Reinforcement Learning

Joern Ploennigs · AI4SC

> The way positive reinforcement is carried out is more important than the amount.
>
> — Burrhus Frederic Skinner

Reinforcement Learning (RL) ist ein Bereich des maschinellen Lernens, bei dem ein Agent lernt, in einer Umgebung durch Interaktionen zu handeln, um eine maximale Belohnung zu erzielen. Der Agent nimmt Aktionen basierend auf seinem aktuellen Zustand vor und erhält Belohnungen oder Bestrafungen, die er verwendet, um seine Strategie zu verbessern.

Während Supervised Learning aus Beispielen lernt („Was ist das?“), lernt Reinforcement Learning durch Versuch und Irrtum („Was soll ich tun?“), indem es für gute Entscheidungen belohnt wird. Es benötigt keine gelabelten Daten, sondern eine Umgebung und Rückmeldung (Reward).

RL eignet sich besonders für Aufgaben, bei denen ein Agent eigenständig durch Interaktion mit einer Umgebung lernen muss, optimale Entscheidungen zu treffen. Typische Einsatzgebiete sind dynamische Steuerungs- und Optimierungsprobleme, bei denen klassische regelbasierte Ansätze an ihre Grenzen stoßen. RL wird häufig verwendet, wenn die Umgebung komplex, unsicher oder sich verändernd ist und keine gelabelten Trainingsdaten vorliegen.

Beispiele im Bauwesen sind: Ein Heiz-/Kühlsystem in Gebäuden lernt, abhängig von Wetter und Nutzung, effizient zu regeln. Beim 3D-Druck von Betonelementen kann RL den Materialeinsatz optimieren. Ein autonomer Bagger lernt, wie er Erdbewegungen möglichst ressourcenschonend und schnell ausführt.

Das macht auf der einen Seite RL vom Konzept her einfach zu verstehen und zu implementieren. Die Spezifikation der Belohnungsstrategie ist allerdings meist schwierig, da dies Verständnis des Problems erfordert. Der Ansatz benötigt keine gelabelte Trainingsdaten, ist also kein Überwachter ML-Ansatz. Er ist auch kein unüberwachter Ansatz, da ja durchaus Wissen in Form der Belohnungsstrategie notwendig ist [1].

> ⚠ Warnung
>
> RL ist in der Praxis oft datenintensiv und empfindlich gegenüber Hyperparametern, Belohnungsdefinitionen und Zufall. Im Vergleich zu Supervised Learning benötigt es häufig deutlich mehr Interaktionen mit der Umgebung und das Training kann instabiler sein.

Deshalb wird RL insbesondere bei komplexen Problemen verwendet wo unzureichend Trainingsdaten verfügbar sind, aber sich die Umwelt gut modellieren läst. Deshalb wird es häufig in der Robotik verwendet, aber wird auch immer beliebter in anderen Gebieten wie dem Training großer Sprachmodelle.

Grundlegende Konzepte

Reinforcement Learning basiert auf einer iterativen Lernstrategie, bei der die Qualität der Lösung durch positives oder negatives Feedback bewertet wird. Das RL-Modell wird hierbei als Agent gesehen, der mit der Umgebung interagiert. Die Begriffe in diesem Kontext sind:

- **Agent**: Der Lernende oder Entscheidungsträger, der in der Umgebung agiert.
- **Umgebung (Environment)**: Alles, mit dem der Agent interagiert.
- **Aktion (Action)**: Eine Entscheidung oder Bewegung, die der Agent treffen kann.
- **Belohnung (Reward)**: Rückmeldung aus der Umgebung, die angibt, wie gut eine Aktion im gegebenen Zustand war.
- **Wertfunktion (Value Function)**: Eine Funktion, die angibt, wie gut ein bestimmter Zustand oder eine Aktion ist.
- **Strategie (Policy)**: Eine Strategie, die der Agent verwendet, um Aktionen zu wählen basierend auf dem Zustand und der Wertfunktion.
- **Zustand (State)**: Eine Repräsentation der aktuellen Situation der Umgebung und ggf. der Strategie.

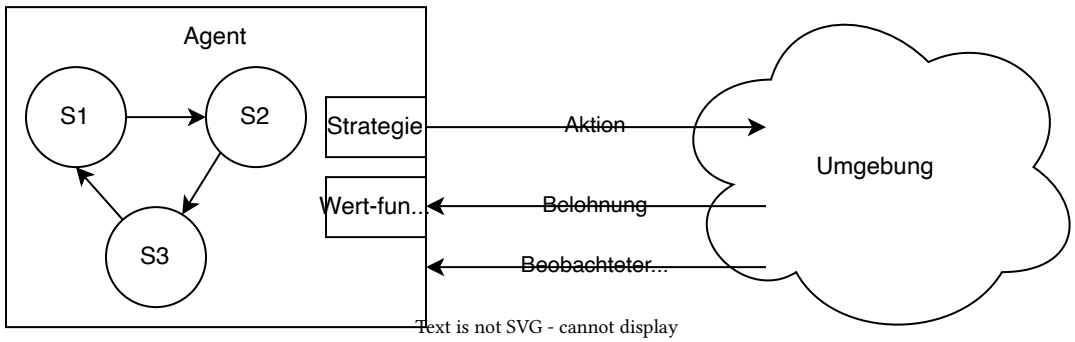


Abbildung 1: Komponenten und Begriffe beim Reinforcement Learning

Zur Modellierung der Umgebung und aktuellen Strategie verwendet man im RL oft ein *Markov Decision Process (MDP)*. Das ist ein diskretes Zustandsmodell, bei dem das System sich immer nur in einem einzigen Zustand befinden kann, der die Umgebung widerspiegelt. Jeder Zustand beschreibt eine bestimmte Bedingung oder Position im Verhalten des Systems. Auf Basis des

aktuellen Zustands und der gewählten Aktion erzeugt die Umgebung den nächsten Zustand sowie eine Belohnung.

Markov-Modelle sind eine sehr beliebte Modelltyp im Maschinellen Lernen. Sie basieren alle auf der *Markov-Bedingung*, dass die Übergangswahrscheinlichkeit von einem Zustand in den anderen nur von dem aktuellen Zustand abhängt und nicht vorhergehenden Zuständen (es gibt also keine Autokorrelation). Man spricht dabei auch von der *Gedächtnislosigkeit*. Dies basiert auf der Idee, dass der Zustand des Systems zu einem bestimmten Zeitpunkt alle Informationen enthält, die notwendig sind, um sein zukünftiges Verhalten vorherzusagen.

Ein MDP besteht aus:

- S: Menge aller möglichen Zustände.
- A: Menge aller möglichen Aktionen.
- $P(s' \mid s, a)$: Übergangswahrscheinlichkeit vom Zustand s zum Zustand s' bei Aktion a.
- $R(s, a)$: Belohnungsfunktion, die die Belohnung angibt, die der Agent erhält, wenn er im Zustand s die Aktion a ausführt.
- γ: Diskontierungsfaktor, der zukünftige Belohnungen abwertet.

