REINFORCEMENT LEARNING

Joern Ploennigs · AI4SC

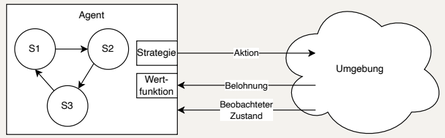
cf. Théodore Géricault "Das Floß der Medusa"

> ⚠ **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.

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.



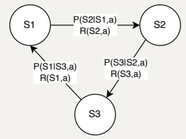
Komponenten und Begriffe beim Reinforcement Learning

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.



Beispiel eines MDP Zustandsdiagrams mit drei Zuständen

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.

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

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 [Bellman, 1957].

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.

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

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

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.

▶ Code

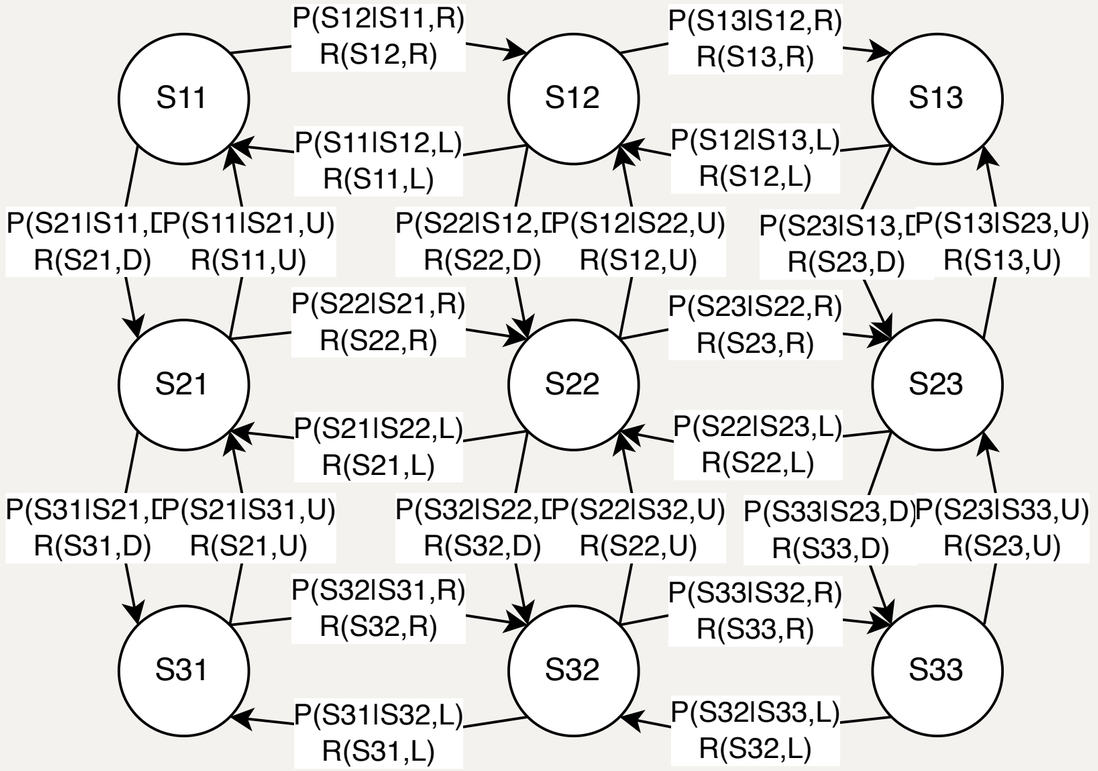
Erstellen wir uns als Beispiel ein 3x3 Gridworld und wollen uns vom Punkt $(0, 0)$ zum Punkt $(3, 3)$ bewegen.

```
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



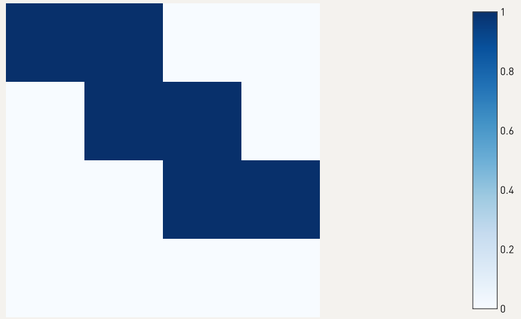
Zustandsraum von Gridworld

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: (0, 1) Belohnung: -0.1 Ziel erreicht: False
Nächster Zustand: (1, 1) Belohnung: -0.1 Ziel erreicht: False
Nächster Zustand: (1, 2) Belohnung: -0.1 Ziel erreicht: False
Nächster Zustand: (2, 2) Belohnung: -0.1 Ziel erreicht: False
Nächster Zustand: (2, 3) Belohnung: -0.1 Ziel erreicht: False
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).updat
```

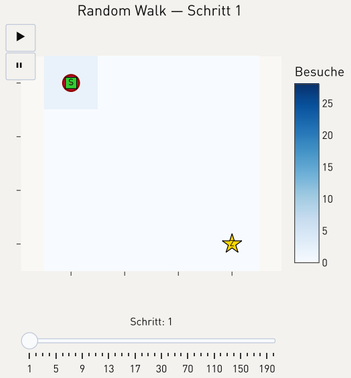


RANDOM WALK

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.

