Huiswerk Opdracht 2

Deze huiswerkopdracht bevat vragen en programmeeropdrachten. Bij elke opdracht staat het aantal punten dat er voor te verdienen is. Iedere student moet de opdrachten zelfstandig uitvoeren. Volg de volgende stappen bij het inleveren van de opdrachten:

- Maak een map met daarin:
  - Een PDF-document met alle vragen en antwoorden.
  - Je Python-bestanden en evt. bijgeleverde scripts (die zullen nodig zijn om de code uit te voeren).
- Comprimeer de map als een zip-bestand met de naam `HW2-{voornaam}_{achternaam}.zip`.
- Stuur het bestand op naar abe_vos@msn.com met het onderwerp `HW1 {voornaam} {achternaam}`.

De deadline is donderdag 19 mei om 23:59. Vragen over het huiswerk kun je sturen naar abe_vos@msn.com.

Monte Carlo Methodes

Blackjack (1 punt)

In deze opdracht gaan we een waardefunctie leren voor Blackjack met Monte Carlo Prediction. OpenAI gym geeft een implementatie van dit spel, we kunnen de omgeving als volgt opzetten:

```
import gym

env = gym.make("Blackjack-v1")
```

We kunnen meer leren over de spelregels van Blackjack door de documentatie van deze omgeving te bekijken:

```
help(env)
```

Om de waardefunctie te leren, moeten we eerst een beleidsfunctie hebben. We gaan hier uit van een simpele strategie waarbij de agent de actie “hit” speelt wanneer de score minder dan 20 is. Bij 20 of hoger speelt de agent “stick” en eindigt de episode. Maak onderstaande functie af om dit beleid te implementeren:

```
def simple_policy(state):
    # SCHRIJF JE CODE HIER
    return 0
```

Let er op dat bij Blackjack de staat niet een enkele integer is maar een tuple. Kijk in de documentatie onder “Observation space” om te zien hoe de staat is opgebouwd.

Hint: tuples kunnen in Python worden “uitgepakt” in losse variabelen:

```python
# Maak een state tuple.
state = (12, 2, False)

# Pak de tuple uit.
a, b, c = state

# TODO: bedenk betere namen voor a, b en c
```

Monte Carlo Prediction (6 punten)

Met onze beleidsfunctie kunnen we episodes van Blackjack simuleren:

```python
done = False

state = env.reset()

while not done:
    action = simple_policy(state)
    state_new, reward, done, _ = env.step(action)
```

Nu gaan we Monte Carlo Prediction implementeren.

```python
def mc_prediction(policy, env, num_episodes, discount_factor=1.0):
    # Keep track of how many times each state has been visited.
    # A defaultdict creates a default value (0) when we want to read a key
    # that is not stored, e.g.: `print(state_count["some state"])` prints `0`.
    N = defaultdict(int)

    # The final value function
    V = defaultdict(float)

    for episode in range(num_episodes):
        state = env.reset()
        done = False

        episode = []

        while not done:
            action = simple_policy(state)
            state_new, reward, done, _ = env.step(action)

            # JOUW CODE HIER
            # Sla informatie van deze stap op in een tuple en voeg die toe
            # aan de lijst.

            state = state_new
```

```
        G = 0

        for step in episode[::-1]:  # Loop through the steps backward.
            ... = step  # JOUW CODE HIER: Pak de tuple uit.

            # JOUW CODE HIER
            # Update G, N en V volgens het MC prediction algoritme

    return V
```

Resultaten (1 punt)

Nu kunnen we de resultaten bekijken. Importeer de functie `create_plots` uit `plot_mc_prediction.py` en voer die uit:

```
from plot_mc_prediction import create_plots

create_plots(env, simple_policy, mc_prediction)
```

Let op, het algoritme wordt voor 500.000 episodes uitgevoerd, dit kan even duren. Sla de gegenereerde plots op en zet ze in het verslag. Omschrijf in eigen woorden wat je ziet.

