Prediction

    Params:
        env - environment
        ep - number of episodes to run
        policy - function in form: policy(state) -> action
        gamma - discount factor [0..1]
        alpha - step size (0..1]
    """
    assert 0 < alpha <= 1
    V = defaultdict(float)    # default value 0 for all states

    for _ in range(ep):
        S = env.reset()
        while True:
            A = policy(S)
            S_, R, done = env.step(A)
            V[S] = V[S] + alpha * (R + gamma * V[S_] - V[S])
            S = S_
            if done: break

    return V
```

For TD prediction to work, **V for terminal states must be equal to zero, always**. Value of terminal states is zero because game is over and there is no more reward to get. Value of next-to-last state is reward for last transition only, and so on.

- If terminal state is initalised to something different than zero, then your resulting V estimates will be offset by that much

- If, V of terminal state is *updated during training* then everything will go wrong.
  - so make *absolutely sure* environment returns different observations for terminal states than non-terminal ones
  - hint: this is not the case for out-of-the-box gym Blackjack, so you need to change it

## Evaluate a Random walk (example 6.2 Sutton's book)

In this example we empirically compare the prediction abilities of TD(0) and constant-$\alpha$ MC when applied to the following Markov reward process:


A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.

```
class LinearEnv:
    """
    State Index:   [ 0    1    2    3    4    5    6 ]
    State Label:   [ .    A    B    C    D    E    . ]
    Type:          [ T    .    .    S    .    .    T ]
    """
    V_true =       [0.0, 1/6, 2/6, 3/6, 4/6, 5/6, 0.0]

    def __init__(self):
        self.reset()

    def reset(self):
        self._state = 3
        self._done = False
        return self._state

    def step(self, action):
        if self._done: raise ValueError('Episode has terminated')
        if action not in [0, 1]: raise ValueError('Invalid action')

        if action == 0: self._state -= 1
        if action == 1: self._state += 1

        reward = 0
        if self._state < 1: self._done = True
```

```
        if self._state > 5: self._done = True; reward = 1

        return self._state, reward, self._done  # obs, rew, done

env = LinearEnv()
```

Plotting helper function:

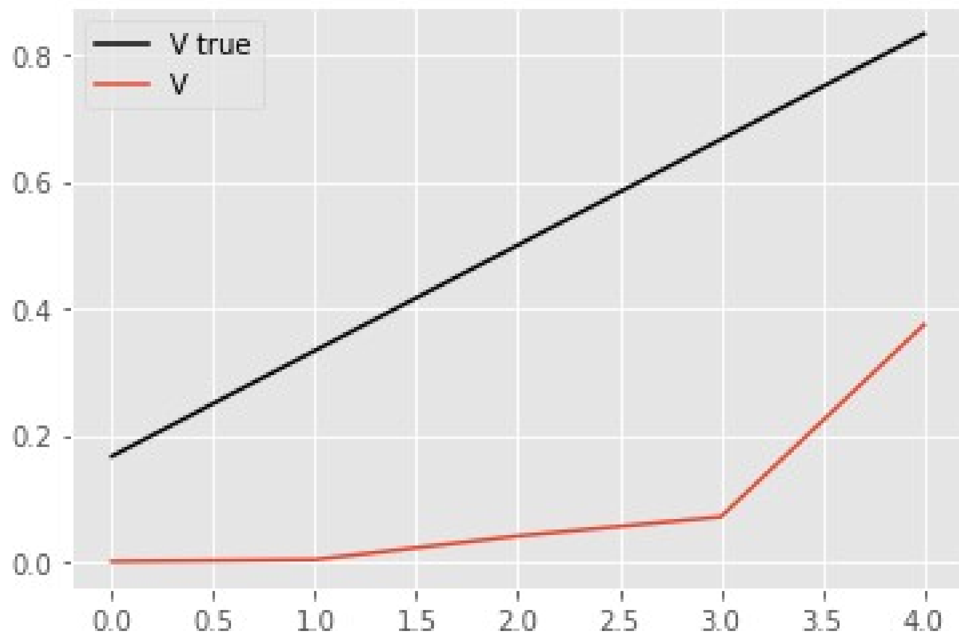
```
def plot(V_dict):
    """Param V is dictionary int[0..7]->float"""

