Reinforcement Learning

Author: Nicklas Hansen (s153077@student.dtu.dk)

Chapter: 5, 6

Exercise 5.5

We are to determine the first-visit and every-visit estimators of the value of a nonterminal state in the case where we transition back to the nonterminal state with probability p and transition to the terminal state with probability $1 - p$. If the reward is $+1$ on all transitions, $\gamma = 1$ and we experience an episode of 10 steps before termination (return of 10), then the first-visit estimator is given by:

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}\, G_t}{|\mathcal{T}(s)|} \Rightarrow V(s_{nt})_{fv} = \frac{1 \cdot 10}{1} = 10$$

where s_{nt} is the nonterminal state and fv denotes that it is the first-visit estimator of V. $|\mathcal{T}(s)| = 1$, as we only consider the first visit to nonterminal state s_{nt}. Note that the ratio ρ is 1, as the probability of taking the single action in the nonterminal state is 1 across all policies. Therefore, ordinary importance sampling and weighted importance sampling are equivalent.
In the every-visit case, the outcome is slightly different due to numerous visits to the same state:

$$V(s_{nt})_{ev} = \frac{1 \cdot \operatorname{avg}(10, 9, 8, \dots, 2, 1)}{1} = 5.5$$

as the nonterminal state s_{nt} is visited 10 times.

Exercise 5.6

If we now consider the action-value function $Q(s, a)$ instead of the state-value function from equation (5.6) of the book, we have that:

$$Q(s, a) = \frac{\sum_{t \in T(s,a)} \rho_{t:T(t)} G_t}{\sum_{t \in T(s,a)} \rho_{t:T(t)}}$$

where the definition of ρ remains the same (as it simply defines the relative probability of a given trajectory following the two policies π and b) and we simply consider state-action pairs (s, a) rather than states alone.

Exercise 5.7

Weighted importance sampling is statistically biased, as the target policy π typically differs from the behavior policy b. Initially, returns that the weighted importance sampling observe is the expectation of v_b rather than v_π which shows a clear bias. This is evident in Figure 5.3 of the book in that the MSE increases slightly over the first 8 episodes or so, and then it decreases again as the bias becomes less dominant due to weighting.

Exercise 5.9

We can rewrite the algorithm for first-visit MC policy evaluation from section 5.1 to use an incremental implementation for samples averages, such that we do not have to store all returns in a list. The last two lines are modified to read as follows:

$$\begin{aligned} n(S_t) &= n(S_t) + 1 \\ V(S_t) &= V(S_t) + \frac{G - V(S_t)}{n(S_t)} \end{aligned}$$

where n is initialized as an occurrence counter for all states $s \in S$. In that way, we can store a sample average of the estimated state-value for all states using $O(S)$ space and perform updates in $O(1)$ time, assuming that states in V are accessible by direct addressing.

Exercise 5.10

In this exercise we are to rewrite the weighted-average update rule (5.7) to get the recursive definition (5.8) in the book. If we write out the first recursion of V we get:

$$\begin{aligned} V_{n+1} &= \frac{\sum_{k=1}^{n} W_k G_k}{\sum_{k=1}^{n} W_k} \\ &= \frac{1}{W_n + \sum_{k=1}^{n-1} W_k} \left[W_n G_n + \sum_{k=1}^{n-1} W_k G_k \right] \\ &= \frac{1}{\sum_{k=1}^{n} W_k} \left[W_n G_n + V_n W_n \right] \end{aligned}$$

If we then define C_n recursively like the authors do in equation (5.8) of the book, we get:

$$\begin{aligned} V_{n+1} &= V_n + \frac{1}{C_n} \left[W_n G_n - V_n W_n \right], \quad C_n = C_{n-1} + W_n \\ &= V_n + \frac{W_n}{C_n} \left[G_n - V_n \right] \end{aligned}$$

