B313 Machine Learning 2 - WS 24/25
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
HTW Berlin - Hochschule für Technik und Wirtschaft Berlin

Beleg 2

Dies ist der zweite Beleg des Kurses. Wie im Moodle angegeben, reichen Sie bitte für diesen Beleg ein `ipynb` oder `py` und ein `pdf` ein. Der Beleg soll die folgenden Inhalte vermitteln:

- Einarbeitung in die gym Umgebung von python
- Implementierung und experimentieren mit dem Q-learning Ansatz sowie verwandten iterativen Verfahren
- Wir lernen weitere grundlegende Begriffe und allgemeine Methoden des maschinellen Lernens kennen

Je nach Vorkenntnis kann der Beleg eine hohe Schwierigkeit aufweisen - lassen Sie sich davon nicht entmutigen. Insbesondere in diesem Fall, empfehle ich Ihnen die Teilnahme an der Übung in der wir uns geleitet durch den Beleg arbeiten werden und umfassende Hilfestellung möglich ist.

Aufgabe 1: gym ToyText Grundlagen

Gym (https://www.gymlibrary.dev/) bietet unter anderem eine umfassende Sammlung an Umgebungen bzw. Aufgaben welche im Rahmen des reinforcement learnings behandelt und untersucht werden können. Wir wollen uns in dieser Übung mit Vertretern aus den ToyText Beispielen auseinandersetzen.

a) Installieren Sie gym für die Verwendung der ToyText Beispiele mittels `pip install gym[toy_text]`

b) Machen Sie sich grundlegend mit den Umgebungen `"Taxi"` sowie `"FrozenLake"` vertraut. Insbesondere benötigen Sie hierfür die Befehle

   - `gym.make()` (erzeugt eine Umgebung `env`, dem Make Befehl muss ein render_mode übergeben werden, `render_mode="ansi"` erlaubt eine Ausgabe der Umgebung als String ins Terminal)
   - `env.reset()` (initialisiert die Umgebung)
   - `env.render()` (erzeugt eine Darstellung des Zustands der Umgebung - z.B. als String)
   - `env.step()` (führt einen Schritt für eine übergebene Aktion aus und gibt u.a. den reward und neuen Zustand des Umgebung zurück)

   Führen Sie u.a. manuelle Schritte (mit der step Funktion) für verschiedene Aktionen aus und beobachten Sie die entsprechende Wirkung (auf reward und andere Rückgabewerte) und ordnen Sie den Aktionen (z.B. 0,1,2,3 bei ForzenLake) die entsprechenden Aktionen im Spiel zu.

c) Die ForzenLake Umgebung hat zusätzlich die Option (in der make Funktion) is_slippery - welche Auswirkung haben hier die verschiedenen Konfigurationen (is_slippery = True / False).

Aufgabe 2: Q-learning

Beim Q-learning nutzen wir ein Q-array $Q \in \mathbb{R}^{|S|\times|A|}$ welches wir iterativ updaten um möglichst nahe an unser optimales Q_* zu kommen. Hierfür gehen wir folgende Schritte durch

```
choose ϵ > 0, γ ∈ [0,1],  α > 0, N_e ∈ ℕ, Q ∈ ℝ^{|𝒮|×|𝒜|}
loop over episodes:
  initalize S
  loop over steps of episode:
    choose action A for state S derived from Q (e.g. eps-greedy from Q)
    take action A and observe S′, R
    update Q(S,A) with S′ and R
    S ← S′
```

d.h.

- wir wählen ein $\epsilon > 0$ für eine ϵ-greedy Strategie um vom aktuellen Q Aktionen zu wählen, den decay-factor $\gamma \in [0, 1]$, eine Lernrate/Schrittweite $\alpha > 0$ und eine Anzahl von Episoden N_e
- wählen ein beliebiges anfängliches Q-Array der shape $(|\mathcal{S}|, |\mathcal{A}|)$

Dann initialisieren wir für jede Episode die Umgebung in einem Startzustand, S und spielen die Episode mit folgenden Schritten bis zum Ende (d.h. bis entweder die Anzahl der maximalen Schritte erreicht oder das Spiel als beendet gilt)

- wählen für den aktuellen Zustand S mittels ϵ-greedy eine Aktion A vom aktuellen Q
- führen die Aktion A aus und erhalten den neuen Zustand S' und reward R
- updaten das Q an der Stelle $Q(S, A)$ mittels

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a\in\mathcal{A}'} Q(S', a) - Q(S, A)] \tag{1}$$

- überschreibe den Zustand S mit S'

Um eine Aktion A für das aktuelle Q mittels ϵ-greedy zu wählen nutzen wir analog wie zuvor:

```
given ϵ > 0, Q ∈ ℝ^{|𝒮|×|𝒜|}, S ∈ 𝒮
draw random number z from [0,1]
if z < ϵ or Q(S,a) same value for all a:
  choose random action A
else:
  choose A = arg max_{a∈𝒜} Q(S,a)
```

a) Implementieren Sie die Q-learning Methode für variable Umgebungen (env) sowie für Variable policies. Z.B. könne Sie dies umsetzen mittels einer