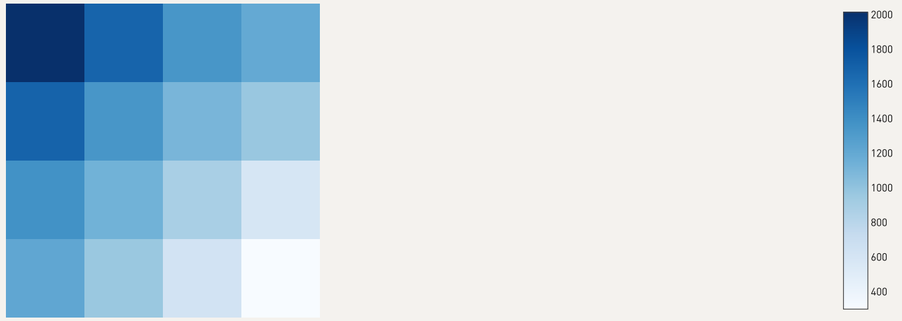
```
env = Gridworld(size=4, start=(0, 0), goal=(3, 3))
trials = 300

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.



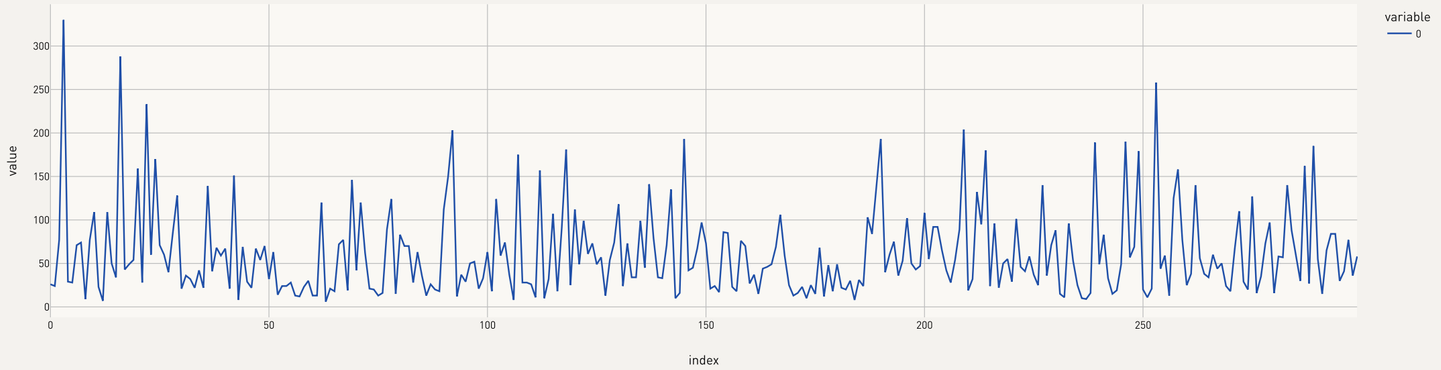
Wenn wir uns die resultierende Häufigkeit der besuchten Rasterpunkte ansehen, so zeigt sich, dass der Großteil der Versuche um dem Starpunkt herum hängen bleibt, aufgrund des Zurücklaufens.

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



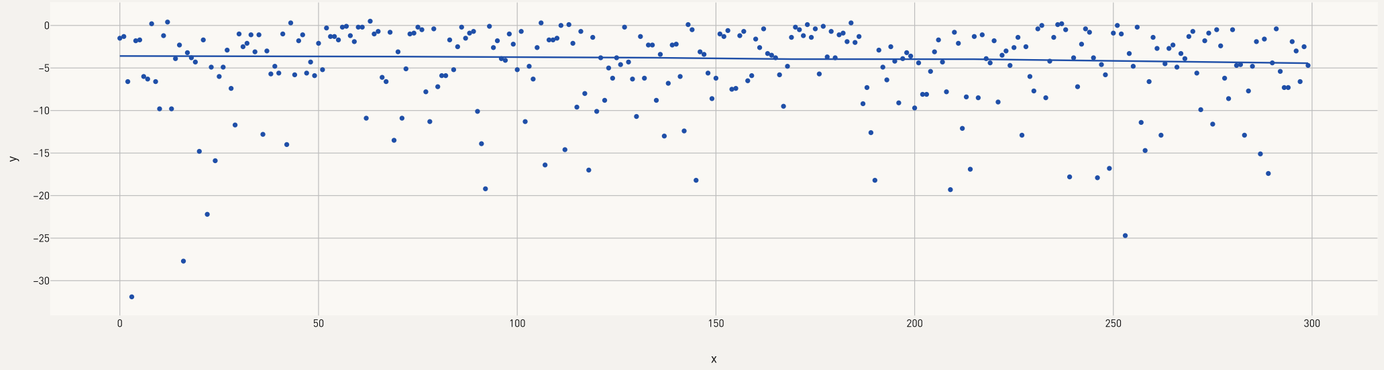
In allen Experimenten ist unser Agent zum Ziel gekommen, was zeigt, dass diese zufällige Explorations-Strategie durchaus erfolgreich ist. Schauen wir auf die Anzahl der Versuche bis zum Ziel über unsere Experimente, so sehen wir, dass diese gleichbleibend stark variiert. Wir haben also kein Lerneffekt.

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



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

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



Q-LEARNING

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.

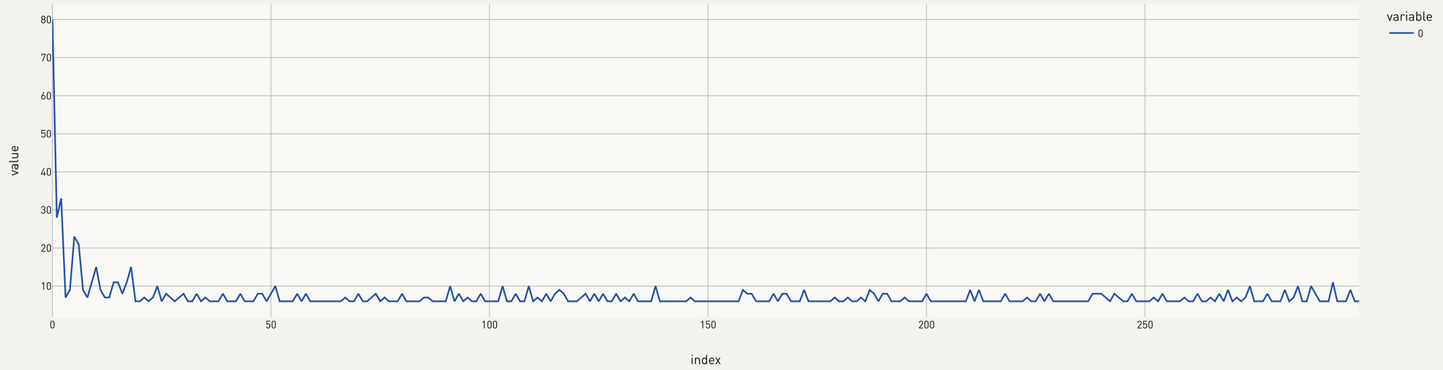
Wir implementieren als nächstes den Q-Learning Ansatz wie er oben beschrieben worden ist. Hierfür initialisieren wir als erstes die Q-Matrix, in welcher wir die erlernten Q-Werte speichern. Da unser Zustandsraum die Größe 4x4 hat und wir 4 Aktionen haben, hat die Q-Matrix eine Größe von 4x4x4.

