Master’s degree in Control System Engineering

Reinforcement Learning
LAB 3

Monte Carlo Methods
&
blackjack problem

DIPARTIMENTO
DI INGEGNERIA
DELL’INFORMAZIONE

Niccolò Turcato
niccolo.turcato@phd.unipd.it

Blackjack

- **Objective**: obtain cards whose sum is as close as possible to 21, without exceeding it.
- **Values**: It is up to each individual player if an ace is worth 1 or 11. Face cards are 10 and any other card is its pip value.

Blackjack

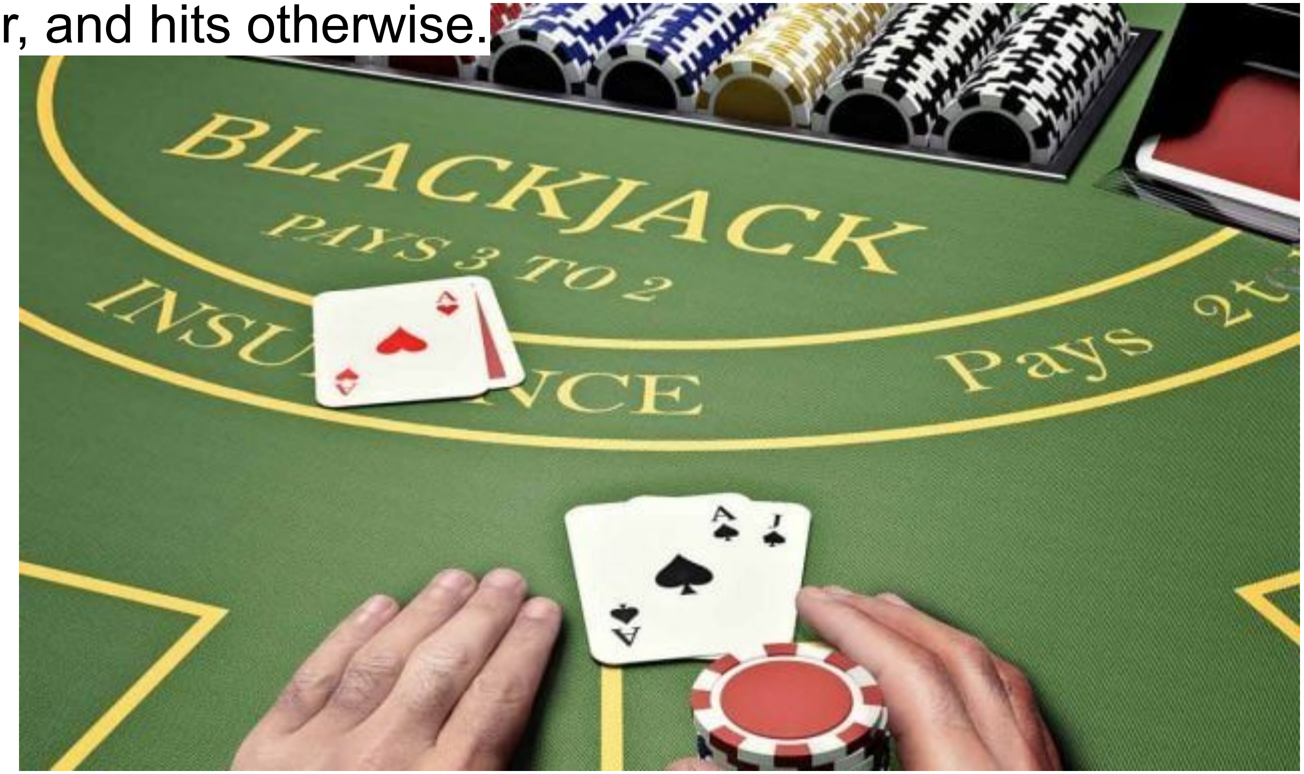
- **Start**: At the beginning each player draws 2 cards. The actual player holds them face up, while the dealer reveals only 1 card to the other player and leave the other face down.

  If the player drew an ace and a 10-card its turn never begins, as it holds already 21 and cannot improve its score: it is called a *natural*.

Blackjack

- **Play**: The player can request additional cards, one by one (hits), until he either stops (sticks) or exceeds 21 (goes bust). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn.

  In standard Blackjack the dealer can perform the same actions as the player, however he hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise.