Exercise 5.11

As we in the off-policy Monte Carlo control algorithm estimate a policy π, we need to compute the weight W at each time step of an episode. We know that W denotes the relative probability of a trajectory for policy π and our behavior policy b. W is computed incrementally as the relative probability of a transition being taken in a given state under the two policies:

$$W_t = W_{t-1} \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

We are evaluating W based on a given state-action pair and know that the probability of that transition being followed under π is 1, so we have that

$$\pi(A_t|S_t) = 1 \Rightarrow W_t = W_{t-1} \frac{1}{b(A_t|S_t)}$$

Exercise 6.2

When utilising MC methods, we rely on the return of the full episode, whereas TD learning also utilises bootstrapping. If we move to a new building and parking lot, i.e. change our starting conditions, we will with MC methods have to perform a full MC simulation again to estimate the value of the new starting state, while TD learning can bootstrap on the remainder of the route to obtain new estimates at a much faster rate. Therefore, estimates by TD learning are likely to be far better than MC methods, at least initially.
In the original scenario, one could also imagine that TD learning will converge faster than the MC-based methods if there is a high degree of randomness to the rewards (e.g. as discussed in the text, if something unexpected happens at a transition with a small probability.

Exercise 6.3

The fact that the value estimate for $V(A)$ changes from 0.5 to about 0.45 tells us that the first episode walked from C (start) to A to the terminal state, yielding a reward of 0 at all states. As we use TD(0) we update a state value at the time-step following the time-step in which we were in that state. As the estimate for all states is 0.5, we have that - moving from e.g. C to B, $V(C) = V(C) + 0.1[0 + \gamma\, V(B) - V(C)] = V(C)$ for $\gamma = 1$. When we move from A to the terminal state, however, we observe the following update:

$$V(A) = V(A) + 0.1[0 + \gamma\, V(T) - V(A)] = V(A) - 0.1V(A) = 0.45$$

where $V(T) = 0$ is the value of the terminal state (defined as 0). In this case, we do not have to make any assumptions about γ, as $V(T) = 0$ anyway.
After a few episodes, all states should start to see a change in their estimated value, as is evident in the figure of the book.

Exercise 6.4

From the empirical RMS error shown in the right graph of example 6.2 in the book, we observe that the TD method has a far steeper learning curve despite larger step-sizes, which can be attributed to the fact that we in TD(0) learning update V_t at time-step V_{t+1}, whereas the MC method updates all estimated state values at the termination of an episode. Therefore, many more rewards and state values influence the update of V_t and in the undiscounted case of MC (as is example 6.2), this requires a smaller step-size than for TD(0) learning.
Even when considering more values of α for comparison of the two algorithms, it should be evident that the conclusion is the same: TD(0) has the potential for learning faster, but may not converge to the true value of V unless the step-size is sufficiently small, whereas MC does require a small step-size but that also has great implications on the rate of convergence. Therefore, neither of the two algorithms should perform significantly better at any other fixed value of α which has not already been covered in the example.

Exercise 6.8

We now consider the action-value form of the TD error

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

and show that the action-value version of equation (6.6) in the book holds for that form of the TD error:

$$\begin{aligned}
G_t - Q(S_t, A_t) &= R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) + \gamma Q(S_{t+1}, A_{t+1}) - \gamma Q(S_{t+1}, A_{t+1}) \\
&= \delta_t + \gamma \left(G_{t+1} - Q(S_{t+1}, A_{t+1})\right) \\
&= \delta_t + \gamma \delta_{t+1} + \gamma^2 \left(G_{t+2} - Q(S_{t+2}, A_{t+2})\right) \\
&= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k
\end{aligned}$$

which is the same result as in the derivation of equation (6.6).

Exercise 6.11

Q-learning is considered an off-policy control method because it greedily approximates q_*, the optimal action-value function, regardless of the policy being followed. Thus, Q-learning allows us to use an ε-greedy policy as behavioral policy (defining actions taken during learning), while our action-value function is updated based on a greedy action selection $\gamma \ \max_a Q(S', a)$.

