Aufgabenblatt 2 - Markov Decision Process

Gruppenmitglieder:

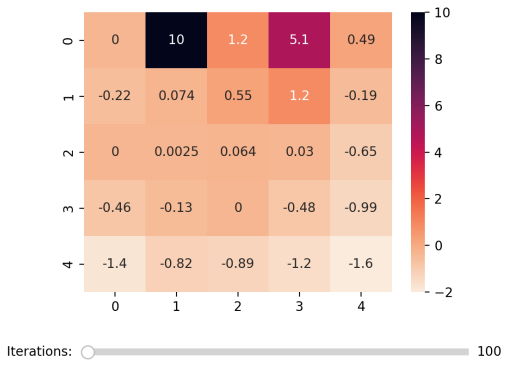
Alina Micke (515183)

Björn Balkow (514317)

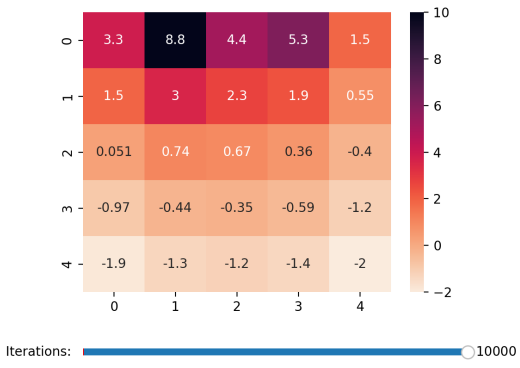
Mathis Hunke (508192)

Aufgabe 2.1 - Value-Funktion für eine Grid-World Umgebung

(a) Bellman mit State Value-Funktion



State Values nach 100 Schritten

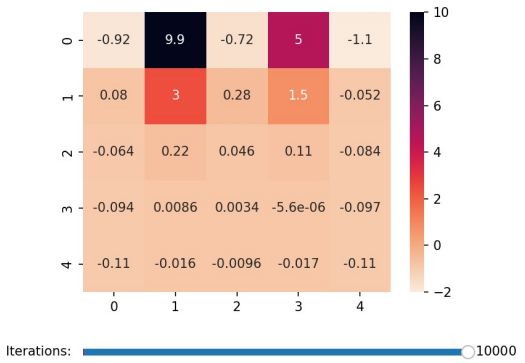


State Values nach 10.000 Schritten

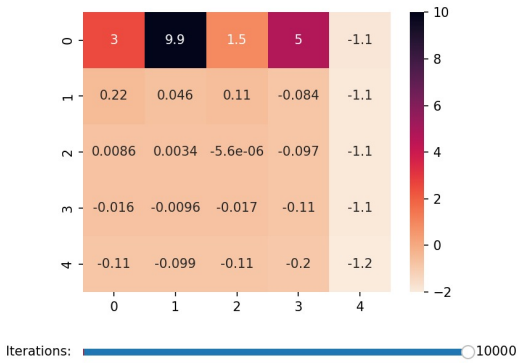
Beschreibung der Ergebnisse der Berechnungen:
Als Ergebnis erhalten wir eine
5×5 Matrix, die vom Aufbau mit unserer Grid-World Umgebung übereinstimmt. Dabei gibt der Wert (i, j) in der `state_value` Matrix, den `state_value` der Position (i, j) in der Grid-World an. Dieser gibt an, wie "gut" es ist, dass wir uns auf diesem Feld befinden. Ist der Wert negativ, so ist bei zufälligem Verhalten in näherer Zukunft ein negativer Reward zu erwarten. Ist der Wert positiv (z.B. an Position A), so können wir mit einem positiven Reward in Zukunft rechnen.
Der Vorteil, den die Verwendung einer zufälligen Policy mit sich bringt, ist, dass genügend Informationen über alle Felder gesammelt werden. Eine Return-orientierte Policy hätte das Problem, dass vorrangig Informationen über "gute" Felder gesammelt werden. Dies führt zu einem ähnlichen Problem, wie bei den Multi-arm-Bandits, i.e. eine gute Alternative, wie Weg B wird evtl. nicht gefunden.