Blackjack

- **Scoring**: When one of the players go bust the game ends and that player loses; otherwise, the outcome [win, lose, or draw] is determined by whose final sum is closer to 21.

We don't take into account the possibility of splitting

Blackjack MDP

Playing blackjack is naturally formulated as an episodic finite MDP: each game of blackjack is an episode.

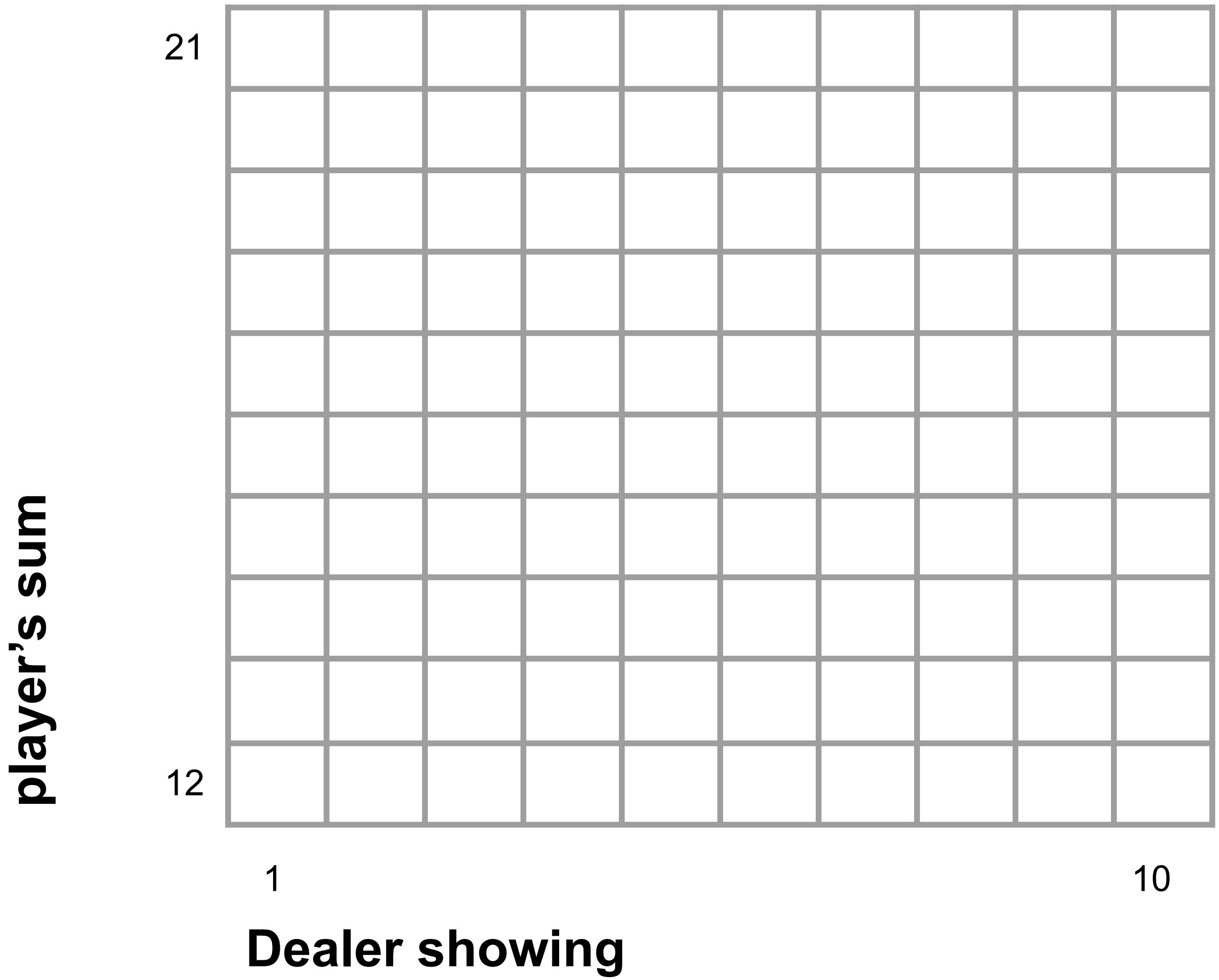
- **Reward: +1, -1, and 0 for winning, losing, and drawing, respectively.**

  All rewards within a game are zero, and we do not discount (γ = 1); therefore these terminal rewards are also the returns.