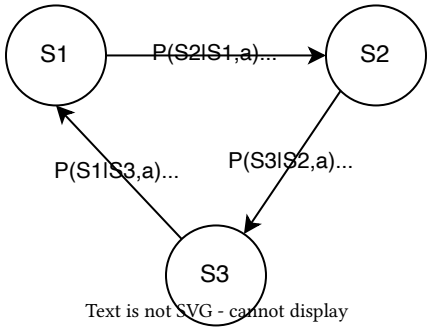


Abbildung 2: Beispiel eines MDP Zustandsdiagrams mit drei Zuständen

Der MDP definiert also die Zustände und die Belohnungsstruktur. Die Übergangswahrscheinlichkeiten $P(s' \mid s, a)$ sind dabei im Reinforcement Learning typischerweise *unbekannt*. Genau das unterscheidet RL von klassischen Verfahren. Der Agent lernt, welche Aktionen zu guten Ergebnissen führen, ausschließlich durch die erhaltenen Belohnungen, ohne das Übergangsmodell zu kennen. Man spricht daher von *modellfreiem* RL. Die entscheidende Frage ist nun, wie man aus dieser reinen Rückmeldung die beste Aktion ableiten kann. Hierbei gibt es das Dilemma, das aufgrund der Markov-Bedingung das Modell zwar einfach ist, aber wir auch gedächtnislos sind, wir also nicht die Historie der Zustände mit in unserer Entscheidung betrachten können. Das ist problematisch, wenn das Ziel nur erreicht werden kann, wenn eine bestimmte Zustandsfolge eintritt, wie in dem Gridworld-Beispiel unten diskutiert [1].

Eine wichtige Gleichung ist hierfür die *Bellman-Gleichung*. Sie beschreibt die Beziehung zwischen dem aktuellen Zustands-Aktionspaar (s, a), der beobachteten Belohnung und den möglichen Nachfolge-Zustands-Aktionspaaren s', a'. Diese Beziehung wird verwendet, um die optimale Wertfunktion zu finden.

Die Bellman-Gleichung löst das Gedächtnisproblem der Markov-Bedingung elegant: Statt die gesamte Zustandshistorie zu speichern, kodiert sie die Zukunft *rekursiv* — der Wert eines Zustands s ergibt sich aus der sofortigen Belohnung plus dem diskontierten Wert des besten erreichbaren Folgezustands. Dieser geht mit $\gamma < 1$ nur teilweise ein. So trägt jede Aktion implizit alle künftigen Konsequenzen in sich, obwohl das Modell nur den aktuellen Zustand kennt. Die *Wertfunktion* $V(s)$ gibt an, wie viel kumulierte Belohnung ein Agent ab Zustand s maximal erwarten kann:

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right)$$

Diese Gleichung besagt, dass der optimale Wert eines Zustands s die maximale erwartete Belohnung ist, die der Agent erhalten kann, wenn er im Zustand s startet, die Aktion a wählt und danach der optimalen Policy folgt. Dadurch berücksichtigen wir bei der Bewertung des Zustandsüberganges durch die Aktion a nicht nur den aktuellen Zustand, sondern auch die durch Aktion a ermöglichten zukünftigen Zustandsübergänge. Damit lösen wir das Dilemma der Gedächtnislosigkeit, das aus der Markov-Bedingung folgt.

Die Wertfunktion $V(s)$ bewertet einen Zustand insgesamt. Die Q-Funktion $Q(s,a)$ bewertet dagegen eine konkrete Aktion in einem Zustand. $Q(s,a)$ ist damit für die Aktionsauswahl oft direkter nutzbar, weil man die beste Aktion über das Maximum der Q-Werte bestimmen kann [2].

Die Bellman-Gleichung wird genutzt, um die optimale Strategie in Form einer *Q-Funktion* $Q(s,a)$ im *Q-Learning* zu erlernen. Das ist ein populärer Off-Policy-Algorithmus, bei dem der Agent eine Q-Funktion $Q(s,a)$ lernt, welche Belohnungen einer Aktion a in einem Zustand s erwartet wird. Off-Policy bedeutet hier, dass das Update den aktuell besten geschätzten nächsten Q-Wert verwendet, auch wenn die tatsächlich ausgeführte Aktion aufgrund von Exploration eine andere war.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \Big(R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \Big)$$

hierbei stellt α die Lernrate dar. Die ist ein sehr wichtiger Parameter, da er bestimmt, wie stark jedes einzelne Update die Q-Werte verändert. Ein niedriger Wert verlangsamt alle Updates und erfordert dadurch längeres Training bis zur Konvergenz. Ein hoher Wert kann dagegen zu instabilen, stark schwankenden Updates führen, sodass das Modell bereits gelernte Werte überschreibt und nicht zuverlässig konvergiert.

Deshalb kombiniert man den Algorithmus man meist mit einer Erkundungsstrategie. Statt immer nur die geschätzte optimale Strategie $Q(s,a)$ zu nehmen, wählt man mit der Wahrscheinlichkeit ϵ zufällige Strategien aus, um somit alternative Strategien zu entdecken. Ein hoher ϵ-Wert sorgt für eine hohe Erkundungsrate (Exploration), während ein niedriger ein verlässlichere Vorhersage erlaubt. Um hier einen Trade-Off zu finden, macht man oft beides und wählt am Anfang des Trainings ein hohes ϵ, um den Lösungsbereich zu erkunden und am Ende des Trainings ein niedriges ϵ um schneller und verlässlicher zu konvergieren.

Beispiel: Gridworld-Umgebung

Als konkretes Lernbeispiel verwenden wir eine *Gridworld*: ein zweidimensionales Raster, auf dem ein Agent von einer Startposition $(0, 0)$ zu einer Zielposition $(3, 3)$ navigieren muss. Der **Zustandsraum** S ist die Menge aller 16 Rasterpositionen (x, y) mit $x, y \in \{0, 1, 2, 3\}$. Der **Aktionsraum** A enthält vier diskrete Aktionen: hoch, runter, links, rechts. Die **Belohnungsfunktion** gibt $+1$ beim Erreichen des Ziels und -0.1 für jeden anderen Schritt, was den Agenten motiviert, den kürzesten Weg zu finden.

Die Umgebung ist in einer Python-Klasse implementiert, die drei zentrale Methoden bereitstellt: `__init__` initialisiert das Raster, `reset` setzt die Episode auf den Startzustand zurück, und `step` führt eine Aktion aus und gibt den neuen Zustand, die Belohnung und ein Done-Flag zurück.

Wir werden eine einfache Gridworld-Umgebung verwenden, um die Grundprinzipien von Reinforcement Learning zu veranschaulichen. In dieser Umgebung versucht ein Agent, in einem Grid von einem Startzustand zu einem Zielzustand zu gelangen und dabei den kürzesten Weg zu finden.

Dies kann zum Beispiel zur Steuerung eines Materialtransports auf einer Baustelle dienen. Ein Agent (z.B. ein Bagger) befindet sich auf einer vereinfachten Baustelle und muss Material von einer Quelle zu einem Zielbereich bringen. Wir simulieren dieses Szenario in einer Gitterwelt mit folgenden Regeln:

- Der Agent kann sich **hoch, runter, links, rechts** bewegen.
- Er erhält **+1 Punkt**, wenn er das Ziel (Z) erreicht.
- Für jeden Schritt wird er mit **-0.1 Punkten** bestraft.
- Die Umgebung ist sehr einfach und zufallsbasiert.