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).

▶ Code

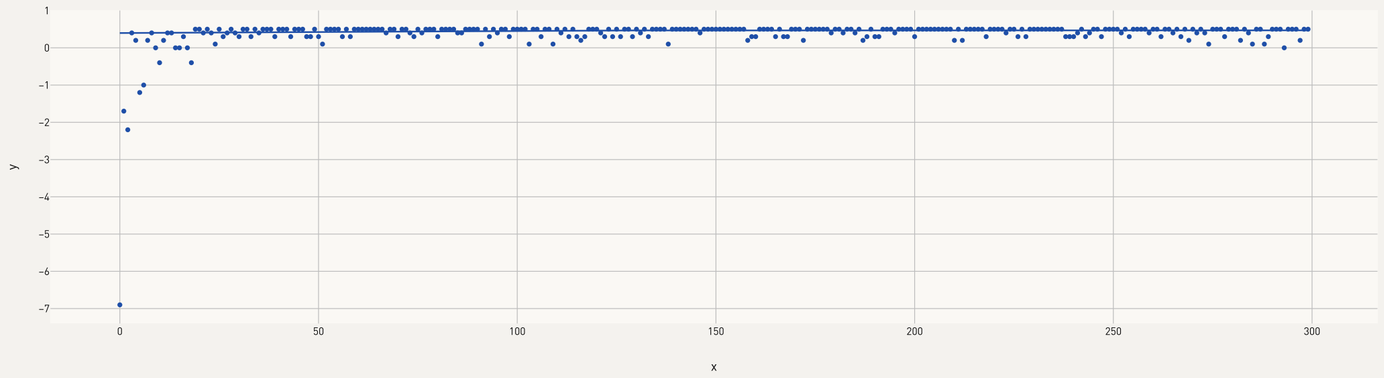
Wenn wir nun die Wiederholungsversuche uns ansehen, dann sehen wir, dass diese schnell auf das Minimum von nur 6 Schritten abfallen. Die verbleibenden Schwankungen entstehen durch den Zufallsanteil ϵ der ϵ-Greedy-Strategie — das ist gewollt, damit der Agent weiterhin gelegentlich andere Wege erkundet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

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

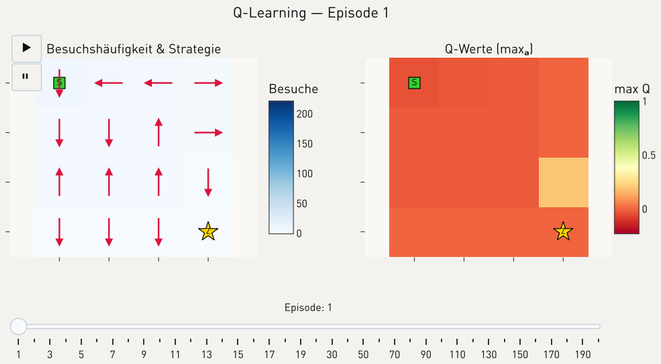


Betrachten wir die Belohnungen, so sehen wir, dass der Ansatz sehr schnell nur noch positive Belohnungen erzieht.

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

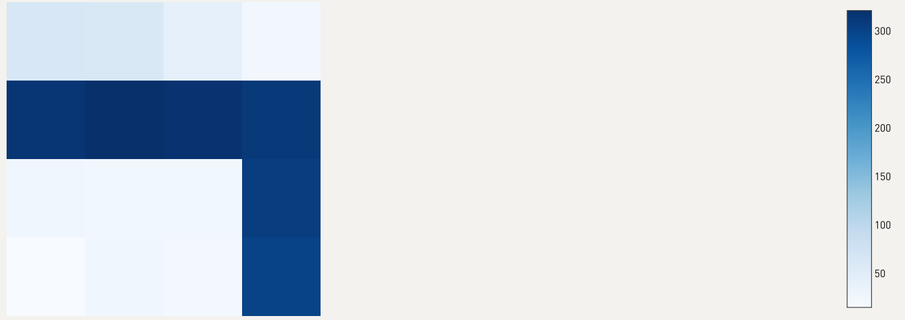


Die folgende Animation zeigt den Q-Learning-Lernprozess: Jeder Frame zeigt die akkumulierten Besuche nach N Trainingsepisoden sowie die aktuelle Greedy-Strategie als Pfeile. Es lässt sich erkennen, wie sich diese von zunächst zufälligen zu immer gerichteteren Pfeilen entwickelt.



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)
```