- **Action:** the player's actions are to hit or to stick.
- **State:** The states depend on the player's cards and the dealer's showing card.
- We assume that cards are dealt from an infinite deck (i.e., with replacement)

Blackjack MDP

Possible states:



$$\in \mathbb{R}^{10\times 10}$$

Blackjack MDP implementation

Remark: if the player's initial sum is <12, then a card is always drawn, so that the cards sum is always in [12, 21] during game.

```
# actions: hit or stand
ACTION_HIT = 0
ACTION_STAND = 1  # "strike" in the book
# Actually it is neither "stand" nor "strike", it is "stick"
ACTIONS = [ACTION_HIT, ACTION_STAND]
```

Blackjack MDP implementation

```python
# policy for player
# Hard-coded policy! Sticks only if sum >= 20!
POLICY_PLAYER = np.zeros(22, dtype=np.int)
POLICY_PLAYER[20] = ACTION_STAND
POLICY_PLAYER[21] = ACTION_STAND
```

```python
# policy for dealer
POLICY_DEALER = np.zeros(22)
for i in range(17, 22):
    POLICY_DEALER[i] = ACTION_STAND
```

Blackjack MDP implementation

```
def play(policy_player, initial_state=None, initial_action=None):
    if initial_state is None:
        # generate a random initial state
        player_sum, usable_ace_player = random_init_player()
        # initialize cards of dealer, suppose dealer will show the first card he gets
        dealer_card1 = get_card()
        dealer_card2 = get_card()
    # If we want instead to initialize the state
    # We cannot impose anything on the second card of the dealer as it is out of our control!
    else:
        # use specified initial state
        usable_ace_player, player_sum, dealer_card1 = initial_state
        dealer_card2 = get_card()

    # initial state of the game
    state = [usable_ace_player, player_sum, dealer_card1]

    # Initial dealer sum may be 22, that's why we keep a variable for its second card

    # initialize dealer's sum
    dealer_sum = card_value(dealer_card1) + card_value(dealer_card2)
    assert dealer_sum <= 21
    assert player_sum <= 21
```