Wir erstellen uns als erstes eine Klasse, welche die Umgebung repräsentiert und die Zustände enthält. Die Umgebung initialisiert zuerst unser Raster (Grid) mit der Start- und Endposition. Dieses Raster definiert unseren Zustandsraum, da der Agent, wenn er sich fort bewegt sich in jeder dieser Zellen im Raster aufhalten kann.

Damit wir Experimente beim Lernen wiederholen können gibt es eine `reset`-Funktion. Mit der `step`-Funktion kann der Agent sich fortbewegen. Wir übergeben der Funktion als `action` die Richtung, in der wir uns bewegen wollen. Dadurch wechselt sich die aktuelle Position, also der aktuelle Zustand in unserem Zustandsmodell.

```python
import numpy as np # Import von NumPy

# Definition der Gridworld-Umgebung
class Gridworld:
    def __init__(self, size, start, goal):
        self.size = size
        self.start = start
        self.goal = goal
        self.state = start
        self.actions = ['up', 'down', 'left', 'right']
        self.grid=np.zeros((self.size,self.size))
```

```
        self.grid[start]=1
        self.grid[goal]=-1

    def reset(self):
        self.state = self.start
        return self.state

    def step(self, action):
        x, y = self.state
        if action == 'up':
            x = max(0, x - 1)
        elif action == 'down':
            x = min(self.size - 1, x + 1)
        elif action == 'left':
            y = max(0, y - 1)
        elif action == 'right':
            y = min(self.size - 1, y + 1)
        self.state = (x, y)
        self.grid[self.state]+=1
        reward = 1 if self.state == self.goal else -0.1
        done = self.state == self.goal
        return self.state, reward, done
```

Erstellen wir uns als Beispiel ein 3x3 Gridworld und wollen uns vom Punkt $(0, 0)$ zum Punkt $(3, 3)$ bewegen.

```
env = Gridworld(size=4, start=(0, 0), goal=(3, 3))
state = env.reset()
print("Startzustand:", state)
env.grid
```

```
Startzustand: (0, 0)
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1.]])
```

Der Zustandsraum ist hierbei ein 2D Grid

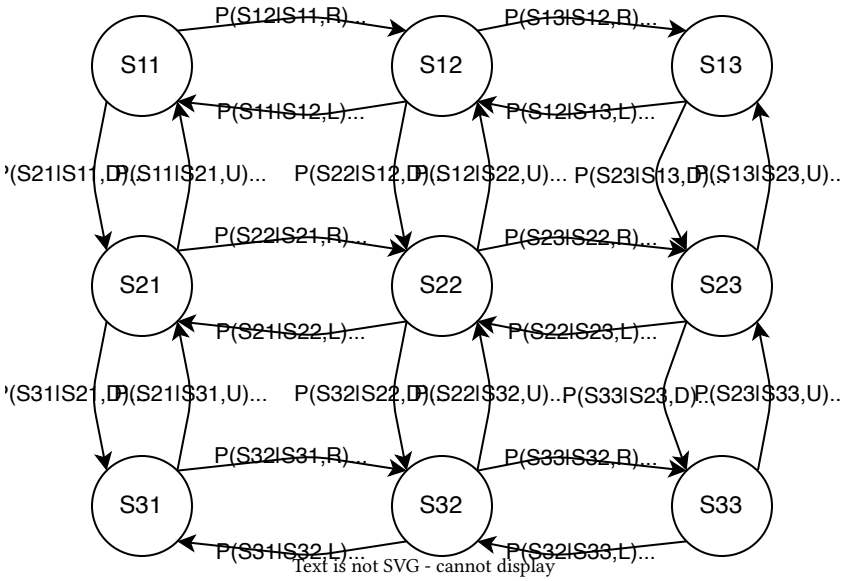


Abbildung 3: Zustandsraum von Gridworld

Bewegen wir uns einmal manuell durch das Grid, so sehen wir wie die Zustände (unsere Position) sich ändern und welche Belohnungen wir erhalten. Wichtig hierbei ist, dass wir eine positive Belohnung erst beim letzten Schritt erhalten, vorher immer nur bestraft werden (z.B. Erschöpfung).

```
# Beispiel für einen Schritt in der Umgebung
steps=['right','down','right','down','right','down']
for step in steps:
    next_state, reward, done = env.step(step)
    print("Nächster Zustand:", next_state, "Belohnung:", reward, "Ziel
erreicht:", done)
```

```
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
Nächster Zustand: (3, 3) Belohnung: 1 Ziel erreicht: True
```

```
import plotly.express as px
px.imshow(env.grid,
color_continuous_scale="Blues").update_xaxes(showticklabels=False, ticks="",
title=None).update_yaxes(showticklabels=False, ticks="", title=None)
```

```
Unable to display output for mime type(s): text/html
```

Random Walk

Als Baseline-Strategie betrachten wir zunächst den *Random Walk*: Der Agent wählt bei jedem Schritt gleichwahrscheinlich eine der vier Aktionen, ohne jede vorherige Erfahrung zu berücksichtigen. Diese Strategie hat kein Gedächtnis und lernt nicht, sondern sie dient als untere Schranke, mit der wir spätere lernende Algorithmen vergleichen. In einer endlichen, verbundenen Gridworld findet der Random Walk das Ziel mit Wahrscheinlichkeit 1, braucht dafür aber im Durchschnitt viele Schritte, da er häufig zurückläuft.

Ein naiver Lösungsansatz ist der so genannte Random Walk (Zufallsweg), bei dem man bei jedem Schritt in eine zufällig gewählte Richtung läuft, was auch bedeutet, dass man ggf. zurück läuft. Untersuchen wir einmal in 300 Experimenten die Erfolgsrate dieser Lösungsstrategie.

```
env = Gridworld(size=4, start=(0, 0), goal=(3, 3))
trials = 300

rewards = []
retries = []
for trial in range(trials):
    env.reset()
    n, r_sum = 0, 0
    done = False
    while not done:
        # Wir wählen eine zufällige Aktion
        action = np.random.choice(env.actions)
        # Führe die Aktion aus und erhalte von der Umgebung den neuen Zustand
und die Belohnung
        new_state, reward, done = env.step(action)
        # Sammel Erfolgsstatistik
        r_sum += reward
        n += 1
    rewards.append(r_sum)
    retries.append(n)
```

Die folgende Animation zeigt, wie sich die besuchten Zustände über die ersten 20 Episoden Schritt für Schritt aufbauen, bevor wir danach jeweils 10 Episode überspringen. Man sieht, dass der Random-Walk-Agent keine Lernkurve hat, sondern immer wieder zurück läuft.

```
Unable to display output for mime type(s): text/html
```

Wenn wir uns die resultierende Häufigkeit der besuchten Rasterpunkte ansehen, so zeigt sich, dass der Großteil der Versuche um dem Starpunkt herum hängen bleibt, aufgrund des Zurücklaufens.

```
px.imshow(env.grid,
color_continuous_scale="Blues").update_xaxes(showticklabels=False, ticks="",
title=None).update_yaxes(showticklabels=False, ticks="", title=None)
```

```
Unable to display output for mime type(s): text/html
```

In allen Experimenten ist unser Agent zum Ziel gekommen, was zeigt, dass diese zufällige Explorations-Strategie durchaus erfolgreich ist. Schauen wir auf die Anzahl der Versuche bis zum Ziel über unsere Experimente, so sehen wir, dass diese gleichbleibend stark variiert. Wir haben also kein Lerneffekt.

```
import plotly.express as px
px.line(retries)
```

```
Unable to display output for mime type(s): text/html
```