    V_arr = np.zeros(7)
    for st in range(7):
        V_arr[st] = V_dict[st]

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(LinearEnv.V_true[1:-1], color='black', label='V true')
    ax.plot(V_arr[1:-1], label='V')

    ax.legend()

    plt.show()
```

Random policy:

```
def policy(state):
    return np.random.choice([0, 1])  # random policy
```

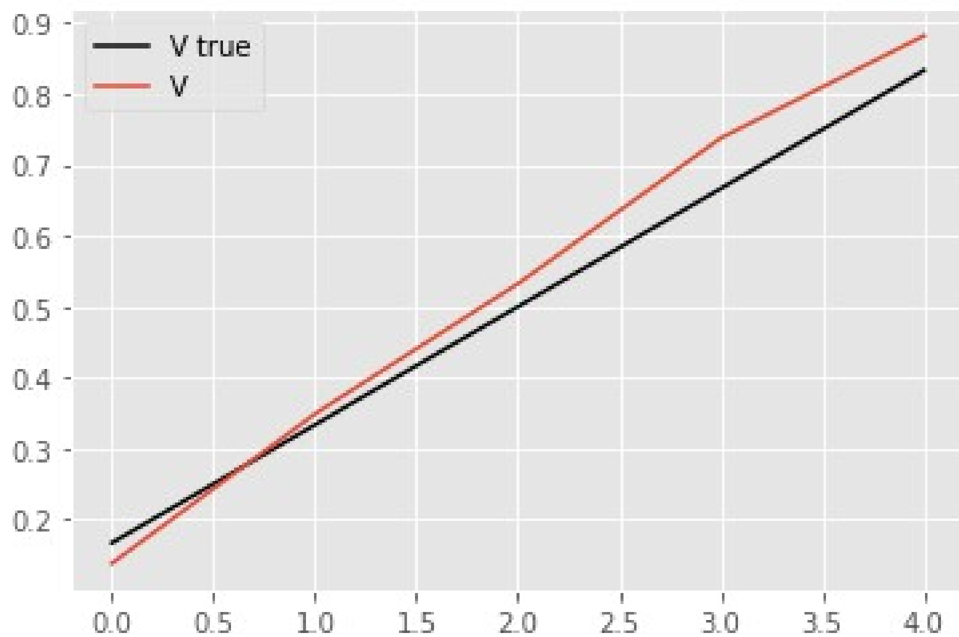
*For 10 episodes*

```
V = td_prediction(env, policy, ep=10, gamma=1.0, alpha=0.1)
plot(V)
```




*For 1000 episodes*

```
V = td_prediction(env, policy, ep=1000, gamma=1.0, alpha=0.1)
plot(V)
```




## Temporal-Difference for BlackJack

Let's start first by fixing the BlackJack environement for TD(0)

As mentioned earlier, there is a problem with Blackjack environment in the gym. If agent sticks, then environment will return exactly the same observation but this time with done==True. This will cause TD prediction to evaluate terminal state to non-zero value belonging to non-terminal state with same observation. We fix this by redefining observation for terminal states with 'TERMINAL'.

```
class BlackjackFixed():
    def __init__(self):
        self._env = gym.make('Blackjack-v1')

    def reset(self):
        return self._env.reset()

    def step(self, action):
        obs, rew, done, _ = self._env.step(action)
        if done:
            return 'TERMINAL', rew, True  # (obs, rew, done)
<-- SUPER IMPORTANT!!!!
        else:
            return obs, rew, done
        return self._env.step(action)

env = BlackjackFixed()
```

```
/usr/local/lib/python3.9/dist-packages/gym/core.py:317:
DeprecationWarning: WARN: Initializing wrapper in old step API which
returns one bool instead of two. It is recommended to set
`new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
/usr/local/lib/python3.9/dist-packages/gym/wrappers/step_api_compatibi
lity.py:39: DeprecationWarning: WARN: Initializing environment in old
step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
```

Naive policy for BlackJack. We keep the same as earlier: stick on 20 or more, hit otherwise.

