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

In [1]:

```
!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
```

```
Requirement already satisfied: gym in /usr/local/lib/python3.7/dist-packages (0.17.3)
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist-packages (from gym) (1.21.5)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/python3.7/dist-packages (from gym) (1.3.0)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python3.7/dist-packages (from gym) (1.5.0)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from gym) (1.4.1)
Requirement already satisfied: future in /usr/local/lib/python3.7/dist-packages (from pyglet<=1.5.0,>=1.4.0->gym) (0.16.0)
--2022-02-23 22:05:35--  https://raw.githubusercontent.com/lcharlin/80-629/master/week13-RL/blackjack.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4251 (4.2K) [text/plain]
Saving to: ‘blackjack.py’

blackjack.py        100%[===================>]   4.15K  --.-KB/s    in 0s

2022-02-23 22:05:35 (31.1 MB/s) - ‘blackjack.py’ saved [4251/4251]

--2022-02-23 22:05:35--  https://raw.githubusercontent.com/lcharlin/80-629/master/week13-RL/plotting.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3457 (3.4K) [text/plain]
Saving to: ‘plotting.py’

plotting.py         100%[===================>]   3.38K  --.-KB/s    in 0s

2022-02-23 22:05:35 (38.1 MB/s) - ‘plotting.py’ saved [3457/3457]
```

In [2]:

```
# imports
%matplotlib inline

import gym
import matplotlib
import numpy as np
import sys

from collections import defaultdict

from blackjack import BlackjackEnv
import plotting

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

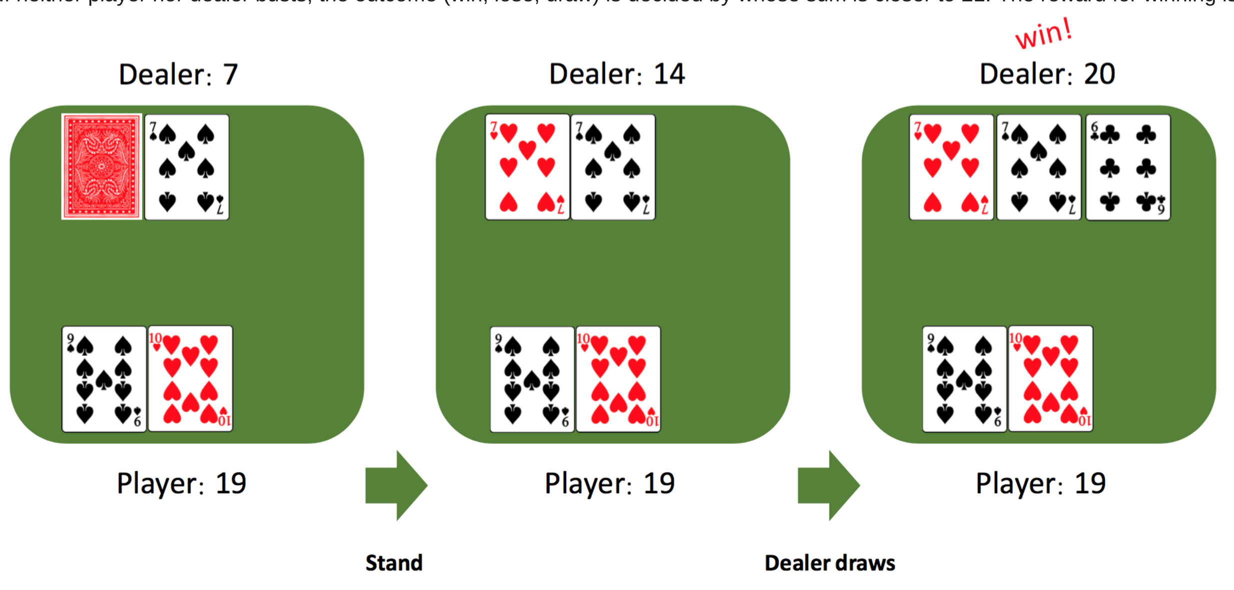
BlackJack Rules

First, we define the Blackjack environment:

- Black Jack is a card game where a player must obtain cards such that their sum is as close to 21 without exceeding it.
- Face cards (Jack, Queen, King) have point value 10. Aces can either count as 11 or 1, and it's called 'usable' at 11.
- In our example below, the player plays against a dealer. The dealer has a fixed policy of always asking for an additional card until the sum of their cards is above 17.
- Stationarity: This game is placed with an infinite deck (or with replacement).

Game Process:

1. The game starts with each (player and dealer) having one face up and one face down card.
2. 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.
3. 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.



In [4]:

```
env = BlackjackEnv()
```

1.2 Monte Carlo prediction

Recall that the Monte Carlo prediction algorithm provides a method for evaluating a given policy (π), 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.

In [8]:

```
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) # YOUR CODE HERE #
            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) # YOUR CODE HERE #
            # Sum up all rewards since the first occurance
            # YOUR CODE HERE #
            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 # YOUR CODE HERE #
            returns_count[state] += 1  # YOUR CODE HERE #
            V[state] = returns_sum[state]/returns_count[state] # YOUR CODE HERE #

        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.**

In [9]:

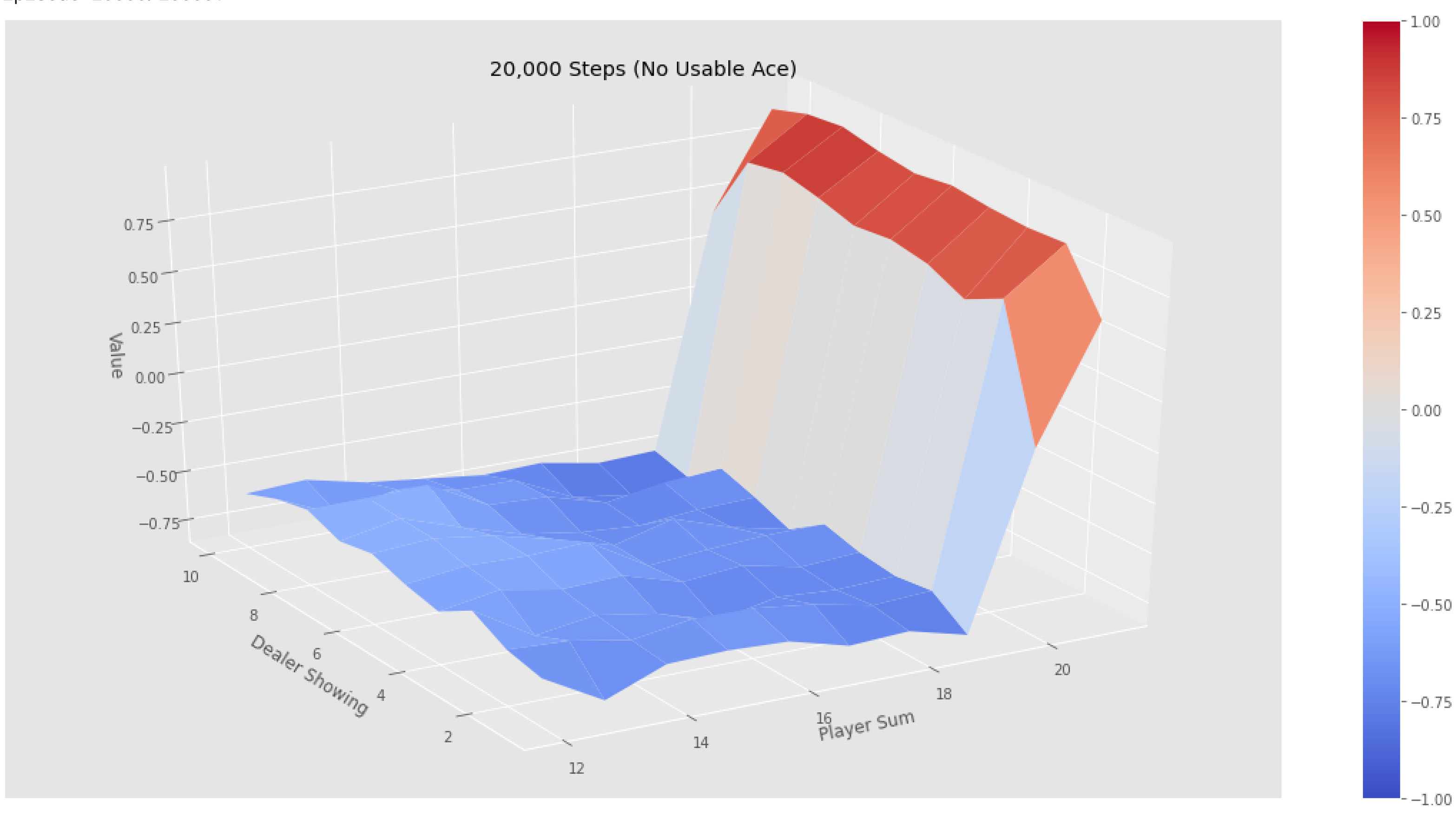
```
def sample_policy(observation):
    """
    A policy that sticks if the player score is >= 20 and hits otherwise.
    """
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

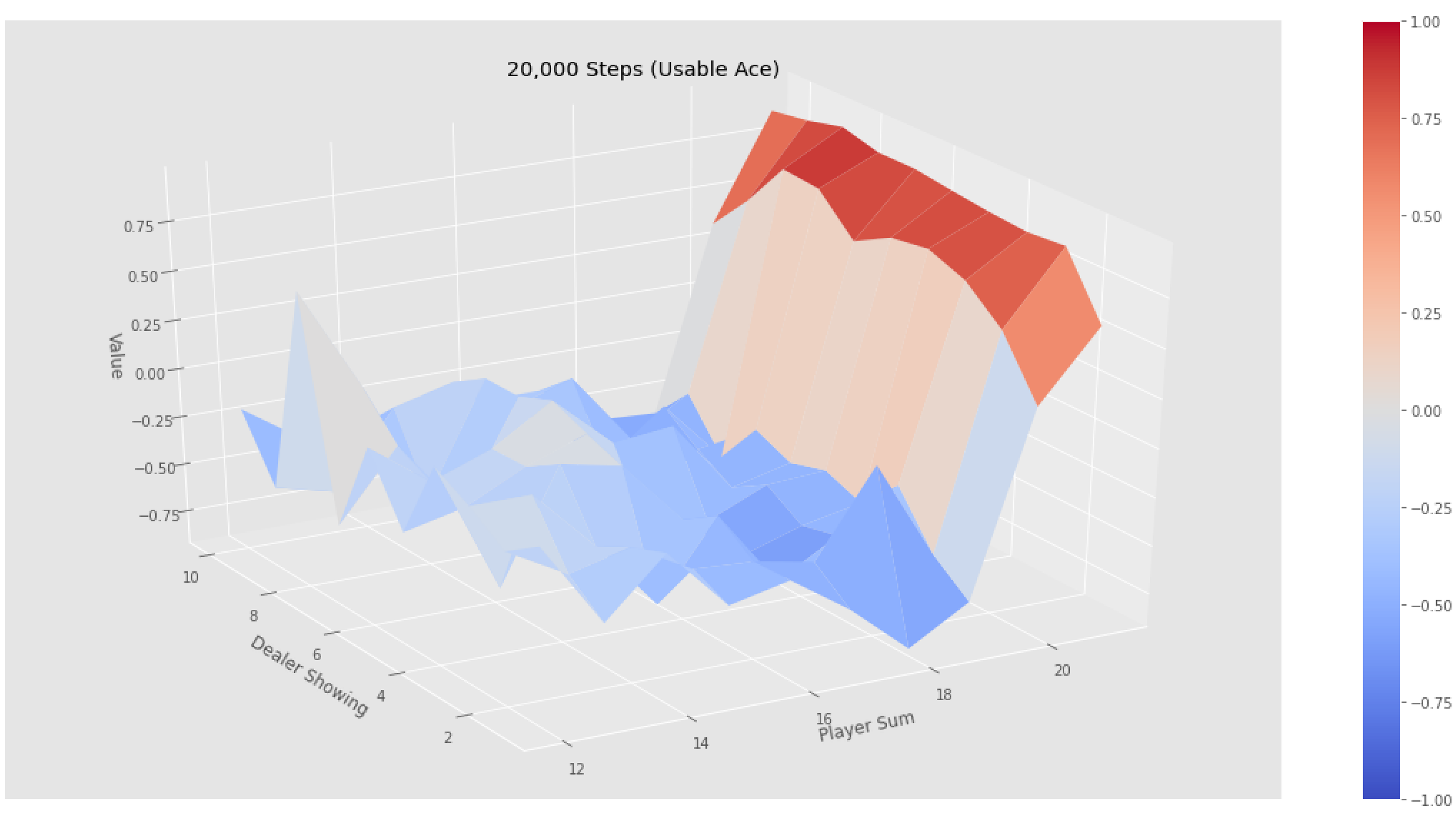
We now evaluate the policy for 20k iterations.

In [10]:

```
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:

When with Ace, there are more variance in the values earned.

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.

In [11]:

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

```
Output hidden; open in https://colab.research.google.com to view.
```

Question

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

Answer:

As the number of episodes sampled increases, the variance of values when a player has a useable ace decreases before getting to 20/21. This indicates that the agent has been able to more precisely learn when to hit even with a useable ace.

Do not forget Part 2 of the Lab, refer to the second notebook

In [ ]:

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

Tabular TD(0) for estimating v_π

Input: the policy π 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 π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$
    $S \leftarrow S'$
  until S is terminal

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

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from collections import defaultdict
import gym  # blacjack
```

Implementation of TD(0)

Start by filling the following blanks in the code below:

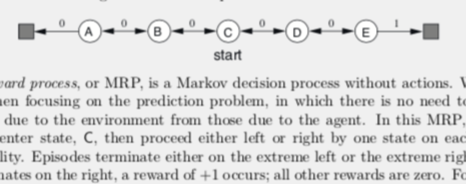
In [4]:
```
def td_prediction(env, policy, ep, gamma, alpha):
    """TD Prediction

    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) # YOUR CODE HERE #
            S_, R, done = env.step(A) # YOUR CODE HERE #
            V[S] = V[S] + alpha * (R + gamma * V[S_] - V[S]) # YOUR CODE HERE #
            S = S_ # YOUR CODE HERE #
            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-α 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 $\frac{5}{6}$.

In [5]:
```
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
```

In [6]:
```
env = LinearEnv()
```

Plotting helper function:

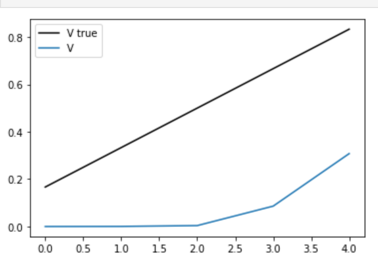
In [7]:
```
def plot(V_dict):
    """Param V is dictionary int[0..7]->float"""

    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:

In [8]:
```
def policy(state):
    return np.random.choice([0, 1])  # random policy
```

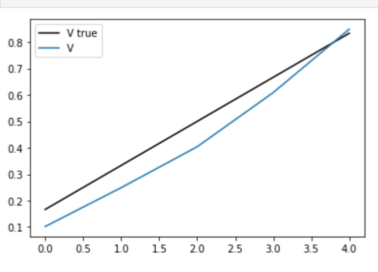
For 10 episodes

In [9]:
```
V = td_prediction(env, policy, ep=10, gamma=1.0, alpha=0.1)
plot(V)
```



For 1000 episodes

In [11]:
```
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'.