Entsprechend niedrig sind auch die Belohnungen, die das Programm pro Experiment sammelt.

```
px.scatter(y=rewards, trendline="lowess")
```

```
Unable to display output for mime type(s): text/html
```

Q-Learning

Q-Learning ist ein modellfreier, Off-Policy-Algorithmus, der die optimale Q-Funktion $Q(s, a)$ direkt aus Erfahrungen lernt [3]. Die Q-Werte werden in einer Tabelle (Q-Matrix) gespeichert und mit der Bellman-Gleichung iterativ aktualisiert:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(R + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$$

Für die Gridworld hat die Q-Matrix die Form $4 \times 4 \times 4$ (Zeile × Spalte × Aktion). Um sowohl zu lernen als auch neue Wege zu entdecken, verwendet der Algorithmus eine ϵ-Greedy-Strategie: Mit Wahrscheinlichkeit ϵ wird eine zufällige Aktion gewählt (Exploration), sonst die Aktion mit dem höchsten Q-Wert (Exploitation).

Im Gegensatz zum Random Walk, der kein Gedächtnis hat und daher nie besser wird, lernt Q-Learning aus jeder Interaktion: Die Q-Werte verbessern sich iterativ, sodass der Agent nach und nach kürzere Wege bevorzugt.

Wir implementieren als nächstes den Q-Learning Ansatz wie er oben beschrieben worden ist. Hierfür initialisieren wir als erstes die Q-Matrix, in welcher wir die erlernten Q-Werte speichern. Da unser Zustandsraum die Größe 4x4 hat und wir 4 Aktionen haben, hat die Q-Matrix eine Größe von 4x4x4.

Der Q-Learning Algorithmus besteht nun zum einen in der Möglichkeit mit der Wahrscheinlichkeit ϵ ein explorativen zufälligen Schritt zu machen oder den vielversprechendsten Schritt unseres aktuellen Zustands s_1, s_2 mit dem maximalen Q-Wert (argmax).

```
env = Gridworld(size=4, start=(0, 0), goal=(3, 3))

# Initialisiere Q-Werte
Q = np.zeros((env.size, env.size, len(env.actions)))

# Hyperparameter
alpha = 0.2  # Lernrate
gamma = 0.9  # Diskontierungsfaktor
epsilon = 0.1  # Epsilon für die Epsilon-Greedy-Strategie

rewardsQ=[]
retriesQ=[]
for trial in range(trials):
    state = env.reset()
    done = False
    n, r_sum=0, 0
    while not done:
        if np.random.uniform(0,1) < epsilon:
            # Wähle Aktion entweder Zufällig aus
            action = np.random.randint(0, len(env.actions))
        else:
            # Wähle Aktion mit maximaler erwarteten Belohnung Q(s)
            action = np.argmax(Q[state[0], state[1], :])
        # Führe die Aktion aus und erhalte von der Umgebung den neuen Zustand
und die Belohnung
        new_state, reward, done = env.step(env.actions[action])
        # Aktualisiere Q-Werte auf Basis der erhaltenen Belohnung
        Q[state[0], state[1], action] += alpha * (reward + gamma *
np.max(Q[new_state[0], new_state[1], :]) - Q[state[0], state[1], action])
        # Aktualisiere den Zustand
        state = new_state
        # Sammel Erfolgsstatistik
        r_sum+=reward
        n += 1
    rewardsQ.append(r_sum)
    retriesQ.append(n)
```

Wenn wir nun die Wiederholungsversuche uns ansehen, dann sehen wir, dass diese schnell auf das Minimum von nur 6 Schritten abfallen. Die verbleibenden Schwankungen entstehen durch den Zufallsanteil ϵ der ϵ-Greedy-Strategie — das ist gewollt, damit der Agent weiterhin gelegentlich andere Wege erkundet und nicht dauerhaft auf einem einzigen Pfad verharrt.

```
px.line(retriesQ)
```

```
Unable to display output for mime type(s): text/html
```

Betrachten wir die Belohnungen, so sehen wir, dass der Ansatz sehr schnell nur noch positive Belohnungen erzieht.

```
px.scatter(y=rewardsQ, trendline="lowess")
```

```
Unable to display output for mime type(s): text/html
```

Die folgende Animation zeigt den Q-Learning-Lernprozess: Jeder Frame zeigt die akkumulierten Besuche nach N Trainingsepisoden sowie die aktuelle Greedy-Strategie als Pfeile. Es lässt sich erkennen, wie sich diese von zunächst zufälligen zu immer gerichteteren Pfeilen entwickelt.

```
Unable to display output for mime type(s): text/html
```

Die Besuchhäufigkeit zeigt, dass der Lernansatz einen der optimalen Wege findet und diesen wiederholt.

```
px.imshow(env.grid,
color_continuous_scale="Blues").update_xaxes(showticklabels=False, ticks="",
title=None).update_yaxes(showticklabels=False, ticks="", title=None)
```

```
Unable to display output for mime type(s): text/html
```

Zeitreihen

RL lässt sich auch auf Zeitreihendaten anwenden, etwa für Handelsentscheidungen. Im Vergleich zur Gridworld ist die Herausforderung hier, dass Aktienkurse *kontinuierlich* sind. Daher wird der Zustandsraum diskretisiert: Die Preisskala wird in 40 gleichmäßige Bins aufgeteilt, und der aktuelle Preis wird auf den nächsten Bin gerundet. Der **Aktionsraum** besteht aus drei Aktionen: kaufen (Long-Position aufbauen), verkaufen (Short-Position eröffnen) und halten. Die **Belohnung** ist der realisierte Gewinn oder Verlust beim Schließen einer Position.

> ⚠ Beispiel
>
> Dies ist ein vereinfachtes Spielzeugbeispiel ohne Transaktionskosten und mit synthetischen Kursdaten — die Ergebnisse sind nicht auf reale Märkte übertragbar.

Mit Reinforcement Learning kann man auch Modelle auf Zeitreihen trainieren. Wir wollen uns zum Beispiel einen Entscheidungsalgorithmus trainieren, welcher lernt, wann er Long-Positionen aufbauen oder Short-Positionen eröffnen soll und wann er sie auflösen soll. Hier ein zufälliger Aktienkurs.

```
# Simulierte Zeitreihe (z.B. Aktienpreise)
np.random.seed(42)
```

```
data = 10 + np.cumsum(np.random.randn(200) + 0.1)

# Daten visualisieren
px.line(data)
```

```
Unable to display output for mime type(s): text/html
```