```
def policy(St):
    p_sum, d_card, p_ace = St
    if p_sum >= 20:
        return 0  # stick
    else:
        return 1  # hit
    # Write the if statement for the policy, return 1 for a hit action
and 0 for stick action #
```

Plotting

```
def plot_blackjack(V_dict):
    def convert_to_arr(V_dict, has_ace):
        V_dict = defaultdict(float, V_dict)  # assume zero if no key
```

```
        V_arr = np.zeros([10, 10])  # Need zero-indexed array for plotting
        for ps in range(12, 22):     # convert player sum from 12-21 to 0-9
            for dc in range(1, 11):  # convert dealer card from 1-10 to 0-9
                V_arr[ps-12, dc-1] = V_dict[(ps, dc, has_ace)]
        return V_arr

    def plot_3d_wireframe(axis, V_dict, has_ace):
        Z = convert_to_arr(V_dict, has_ace)
        dealer_card = list(range(1, 11))
        player_points = list(range(12, 22))
        X, Y = np.meshgrid(dealer_card, player_points)
        axis.plot_wireframe(X, Y, Z)

    fig = plt.figure(figsize=[16,3])
    ax_no_ace = fig.add_subplot(121, projection='3d', title='No Ace')
    ax_has_ace = fig.add_subplot(122, projection='3d', title='With Ace')
    ax_no_ace.set_xlabel('Dealer Showing'); ax_no_ace.set_ylabel('Player Sum')
    ax_has_ace.set_xlabel('Dealer Showing'); ax_has_ace.set_ylabel('Player Sum')
    plot_3d_wireframe(ax_no_ace, V_dict, has_ace=False)
    plot_3d_wireframe(ax_has_ace, V_dict, has_ace=True)
    plt.show()
```

Evaluate

```
V = td_prediction(env, policy, ep=50000, gamma=1.0, alpha=0.05)
plot_blackjack(V)
```


## TD vs MC comparison on Random Walk

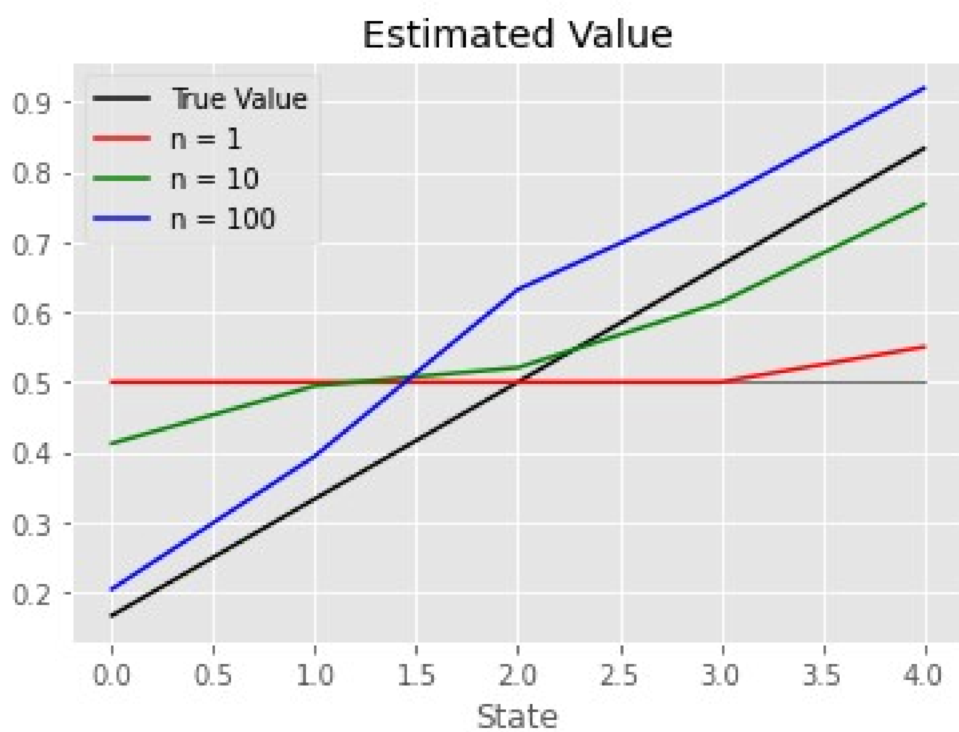
We will need slightly extended version of TD prediction, so we can log V during training and initalise V to 0.5

