Training a Blackjack Agent Using Proximal Policy Optimization

By: Bradley Friedrich

In this project, I use the Reinforcement Learning Algorithm of Proximal Policy Optimization(PPO) to determine the optimal policy in the game of blackjack. This project was inspired by the book "Reinforcement Learning: An Introduction" (Sutton and Barto). In the book, they use an alternative algorithm to create their agent, but I was inspired to use PPO for 2 reasons:

1. I wanted to implement an algorithm by hand to get a thorough understanding of the world of reinforcement learning, and I hadn't seen anyone use PPO on blackjack, so it it seemed like a fun way to solve this problem.

2. PPO seems to be the "state of the art" RL algorithm, and it seemed useful and fun to learn.

Blackjack works by being dealt 2 cards and playing against the dealer. You can either choose to add cards or stay with your hand, and get as close to 21 as possible. If you exceed 21, however, you lose. You also lose if the dealer has a bigger number than you. Yopu win by beating the deaeler, or if the dealer goes over the value fo 21. It is also possible to tie the dealer if you end up with the same value. In this blackjack environment, we consider three inputs:

1. The sum of our hand, which takes on the integer values between 12 and 21 (we will always hit when our hand is an 11 or lower, as we cannot lose on that hit so we might as well increase our hand)

2. The card we can see in the dealers hand, which can be any integer from 2 to 11.

3. Whether or not we have an ace that can be used as either an 11 or a 1, giving us more flexibility.

The agent will consider these 3 factors and determine whether it is optimal for us to "hit" and draw another card, or "stick" with our current hand. The agent uses gymnasium, an open source library created by OpenAI to instantiate the environment and synthetically generate training data and rewards. The agent will receive a 1 if they win the hand and a -1 if they do not, with a 0 if they tie the dealer. In practice, payout for a natural blackjack (being dealt a card valued 10 or an ace) is 1.5X, but this will be ignored, as we are focused on the decisions you make to optimally play blackjack, and a natural blackjack represents no decisions to be made.

The PPO algorithm works by ensuring that the policy does not change too much between training iterations by a clipping parameter ϵ. If the policy changes outside this threshold, its change will not be implemented and the old policy will still be in effect. The algorithm is defined by:

$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A},\ clip(r_t, 1-\epsilon, 1+\epsilon))]$,

where $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t}$ (the part that accounts for policy change from old to new)

and the advantage term $\hat{A}_t$ attempts to quantify the "advantage" of taking one action over another.

In [ ]:
```
## Imports
import gymnasium as gym
from __future__ import annotations
from collections import defaultdict
import matplotlib.pyplot as plt
import numpy as np
from torch import nn, optim
import torch
from torch.distributions import Categorical
import seaborn as sns
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
```

In [ ]:
```
## Use OpenAI gymnasium API
env = gym.make('Blackjack-v1', natural = False, sab = False)
```

In [ ]:
```
## Define an actor critic network to train the policy gradient and advantage function

class ActorCritic(nn.Module):
    def __init__(self, observation_dim, action_dim):
```

```
        super().__init__()
        self.shared_layers = nn.Sequential(
            nn.Linear(observation_dim, 200),  # Correctly initialize with observation_dim
            nn.ReLU(),
        )
        self.policy_layers = nn.Sequential(
            nn.Linear(200, action_dim)
        )
        self.value_layers = nn.Sequential(
            nn.Linear(200, 1)
        )

    def forward(self, observation):
        z = self.shared_layers(observation)
        policy_logits = self.policy_layers(z)
        value = self.value_layers(z)
        return policy_logits, value
```

In [ ]:

```
## Create PPO trainer

class PPO_Agent_Trainer:
    def __init__(self,
             actor_critic,
             epsilon: float  = 0.2,
             kl_delta: float = 0.01,
             policy_iter: int = 50,
             value_train_iters: int = 50,
             policy_learning_rate: float = 2e-4,
             value_learning_rate: float = 2e-4):


        '''Inputs:
        actor_critic: Neural Network defined above
        epsilon: Clippping parameter. The higher the value the more movement allowed for the policy
        kl_delta: If KL divergence exceeds this threshold stop training
        policy_iter: Iterations for policy in an episode
```

```
        value_train_iter: above for values
        policy_learning_rate: Learning rate for policy network
        value_learning_rate: learning rate for value network'''

        self.actor_critic = actor_critic
        self.epsilon = epsilon
        self.kl_delta = kl_delta
        self.policy_iter= policy_iter
        self.value_train_iters = value_train_iters
        self.learning_rate = policy_learning_rate
        self.value_learning_rate = value_learning_rate

        policy_params = list(self.actor_critic.shared_layers.parameters())
        + list(self.actor_critic.policy_layers.parameters())
        self.policy_optimizer = optim.Adam(policy_params, lr = policy_learning_rate)

        value_params = list(self.actor_critic.shared_layers.parameters())
        + list(self.actor_critic.value_layers.parameters())
        self.value_optimizer = optim.Adam(value_params, lr = value_learning_rate)

        for param in policy_params:
            param.requires_grad = True

        for param in value_params:
            param.requires_grad = True




    ## Train policy network
    def train_policy(self, observations, actions, prev_log_probs, gaes):
        for _ in range(self.policy_iter):
            self.policy_optimizer.zero_grad()

            new_logits = self.actor_critic.forward(observations)[0]
            new_logit_dist = Categorical(logits=new_logits)
            new_log_probs = new_logit_dist.log_prob(actions)
```

```
            policy_ratio = torch.exp(new_log_probs - prev_log_probs)
            total_loss = policy_ratio*gaes

            #Clip the policy and compute ratio
            clipped_policy_ratio = policy_ratio.clamp(1- self.epsilon, 1 + self.epsilon)
            clipped_loss = clipped_policy_ratio*gaes

            #Compute loss
            overall_policy_loss = -1*torch.min(total_loss, clipped_loss).mean()

            #Add entropy term to ensure sufficient exploration of possible states and outcomes
            entropy = Categorical(logits=new_logits).entropy().mean()

            overall_policy_loss = overall_policy_loss - 0.1 * entropy

            overall_policy_loss.backward()
            self.policy_optimizer.step()

            # If KL divergence is above a threshold, end training
            kl_divergence = (prev_log_probs - new_log_probs).mean()
            if kl_divergence > self.kl_delta:
                break



    #Train value network
    def train_value(self, observations, returns):
        for _ in range(self.value_train_iters):
            self.value_optimizer.zero_grad()
            values = self.actor_critic(observations)[1]
            value_loss = ((returns - values)**2).mean()
            value_loss.backward()
            self.value_optimizer.step()
```

In [ ]:

```
# Compute and discount rewards generated by model
```

```
def discount_rewards(rewards, gamma: float = 0.95) -> np.array:

    '''rewards: list of model rewards
    gamma: discoutning parameter between 0 and 1'''

    new_rewards = [float(rewards[-1])]
    for i in reversed(range(len(rewards) -1)):
        new_rewards.append(float(rewards[i]) + gamma*new_rewards[-1])
    return np.array(new_rewards[::-1])

#Calculate advantage of taking action
def calculate_advantage(rewards, values, gamma: float = 0.95, decay: float = 0.99) -> np.array:

    '''rewards: list of model rewards
    values: state of enviroment
    gamma: discoutning parameter between 0 and 1
    decay: parameter between 0 and 1'''

    rewards = np.array(rewards).flatten()
    values = np.array(values).flatten()
    next_vals = np.concatenate([values[1:], [0]])
    deltas = rewards + gamma * next_vals - values
    gaes = []
    cumulative_advantage = 0.0
    for delta in reversed(deltas):
        cumulative_advantage = delta + decay * gamma * cumulative_advantage
        gaes.insert(0, cumulative_advantage)
    return np.array(gaes)
```

In [ ]:

```
## Generate data/reward, evaluate state

def sample_and_reward(model, max_steps=100):
    train_data = [[] for _ in range(5)]

    #Reset environment, generate observation
    observation, _ = env.reset()
```

```
    episode_reward = 0

    for num in range(max_steps):

        #Run prediction on state
        observation_tensor = torch.tensor(observation, dtype=torch.float32).unsqueeze(0)
        logits, value = model.forward(observation_tensor)
        action_dist = Categorical(logits=logits)
        action = action_dist.sample()
        action_log_prob = action_dist.log_prob(action)
        next_observation, reward, done, _, a = env.step(action.item())
        observation = next_observation

        #compute episode reward
        episode_reward += reward

        #Keep track of training data
        for ind, cat in enumerate([observation, action, reward, value, action_log_prob]):
            train_data[ind].append(cat)

        #Break loop if terminal state is reached
        if done:
            break

    #Compute advantage
    train_data[3] = calculate_advantage(train_data[2], train_data[1])

    return train_data, episode_reward
```

In [ ]:

```
obs_space = env.observation_space
obs_dim = 3

#Ensure all weights of neural networks are initialized and nonzero
def init_weights(m):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
```

```
        if m.bias is not None:
            m.bias.data.fill_(0.01)

model = ActorCritic(observation_dim=3, action_dim=env.action_space.n)
model.apply(init_weights)
train_data, reward = sample_and_reward(model)

#Number of training epsiodes
episodes = 750000

#Instantiate trainer
ppo_blackjack = PPO_Agent_Trainer(model,
                                  policy_learning_rate=2e-2,
                                  value_learning_rate=2e-2,
                                  kl_delta=0.05,
                                  policy_iter=50,
                                  value_train_iters=50)

episode_rewards = []


#Initialize to keep track of training
usable_ace = {(player_sum, dealer_sum): {'hit': [0, 0], 'stick': [0, 0]}
              for dealer_sum in range(2,11) for player_sum in range(12,22)}
no_usable_ace = {(player_sum, dealer_sum): {'hit': [0, 0], 'stick': [0, 0]}
                 for dealer_sum in range(2,11) for player_sum in range(12,22)}


for i in range(episodes):
    #Generate sample and reward
    train_data, reward = sample_and_reward(model)


    #If we have more than 21, we have less than 12, dealer has less than 2
    #skip as these options do not represent choices we have to make
    # and dealer must treat ace as 11 not 1
    if train_data[0][0][0] > 21 or train_data[0][0][0] < 12 or train_data[0][0][1] < 2:
```

```
            continue


        #Track data for usable ace
        if train_data[0][0][2] == 1:


            last_hit = len(train_data[0]) - 1
            #hits
            if last_hit > 0:
                for num in range(last_hit):
                    if train_data[0][num +1][0] > 21:
                        continue
                    usable_ace[train_data[0][num][:2]]['hit'][0] += reward
                    usable_ace[train_data[0][num][:2]]['hit'][1] += 1
            #sticks
            usable_ace[train_data[0][0][:2]]['stick'][0] += reward
            usable_ace[train_data[0][0][:2]]['stick'][1] += 1

        #Data for no usable ace
        elif train_data[0][0][2] == 0:
            last_hit = len(train_data[0]) - 1
            if last_hit > 0:
                #hits
                for num in range(last_hit):
                    if train_data[0][num +1][0] > 21:
                        continue
                    no_usable_ace[train_data[0][num][:2]]['hit'][0] += reward
                    no_usable_ace[train_data[0][num][:2]]['hit'][1] += 1
                if train_data[0][-1][0] > 21:
                    continue
            #sticks
            no_usable_ace[train_data[0][0][:2]]['stick'][0] += reward
            no_usable_ace[train_data[0][0][:2]]['stick'][1] += 1


        episode_rewards.append(reward)
```

```
    selected_indices= np.random.permutation(len(train_data[0]))
    train_data[0] = np.array(train_data[0])  # Observations
    train_data[1] = np.array(train_data[1])  # Actions
    train_data[3] = np.array(train_data[3])
    train_data[4] = np.array([tensor.detach().numpy() for tensor in train_data[4]])
    observation = torch.tensor(train_data[0][selected_indices], dtype=torch.float32, requires_grad=True)
    actions = torch.tensor(train_data[1][selected_indices], dtype=torch.float32)
    old_log_probs = torch.tensor(train_data[4][selected_indices], dtype=torch.float32)
    advantages = torch.tensor(train_data[3][selected_indices], dtype=torch.float32)
    returns = torch.tensor(discount_rewards(train_data[2]), dtype=torch.float32)


    # Train the policy network using PPO
    ppo_blackjack.train_policy(observation, actions, old_log_probs, advantages)

    # Train the value network
    ppo_blackjack.train_value(observation, returns)
```

In [ ]:
```
print(f'Final Episode Reward is {round(np.mean(episode_rewards), 2)}')
```

```
Final Episode Reward is 0.08
```

Here, we get a final reward of 0.08. Meaning, implementing this strategy, you are expected to win 0.08 times your bet every time (in addition to your bet). Considering blackjack is not a zero-sum game and that the house always has an advantage, this strategy is a vast improvement on playing with no strategy, and I am very satisfied with this result. Typically, you expect to lose a game of blackjack, so the fact that this strategy generates a winning value on average shows the powers of thw PPO algorithm and RL in general.

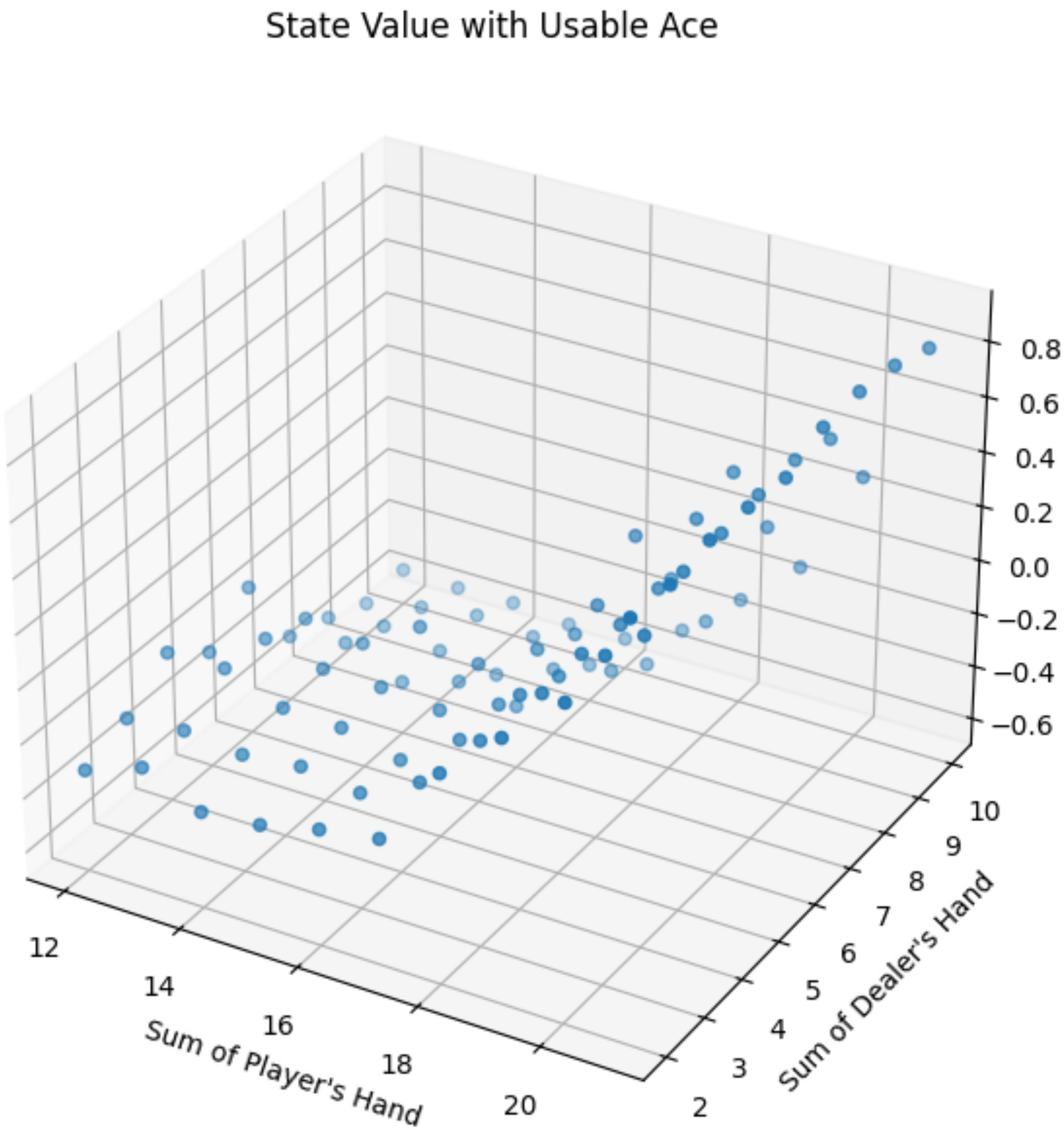
Below I will show some visualizations for the scenarios where we have and do not have a usable ace. One graph shows the state value for a combination of our hand and the dealer's hand. This represents how good each state is, with a value above 0 being favorable to the plyaer. The other chart shows whether or not the agent recommends hitting or sticking in a situation, highlighted blue if you should hit and yellow if not.

In [ ]:
```
#Prepare data for usable ace
```

```
usable_ace_comb = {key: [usable_ace[key]['hit'][0],
                         usable_ace[key]['hit'][1],
                         usable_ace[key]['stick'][0],
                         usable_ace[key]['stick'][1]] for key in usable_ace}

usable_ace_df = pd.DataFrame.from_dict(usable_ace_comb, orient = 'index',
                                       columns=['hit value', 'num hits', 'stick value', 'num sticks'])

usable_ace_df['player sum'] = [a[0] for a in list(usable_ace_df.index)]
usable_ace_df['dealer showing'] = [a[1] for a in list(usable_ace_df.index)]


#Calculate value of hit/stick and overall value for each state combination for usable ace
usable_ace_df['overall hit'] = usable_ace_df['hit value']/usable_ace_df['num hits']
usable_ace_df['overall stick'] = usable_ace_df['stick value']/usable_ace_df['num sticks']

usable_ace_df['overall value'] = (usable_ace_df['num hits']*usable_ace_df['overall hit'] +
                                    usable_ace_df['num sticks']*usable_ace_df['overall stick'])

usable_ace_df['overall value'] = usable_ace_df['overall value']/(usable_ace_df['num hits']
                                                                 + usable_ace_df['num sticks'])

#Calculate what action should be taken for each state combination for usable ace
#This is just which overall value is higher, according to the training
best_option_usable_ace = np.argmax(usable_ace_df[['overall stick', 'overall hit']].values, axis=1)
usable_ace_df['optimal'] = best_option_usable_ace


#Prepare data for no usable ace
no_usable_ace_comb = {key: [no_usable_ace[key]['hit'][0],
                            no_usable_ace[key]['hit'][1],
                            no_usable_ace[key]['stick'][0],
                            no_usable_ace[key]['stick'][1]] for key in no_usable_ace}

no_usable_ace_df = pd.DataFrame.from_dict(no_usable_ace_comb, orient = 'index',
                                    columns=['hit value', 'num hits', 'stick value', 'num sticks'])
```

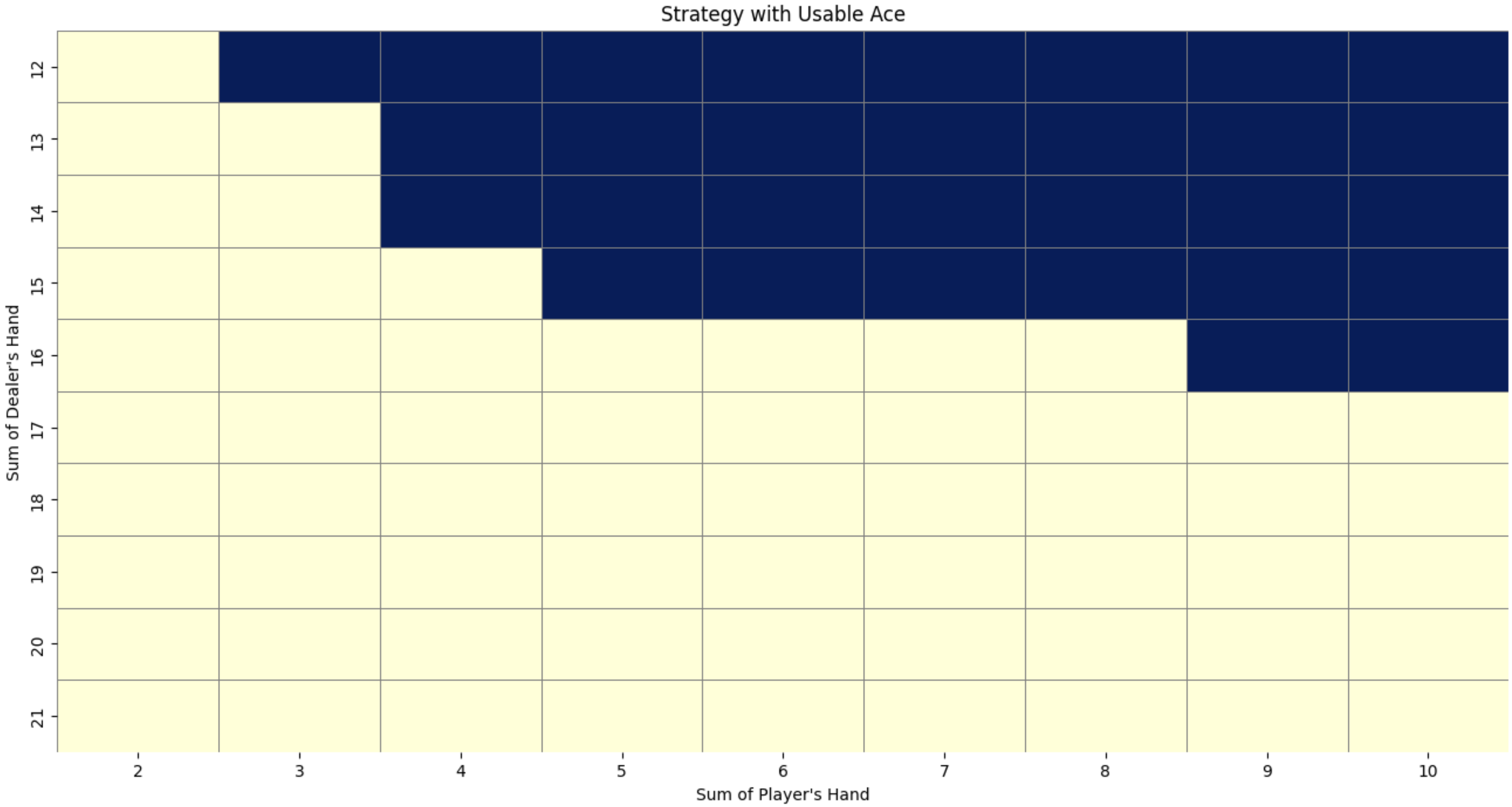
```
no_usable_ace_df['player sum'] = [a[0] for a in list(no_usable_ace_df.index)]
no_usable_ace_df['dealer showing'] = [a[1] for a in list(no_usable_ace_df.index)]

#Calculate value of hit/stick and overall value for each state combination for no usable ace
no_usable_ace_df['overall hit'] = no_usable_ace_df['hit value']/no_usable_ace_df['num hits']
no_usable_ace_df['overall stick'] = no_usable_ace_df['stick value']/no_usable_ace_df['num sticks']
no_usable_ace_df['overall value'] = (no_usable_ace_df['num hits']*no_usable_ace_df['overall hit'] +
                                     no_usable_ace_df['num sticks']*no_usable_ace_df['overall stick'])

no_usable_ace_df['overall value']=no_usable_ace_df['overall value']/(no_usable_ace_df['num hits']
                                                                     +no_usable_ace_df['num sticks'])

#Calculate what action should be taken for each state combination for no usable ace
best_option_no_usable_ace = np.argmax(no_usable_ace_df[['overall hit', 'overall stick']].values, axis=1)
no_usable_ace_df['optimal'] = best_option_no_usable_ace
```

In [ ]:
```
fig = plt.figure(figsize=(16,8))
ax = fig.add_subplot(122, projection='3d')

# Plot the data
ax.scatter(usable_ace_df['player sum'],
           usable_ace_df['dealer showing'],
           usable_ace_df['overall value'])

# Add labels
ax.set_xlabel("Sum of Player's Hand")
ax.set_ylabel("Sum of Dealer's Hand")
ax.set_title('State Value with Usable Ace')
plt.show()
```


State Value with Usable Ace
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
12
14
16
18
20
Sum of Player's Hand
2
3
4
5
6
7
8
9
10
Sum of Dealer's Hand

As we see here, the value of the state increases as our hand increases, and increases as the sum of the dealer's hand decreases. This is expected.

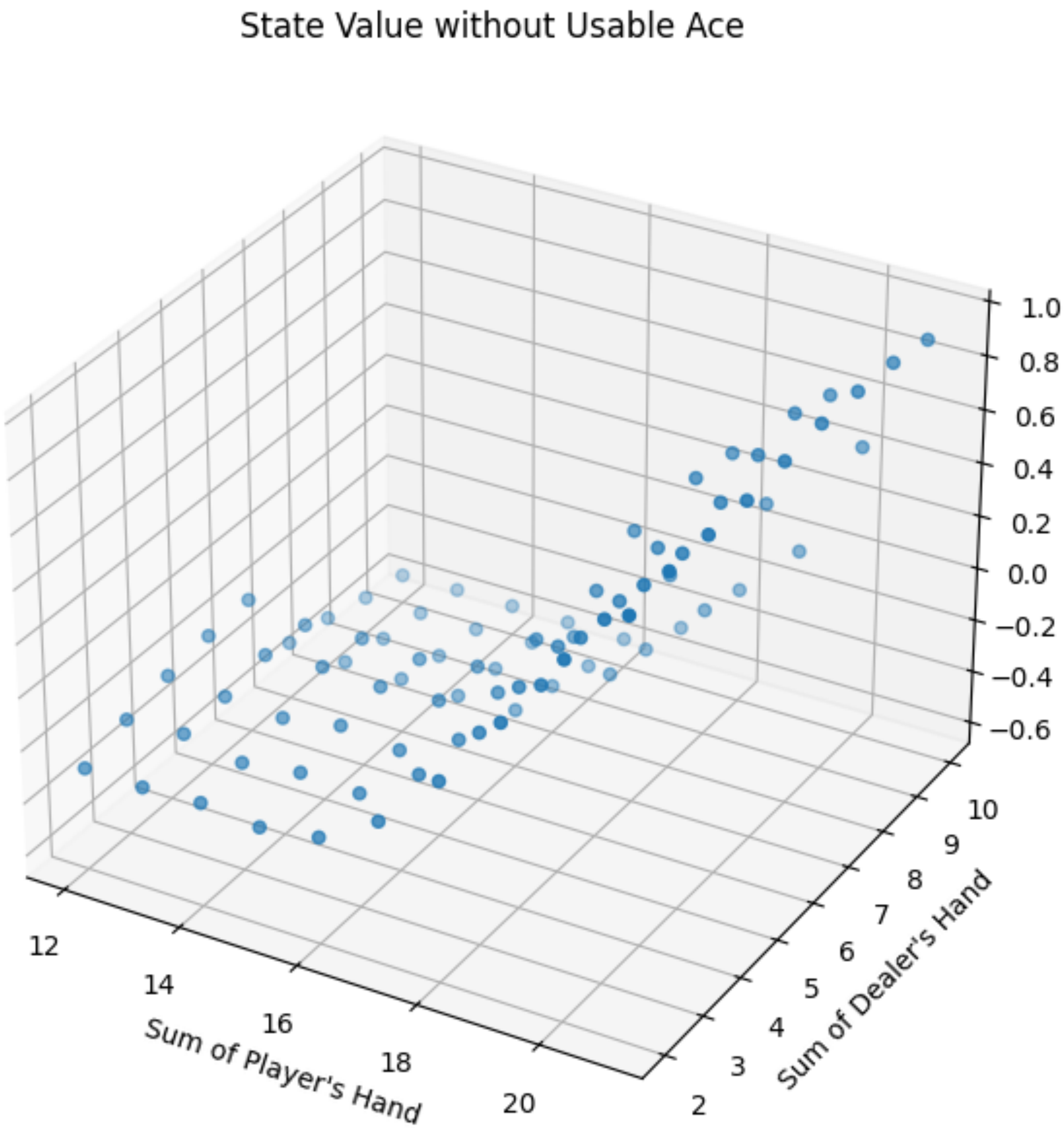
In [ ]:
```
heatmap_data = usable_ace_df.pivot(index = 'player sum', columns = 'dealer showing',
                                   values = 'optimal').fillna(0)

# Plotting the heatmap
plt.figure(figsize=(16, 8))
sns.heatmap(heatmap_data, cmap='YlGnBu', linewidths=0.5, linecolor='gray', cbar=False)


# Label the axes
plt.xlabel("Sum of Player's Hand")
plt.ylabel("Sum of Dealer's Hand")
plt.title('Strategy with Usable Ace')

# Show the plot
plt.show()
```



Here, we see that the agent is aggressive until it reaches 16, where it starts to play more conservatively ands stick more. Additionally, it is aggressive until the dea;er starts to show between 4-8, depending on its own hand, as there the dealer is more likely to bust himself, so the agent must only need to not bust to win.

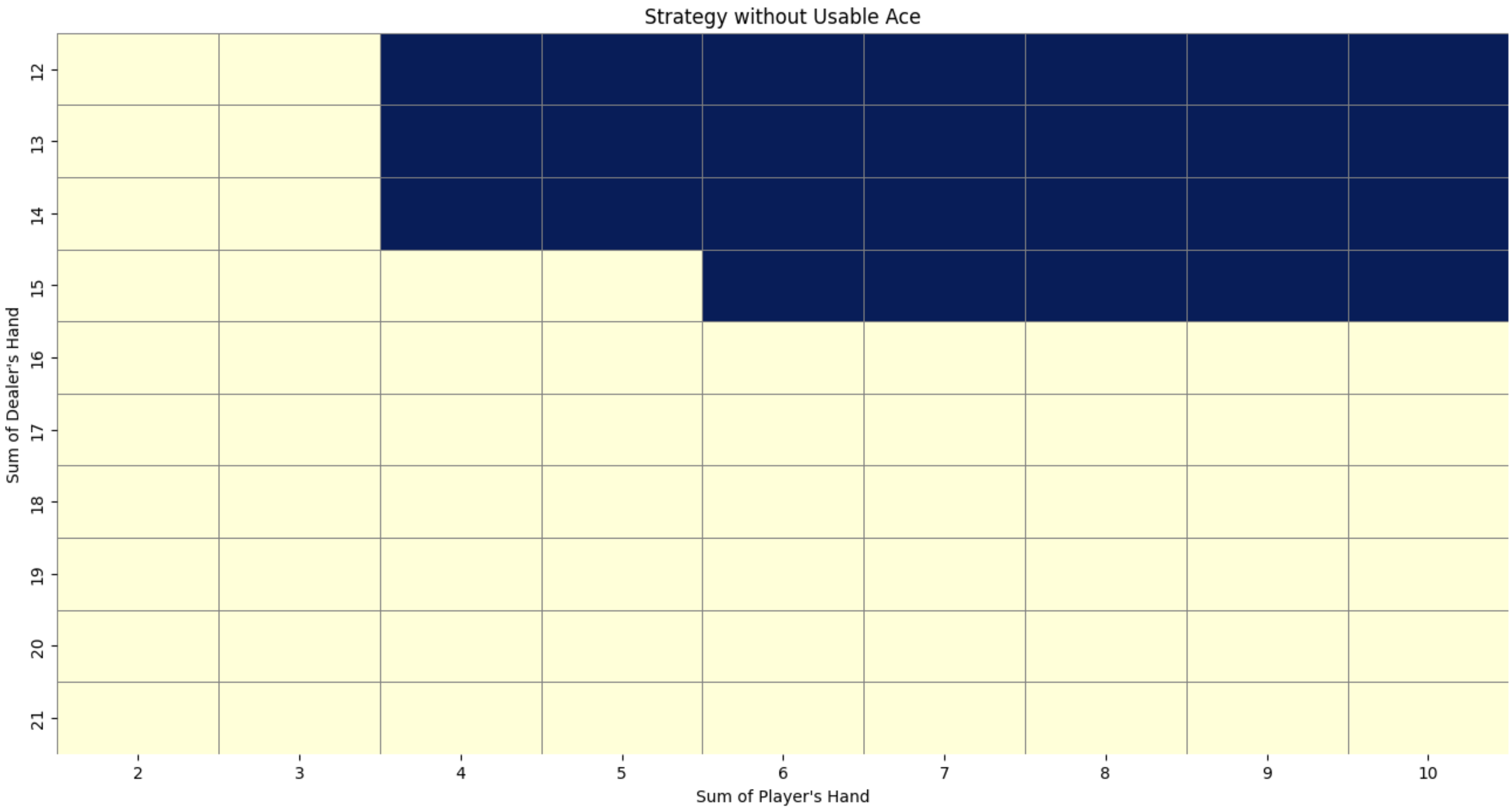
In [ ]:
```
fig = plt.figure(figsize=(16,8))
ax = fig.add_subplot(122, projection='3d')

# Plot the data
```

```
ax.scatter(no_usable_ace_df['player sum'],
           no_usable_ace_df['dealer showing'],
           no_usable_ace_df['overall value'])

# Add labels
ax.set_xlabel("Sum of Player's Hand")
ax.set_ylabel("Sum of Dealer's Hand")
ax.set_title('State Value without Usable Ace')
plt.show()
```


State Value without Usable Ace
Sum of Player's Hand
12
14
16
18
20
Sum of Dealer's Hand
2
3
4
5
6
7
8
9
10
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

The same general trend appears here as it does with a usable ace, albeit marginally worse, as having the ace provides additional flexibility.

In [ ]:
```
heatmap_data = no_usable_ace_df.pivot(index = 'player sum', columns = 'dealer showing',
                                      values = 'optimal').fillna(0)

# Plotting the heatmap
plt.figure(figsize=(16, 8))
sns.heatmap(heatmap_data, cmap='YlGnBu', linewidths=0.5, linecolor='gray', cbar=False)

# Label the axes
plt.xlabel("Sum of Player's Hand")
plt.ylabel("Sum of Dealer's Hand")
plt.title('Strategy without Usable Ace')

# Show the plot
plt.show()
```



Without a usable ace, the agent is less aggressive with its hits, albeit only marginally. This makes sense, as having an ace helps your odds but drawing a high card right after that happens nullifies that help, so it makes sense that the agent is not that much more conservative with the cushion of an ace.

Overall, this was a fun project. I learned a lot about Reinforcment Learning, coding in PyTorch, and even data visualization. It is fascinating to apply applying mathematical techniques to non-academic subjects like blackjack, especially as it is such a common, fun, and simple game. I hope to implement the strategy of my PPO agent next time I play!