Exercise 6.12

If action selection for the Q-learning algorithm is greedy, it becomes an on-policy method since its action selection and action-value function update is based on a common policy, $\pi = b$. The only difference between the two algorithms, then, is that Sarsa uses an ε-greedy policy, whereas Q-learning uses a greedy policy, and their actions are not always identical then due to exploration (and lack thereof).

Implementation (***MC Prediction***)

Code and results are displayed on the last few pages.
It is easy to see that the value function for no usable ace is smoother than the one for usable ace, which is due to the fact that usable ace is far less likely to occur and the MC-estimation is thus done over a smaller number of samples.

Implementation (***MC Control with Epsilon-Greedy Policy***)

Code and results are displayed on the last few pages.
With some slight alterations to the code for the previous programming exercise, we have an MC control. Rather than estimating the state-value function V, we instead consider state-action pairs and estimate the action-value function Q. Additionally, we switch out the predefined policy from the previous exercise with an ε-greedy policy, such that all actions are considered with non-zero probability. After 5,000,000 episodes, we have a pretty good estimate and a smooth curve for scenarios both with and without a usable ace.

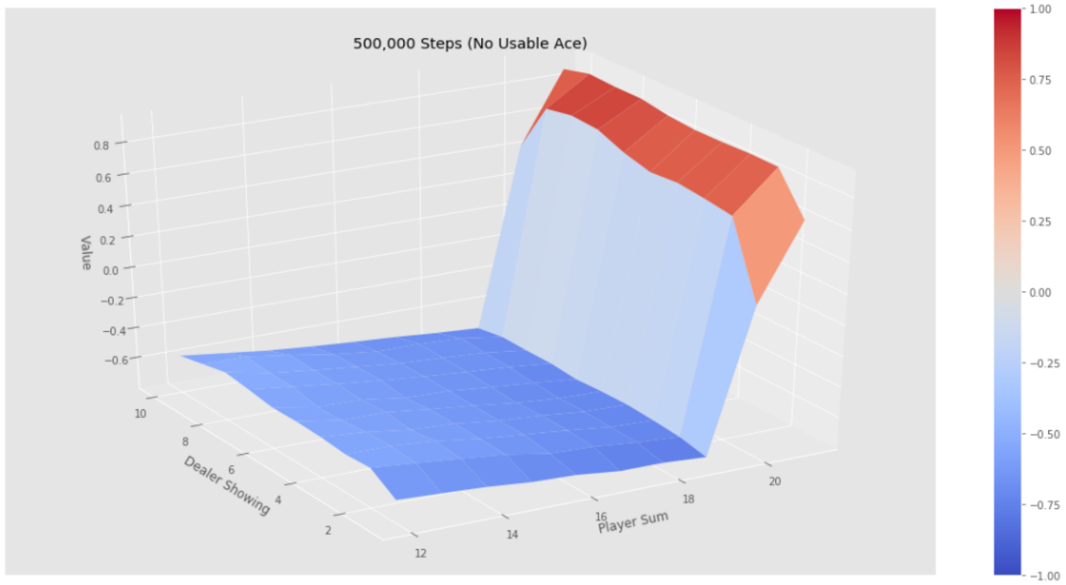
Implementation (***Off-Policy MC Control with Weighted Importance Sampling***)