Monte Carlo Control (1 punt)

Waarom kunnen we met de geleerde waardefunctie niet een beter beleid leren zoals we dat met Policy Improvement deden?

Hint: kijk naar de update stap van Policy Improvement.

MC en TD (3 punten)

Noem een voordeel en een nadeel van Monte Carlo methodes. Noem ook een situatie waarbij je beter TD learning kan gebruiken.

TD Control

Bij deze oefening gaan we Sarsa en Q-learning vergelijken. We proberen de algoritmes uit op de `CliffWalking` omgeving. Dit is een gridworld met 4 rijen en 12 kolommen. De agent begint linksonder in staat S en moet rechtsonder in staat G terecht komen. Alle tegels tussen S en G vormen een klif, als de agent op een van deze tegels belandt, valt die naar benenden, krijgt een beloning van -100 en wordt teruggezet op staat S. Elke andere overgang levert een beloning van -1 op. Wanneer de agent staat G bereikt, eindigt de episode. We gaan uit van $\gamma = 1$. De `CliffWalking` omgeving staat in `cliffwalking.py` en kan als volgt opgezet worden:

```
from cliffwalking import CliffWalking

env = CliffWalking()
```

De omgeving kan in beeld gebracht worden met `env.render()`. Hier zijn de kliftegels aangegeven met `#` en de huidige staat van de agent is `O`.

Epsilon-greedy (1 punt)

Voor zowel Sarsa als Q-learning is het belangrijk dat we alle staten blijven bezoeken tijdens het trainen. Dit doen we met een epsilon-greedy beleid. Bij dit beleid genereren we een waarde tussen 0 en 1 met `random.random()`. Als de waarde lager is dan `epsilon`, kiezen we een willekeurige actie. Als de waarde gelijk of hoger is dan `epsilon`, kiezen we de actie met de hoogste Q-waarde voor de gegeven staat. Implementeer dit beleid, de acties worden uitgedrukt als integers, dus de functie geeft een waardes van `0` tot en met `nA - 1`.

```
def egreedy_policy(Q, s, nA, epsilon=0.1):
    pass
```

Hint: In de functie opzet hierboven is `Q` een dictionary met een staat als key en een lijst als value. De lijst bevat een Q-waarde voor elke actie. Er zijn `nA` mogelijke acties, dus dat is ook de lengte van de lijst.

Sarsa (6 punten)

Eerst implementeren we Sarsa. Vul de code aan om de Q functie te updaten met de Sarsa update stap.

We slaan de som van alle beloningen in een episode op als `total_reward`. Deze variabele heb je voor nu nog niet nodig.

```
def sarsa(env, num_episodes, learning_rate=0.5, discount_factor=1.0):
    Q = {}
    nA = env.action_space.nA
    rewards = []

    for episode in range(num_episodes):
        done = False
        total_reward = 0

        state = env.reset()
        action = egreedy_policy(Q, state, nA)

        while not done:
            state_new, reward, done, _ = env.step(action)
            total_reward += reward

            # JOUW CODE HIER
```

```
            # Kies een nieuwe actie met het egreedy beleid.
            # Update `Q[state][action]`
            # Update `state` en `action` voor de volgende iteratie.

        rewards.append(total_reward)

        print(f"Episode {episode}, sum reward: {total_reward}")

    return Q, rewards
```

Hint: wanneer je een staat voor het eerst observeert, zal deze nog niet als key in `Q` bestaan. Zorg dus dat je deze aanmaakt *voordat* je die probeert uit te lezen. Onthoud: `Q` is een dict van lijsten, waar elke lijst `Q[s]` een Q-waarde heeft voor elke actie in `s`.