Wir definieren uns wieder eine Trainingsumgebung. Dieses Mal haben wir allerdings die Schwierigkeit, dass der Wert kontinuierlich sind und nicht diskret, wir also keine natürliche Zustandsraumdarstellung haben. Deshalb müssen wir die kontinuierliche Zeitreihe des Aktienwertes zuerst diskretisieren.

```
class StockTradingEnv:
    def __init__(self, data, num_states=40):
        self.data = data
        self.current_step = 0
        self.state = None
        self.states = np.linspace(min(data), max(data), num_states)
        self.done = False
        self.actions = ['verkaufen', 'halten', 'kaufen']  # 0→-1, 1→0, 2→+1
        self.position = 0  # 1 = long, -1 = short, 0 = neutral
        self.balance = 0
        self.current_price=0
        self.old_price = 0

    def discretize(self):
        price_bin = np.digitize(self.data[self.current_step], self.states) - 1
        return price_bin * 3 + (self.position + 1)  # position: -1→0, 0→1, 1→2

    def reset(self):
        self.current_step = 0
        self.done = False
        self.position = 0
        self.balance = 0
        self.old_price = 0
        self.current_price = 0
        self.state = self.discretize()
        return self.state

    def step(self, action):
        self.current_step += 1
        if self.current_step > len(self.data) - 2:
            self.done = True
        self.current_price = self.data[self.current_step]
        reward = 0
        if action == 2:  # Kaufen (+1)
```

```
            if self.position == 0: # Kaufe Aktie
                self.position = 1
                self.balance -= self.current_price
                self.old_price = self.current_price
            elif self.position == -1: # Verkaufe Short
                reward = 2 * (self.old_price - self.current_price)
                self.position = 0
                self.balance -= self.current_price
        elif action == 0:  # Verkaufen (-1)
            if self.position == 0: # Kaufe Short
                self.position = -1
                self.balance += self.current_price
                self.old_price = self.current_price
            elif self.position == 1: # Verkaufe Aktie
                reward = 2 * (self.current_price - self.old_price)
                self.position = 0
                self.balance += self.current_price
        elif action == 1:  # Halten (0)
            pass
        self.state = self.discretize()
        return self.state, reward, self.done
```

Als Beispiel kaufen wir Aktien, halten sie für 6 Züge und verkaufen sie. Dann kaufen wir Shorts, um auf fallende Aktien zu wetten, halten sie wieder und verkaufen sie.

```
num_states=40

# Beispiel für die Erstellung und Verwendung der StockTradingEnv-Umgebung
env = StockTradingEnv(data, num_states)
state = env.reset()
print("Startzustand:", state)

# Beispiel für einen Schritt in der Umgebung
for action in [2,1,1,1,1,1,0,0,1,1,1,1,1,2]:#  2 = Kaufen, 1 = Halten, 0 =
Verkaufen
    next_state, reward, done = env.step(action)
    print("Nächster Zustand:", next_state, "Belohnung:", reward, "Ziel
erreicht:", done)
```

```
Startzustand: 40
Nächster Zustand: 41 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 44 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 53 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 53 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 50 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 59 Belohnung: 0 Ziel erreicht: False
```

```
Nächster Zustand: 64 Belohnung: 9.298151214993005 Ziel erreicht: False
Nächster Zustand: 60 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 66 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 63 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 60 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 63 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 54 Belohnung: 0 Ziel erreicht: False
Nächster Zustand: 46 Belohnung: 6.365646416803102 Ziel erreicht: False
```

Als nächstes implementieren wir wieder den Q-Learning Algorithmus. Der ist fast identisch zu dem Gridworld-Beispiel. Unterschiede gibt es nur, da wir nur eine zweidimensionale Q-Matrix haben, statt einer dreidimensionalen, da der Zustandsraum weniger Dimensionen hat.

```
import time

# Beispiel für die Erstellung und Verwendung der StockTradingEnv-Umgebung
tic=time.time()
env = StockTradingEnv(data, num_states)

# Initialisiere Q-Werte
Q = np.zeros((num_states * 3, 3))  # Zustände: price_bin × 3 Positionen

# Hyperparameter
alpha = 0.1  # Lernrate
gamma = 0.9  # Diskontierungsfaktor
epsilon = 0.1  # Epsilon für die Epsilon-Greedy-Strategie

# Training des Q-Learning-Agenten
rewards = []
for episode in range(2000):
    state = env.reset()
    done = False
    total_reward = 0

    while not done:
        if np.random.random() < epsilon:
            # Wähle Aktion entweder Zufällig aus
            action = np.random.randint(0, len(env.actions))
        else:
            # Wähle Aktion mit maximaler erwarteten Belohnung Q(s)
            action = np.argmax(Q[env.state])
        # Führe die Aktion aus und erhalte von der Umgebung den neuen Zustand
und die Belohnung
        new_state, reward, done = env.step(action)
        # Aktualisiere Q-Werte auf Basis der erhaltenen Belohnung
        Q[state, action] += alpha * (reward + gamma * np.max(Q[new_state]) -
Q[state, action])
```

```
        state = new_state
        total_reward += reward
    rewards.append(total_reward)
print(f"Execution Time: {time.time()-tic}")
```

```
Execution Time: 1.6178967952728271
```

```
# Evaluierung des Q-Learning Agents
state = env.reset()
done = False
total_reward = 0
actions = []
portfolio = []
while not done:
    action = np.argmax(Q[env.state])  # reines Greedy, kein Epsilon
    state, reward, done = env.step(action)
    actions.append(action)
    total_reward += reward
    # Kassenstand + Marktwert der offenen Position
    portfolio.append(env.balance + env.position * env.current_price)
print("Gesamtbelohnung:", total_reward)
```

Auch hier zeigt sich, dass der Q-Learning Algorithmus auf diesem synthetischen Datensatz lernt, Kauf- und Verkaufssignale zuzuordnen. Zu beachten ist, dass dies ein vereinfachtes Spielzeugbeispiel ist: Die Kursdaten sind zufällig generiert, es gibt keine Transaktionskosten, und die Ergebnisse lassen sich nicht auf reale Märkte übertragen.

```
px.scatter(y=rewards, trendline="lowess")
```

```
Unable to display output for mime type(s): text/html
```