(b) Bellman mit Action Value-Funktion

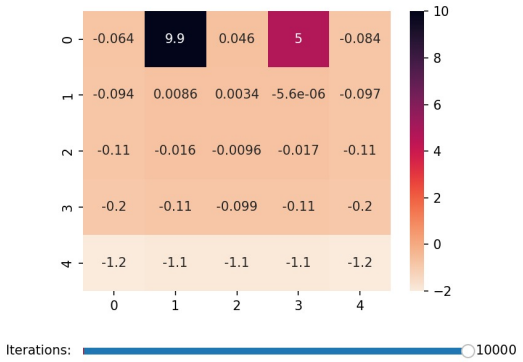
Action Value-Matrizen für die vier Bewegungsrichtungen nach 10.000 Iterationen



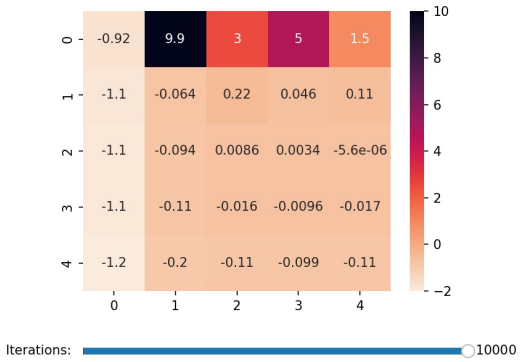
Nord-Bewegung



Ost-Bewegung



Süd-Bewegung



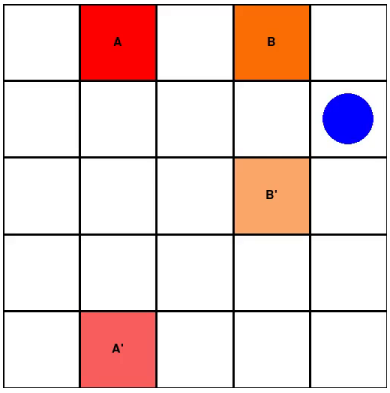
West-Bewegung

Als Ergebnis der Action-Value-Funktion erhalten wir in Abhängigkeit der Aktion eine Matrix. Dabei beschreibt der Wert in der Matrix zur Aktion a in dem jeweiligen Feld, wie "gut" das Feld ist. Dies ist so zu interpretieren, dass umso höher der Wert, umso sinnvoller ist es die jeweilige Aktion auszuführen, wenn man sich auf dem Feld befindet.
Die state-value Werte zeigen also an, wie hoch das Feld bei einer zufälligen Aktion zu bewerten ist, die Action-Value Werte beschreiben, wie hoch die Aktion zu bewerten ist, wenn man in dem jeweiligen Feld steht.
Dabei gibt jeder Value eine Bewertung für das jeweilige Feld ab, einmal bezüglich einer zufälligen Aktion, einmal in Abhängigkeit von dieser.

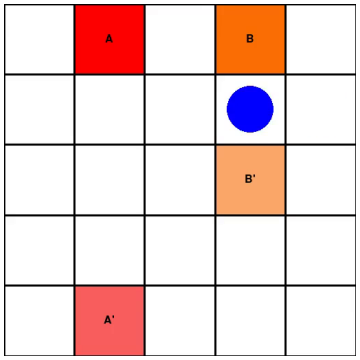
Aufgabe 2.2 - OpenAI Gym: Implementieren der Grid-World

b)

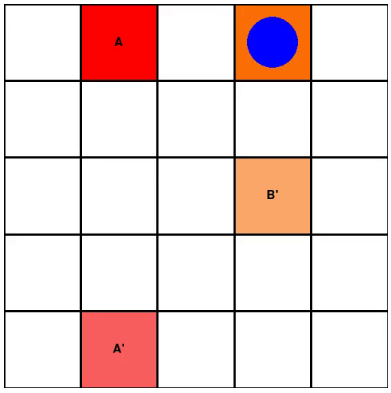
Visualisierung der GridWorld Umgebungs-Interaktion über 6 Zeitschritte