Q-learning (4 punten)

Q-learning lijkt erg op Sarsa. Dit keer halen we de tweede actie niet uit ons beleid, maar kiezen we altijd de actie met de hoogste Q-waarde voor de volgende staat. Hieronder is een functie die voor een gegeven staat, de maximale Q-waarde teruggeeft.

```
def get_optimal_value(Q, s, nA):
    action_values = Q.get(s, [0 for _ in range(nA)])
    return max(action_values)
```

Implementeer nu het Q-learning algoritme.

```
def q_learning(env, num_episodes, learning_rate=0.5, discount_factor=1.0):
    Q = {}
    nA = env.action_space.nA
    rewards = []

    for episode in range(num_episodes):
        done = False
        total_reward = 0

        state = env.reset()

        while not done:
            # JOUW CODE HIER
            # Kies een nieuwe actie met het egreedy beleid.
            action = ...

            state_new, reward, done, _ = env.step(action)
            total_reward += reward

            # JOUW CODE HIER
```

```
            # Update `Q[state][action]`, gebruik `get_optimal_value`.
            # Update `state` voor de volgende iteratie.

        rewards.append(total_reward)

        print(f"Episode {episode}, sum reward: {total_reward}")

    return Q, rewards
```

On-policy & off-policy (2 punten)

Omschrijf in eigen woorden het verschil tussen on-policy en off-policy methodes.

Vergelijken (4 punten)

Als we beide algoritmes laten trainen op de `CliffWalking` omgeving en de `rewards` lijsten plotten, krijgen we een afbeelding als hieronder. Zoals je kan zien doet Sarsa het gemiddeld een stuk beter dan Q-learning.

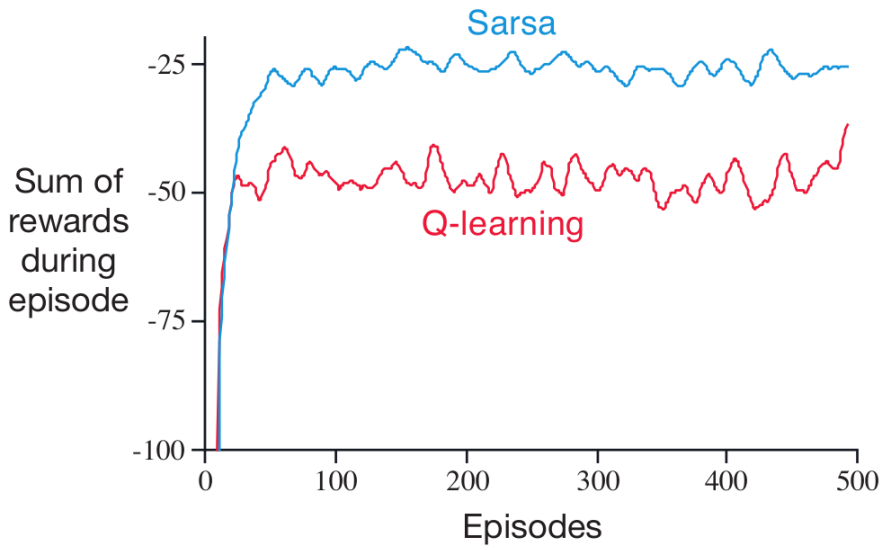

Figure 1: Sarsa vs. Q-learning

Probeer je implementaties zelf uit op de omgeving en plot de beloningen. Dit kan je doen met `matplotlib`:

```
import matplotlib.pyplot as plt

values = [1, 2, 4, 9, 15]
```

```
plt.plot(values, label="x^2")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.plot()
```

Als de plots niet overeen komen, probeer dan de learning rate aan te passen.

Alhoewel het lijkt alsof Q-learning slechter presteert dan Sarsa, is eigenlijk het tegengestelde het geval! Kan je verklaren waarom dit zo is?

Hint: render het verloop van een episode nadat de algoritmes zijn getraind met `epsilon=0` (zodat je een greedy beleid krijgt). Hoe verschillen de twee geleerde beleidsfuncties?