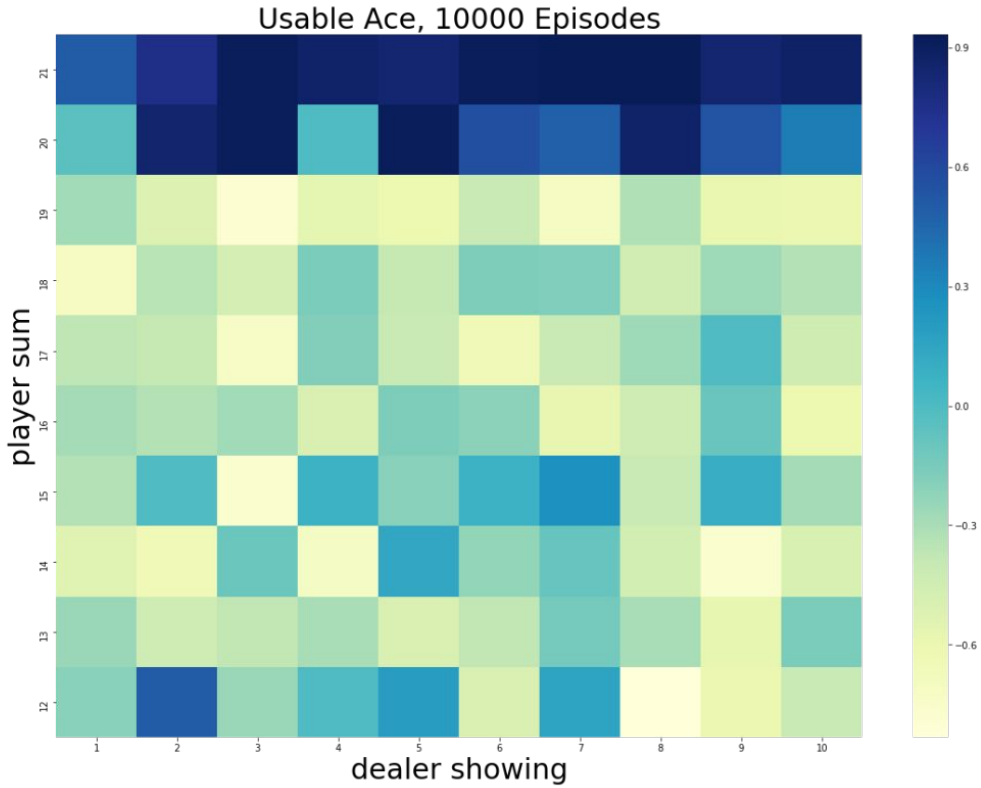
Blackjack MDP implementation

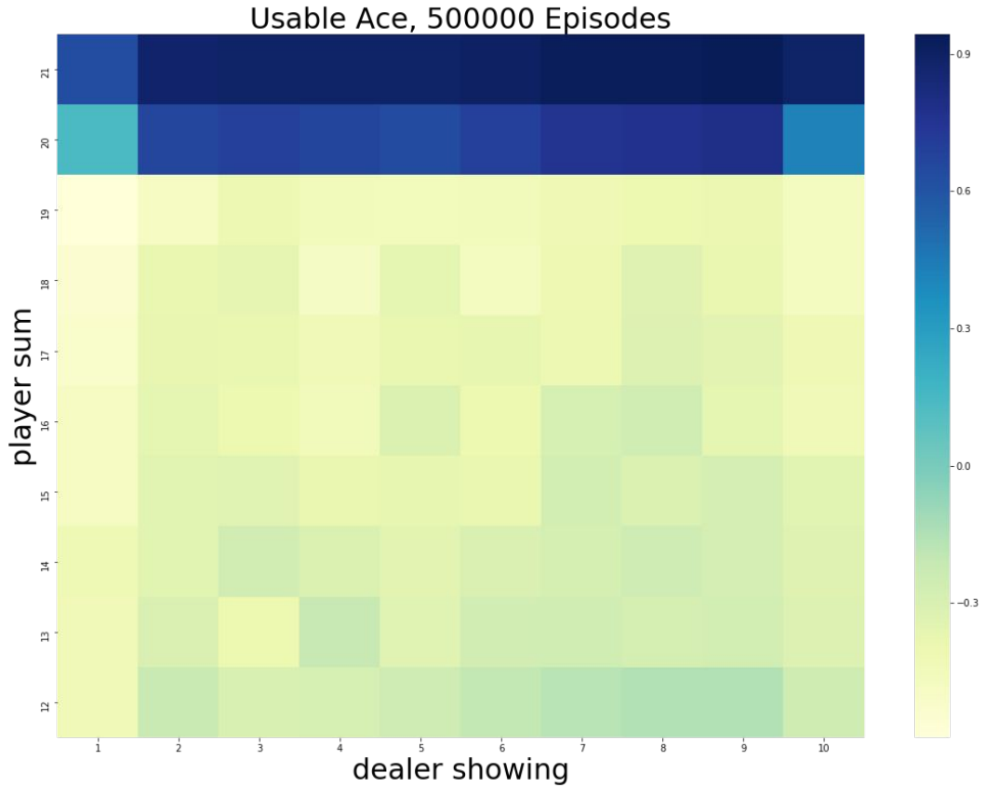
```python
    ...

    # game starts!

    # player's turn
    player_sum, player_trajectory = player()

    # dealer's turn
    dealer_sum, _ = dealer()

    # Determine the winner if no one busted out
    # compare the sum between player and dealer
    assert player_sum <= 21 and dealer_sum <= 21
    if player_sum > dealer_sum:
        return state, 1, player_trajectory  # Win
    elif player_sum == dealer_sum:
        return state, 0, player_trajectory  # Draw
    else:
        return state, -1, player_trajectory  # Lose
```