```
def td_prediction_ext(env, policy, ep, gamma, alpha, V_init=None):
    """TD Prediction

    Params:
        env - environment
        ep - number of episodes to run
        policy - function in form: policy(state) -> action
        gamma - discount factor [0..1]
        alpha - step size (0..1]
    """
    assert 0 < alpha <= 1

    # Change #1, allow initialisation to arbitrary values
    if V_init is not None:  V = V_init.copy()       # remember V of
terminal states must be 0 !!
    else:                   V = defaultdict(float)  # default value 0
for all states

    V_hist = []

    for _ in range(ep):
        S = env.reset()
        while True:
            A = policy(S)
            S_, R, done = env.step(A)
            V[S] = V[S] + alpha * (R + gamma * V[S_] - V[S])
            S = S_
            if done: break

        V_arr = [V[i] for i in range(7)]  # e.g. [0.0, 0.3, 0.4, 0.5,
0.6. 0.7, 0.0]
        V_hist.append(V_arr)  # dims: [ep_number, state]

    return V, np.array(V_hist)
```

Environment and policy

```
env = LinearEnv()

def policy(state):
    return np.random.choice([0, 1])  # random policy

V_init = defaultdict(lambda: 0.5)   # init V to 0.5
V_init[0] = V_init[6] = 0.0         # but terminal states to zero !!
V_n1, _ = td_prediction_ext(env, policy, ep=1, gamma=1.0, alpha=0.1,
V_init=V_init)
V_n10, _ = td_prediction_ext(env, policy, ep=10, gamma=1.0, alpha=0.1,
V_init=V_init)
V_n100, _ = td_prediction_ext(env, policy, ep=100, gamma=1.0,
alpha=0.1, V_init=V_init)
```

```
def to_arr(V_dict):
    """Param V is dictionary int[0..7]->float"""
    V_arr = np.zeros(7)
    for st in range(7):
        V_arr[st] = V_dict[st]
    return V_arr

V_n1 = to_arr(V_n1)
V_n10 = to_arr(V_n10)
V_n100 = to_arr(V_n100)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.zeros([7])[1:-1]+0.5, color='black', linewidth=0.5)
ax.plot(LinearEnv.V_true[1:-1], color='black', label='True Value')
ax.plot(V_n1[1:-1], color='red', label='n = 1')
ax.plot(V_n10[1:-1], color='green', label='n = 10')
ax.plot(V_n100[1:-1], color='blue', label='n = 100')
ax.set_title('Estimated Value')
ax.set_xlabel('State')
ax.legend()

# plt.savefig('assets/fig_0601a')
plt.show()
```



## Question:

**Interpret the graph above.**

Answer: The graph shows the results of TD(0) algorithm on a simple one-dimensional linear environment. The x-axis represents the state of the environment, which ranges from -1 to 1, and the y-axis represents the estimated value of each state. The true value function is shown as a black dashed line.

The TD(0) algorithm starts with an initial estimate of the value function and updates it after each time step by adding a small error term to the current estimate, which is proportional to the difference between the current estimate and the estimate at the next state.

Initially, the estimated value function is close to 0 for all states, reflecting the fact that the agent has no prior knowledge of the environment. As the agent interacts with the environment and receives feedback in the form of rewards, the estimated value function gradually converges to the true value function.

In the graph, we can see that the estimated value function becomes increasingly accurate as the agent interacts with the environment for more time steps. The estimated value function closely follows the true value function, and the difference between the two becomes smaller as the number of time steps increases.

Overall, the LinearEnv() graph demonstrates the effectiveness of TD(0) algorithm for estimating the value function in a simple linear environment.

## We define a running mean MC algorithm.