Hierbei macht der Algorithmus gar nicht so viele Transaktionen, wie er könnte, um das Ergebnis zu maximieren.

```
from plotly.subplots import make_subplots
import plotly.graph_objects as go

fig = make_subplots(rows=2, cols=1, shared_xaxes=True,
                    subplot_titles=("Portfoliowert", "Aktionen (1=Kaufen,
0=Halten, -1=Verkaufen)"))
fig.add_trace(go.Scatter(y=portfolio, name="Portfolio"), row=1, col=1)
fig.add_trace(go.Scatter(y=[a-1 for a in actions], line_shape="hvh",
name="Aktion"), row=2, col=1)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

RL Frameworks

Der tabellarische Q-Learning-Ansatz, den wir bisher verwendet haben, skaliert gut für kleine Probleme wie das 4×4-Gridworld. Seine grundlegende Einschränkung ist jedoch, dass die Q-Tabelle für jeden möglichen Zustand einen Eintrag benötigt. Bei realen Problemen — etwa Bilddaten aus einer Kamera oder hochdimensionalen Sensorwerten — wächst der Zustandsraum so schnell, dass eine vollständige Tabelle nicht mehr in den Speicher passt und auch nicht ausreichend gut abgedeckt werden kann. Die Lösung ist, die Q-Funktion nicht als Tabelle, sondern als neuronales Netz zu approximieren (Deep Q-Network, DQN) [4]. Diese Erweiterung auf neuronale Funktionsapproximatoren ist der Kern des modernen Deep Reinforcement Learning und wird von spezialisierten Bibliotheken wie `Gymnasium` und Ray RLlib unterstützt.

RL gehört nicht zu den traditionellen ML-Verfahren und ist deshalb auch nicht in SciKit-Learn oder Statsmodels enthalten. Das liegt unter anderem daran, dass man beim RL nicht einfach ein Modell auf einem Datensatz trainiert, wie es von der Standard-API von SciKit-Learn erwartet wird, sondern manuell ein Umgebungsmodell erstellen muss.

Mit der wachsenden Popularität von RL in den letzten Jahren haben sich aber spezielle Bibliotheken für RL entwickelt, die dies vereinfachen und das Lösen komplexer Probleme vereinfachen. Eine Bibliothek dafür ist `Gymnasium` oder kurz `Gym` von OpenAI. Die `Gym`-Bibliothek ist eine Open-Source-Bibliothek, die speziell für die Entwicklung und das Testen von RL-Algorithmen entwickelt wurde. Sie bietet eine standardisierte API und eine Vielzahl von vordefinierten Umgebungen, die es ermöglichen RL-Algorithmen effizient zu implementieren und zu evaluieren. Hierbei spielt insbesondere die Definition der Umgebung eine wichtige Rolle.

Das Standard-Interface in `Gym` für eine Umgebung für den Börsenfall ist mehr oder weniger identisch mit der Klasse, die wir oben definiert haben.

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np
import pandas as pd

class StockTradingEnvGym(gym.Env):
    def __init__(self, data, num_states=40):
        super(StockTradingEnvGym, self).__init__()
        self.data = data.copy()
        self.num_states = num_states
        self.bins = np.linspace(min(data), max(data), num_states)
        self.current_step = 0
        self.balance = 0  # Startguthaben
        self.position = 0  # 0: neutral, 1: long, -1: short
        self.old_price = 0
        self.current_price = self.data[self.current_step]
```

```
        # Actions: 2 = Kaufen (+1), 1 = Halten (0), 0 = Verkaufen (-1)
        self.action_space = spaces.Discrete(3)
        # Observation space: [current price, position, balance]
        self.observation_space = spaces.Box(
            low=np.array([-np.inf, -1, -np.inf]),
            high=np.array([np.inf, 1, np.inf]),
            dtype=np.float32
        )

    def discretize(self):
        return np.digitize(self.current_price, self.bins) - 1

    def reset(self, seed=0, options=None):
        self.current_step = 0
        self.balance = 0
        self.position = 0
        self.old_price = 0
        self.current_price = self.data[self.current_step]
        return self._get_obs(), {}

    def _get_obs(self):
        return np.array([self.current_price, self.position, self.balance])

    def step(self, action):
        self.current_step += 1
        self.current_price = self.data[self.current_step]
        reward = 0
        if action == 2:  # Kaufen (+1)
            if self.position == 0: # Kaufe Aktie
                self.position = 1
                self.balance -= self.current_price
                self.old_price = self.current_price
            elif self.position == -1: # Verkaufe Short
                reward = 2 * (self.old_price - self.current_price)
                self.position = 0
                self.balance -= self.current_price
        elif action == 0:  # Verkaufen (-1)
            if self.position == 0: # Kaufe Short
                self.position = -1
                self.balance += self.current_price
                self.old_price = self.current_price
            elif self.position == 1: # Verkaufe Aktie
                reward = 2 * (self.current_price - self.old_price)
                self.position = 0
                self.balance += self.current_price
        elif action == 1:  # Halten (0)
            reward = 0
        done = self.current_step >= len(self.data) - 1
```

```
        return self._get_obs(), reward, done, False, {}

    def render(self, mode='human'):
        print(f'Step: {self.current_step}, Price: {self.current_price},
Position: {self.position}, Balance: {self.balance}')
```

Ein Aspekt warum RL in den letzten Jahren so beliebt geworden ist, ist dass sich der Lernvorgang gut parallelisieren lässt. Da wir zum Lernen viele Experimente machen müssen, können wir diese natürlich auch parallel ausführen und dadurch gut in einem Rechenzentrum in der Cloud oder auf Grafikkarten mit ihren tausenden kleinen Prozessoren verteilen.

Eine Bibliothek, die hierbei viel genutzt wird, ist *Ray* welche auch gerne zur Parallelisierung von ML-Aufgaben genutzt wird, da die Bibliothek das Verteilen der Lernaufgaben im Cluster und das Sammeln der Ergebnisse übernimmt.

Proximale Policy Optimization (PPO) ist ein Policy-Gradient-Verfahren, das direkt eine parametrisierte Policy $\pi_\theta(a \mid s)$ optimiert, statt eine Q-Tabelle zu lernen [5]. Die zentrale Idee ist ein Clip-Mechanismus: Das Verhältnis $r_t(\theta) = \pi_\theta / \pi_{\theta_{\text{alt}}}$ wird auf das Intervall $[1-\varepsilon, 1+\varepsilon]$ begrenzt, um zu große Policy-Updates zu verhindern und so das Training zu stabilisieren. PPO ist ein *On-Policy*-Verfahren — Trainingsdaten werden von der aktuellen Policy erzeugt und nach dem Update verworfen.

Mit Ray RLlib wird das Training durch `.env_runners(num_env_runners=N)` parallelisiert: Mehrere Umgebungsinstanzen laufen gleichzeitig und sammeln Erfahrungen, die dann gebündelt für das Policy-Update genutzt werden. Die wichtigsten Konfigurationsparameter sind die Netzwerkarchitektur (`fcnet_hiddens`), der Clip-Parameter `clip_param`, der Diskontierungsfaktor `gamma` und die Anzahl der SGD-Iterationen pro Update (`num_sgd_iter`).

Wir nutzen hier den *Proximale Policy Optimization (PPO)* Algorithmus. Er ist ein iteratives Verfahren, welches zur Optimierung von Richtlinien in Agenten verwendet wird und instabile Updates der Policy vermeidet und ermöglicht so die effiziente Handhabung komplexer Aufgaben mit kontinuierlichen Aktionsräumen. Dies wird durch die Einführung eines Clip-Parameters ε erreicht, der die maximale Änderung der Policy-Parameter begrenzt. PPO ist ein On-Policy-Verfahren, weil die Policy mit Daten aktualisiert wird, die von der aktuellen Policy selbst erzeugt wurden.

```
from ray.rllib.algorithms.ppo import PPOConfig
from ray.tune.registry import register_env

tic=time.time()
def env_creator(env_config):
    return StockTradingEnvGym(data)  # return an env instance

register_env("StockTradingEnvGym", env_creator)

if not ray.is_initialized():
```

```
    ray.init(ignore_reinit_error=True, log_to_driver=False)

config = (
    PPOConfig()
    .api_stack(
        enable_rl_module_and_learner=False,
        enable_env_runner_and_connector_v2=False
    )
    .environment("StockTradingEnvGym")
    .env_runners(num_env_runners=2)
    .framework("torch")
    .training()
)

algo = config.build_algo()  # 2. build the algorithm,

for _ in range(40):
    algo.train()  # 3. train it,

#algo.evaluate()  # 4. and evaluate it.
print(f"Execution Time: {time.time()-tic}")
```

```
2026-06-30 18:34:19,372 INFO worker.py:2013 -- Started a local Ray instance.
/Users/jplonnigs/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/
python3.12/site-packages/ray/rllib/algorithms/algorithm.py:527:
RayDeprecationWarning: This API is deprecated and may be removed in future Ray
releases. You could suppress this warning by setting env variable
PYTHONWARNINGS="ignore::DeprecationWarning"
`UnifiedLogger` will be removed in Ray 2.7.
  return UnifiedLogger(config, logdir, loggers=None)
/Users/jplonnigs/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/
python3.12/site-packages/ray/tune/logger/unified.py:53: RayDeprecationWarning:
This API is deprecated and may be removed in future Ray releases. You could
suppress this warning by setting env variable
PYTHONWARNINGS="ignore::DeprecationWarning"
The `JsonLogger interface is deprecated in favor of the
`ray.tune.json.JsonLoggerCallback` interface and will be removed in Ray 2.7.
  self._loggers.append(cls(self.config, self.logdir, self.trial))
/Users/jplonnigs/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/
python3.12/site-packages/ray/tune/logger/unified.py:53: RayDeprecationWarning:
This API is deprecated and may be removed in future Ray releases. You could
suppress this warning by setting env variable
PYTHONWARNINGS="ignore::DeprecationWarning"
The `CSVLogger interface is deprecated in favor of the
`ray.tune.csv.CSVLoggerCallback` interface and will be removed in Ray 2.7.
  self._loggers.append(cls(self.config, self.logdir, self.trial))
/Users/jplonnigs/Documents/code/Lehre/IntroductionMachineLearning/.venv/lib/
```

```
python3.12/site-packages/ray/tune/logger/unified.py:53: RayDeprecationWarning:
This API is deprecated and may be removed in future Ray releases. You could
suppress this warning by setting env variable
PYTHONWARNINGS="ignore::DeprecationWarning"
The `TBXLogger interface is deprecated in favor of the
`ray.tune.tensorboardx.TBXLoggerCallback` interface and will be removed in Ray
2.7.
  self._loggers.append(cls(self.config, self.logdir, self.trial))
[2026-06-30 18:34:20,584 E 23174 77412954] core_worker.cc:2232: Actor with
class name: 'RolloutWorker' and ID: 'b5667cbfd32c387a12cbf7f701000000' has
constructor arguments in the object store and max_restarts > 0. If the
arguments in the object store go out of scope or are lost, the actor restart
will fail. See https://github.com/ray-project/ray/issues/53727 for more
details.
2026-06-30 18:34:26,526 WARNING train_ops.py:114 -- DeprecationWarning:
`ray.rllib.execution.train_ops.multi_gpu_train_one_step` has been deprecated.
This will raise an error in the future!
```

```
Execution Time: 121.3272500038147
```

```python
# Evaluierung des Agents
env = StockTradingEnvGym(data)
state = env.reset()
done = False
total_reward = 0

actions = []
portfolio = []
while not done:
    action, _, _ = algo.get_policy().compute_single_action(env._get_obs())
    state, reward, done, _, _ = env.step(action)
    total_reward += reward
    actions.append(action)
    # Kassenstand + Marktwert der offenen Position
    portfolio.append(env.balance + env.position * env.current_price)
    env.render()

print("Gesamtbelohnung:", total_reward)
```

```python
fig = make_subplots(rows=2, cols=1, shared_xaxes=True,
                    subplot_titles=("Portfoliowert", "Aktionen (1=Kaufen,
0=Halten, -1=Verkaufen)"))
fig.add_trace(go.Scatter(y=portfolio, name="Portfolio"), row=1, col=1)
fig.add_trace(go.Scatter(y=[a-1 for a in actions], line_shape="hvh",
```

```
name="Aktion"), row=2, col=1)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Robotik

Die bisherigen Beispiele: Gridworld und Aktienhandel verwendeten *diskrete* Aktionsräume: Der Agent wählt aus einer endlichen Liste von Optionen (hoch/runter/links/rechts oder kaufen/verkaufen/halten). In der Robotik hingegen sind Aktionen typischerweise *kontinuierlich*: Gelenkwinkel, Kräfte und Drehmomente nehmen beliebige reelle Werte an. Ein diskreter Q-Table-Ansatz ist dafür ungeeignet, da man den kontinuierlichen Raum nicht vollständig abdecken kann. Stattdessen verwendet man Policy-Gradient-Methoden wie PPO, die direkt eine kontinuierliche Ausgabe als Policy lernen.

DL ist insbesondere in der Robotik ein beliebtes Lernverfahren. Das liegt daran, das zum einen einfache Aufgaben, wie das Greifen von Objekten für Roboter hochkomplex sind und die Regelungen und Steuerungen sehr komplex zu entwickeln sind. Auch wir Menschen brauchen Wochen als Kleinkind um die Grob- und Feinmotorik dafür zu lernen. Trotzdem ist die Aufgabe und die Umgebung eines Roboters einfach zu simulieren. Deshalb setzt man vermehrt auf RL, um solche komplexen Steuerungsprobleme zu erlernen, statt selbstständisch Steuerungen zu entwickeln.

Robot Pusher

Pusher-v5 ist eine Umgebung aus der Gymnasium-Bibliothek, die zur Simulation von Aufgaben im Bereich der Robotersteuerung verwendet wird. In dieser speziellen Umgebung wird ein Roboterarm simuliert, der darauf trainiert wird, ein Objekt zu einer bestimmten Zielposition zu schieben.

Das Ziel des Agenten (Roboterarms) in der Pusher-v5-Umgebung ist es, ein Objekt (in der Regel ein Block) zu einer festgelegten Zielposition zu schieben. Der Agent muss lernen, wie er seine Gelenke bewegen kann, um das Objekt erfolgreich zu schieben.

Der Aktionsraum ist kontinuierlich und repräsentiert die Steuerung des Roboterarms. Typischerweise handelt es sich um einen Vektor von Gelenkbewegungen, die der Agent ausführen kann.

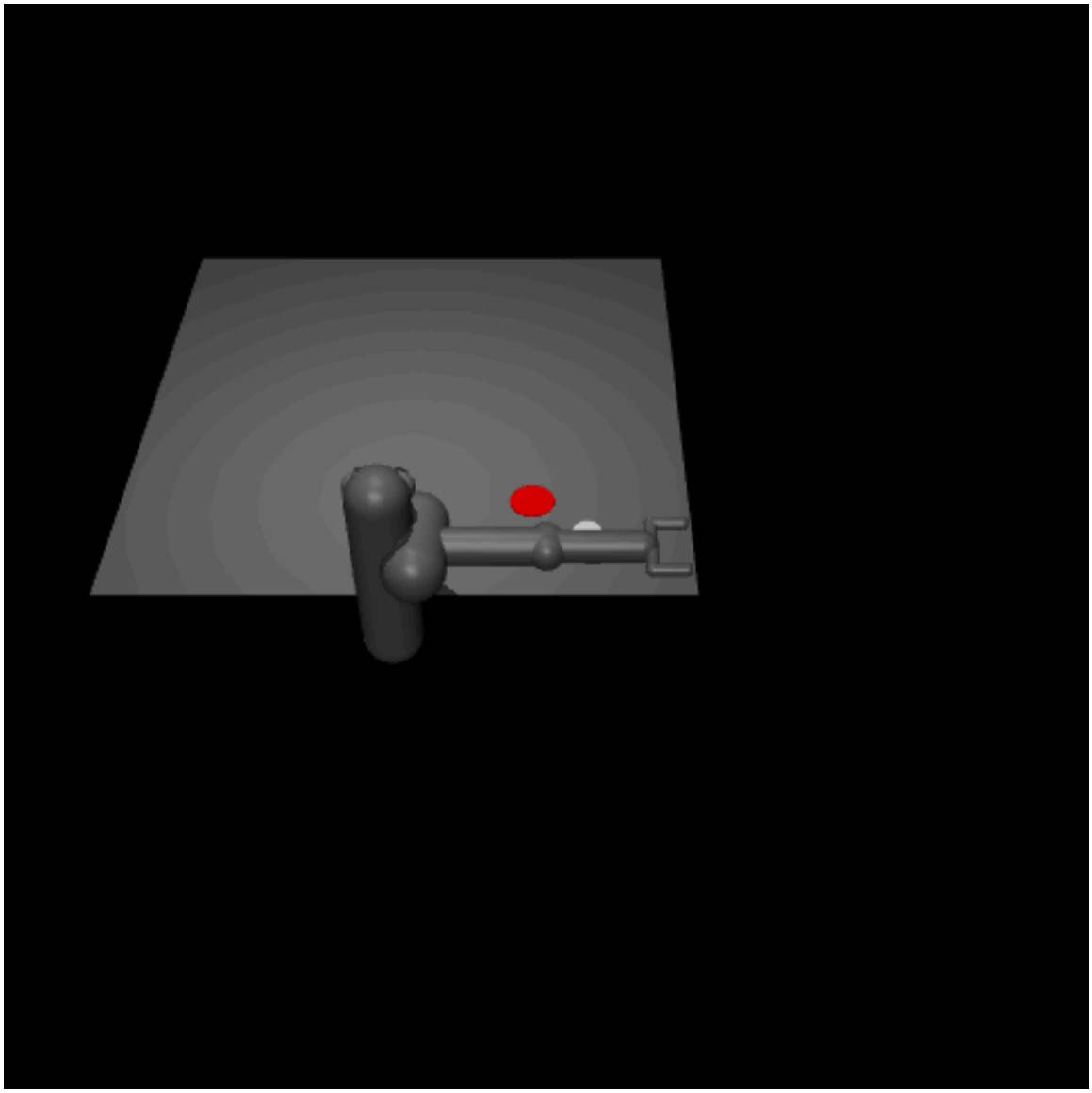
Der Beobachtungsraum umfasst verschiedene Aspekte des Zustands der Umgebung, einschließlich der Position des Endeffektors des Roboterarms; die Position des zu schiebenden Objekts und die Zielposition, zu der das Objekt geschoben werden soll.

Die Belohnung in der Pusher-v5-Umgebung basiert darauf, wie nah das Objekt an der Zielposition ist. Der Agent erhält eine höhere Belohnung, wenn das Objekt näher an der Zielposition ist, und eine geringere Belohnung, wenn es weiter entfernt ist.

Da das Modell in `Gym` enthalten ist, ist die Initialisierung einfach.