- `update_Q` Funktion welche das Q-Array für einen gegebenen Zustand, reward, gewählte Aktion sowie ein α und γ aktualisiert
- `epsilon_greedy_agent` Funktion welche eine Aktion ϵ-greedy aus dem aktuellen Q wählt
- einer Trainingsschliefe welche für eine gegebene Anzahl an Episoden, das Spiel solange spielt, bis die Umgebung einen finalen Zustand meldet oder eine maximale Anzahl an Schritten überschritten wurde (da die Umgebungen auch unendlich Spiele zulassen würden, sollten Sie die Episodenlänge auf z.B. 1000 begrenzen)

Aufgabe 3: Hyperperametersuche
Vielen Methoden des maschinellen Lernens hängen von sogenannten Hyperperametern ab. Dies sind Parameter für die Rechnung (z.B. die Schrittweite α) welche im einfachsten Fall für eine Rechnung konstant sind, deren Wert jedoch das Ergebnis der Rechnung beeinflussen. Diese Hyperparameter müssen geeignet gewählt werden. Eine Liste der Hyperparameter in unserem Setting wäre:

- die Schrittweite α
- der Decayfactor γ
- die Anzahl der gespielten Episoden N_{eps}
- das ϵ für den `epsilon_greedy_agent`
- der Startwert Q_0 für das Q-Array

Die einfachste Form der Suche nach geeigneten Hyperparametern wäre einen Parameter zu optimieren, während alle anderen Parameter fixiert bleiben (eine Art der Linesearch). Im simpelsten Fall definieren wir z.B. für α die möglichen Werte $\alpha \in \{0.001, 0.01, 0.1, 1\}$ und trainieren für jeden ein Q-Array und vergleichen dieses. (Komplexer wäre eine Gridsearch bei der für mehrere/alle Parameter entsprechende Mengen definiert werden und alle Kombinationen getestet werden).

Um zwei Q-Arrays miteinander zu vergleichen brauchen wir eine Metrik. Im einfachsten Fall können wir hier die Strategie wählen, dass wir wiederholt mit einem `greedy_agent` für ein entsprechendes Q-Array spielen und vergleichen, wie oft diese das Spiel erfolgreich abschließen.

a) Implementieren Sie eine `eval_q` Funktion welche für eine übergebenes Q-Array und eine gegebene Umgebung ausgibt, wie erfolgreich die vom Q-Array abgeleitete policy beim Lösen der Aufgabe ist (Anteil der Erfolge bei z.B. 200 Versuchen).

b) Nutzen Sie die Aufgabe `"FrozenLake"` mit `is_slippery = False` um einen Hyperparametersuche für den Parameter γ zu implementieren. Verwenden Sie zum Beispiel: $\alpha = 0.1$, $\epsilon = 0.1$, $Q_0 = 0$, $N_{eps} = 5000$ sowie $\gamma \in \{0, 0.1, 0.5, 0.9, 1\}$. Plotten Sie hierfür den Graphen γ gegen die Erfolgsrate des finalen Q-Arrays.
(Optional: Sie sollten finden, dass das Training in fast allen Fällen zum Erfolg führt - untersuchen Sie ob es einen Zusammenhang zwischen γ und dem Zeitpunkt ab wann der Erfolg eintritt gibt.)

c) Nutzen Sie nun die Umgebung `"FrozenLake"` mit `is_slippery = True` und beobachten Sie die Veränderung. Erstellen Sie erneut den Plot von γ gegen die Erfolgsrate des finalen Q-Arrays (falls auf Ihrer Hardware vertretbar: mitteln Sie die Rate über 10 Wiederholungen um Fluktuationen zu vermindern).

d) Plotten Sie eine Lernkurve für das beste Setting aus c) in dem Sie nach z.B. stets 20 oder 50 Schritten die `eval_q` Funktion für das aktuelle Q-Array aufrufen und die Erfolgsrate über den Verlauf des Trainings plotten.

e) Wiederholen Sie b) für die Umgebung `"Taxi"`.

Aufgabe 4: Diskussion
Diskutieren Sie kurz (max 3-Seiten mit Abbildungen) Ihre Ergebnisse aus der Aufgabe "Hyperparametersuche" in einem pdf. Versuchen Sie hierbei insbesondere darauf einzugehen, warum

Sie die entsprechenden Kurven erhalten bzw. diese sinnvoll erscheinen, welche Besonderheiten Ihnen auffallen und wieso diese auftreten, und welchen Rolle Zufall in den verschiedenen Settings spielt.

Aufgabe 5: Extra (optional)

a) Implementieren Sie SARSA sowie expected SARSA anstelle des Q-learnings und wiederholen Sie z.B. Aufabe c) der Hyperparametersuche.

b) Testen Sie den Zusammenhang von weiteren Hyperparemetern zur Erfolgsrate