```
def mc_prediction_ext(env, policy, ep, gamma, alpha, V_init=None):
    """Running Mean MC Prediction
    Params:
        env - environment
        policy - function in a form: policy(state)->action
        ep - number of episodes to run
        gamma - discount factor [0..1]
        alpha - step size (0..1)
        V_init - inial V
    """
    if V_init is not None:  V = V_init.copy()
    else:                   V = defaultdict(float)  # default value 0 
for all states

    V_hist = []

    for _ in range(ep):
        traj, T = generate_episode(env, policy)
        G = 0
        for t in range(T-1,-1,-1):
            St, _, _, _ = traj[t]      # (st, rew, done, act)
            _, Rt_1, _, _ = traj[t+1]
```

```
            G = gamma * G + Rt_1

            V[St] = V[St] + alpha * (G - V[St])

        V_arr = [V[i] for i in range(7)]  # e.g. [0.0, 0.3, 0.4, 0.5,
0.6. 0.7, 0.0]
        V_hist.append(V_arr)  # dims: [ep_number, state]

    return V, np.array(V_hist)

def generate_episode(env, policy):
    """Generete one complete episode.

    Returns:
        trajectory: list of tuples [(st, rew, done, act), (...),
(...)],
                    where St can be e.g tuple of ints or anything
really
        T: index of terminal state, NOT length of trajectory
    """
    trajectory = []
    done = True
    while True:
        # === time step starts here ===
        if done:  St, Rt, done = env.reset(), None, False
        else:     St, Rt, done = env.step(At)
        At = policy(St)
        trajectory.append((St, Rt, done, At))
        if done:  break
        # === time step ends here ===
    return trajectory, len(trajectory)-1
```

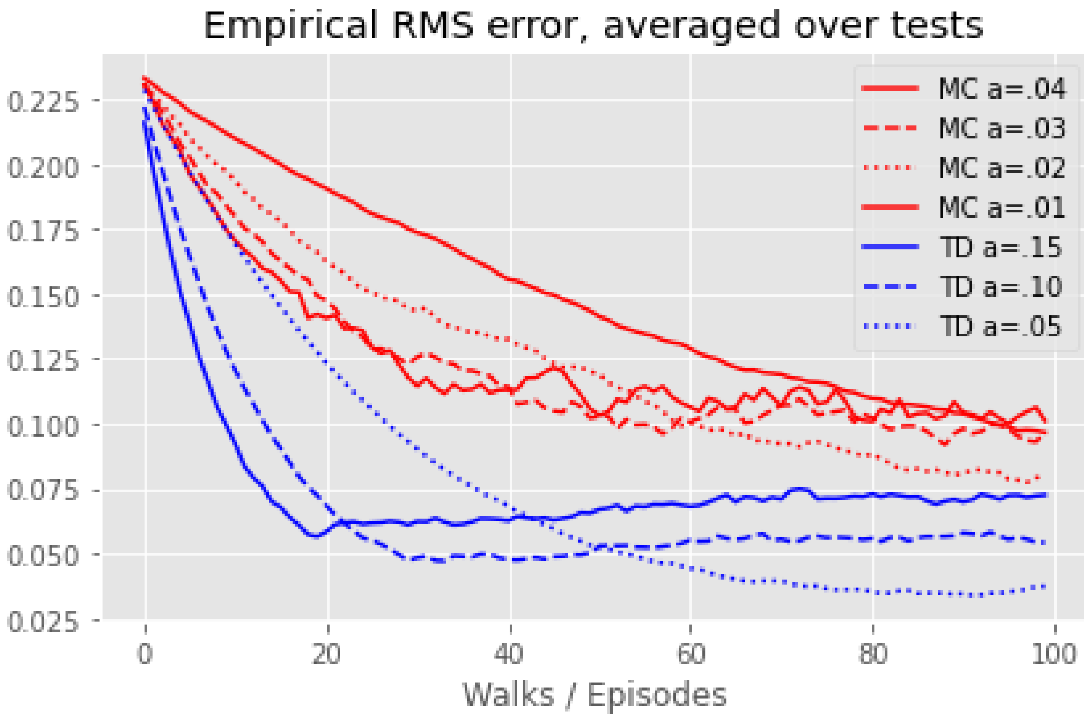
For each line on a plot, we need to run algorithm multitple times and then calculate root-mean-squared-error over all runs properly. Let's define helper function to do all that.