In [12]:
```
class BlackjackFixed():
    def __init__(self):
        self._env = gym.make('Blackjack-v0')

    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)
```

In [13]:
```
env = BlackjackFixed()
```

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

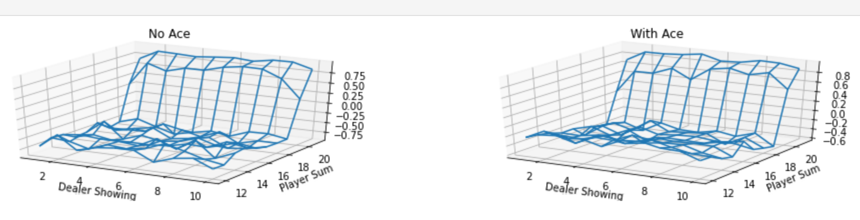
In [15]:
```
def policy(St):
    p_sum, d_card, p_ace = St
    if p_sum >= 20:
        return 0  # don't hit when p_sum is equal or larger than 20
    else:
        return 1  # otherwise, hit
    # YOUR CODE HERE #
    # Write the if statement for the policy, return 1 for a hit action and 0 for stick action #
```

Plotting

In [16]:
```
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

In [17]:
```
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

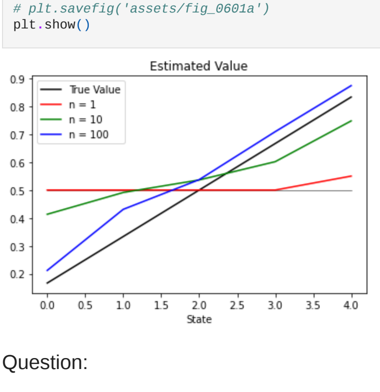
In [20]:
```
def td_prediction_ext(env, policy, ep, gamma, alpha, V_init=None):
    """TD Prediction

    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) # YOUR CODE HERE #
            S_, R, done = env.step(A) # YOUR CODE HERE #
            V[S] = V[S] + alpha * (R + gamma * V[S_] - V[S]) # YOUR CODE HERE #
            S = S_ # YOUR CODE HERE #
            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

In [21]:
```
env = LinearEnv()
```

In [22]:
```
def policy(state):
    return np.random.choice([0, 1])  # random policy
```

In [23]:
```
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 more episodes the updated td algorithm has, the closer it is to converge to the true value. The random walk has been able to add more exploration to the agent's behavior which helps it find the actions that lead closer to the true value

We define a running mean MC algorithm.

In [24]:
```
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)
```

In [25]:
```
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
```

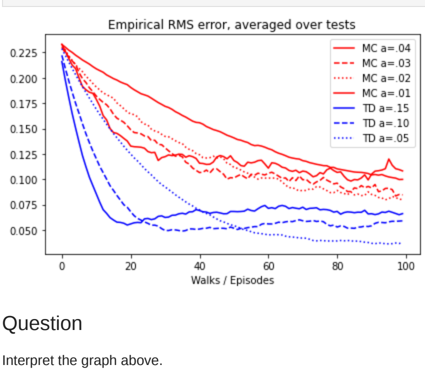
For each line on a plot, we need to run algorithm multitple times and then calculate root-mean-squared-error over all runs properly. Let's define helper function to do all that.

In [26]:
```
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

In [27]:
```
#                                               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)
```

In [28]:
```
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:

In general, TD is better than MC algorithm in terms of lower RMS error from the beginning til the end. For TD, as the influence of the latest action decreased (alpha decreased), the performance converged closer to the true value. It means a large alpha will make the agent pay too much attention on the newest rewards and it seems not good.

From the graph we can see that, MC performs so poor with it has a large alpha. It is understandable because it is model free.

Moreover, since the TD can learn directly from experiences and can learn before knowing the outcome, it can outperform MC. Nevertheless, TD does have low variance but some bias, while MC has zero bias and higher variance. TD also is dependent on initial value, evident from its different expectations near the initials walks

In [ ]: