Kelompok21_RL_MonteCarlo_Blackjack

April 17, 2022

1 Nama Kelompok

- Diah Siti Fatimah Azzahrah ~ Better
- Annisa Kunarji Sari ~ Altas
- Ade Irvan Kurniawan ~ Persevere
- Fazeel Muhammad Zuhdi ~ Visioner
- Michael Chandra ~ Alan Turing

1.0.1 Part 0: Explore BlackjackEnv

Use the code cell below to create an instance of the Blackjack environment.

[1]:
```
import gym
env = gym.make('Blackjack-v0')
```

Setiap State adalah tuple dari: - jumlah pemain terkini $\in \{0, 1, \ldots, 31\}$, - kartu menhhadap ke atas dealer $\in \{1, \ldots, 10\}$, dan - apakah pemain memiliki kartu as yang dapat digunakan (`no` $= 0$, `yes` $= 1$).

agent punya 2 potensi aksi:

```
STICK = 0
HIT = 1
```

Verifikasi dengan menjalankan kode cell dibawah ini

[2]:
```
print(env.observation_space)
print(env.action_space)
```

```
Tuple(Discrete(32), Discrete(11), Discrete(2))
Discrete(2)
```

Jalankan sel kode di bawah ini untuk bermain Blackjack dengan kebijakan acak.

(*Kode saat ini memainkan Blackjack tiga kali - jangan ragu untuk mengubah nomor ini, atau menjalankan sel beberapa kali. Sel dirancang agar Anda mendapatkan pengalaman dengan keluaran yang dikembalikan saat agen berinteraksi dengan lingkungan.*)

[3]:
```
for i_episode in range(3):
    state = env.reset()
```

```
    while True:
        print(state)
        action = env.action_space.sample()
        state, reward, done, info = env.step(action)
        if done:
            print('End game! Reward: ', reward)
            print('You won :)\n') if reward > 0 else print('You lost :(\n')
            break
```

```
(15, 10, False)
End game! Reward:  -1.0
You lost :(

(5, 6, False)
End game! Reward:  -1.0
You lost :(

(16, 7, False)
End game! Reward:  -1.0
You lost :(
```

1.0.2 Part 1: MC Prediction: State Values

Di bagian ini, kita akan menulis implementasi prediksi MC Anda sendiri (untuk memperkirakan fungsi nilai keadaan).

dimulai dengan menyelidiki kebijakan di mana pemain selalu menempel jika jumlah kartunya melebihi 18. Fungsi `generate_episode_from_limit` mengambil sampel episode menggunakan kebijakan ini.

Fungsi menerima sebagai **input**: - `bj_env`: ini adalah instance dari OpenAI Gym's Blackjack environment.

dikembalikan sebagai **output**: - `episode`: ini adalah list dari (state, action, reward) tuples (of tuples) dan corresponds terhadap $(S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T)$, dimana T final time step. secara khusus, `episode[i]` mengembalikan (S_i, A_i, R_{i+1}), dan `episode[i][0]`, `episode[i][1]`, dan `episode[i][2]` mengembalikan S_i, A_i, dan R_{i+1}, masing - masing.

[4]:
```
def generate_episode_from_limit(bj_env):
    episode = []
    state = bj_env.reset()
    while True:
        action = 0 if state[0] > 18 else 1
        next_state, reward, done, info = bj_env.step(action)
        episode.append((state, action, reward))
        state = next_state
        if done:
```

```
            break
    return episode
```

Jalankan sel kode di bawah ini untuk bermain Blackjack dengan kebijakan tersebut.

(*Kode saat ini memainkan Blackjack tiga kali - jangan ragu untuk mengubah nomor ini, atau menjalankan sel beberapa kali. Sel dirancang untuk Anda agar terbiasa dengan output dari* `generate_episode_from_limit` *function.*)

[5]:
```
for i in range(3):
    print(generate_episode_from_limit(env))
```

```
[((14, 3, False), 1, 0.0), ((17, 3, False), 1, 0.0), ((18, 3, False), 1, -1.0)]
[((20, 7, True), 0, 1.0)]
[((19, 1, True), 0, 0.0)]
```

menuliskan implementasi dari prediksi MC pada first-visit atau every-visit MC prefiksi; dalam kasus Blackjack environment, tekniknya setara.

algoritma memiliki 3 argument: - `num_episodes`: Ini adalah jumlah episode yang dihasilkan melalui interaksi agen-lingkungan. - `generate_episode`: Ini adalah fungsi yang mengembalikan episode interaksi. - `gamma`: Ini adalah tingkat diskon. Berupa nilai antara 0 dan 1, inklusif (nilai default: `1`).

algoritma mengembalikan sebagai output: - `V`: ini adalah dictionary dimana `V[s]` adalah estimasi dari state `s`. Sebagai contoh, jika kode mengembalikan sebagai output berikut:

```
{(4, 7, False): -0.38775510204081631, (18, 6, False): -0.58434296365330851, (13, 2, False): -0
```

kemudian nilai dari state `(4, 7, False)` akan diestimasikan sebagai `-0.38775510204081631`.

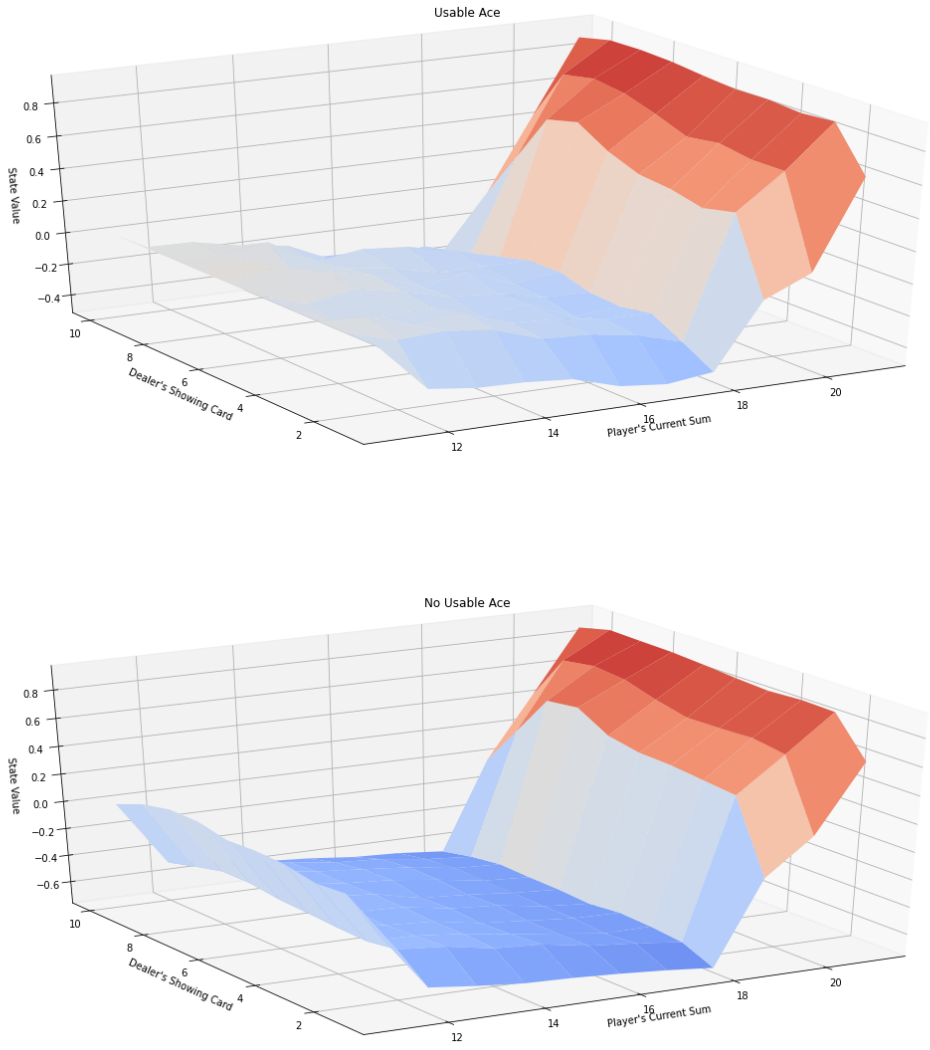
[6]:
```
from collections import defaultdict
import numpy as np
import sys

def mc_prediction_v(env, num_episodes, generate_episode, gamma=1.0):
    # initialize empty dictionary of lists
    returns = defaultdict(list)
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()
        # generate an episode
        episode = generate_episode(env)
        # obtain the states, actions, and rewards
        states, actions, rewards = zip(*episode)
        # prepare for discounting
        discounts = np.array([gamma**i for i in range(len(rewards)+1)])
```

```
        # calculate and store the return for each visit in the episode
        for i, state in enumerate(states):
            returns[state].append(sum(rewards[i:]*discounts[:-(1+i)]))
    # calculate the state-value function estimate
    V = {k: np.mean(v) for k, v in returns.items()}
    return V
```

Kalkulasi dan plotting fungsi estimasi state-value

[9]:
```
from plot_utils import plot_blackjack_values

# obtain the value function
V = mc_prediction_v(env, 500000, generate_episode_from_limit)

# plot the value function
plot_blackjack_values(V)
```

```
Episode 500000/500000.
```



1.0.3 Part 2: MC Prediction: Action Values

Di bagian ini, kita akan menulis implementasi prediksi MC (untuk memperkirakan fungsi value action).

dimulai dengan menyelidiki kebijakan di mana pemain *hampir* selalu menempel jika jumlah kartunya melebihi 18. Secara khusus, dia memilih tindakan `STICK` dengan probabilitas 80% jika jumlahnya lebih besar dari 18; dan, jika jumlahnya 18 atau kurang, dia memilih aksi `HIT` dengan probabilitas 80%. Fungsi `generate_episode_from_limit_stochastic` mengambil sampel episode menggunakan kebijakan ini

fungsi menerima sebagai **input**: - `bj_env`: ini adalah instance dari OpenAI Gym's Blackjack environment

mengembalikan sebagai **output**: - `episode`: ini adalah list dari (state, action, reward) tuples (of tuples) dan corresponds terhadap $(S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T)$,dimana T adalah final time step. secara khsusu, `episode[i]` mengembalikan (S_i, A_i, R_{i+1}), dan `episode[i][0]`, `episode[i][1]`, dan `episode[i][2]` mengembalikan S_i, A_i, dan R_{i+1}, masing - masing.

[10]:
```
def generate_episode_from_limit_stochastic(bj_env):
    episode = []
    state = bj_env.reset()
    while True:
        probs = [0.8, 0.2] if state[0] > 18 else [0.2, 0.8]
        action = np.random.choice(np.arange(2), p=probs)
        next_state, reward, done, info = bj_env.step(action)
        episode.append((state, action, reward))
        state = next_state
        if done:
            break
    return episode
```

menuliskan implementasi dari prediksi MC pada first-visit atau every-visit MC prefiksi; dalam kasus Blackjack environment, tekniknya setara.

algoritma memiliki 3 argument:

- `num_episodes`: Ini adalah jumlah episode yang dihasilkan melalui interaksi agen-lingkungan.
- `generate_episode`: Ini adalah fungsi yang mengembalikan episode interaksi.
- `gamma`: Ini adalah tingkat diskon. Berupa nilai antara 0 dan 1, inklusif (nilai default: 1).

algortima mengambalikan sebagai output: - `Q`: adalah dictionary dari (one-dimensional arrays) dimana `Q[s][a]` adalah estimasi action value korespondensi terhadap state `s` dan action `a`.

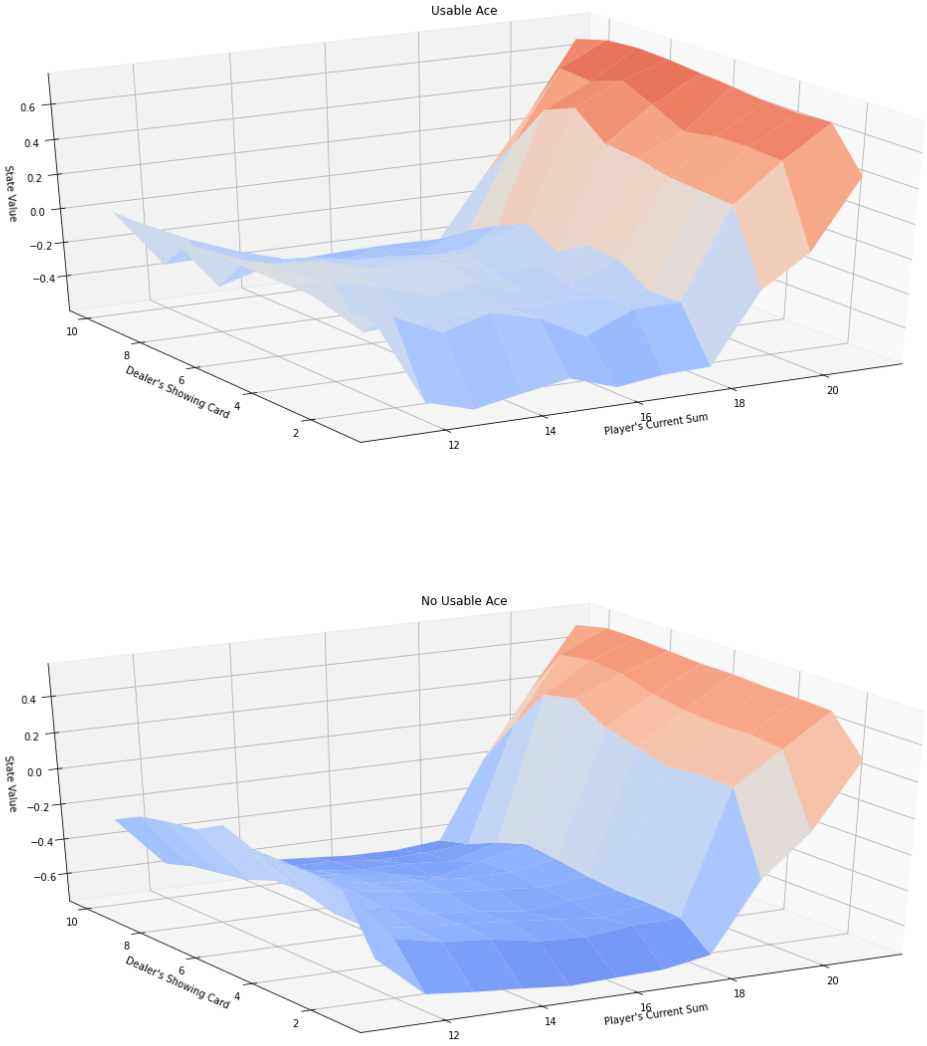
[11]:
```
def mc_prediction_q(env, num_episodes, generate_episode, gamma=1.0):
    # initialize empty dictionaries of arrays
    returns_sum = defaultdict(lambda: np.zeros(env.action_space.n))
    N = defaultdict(lambda: np.zeros(env.action_space.n))
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()
        # generate an episode
        episode = generate_episode(env)
        # obtain the states, actions, and rewards
        states, actions, rewards = zip(*episode)
        # prepare for discounting
        discounts = np.array([gamma**i for i in range(len(rewards)+1)])
        # update the sum of the returns, number of visits, and action-value
        # function estimates for each state-action pair in the episode
```

```
        for i, state in enumerate(states):
            returns_sum[state][actions[i]] += sum(rewards[i:]*discounts[:
↪-(1+i)])
            N[state][actions[i]] += 1.0
            Q[state][actions[i]] = returns_sum[state][actions[i]] /␣
↪N[state][actions[i]]
    return Q
```

Kalkulasi dan plotting fungsi estimasi action-value

[12]:
```
# obtain the action-value function
Q = mc_prediction_q(env, 500000, generate_episode_from_limit_stochastic)

# obtain the state-value function
V_to_plot = dict((k,(k[0]>18)*(np.dot([0.8, 0.2],v)) + (k[0]<=18)*(np.dot([0.2,␣
↪0.8],v))) \
         for k, v in Q.items())

# plot the state-value function
plot_blackjack_values(V_to_plot)
```

```
Episode 500000/500000.
```



1.0.4 Part 3: MC Control: GLIE

pada part ini, di implementasikan konstan dari -α MC kontrol.

algoritma memiliki 3 argument:

- `num_episodes`: Ini adalah jumlah episode yang dihasilkan melalui interaksi agen-lingkungan.
- `generate_episode`: Ini adalah fungsi yang mengembalikan episode interaksi.
- `gamma`: Ini adalah tingkat diskon. Berupa nilai antara 0 dan 1, inklusif (nilai default: 1).

algortima mengambalikan sebagai output: - `Q`: adalah dictionary dari(one-dimensional arrays) dimana `Q[s][a]` adalah estimasi action value korespondensi terhadap state `s` dan action `a`. - `policy`: merupakan dictionary dimana `policy[s]` mengembalikan action yang agen pilih setelah mengamati state `s`.

[13]:
```
def generate_episode_from_Q(env, Q, epsilon, nA):
    """ generates an episode from following the epsilon-greedy policy """
    episode = []
    state = env.reset()
    while True:
        action = np.random.choice(np.arange(nA), p=get_probs(Q[state], epsilon,␣
↪nA)) \
                                  if state in Q else env.action_space.sample()
        next_state, reward, done, info = env.step(action)
        episode.append((state, action, reward))
        state = next_state
        if done:
            break
    return episode

def get_probs(Q_s, epsilon, nA):
    """ obtains the action probabilities corresponding to epsilon-greedy policy␣
↪"""
    policy_s = np.ones(nA) * epsilon / nA
    best_a = np.argmax(Q_s)
    policy_s[best_a] = 1 - epsilon + (epsilon / nA)
    return policy_s

def update_Q_GLIE(env, episode, Q, N, gamma):
    """ updates the action-value function estimate using the most recent␣
↪episode """
    states, actions, rewards = zip(*episode)
    # prepare for discounting
    discounts = np.array([gamma**i for i in range(len(rewards)+1)])
    for i, state in enumerate(states):
        old_Q = Q[state][actions[i]]
        old_N = N[state][actions[i]]
        Q[state][actions[i]] = old_Q + (sum(rewards[i:]*discounts[:-(1+i)]) -␣
↪old_Q)/(old_N+1)
        N[state][actions[i]] += 1
    return Q, N
```

[14]:
```
def mc_control_GLIE(env, num_episodes, gamma=1.0):
    nA = env.action_space.n
    # initialize empty dictionaries of arrays
    Q = defaultdict(lambda: np.zeros(nA))
    N = defaultdict(lambda: np.zeros(nA))
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
```

```
            sys.stdout.flush()
        # set the value of epsilon
        epsilon = 1.0/((i_episode/8000)+1)
        # generate an episode by following epsilon-greedy policy
        episode = generate_episode_from_Q(env, Q, epsilon, nA)
        # update the action-value function estimate using the episode
        Q, N = update_Q_GLIE(env, episode, Q, N, gamma)
    # determine the policy corresponding to the final action-value function␣
↪estimate
    policy = dict((k,np.argmax(v)) for k, v in Q.items())
    return policy, Q
```

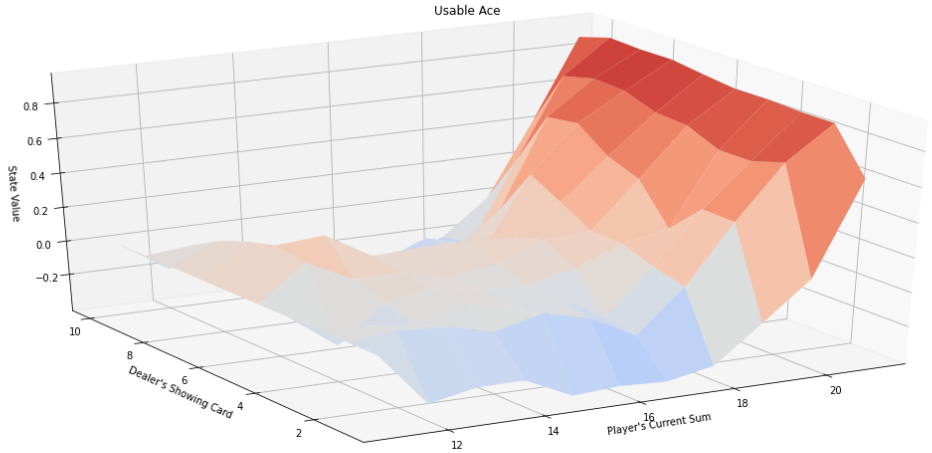
Mencari estimasi dari optimal policy dan fungsi action-value

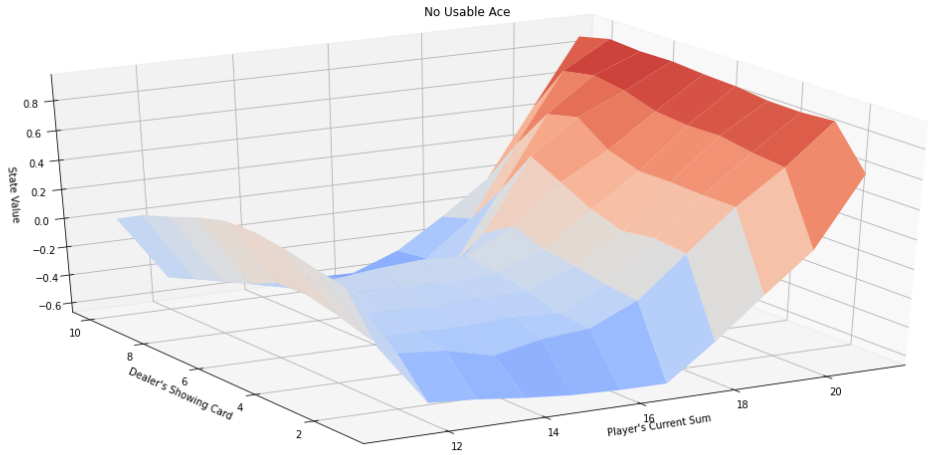
[15]:
```
# obtain the estimated optimal policy and action-value function
policy_glie, Q_glie = mc_control_GLIE(env, 500000)
```

```
Episode 500000/500000.
```

plotting korespondensi fungsi state-value.

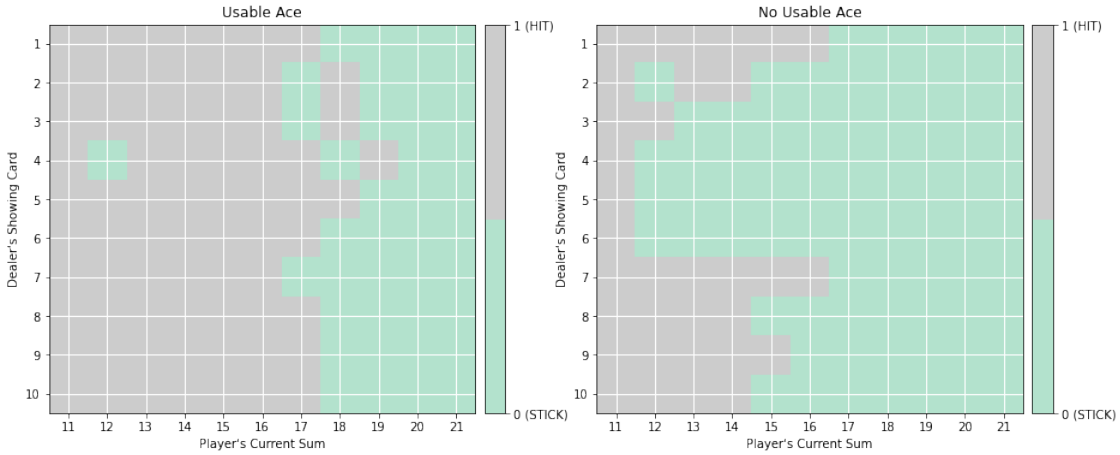
[16]:
```
# obtain the state-value function
V_glie = dict((k,np.max(v)) for k, v in Q_glie.items())

# plot the state-value function
plot_blackjack_values(V_glie)
```





visualisasi policy yang diestimasikan secara opetimal.

[17]:
```
from plot_utils import plot_policy

# plot the policy
plot_policy(policy_glie)
```



1.0.5 Part 4: MC Control: Constant-α

implementasi dari constant-α MC kontrol.

algoritma memiliki 4 argument: - `env`: merupakan instance dari OpenAI Gym environment. - `num_episodes`: Ini adalah jumlah episode yang dihasilkan melalui interaksi agen-lingkungan. - `alpha`: merupakan parameter ukuran step untuk update step/learning rate. - `gamma`: Ini adalah tingkat diskon. Berupa nilai antara 0 dan 1, inklusif (nilai default: 1).

algoritma mengembalikan sebagai output: - `Q`: ini adalah dictionary dari(one-dimensional arrays) dimana `Q[s][a]` adalah estimasi action value korespondensi terhadap state `s` dan action `a`. - `policy`: ini adalah dictionary dimana `policy[s]` mengembalikan action yang agen pilih setelah mengamati state `s`.

[18]:
```
def update_Q_alpha(env, episode, Q, alpha, gamma):
    """ updates the action-value function estimate using the most recent␣
↪episode """
    states, actions, rewards = zip(*episode)
    # prepare for discounting
    discounts = np.array([gamma**i for i in range(len(rewards)+1)])
    for i, state in enumerate(states):
        old_Q = Q[state][actions[i]]
        Q[state][actions[i]] = old_Q + alpha*(sum(rewards[i:]*discounts[:
↪-(1+i)]) - old_Q)
    return Q
```

[19]:
```
def mc_control_alpha(env, num_episodes, alpha, gamma=1.0):
    nA = env.action_space.n
    # initialize empty dictionary of arrays
    Q = defaultdict(lambda: np.zeros(nA))
    # loop over episodes
```

```
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()
        # set the value of epsilon
        epsilon = 1.0/((i_episode/8000)+1)
        # generate an episode by following epsilon-greedy policy
        episode = generate_episode_from_Q(env, Q, epsilon, nA)
        # update the action-value function estimate using the episode
        Q = update_Q_alpha(env, episode, Q, alpha, gamma)
    # determine the policy corresponding to the final action-value function␣
↪estimate
    policy = dict((k,np.argmax(v)) for k, v in Q.items())
    return policy, Q
```

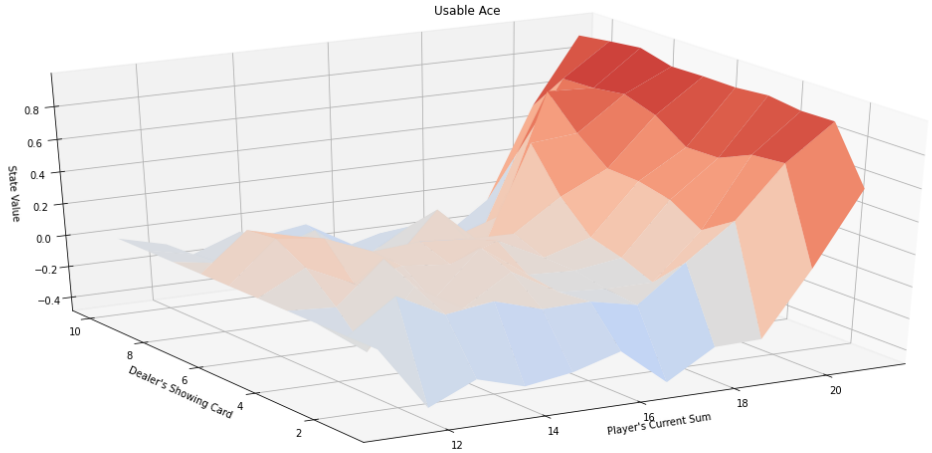
Estimasi optimal policy dan fungsi action-value

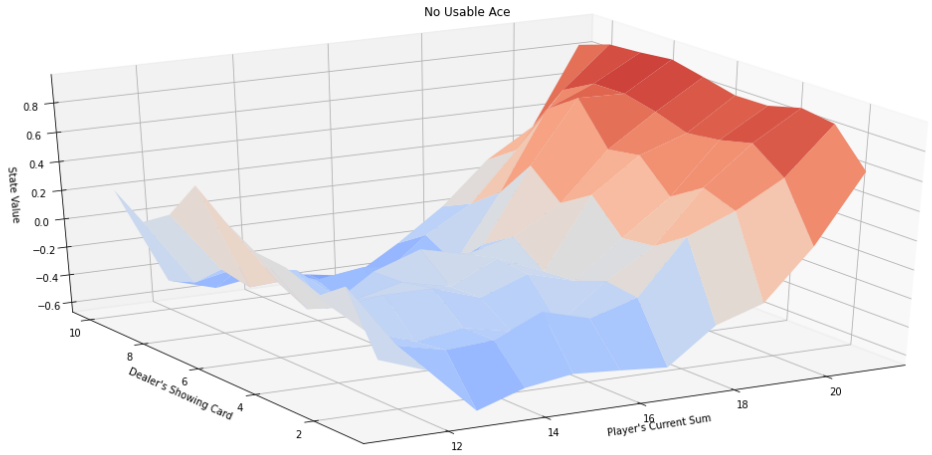
[20]:
```
# obtain the estimated optimal policy and action-value function
policy_alpha, Q_alpha = mc_control_alpha(env, 500000, 0.02)
```

```
Episode 500000/500000.
```

plot korespondensi fungsi state-value.

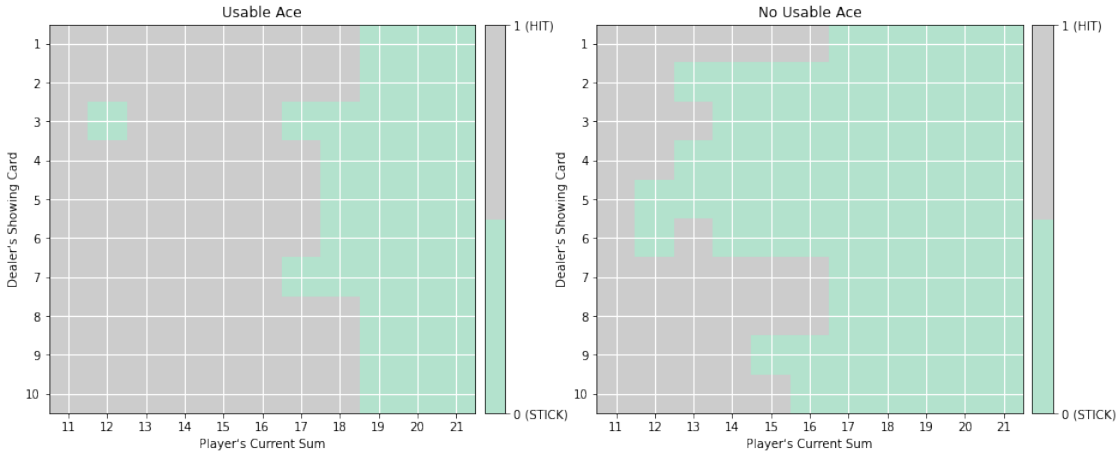
[21]:
```
# obtain the state-value function
V_alpha = dict((k,np.max(v)) for k, v in Q_alpha.items())

# plot the state-value function
plot_blackjack_values(V_alpha)
```





visualisasi policy yang diestimasi secara optimal.

[22]:
```
# plot the policy
plot_policy(policy_alpha)
```



2 Kesimpulan

Untuk membangun policy yang lebih baik, pertama-tama perancang harus bisa mengevaluasi policy apa pun. Jika agen mengikuti kebijakan untuk banyak episode, menggunakan Prediksi Monte-Carlo, kita dapat membuat tabel-Q (yaitu, "memperkirakan" fungsi action-value) dari hasil episode.

Jadi kita bisa mulai dengan kebijakan stokastik seperti `stick` dengan probabilitas 80% jika jumlahnya lebih besar dari 18 karena kita tidak ingin melebihi 21. Lain jika jumlahnya kurang dari 18, kita akan `HIT` dengan probabilitas 80%.