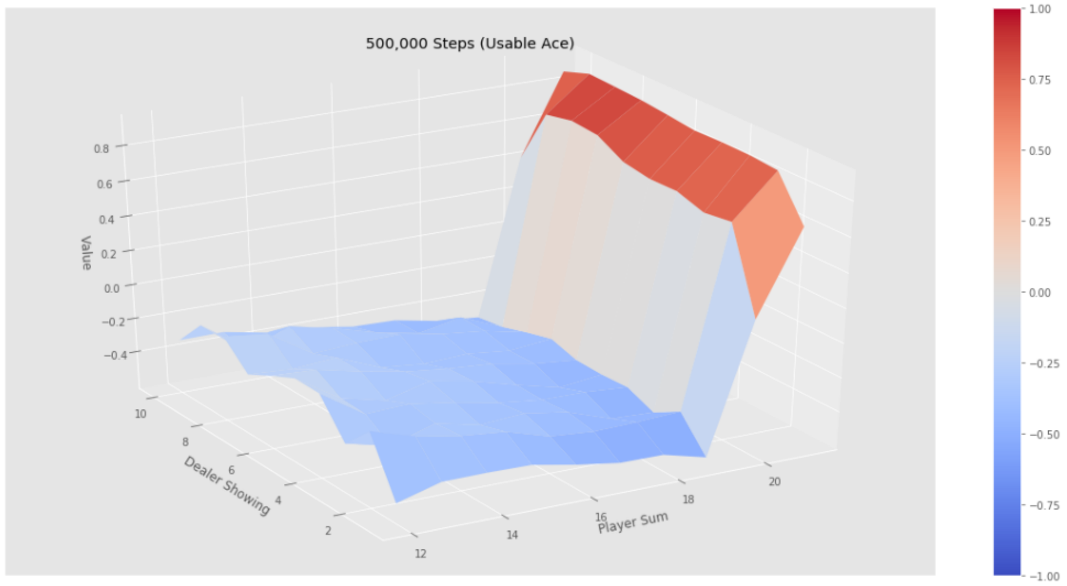
Code and results are displayed on the last few pages.
The solutions to the two previous exercises are recycled to produce an off-policy MC control using weighted importance sampling. The resulting value function estimations are slightly more noisy than those learned on-policy.

```python
def mc_prediction(policy, env, num_episodes, gamma=1.0):
    """
    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.
        gamma: 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 = {}
    returns_count = {}

    # The final value function
    V = {}

    for episode in range(num_episodes):

        # env.reset() resets the env by dealing new cards and returns the
        # starting state (sum of hand, one dealer card, usable ace) as a tupe.
        St = env.reset()
        seq_states, seq_rewards, terminate = [], [], False

        # Iterate through episode until termination
        while not terminate:

            At = policy(St)

            # env.step() takes action At and returns the new state
            # as a tuple, the reward for taking action At and a flag
            # that is True if we end up in the terminal state. For some
            # reason, step() also returns an empty dict that we ignore.
            St_new, Rt, terminate, _ = env.step(At)

            # Append state and given reward to sequence and
            # move to the new state after taking action At.
            seq_states.append(St)
            seq_rewards.append(Rt)
            St = St_new

        # Now that we have terminated we update the average return
        # of all states that we visited during this particular episode.
        for idx, s in enumerate(set(seq_states)):

            # Get index of first visit
            fv = seq_states.index(s)

            # Create key in dict if not exist
            if s not in returns_sum:
                returns_sum[s] = 0
                returns_count[s] = 0

            # Compute sampled return from state
            G = 0
            for t, Rt in enumerate(seq_rewards[fv:]):
                G += (gamma ** t) * Rt

            # Update value function estimation of state
            returns_sum[s] += G
            returns_count[s] += 1
            V[s] = returns_sum[s] / returns_count[s]

    return V
```

```python
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
```

```python
V_10k = mc_prediction(sample_policy, env, num_episodes=10000)
plotting.plot_value_function(V_10k, title="10,000 Steps")

V_500k = mc_prediction(sample_policy, env, num_episodes=500000)
plotting.plot_value_function(V_500k, title="500,000 Steps")
```