Monte Carlo on-policy

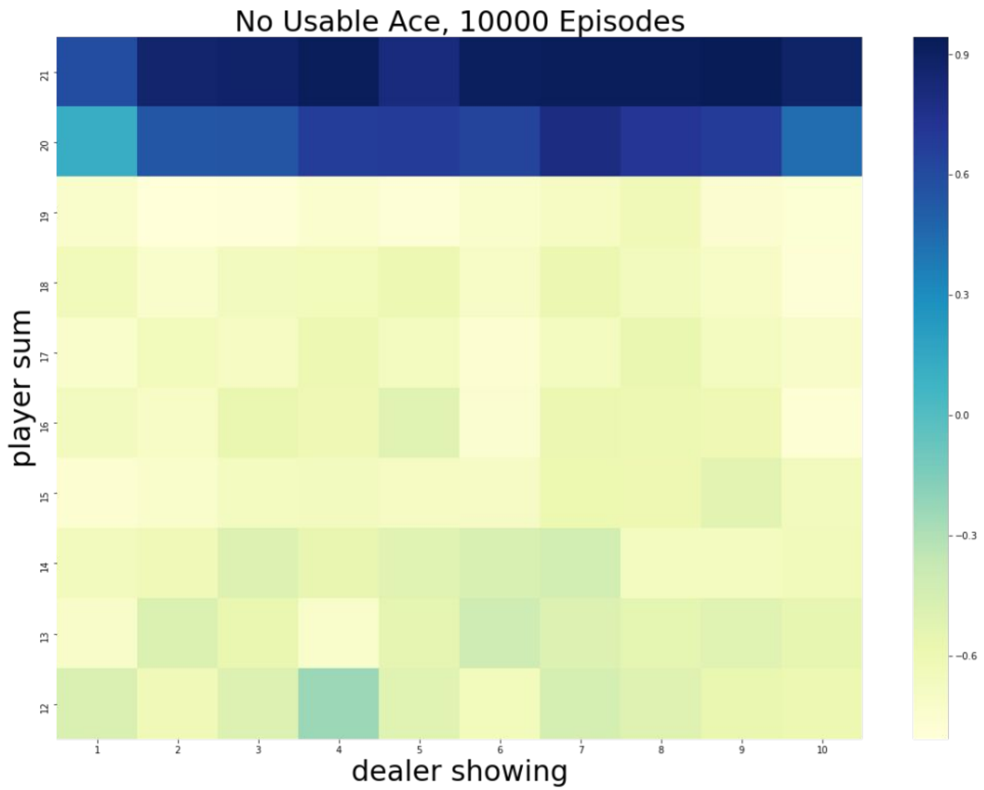
```
# Monte Carlo Sample with On-Policy
def monte_carlo_on_policy(episodes):
    # all states given that the player have a usable ace: 10 possible values of sum
    # times ten possible values for the card face up = 100
    states = np.zeros((10, 10))
    # initialize counts to 1 to avoid 0 being divided
    # counting the visits to any state
    states_count = np.ones((10, 10))

    for i in tqdm(range(0, episodes)):
        _, reward, player_trajectory = play(target_policy_player)
        for (usable_ace, player_sum, dealer_card), _ in player_trajectory:
            player_sum -= 12  # so that it ranges from 0 to 9
            dealer_card -= 1  # same: ranging from 0 to 9
            if usable_ace:
                states_count[player_sum, dealer_card] += 1
                states[player_sum, dealer_card] += reward  # sum of returns

    return states / states_count
```