```
#| echo: true
#| max-height: 200
if not ray.is_initialized():
    ray.init(ignore_reinit_error=True, log_to_driver=False)

config = (  # 1. Configure the algorithm,
    PPOConfig()
    .api_stack(
        enable_rl_module_and_learner=False,
        enable_env_runner_and_connector_v2=False
    )
    .environment("Pusher-v5")
    .env_runners(num_env_runners=4)
    .framework("torch")
    .training()
)

algo = config.build_algo()  # 2. build the algorithm,

for _ in range(5):
    algo.train()  # 3. train it,

#algo.evaluate()  # 4. and evaluate it.
print(f"Execution Time: {time.time()-tic}")
```

Betrachten wir einmal das Ergebnis einer zufälligen Bewegung, so sehen wir wie der Arm vorerst orientierungslos agiert.



Nach mehreren Trainingsepisoden lernt der Algorithmus allerdings den Arm gut zu benutzen. Hier ein Vergleich unterschiedlicher RL-Ansätze. Zu beobachten ist, dass über die Trainings-Episoden die Modelle durch Zufall die Lösung entdecken und dann wiederholen können. Die finalen Lösungen sind dabei aber auch nach vielen Trainings-Episoden nicht perfekt.

```
#| echo: false
from IPython.display import IFrame
IFrame(width="800", height="413", src="https://www.youtube.com/embed/_QmcH1
TyNwg", title="Gymnasium - Pusher-v4, Test with different algorithms",
        frameborder="0", allow="accelerometer; autoplay; clipboard-write;
encrypted-media; gyroscope; picture-in-picture; web-share",
referrerpolicy="strict-origin-when-cross-origin", allowfullscreen=True)
```

Acrobot

`Acrobot-v1` ist eine klassische Gymnasium-Umgebung, in der ein zweigliedriges Pendelsystem durch geeignete Steuerimpulse in eine aufrechte Position gebracht werden soll.

Das Hauptziel des Agenten in der `Acrobot-v1`-Umgebung ist es, das gekoppelte Pendelsystem so zu steuern, dass das freie Ende eine Zielhöhe erreicht. Der Agent muss lernen, wie die verfügbaren Drehmomente eingesetzt werden, um Schwung aufzubauen und das System kontrolliert aufzurichten.

Der Aktionsraum ist diskret und beschreibt wenige mögliche Steuerimpulse am Gelenk. Der Beobachtungsraum enthält die Winkel- und Geschwindigkeitsinformationen des Systems in kompakter Form, sodass der Agent den aktuellen Zustand des Pendels einschätzen kann.

Die Belohnungsstruktur ist einfach: Der Agent erhält in jedem Zeitschritt eine negative Rückmeldung, bis das Ziel erreicht ist. Dadurch wird er dazu motiviert, die Aufgabe mit möglichst wenigen Schritten zu lösen.

Auch hier ist die Initialisierung einfach.

```
#| echo: false
#| max-height: 200
from ray.rllib.algorithms.ppo import PPOConfig

if not ray.is_initialized():
    ray.init(ignore_reinit_error=True)

config = (  # 1. Configure the algorithm,
    PPOConfig()
    .api_stack(
        enable_rl_module_and_learner=False,
        enable_env_runner_and_connector_v2=False
    )
    .environment('Acrobot-v1')
    .env_runners(num_env_runners=2)
    .framework("torch")
    .training()
    .evaluation(evaluation_num_env_runners=1)
)

algo = config.build_algo()  # 2. build the algorithm,

for _ in range(5):
    algo.train()  # 3. train it,

#algo.evaluate()  # 4. and evaluate it.
print(f"Execution Time: {time.time()-tic}")
```

Das RL-Modell lernt auch hier, ein dynamisches System schrittweise besser zu steuern.

Referenzen

Bibliographie

[1] R. S. Sutton und A. G. Barto, *Reinforcement Learning: An Introduction*, 2. Aufl. Cambridge, MA: MIT Press, 2018.

[2] R. Bellman, „A Markovian Decision Process“, *Journal of Mathematics and Mechanics*, Bd. 6, Nr. 5, S. 679–684, 1957.

[3] C. J. C. H. Watkins und P. Dayan, „Q-learning“, *Machine Learning*, Bd. 8, Nr. 3–4, S. 279–292, 1992.

[4] V. Mnih *u. a.*, „Human-level control through deep reinforcement learning“, *Nature*, Bd. 518, Nr. 7540, S. 529–533, 2015.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, und O. Klimov, „Proximal Policy Optimization Algorithms“, *CoRR*, 2017.