500,000 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

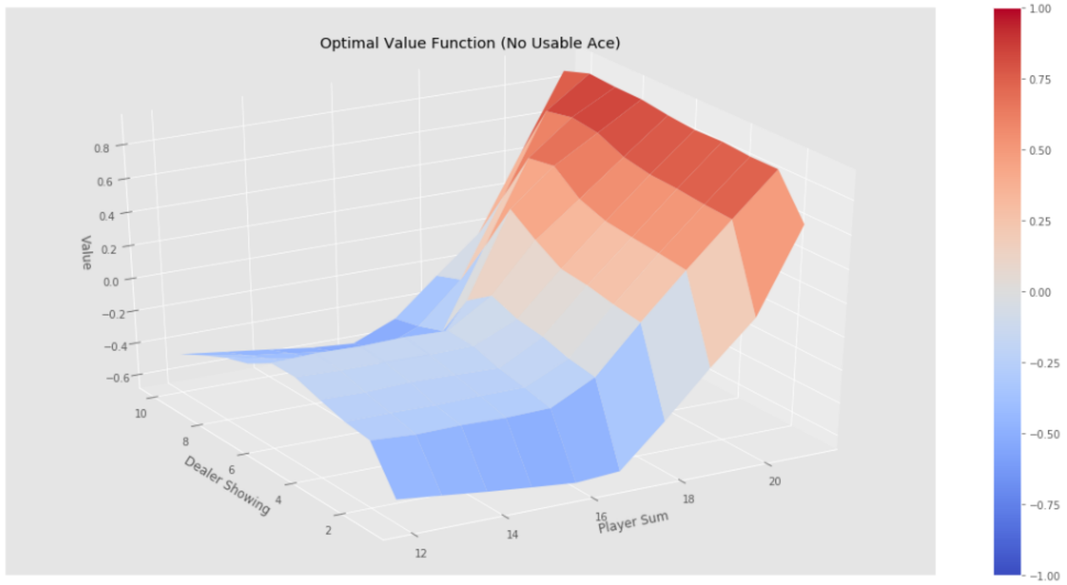

500,000 Steps (Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
10
8
6
4
2
12
14
16
18
20
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