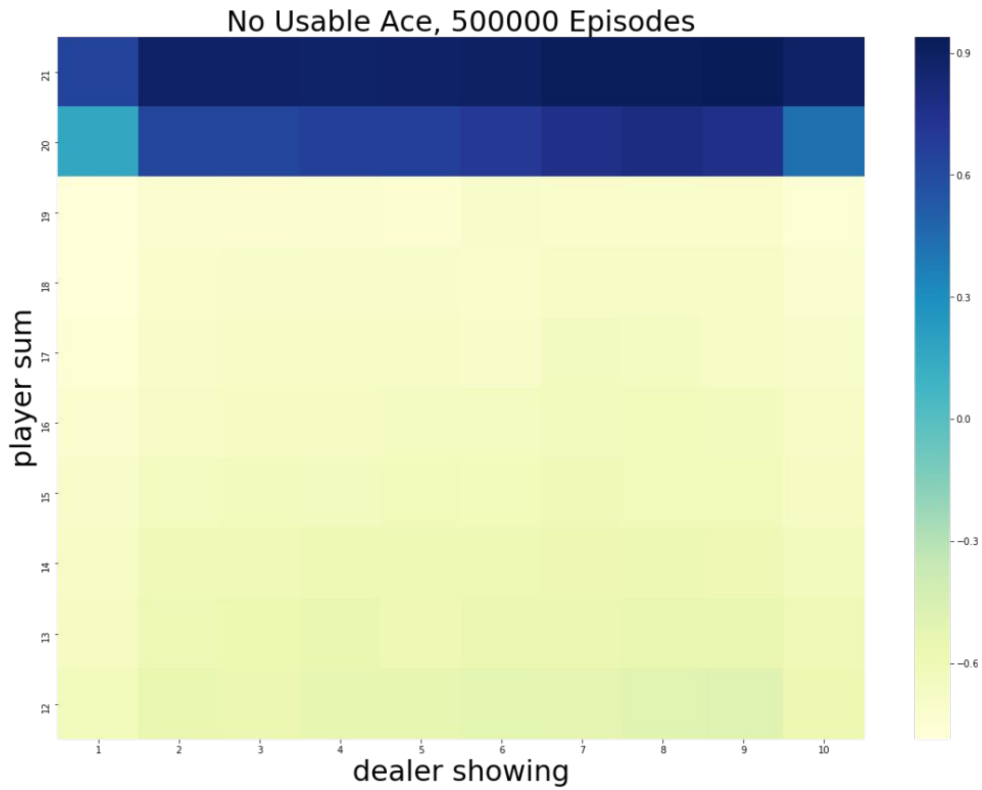
Monte Carlo on-policy





Monte Carlo on-policy





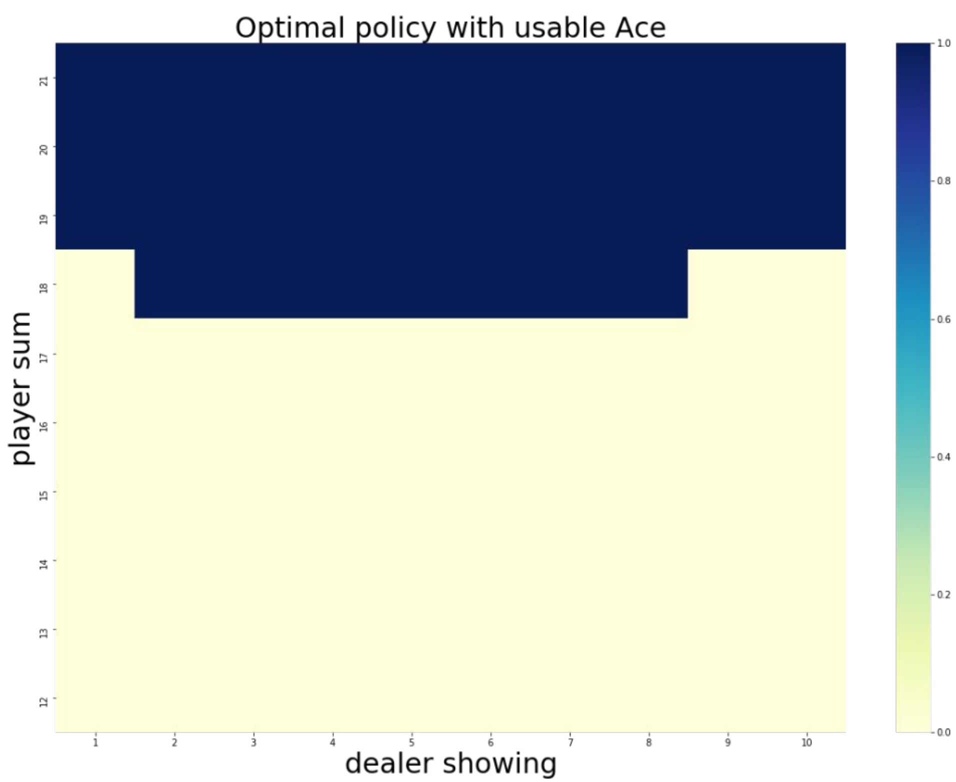
Monte Carlo with exploring starts

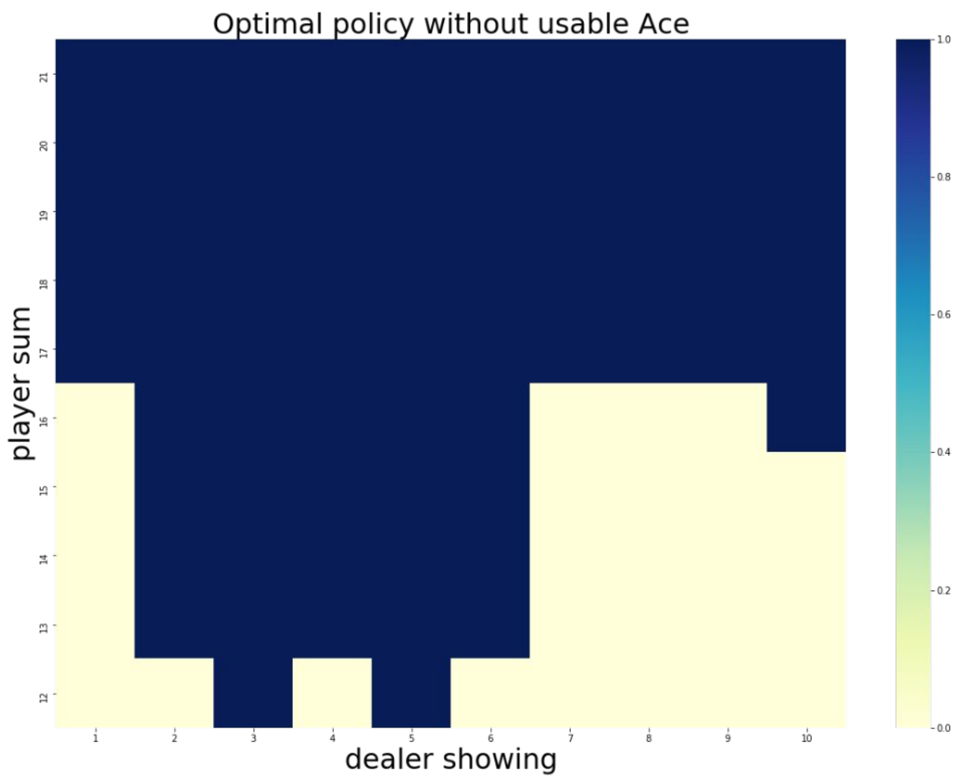
```python
# Monte Carlo with Exploring Starts
def monte_carlo_es(episodes):
    # here we estimate the action-value function!
    # (playerSum, dealerCard, usableAce, action)
    state_action_values = np.zeros((10, 10, 2, 2))
    # initialize counts to 1 to avoid division by 0
    state_action_pair_count = np.ones((10, 10, 2, 2))

    # behavior policy is greedy
    def behavior_policy(usable_ace, player_sum, dealer_card):
        return greedy_choice(usable_ace, player_sum, dealer_card)


    ...
```

Monte Carlo with exploring starts

```
    ...

    # play for several episodes
    for episode in tqdm(range(episodes)):
        # for each episode, use a randomly initialized state and action
        # it is not the same as initializing the initial state to None!
        initial_state = random_init()
        initial_action = np.random.choice(ACTIONS)

        #Play an episode
        _, reward, trajectory = play(current_policy, initial_state, initial_action)

        # The set is probably here just to stress the fact that we want a first-visit algorithm!
        first_visit_check = set()
        for (usable_ace, player_sum, dealer_card), action in trajectory:

            # update values of state-action pairs
            state_action_values[player_sum, dealer_card, usable_ace, action] += reward
            state_action_pair_count[player_sum, dealer_card, usable_ace, action] += 1

    return state_action_values / state_action_pair_count
```