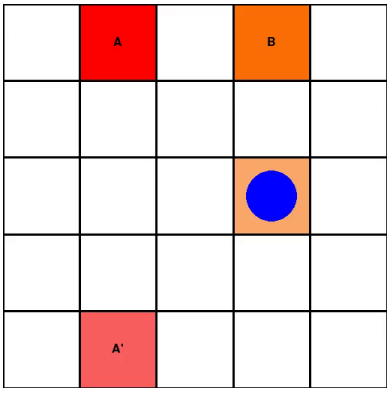
Zeitschritt 1



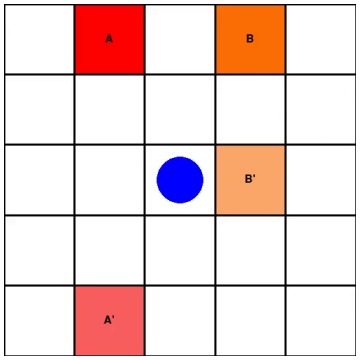
Zeitschritt 2



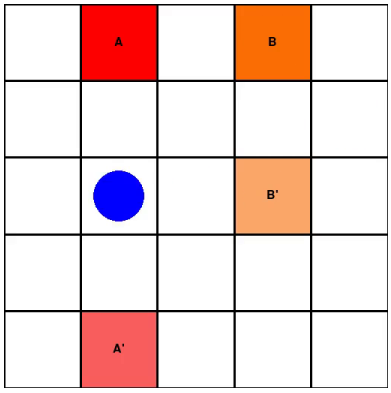
Zeitschritt 3



Zeitschritt 4



Zeitschritt 5



Zeitschritt 6

Für eine GIF-Animation, s. beigefügte Dateien.

Aufgabe 2.3 - Controller für die CartPole Gymnasium Umgebung

(a) Konzeptualisieren des RL Problems

Zustandsraum

Der Zustandsraum ist ein Vier-Tupel, welches die folgenden Variablen bündelt:

- Horizontale Position des Wagens $x \in [-4.8, 4.8]$
- Geschwindigkeit des Wagens $\dot{x} \in [-\infty, +\infty]$
- Winkel des Stabs zur Vertikalen $\theta \in [-24^\circ, +24^\circ]$

- Geschwindigkeit des Stabs $\dot{\theta} \in [-\infty, +\infty]$

Anmerkung: Als Standardeinstellung terminiert die Episode, wenn der Winkel des Stabs θ außerhalb des Intervalls $\pm 12^\circ$ liegt.

Aktionsraum

Diskreter Aktionsraum, der zwischen zwei Aktionen unterscheidet: Links (0) und Rechts (1). Je nach ausgewählter Aktion wird der Wagen in die linke oder rechte Richtung bewegt, ohne dabei Einfluss auf die Bewegungsgeschwindigkeit zu nehmen.

Umgebungsdynamik

Die Umgebungsdynamik $P(s'|s, a)$ modelliert die Auswirkungen der Kräfte auf den Wagen und das Pendel und berechnet daraus den Folgezustand s'.

Physik des Systems: Die Geschwindigkeit des Wagens, die durch die aufgebrachte Kraft verringert oder erhöht wird, ist nicht festgelegt und hängt vom Winkel ab, in dem die Stange ausgerichtet ist. Der Schwerpunkt der Stange variiert die Menge an Energie, die benötigt wird, um den Wagen unter der Stange zu bewegen.

Beispiel:

Angenommen, die Stange hat sich um 30° von der Vertikalen geneigt (positive Neigung). Der Agent übt eine Kraft nach rechts auf den Wagen aus. Da die Stange nach links geneigt ist, wird die Schwerkraft versuchen, die Stange wieder nach unten zu ziehen, was dazu führt, dass die Stange sich weiter nach links neigt. In diesem Fall muss der Agent eine größere Kraft aufbringen, um den Wagen zu beschleunigen, weil ein Teil dieser Kraft der Schwerkraft entgegenwirken muss.

Belohnungsstruktur

Das Ziel ist es, den Stab so lange wie möglich zu balancieren. Ein Reward von +1 wird für jeden Zeitschritt gegeben, in dem der Wagen es schafft den Stab zu balancieren (inklusive des Terminierungs-Zeitschritts) und innerhalb der Umgebungsbegrenzung zu bleiben. Der Return wird maximiert, wenn der Stab möglichst lange balanciert wird.

(b) Intuitiver Lösungsansatz

Beschreibung der Policies:

Die naive Policy benutzt lediglich den Winkel des Stabs. Ist der Winkel des Stabs negativ, soll der Wagen sich nach links bewegen, ist der Winkel positiv, soll sich der Wagen nach rechts bewegen.

```
def naive_policy(observations: npt.NDArray[np.float64]) -> int:
    """
    Naive policy only checking for pole angle.
    :param observations: Array of environmental observations.
    :return: direction encoding
    """
    _, _, pole_angle, _ = observations
    # move left if pole angle is negative, otherwise move right
    return 0 if pole_angle < 0 else 1
```

Die erweiterte Policy (`custom_policy`) nutzt zusätzlich noch die Geschwindigkeit des Wagens. Hier wird zunächst ein einfacher Wert berechnet, der den Winkel des Stabs ins Verhältnis zur Wagengeschwindigkeit setzt. Der Wert ermittelt sich wie folgt:

`cart_velocity > pole_angle * 20`

Der Faktor 20 wurde durch manuelles Ausprobieren ermittelt.