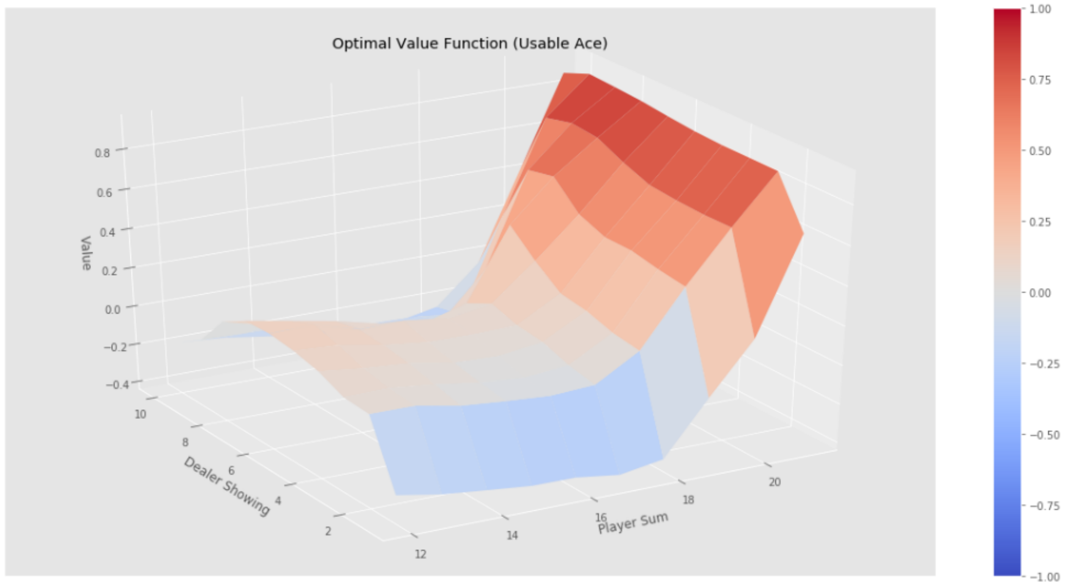
```
def make_epsilon_greedy_policy(Q, epsilon, nA):
    """
    Creates an epsilon-greedy policy based on a given Q-function and epsilon.

    Args:
        Q: A dictionary that maps from state -> action-values.
            Each value is a numpy array of length nA (see below)
        epsilon: The probability to select a random action . float between 0 and 1.
        nA: Number of actions in the environment.

    Returns:
        A function that takes the observation as an argument and returns
        the probabilities for each action in the form of a numpy array of length nA.

    """
    def policy_fn(observation):

        # Allow any action to be taken with at least epsilon/nA probability
        A = np.array([epsilon/nA] * nA)

        # Create action-value if not exist in Q
        if observation not in Q:
            Q[observation] = np.zeros(env.action_space.n)

        # Allow greedy action to be taken with 1 - epsilon + epsilon/nA probability
        A[np.argmax(Q[observation])] += (1 - epsilon)

        return A


    return policy_fn
```

```
def mc_control_epsilon_greedy(env, num_episodes, gamma=1.0, epsilon=0.1):
    """
    Monte Carlo Control using Epsilon-Greedy policies.
    Finds an optimal epsilon-greedy policy.

    Args:
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        discount_factor: Gamma discount factor.
        epsilon: Chance the sample a random action. Float betwen 0 and 1.

    Returns:
        A tuple (Q, policy).
        Q is a dictionary mapping state -> action values.
        policy is a function that takes an observation as an argument and returns
        action probabilities
    """

    # 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 = {}
    returns_count = {}

    # The final action-value function
    Q = {}

    # Use epsilon-greedy policy
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for episode in range(num_episodes):

        # env.reset() resets the env by dealing new cards and returns the
        # starting state (sum of hand, one dealer card, usable ace) as a tupe.
        St = env.reset()
        seq_sa, seq_rewards, terminate = [], [], False

        # Iterate through episode until termination
        while not terminate:

            p = policy(St)
            At = np.random.choice(np.arange(len(p)), p=p)

            # env.step() takes action At and returns the new state
            # as a tuple, the reward for taking action At and a flag
            # that is True if we end up in the terminal state. For some
            # reason, step() also returns an empty dict that we ignore.
            St_new, Rt, terminate, _ = env.step(At)

            # Append state and given reward to sequence and
            # move to the new state after taking action At.
            seq_sa.append((St, At))
            seq_rewards.append(Rt)
            St = St_new

        # Now that we have terminated we update the average return
        # of all states that we visited during this particular episode.
        for idx, sa in enumerate(set(seq_sa)):

            # Get index of first visit
            fv = seq_sa.index(sa)

            # Create key in dict if not exist
            if sa not in returns_sum:
                returns_sum[sa] = 0
                returns_count[sa] = 0

            # Compute sampled return from state
            G = 0
            for t, Rt in enumerate(seq_rewards[fv:]):
                G += (gamma ** t) * Rt

            # Update action-value function estimation for pair
            returns_sum[sa] += G
            returns_count[sa] += 1
            Q[sa[0]][sa[1]] = returns_sum[sa] / returns_count[sa]

    return Q, policy
```

```
Q, policy = mc_control_epsilon_greedy(env, num_episodes=5000000, epsilon=0.1)
```

```
# For plotting: Create value function from action-value function
# by picking the best action at each state
V = defaultdict(float)
for state, actions in Q.items():
    action_value = np.max(actions)
    V[state] = action_value
plotting.plot_value_function(V, title="Optimal Value Function")
```


Optimal Value Function (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


Optimal Value Function (Usable Ace)
Value
Dealer Showing
Player Sum
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
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