Monte Carlo with exploring starts

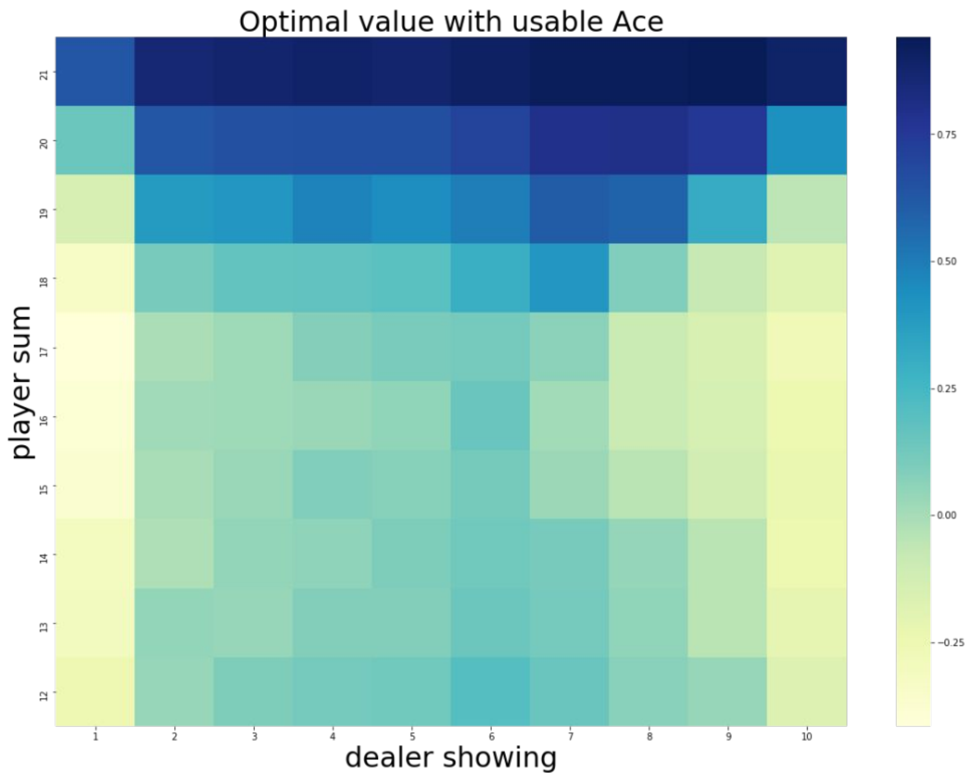


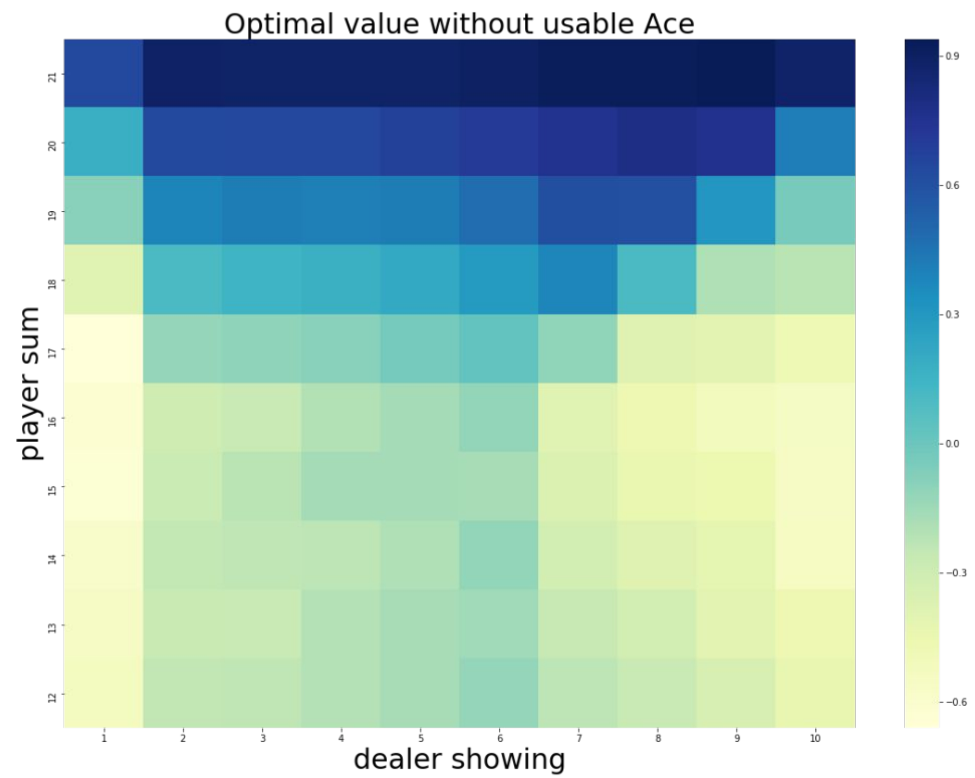


With usable ace **Without usable ace**

```
# actions: hit or stand
ACTION_HIT = 0
ACTION_STAND = 1  # "strike" in the book
```

Monte Carlo with exploring starts





With usable ace

Without usable ace