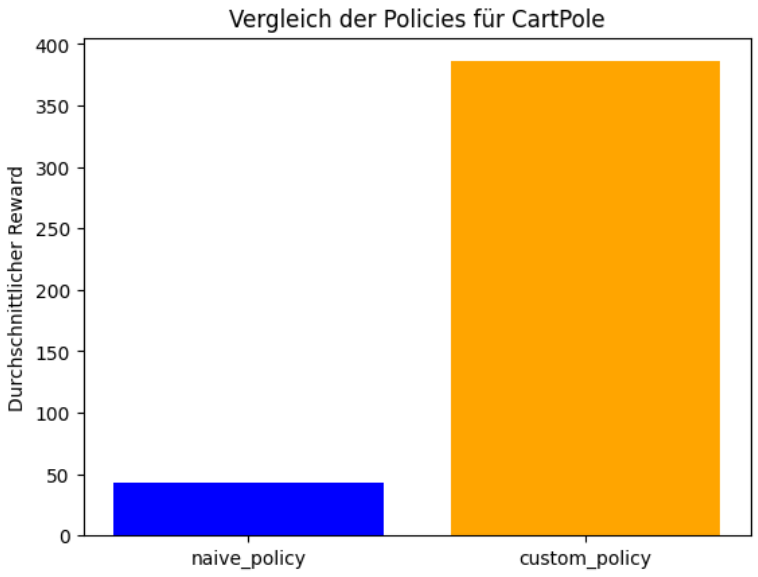
Wenn der Winkel des Stabs relativ klein ist im Verhältnis zur Wagengeschwindigkeit, wird zunächst überprüft, ob der Wagen noch beschleunigt. Ist dies der Fall, soll der Wagen sich nach Rechts bewegen. Andernfalls nach links.

Ist der Winkel des Stabs relativ betrachtet zu groß für die beobachtete Wagengeschwindigkeit, wird zunächst erneut überprüft, ob der Wagen noch beschleunigt. Falls ja, soll der Wagen sich nach links bewegen, ansonsten nach rechts.

```
def custom_policy(observations: npt.NDArray[np.float64]) -> int
    """
    Policy taking the cart velocity and pole angle into account
    :param: observations as numpy array
    :return: direction
```

```
    """
    # extract required observations
    _, cart_velocity, pole_angle, _ = observations
    # if pole angle is relatively small to the cart velocity
    if cart_velocity > pole_angle * 20:
        # check if we are still accelerating
        if cart_velocity >= 1:
            # move right
            return 1
        # otherwise move left
        return 0
    # otherwise, if we are still accelerating enough
    elif cart_velocity >= 1:
        # move left
        return 0
    # else, move right
    return 1
```

Vergleich der Policies



Effektivität der Policies

Die naive Policy ist nicht besonders effektiv, da sie wesentliche Beobachtungen außer Acht lässt und damit zu einfache Anweisungen für den Agenten gibt.

Die `custom_policy` ist hingegen wesentlich effektiver, und das, obwohl sie nur einen Parameter mehr verwendet. Anhand der Abbildung ist zu sehen, dass dieser eine Parameter in einem deutlichen Anstieg des durchschnittlichen Returns führt.

Schwierigkeiten

Einen geeigneten Faktor zu suchen für die `custom_policy` erwies sich als Trial-and-Error Lösung. Dieser Faktor beeinflusst maßgeblich, wie der maximale Reward pro Episode ausfällt.