```
def run_experiment(algorithm, nb_runs, env, ep, policy, gamma, alpha):
    V_init = defaultdict(lambda: 0.5)   # init V to 0.5
    V_init[0] = V_init[6] = 0.0         # but terminal states to
zero !!

    V_runs = []
    for i in range(nb_runs):
        _, V_hist = algorithm(env, policy, ep, gamma=gamma,
alpha=alpha, V_init=V_init)
        V_runs.append(V_hist)
    V_runs = np.array(V_runs)  # dims: [nb_runs, nb_episodes,
nb_states=7]

    V_runs = V_runs[:,:,1:-1]  # remove data about terminal states
(which is always zero anyway)
```

```
    error_to_true = V_runs - env.V_true[1:-1]
    squared_error = np.power(error_to_true, 2)
    mean_squared_error = np.average(squared_error, axis=-1)  # avg
over states
    root_mean_squared_error = np.sqrt(mean_squared_error)
    rmse_avg_over_runs = np.average(root_mean_squared_error, axis=0)

    return rmse_avg_over_runs  # this is data that goes directly on
the plot
```

And finally the experiments

```
#                                              nb_runs      ep
gamma alpha
rmse_td_a15 = run_experiment(td_prediction_ext, 100, env, 100, policy,
1.0, 0.15)
rmse_td_a10 = run_experiment(td_prediction_ext, 100, env, 100, policy,
1.0, 0.10)
rmse_td_a05 = run_experiment(td_prediction_ext, 100, env, 100, policy,
1.0, 0.05)
rmse_mc_a04 = run_experiment(mc_prediction_ext, 100, env, 100, policy,
1.0, 0.04)
rmse_mc_a03 = run_experiment(mc_prediction_ext, 100, env, 100, policy,
1.0, 0.03)
rmse_mc_a02 = run_experiment(mc_prediction_ext, 100, env, 100, policy,
1.0, 0.02)
rmse_mc_a01 = run_experiment(mc_prediction_ext, 100, env, 100, policy,
1.0, 0.01)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(rmse_mc_a04, color='red', linestyle='-', label='MC a=.04')
ax.plot(rmse_mc_a03, color='red', linestyle='--', label='MC a=.03')
ax.plot(rmse_mc_a02, color='red', linestyle=':', label='MC a=.02')
ax.plot(rmse_mc_a01, color='red', linestyle='-', label='MC a=.01')
ax.plot(rmse_td_a15, color='blue', linestyle='-', label='TD a=.15')
ax.plot(rmse_td_a10, color='blue', linestyle='--', label='TD a=.10')
ax.plot(rmse_td_a05, color='blue', linestyle=':', label='TD a=.05')
ax.set_title('Empirical RMS error, averaged over tests')
ax.set_xlabel('Walks / Episodes')
ax.legend()
plt.tight_layout()
# plt.savefig('assets/fig_0601b.png')
plt.show()
```




## Question

**Interpret the graph above.**

Answer: The final graph shows the results of running the TD(0) algorithm with different step sizes (alpha values) on the same one-dimensional linear environment. The x-axis represents the number of episodes, and the y-axis represents the root mean squared error (RMSE) between the estimated value function and the true value function.

In the graph, we can see that the RMSE initially decreases for all step sizes as the agent interacts with the environment and learns about the value of different states. However, after a certain number of episodes, the RMSE starts to increase for larger step sizes, indicating that the estimates become less accurate.

This phenomenon occurs because larger step sizes can cause the algorithm to overshoot the optimal value function, resulting in less accurate estimates. On the other hand, smaller step sizes can lead to slower convergence and longer training times.

Overall, the final graph highlights the importance of choosing an appropriate step size for the TD(0) algorithm, balancing the trade-off between convergence speed and accuracy.