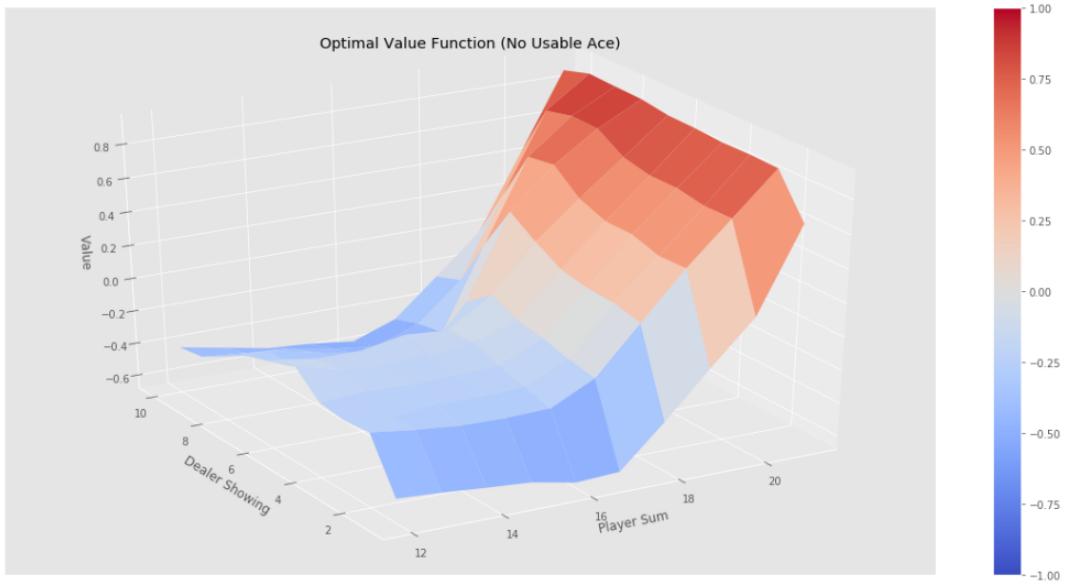
```python
def create_random_policy(nA):
    """
    Creates a random policy function.

    Args:
        nA: Number of actions in the environment.

    Returns:
        A function that takes an observation as input and returns a vector
        of action probabilities
    """

    def policy_fn(observation):
        return np.ones(nA) / nA

    return policy_fn
```