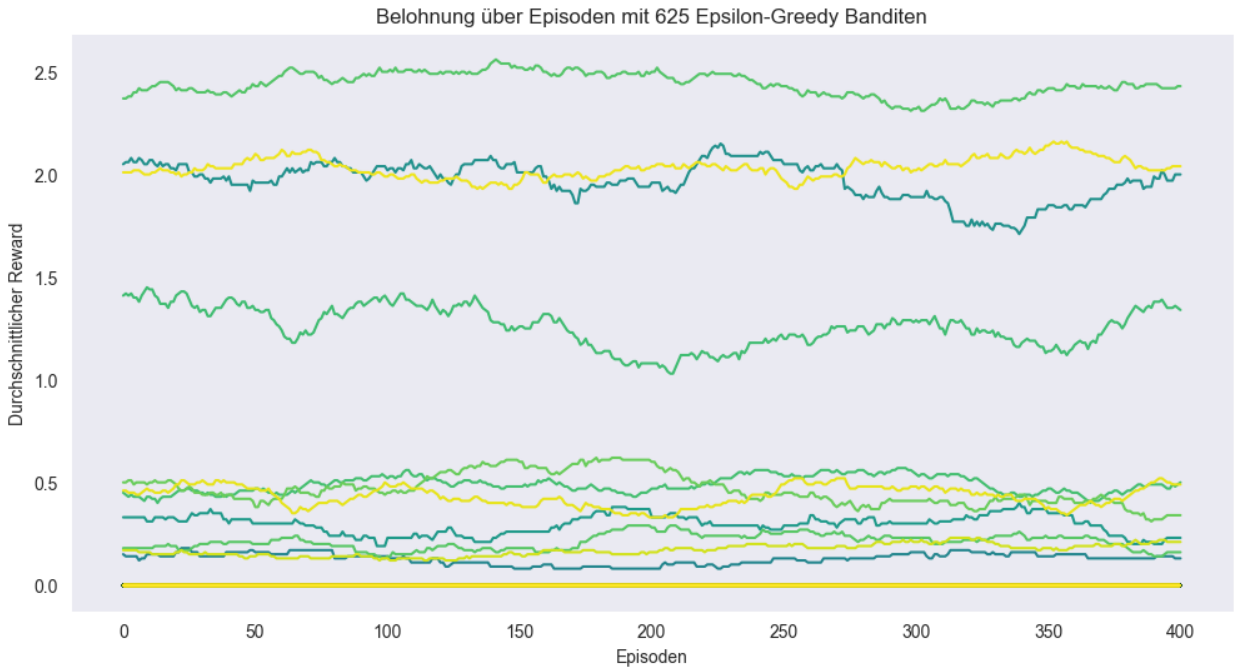
Bei der naiven Policy gab es keine besonderen Schwierigkeiten. Lediglich die schlechte Effektivität stellt ein Problem zur Maximierung des Returns dar.

Performanz

Die naive Policy ist in Hinblick auf Stabilität und Zielerreichung nicht effektiv, da der durchschnittliche Reward pro Episode unter 50 liegt. D.h., der Agent schafft es mit dieser Policy durchschnittlich weniger als 50 Zeitschritte den Stab zu balancieren. Diese Performanz wird deutlich im Vergleich zur `custom_policy`, welche es im Durchschnitt schafft, mehr als 350 Zeitschritte den Stab zu balancieren. D.h., die `custom_policy` ist wesentlich stabiler und erfüllt das Ziel bzw. die Aufgabe besser als die naive Policy.

(c) RL Lösungsansatz

Der Zustandsraum wurde in 5 Bins, also $5^4 = 625$ diskrete Zustände, unterteilt.



Performanz- und Konvergenzverhalten

Es wird immer nur der Bandit trainiert, der zum nächsten Zustand passt. Um alle Banditen zu trainieren, muss eine hohe Episoden-Anzahl gewählt werden. Dies stellt allerdings nicht sicher, dass alle Banditen gleich gut trainiert werden. Es ist sehr wahrscheinlich, dass einige Banditen gar nicht im Laufe einer Episode ausgewählt und trainiert werden (z.B. Banditen, die den Rand der Umgebung darstellen). Allerdings werden Banditen für wahrscheinliche Zustände (z.B. die mittleren) intensiv trainiert.

Bezüglich der Performanz lässt sich sagen, dass der Algorithmus sich nur auf die relevanten Zustände, und damit die Banditen, konzentriert und diese optimiert. Unwahrscheinliche Randfälle werden hier (fast) nie betrachtet, weshalb auch keine unnötige (Rechen-)Zeit für diese Fälle aufgewendet wird.

Zusammenfassend lässt sich sagen, dass das Konvergenzverhalten suboptimal ist, da die Optimierung von der Anzahl der Episoden und der Anzahl an Zuständen abhängt, die Performanz jedoch sinnvoll erscheint. Es werden nur die Banditen trainiert, die zu einer schnellen Lösung des Problems beitragen (i.e., wahrscheinliche Zustände und keine Randfälle).

Effizienz der Diskretisierung

Je stärker der Zustandsraum diskretisiert wird, desto genauer kann der Agent agieren und optimiert werden, da sich im Unendlichen der diskrete Zustandsraum dem stetigen immer weiter annähert. Dies schränkt die Lernfähigkeit jedoch in der Hinsicht ein, dass der Agent über mehr Episoden trainiert werden muss, um die relevanten Zustände (bzw. Banditen) zu optimieren.