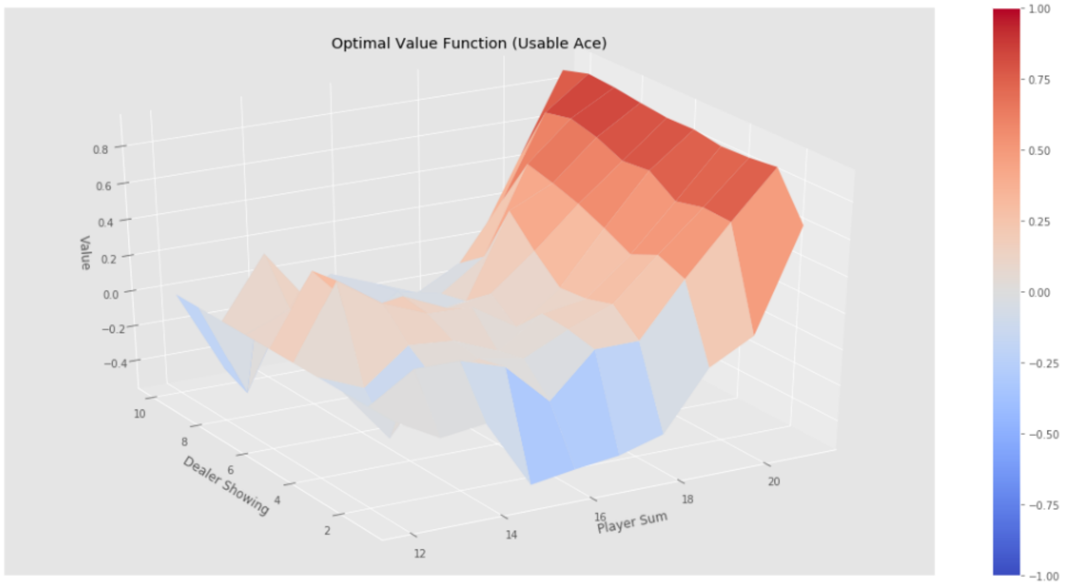
```python
def create_greedy_policy(Q):
    """
    Creates a greedy policy based on Q values.

    Args:
        Q: A dictionary that maps from state -> action values

    Returns:
        A function that takes an observation as input and returns a vector
        of action probabilities.
    """

    def policy_fn(observation):

        A = np.array([0] * len(Q[observation]))

        # Create action-value if not exist in Q
        if observation not in Q:
            Q[observation] = np.zeros(env.action_space.n)
            C[observation] = np.zeros(env.action_space.n)

        # Allow greedy action to be taken with a probability of 1
        A[np.argmax(Q[observation])] += 1

        return A

    return policy_fn
```

```python
def mc_control_importance_sampling(env, num_episodes, behavior_policy, gamma=1.0):
    """
    Monte Carlo Control Off-Policy Control using Weighted Importance Sampling.
    Finds an optimal greedy policy.

    Args:
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        behavior_policy: The behavior to follow while generating episodes.
            A function that given an observation returns a vector of probabilities for each action.
        discount_factor: Gamma discount factor.

    Returns:
        A tuple (Q, policy).
        Q is a dictionary mapping state -> action values.
        policy is a function that takes an observation as an argument and returns
        action probabilities. This is the optimal greedy policy.
    """

    # The final action-value function and
    # C from importance sampling formula
    Q, C = {}, {}

    # Our greedy policy that we want to learn
    target_policy = create_greedy_policy(Q)

    for episode in range(num_episodes):

        # env.reset() resets the env by dealing new cards and returns the
        # starting state (sum of hand, one dealer card, usable ace) as a tupe.
        St = env.reset()
        seq_all, terminate = [], False

        # Instantiate return and weight of returns (ratio)
        G, W = 0, 1

        # Iterate through episode until termination
        while not terminate:

            p = behavior_policy(St)
            At = np.random.choice(np.arange(len(p)), p=p)

            # env.step() takes action At and returns the new state
            # as a tuple, the reward for taking action At and a flag
            # that is True if we end up in the terminal state. For some
            # reason, step() also returns an empty dict that we ignore.
            St_new, Rt, terminate, _ = env.step(At)
            # Append state, action and given reward to sequence and
            # move to the new state after taking action At.
            seq_all.append((St, At, Rt))
            St = St_new

        # Now that we have terminated we update the average return
        # of all state-action pairs that we took during this particular episode.
        for t in range(len(seq_all)-1, 0, -1):

            # Get state, action and reward of step t
            s, a, r = seq_all[t]

            # Create key in dict if not exist
            if s not in C:
                C[s] = np.zeros(env.action_space.n)
                Q[s] = np.zeros(env.action_space.n)

            # Update return and weight
            G = r + gamma * G
            C[s][a] += W

            # Update action-value function estimation for pair.
            # This time, we have to do it incrementally.
            Q[s][a] += (W / C[s][a]) * (G - Q[s][a])

            # If action taken by behavior policy is not the greedy action, break
            if a != np.argmax(target_policy(s)):
                break

            # Otherwise update weight based on action taken
            W = W * 1./behavior_policy(s)[a]

    return Q, target_policy
```

```python
random_policy = create_random_policy(env.action_space.n)
Q, policy = mc_control_importance_sampling(env, num_episodes=5000000, behavior_policy=random_policy)
```

```python
# For plotting: Create value function from action-value function
# by picking the best action at each state
V = defaultdict(float)
for state, action_values in Q.items():
    action_value = np.max(action_values)
    V[state] = action_value
plotting.plot_value_function(V, title="Optimal Value Function")
```


Optimal Value Function (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


Optimal Value Function (Usable Ace)
Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
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