ZEITREISEN

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)
```

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.

▶ Code

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.

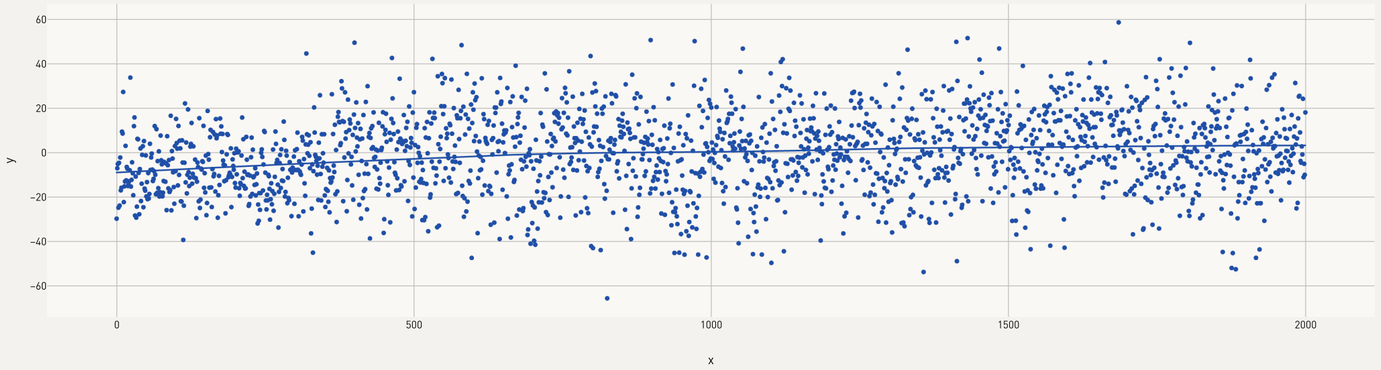
▶ Code

```
Execution Time: 1.6105659008026123
```

```
# 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)
```

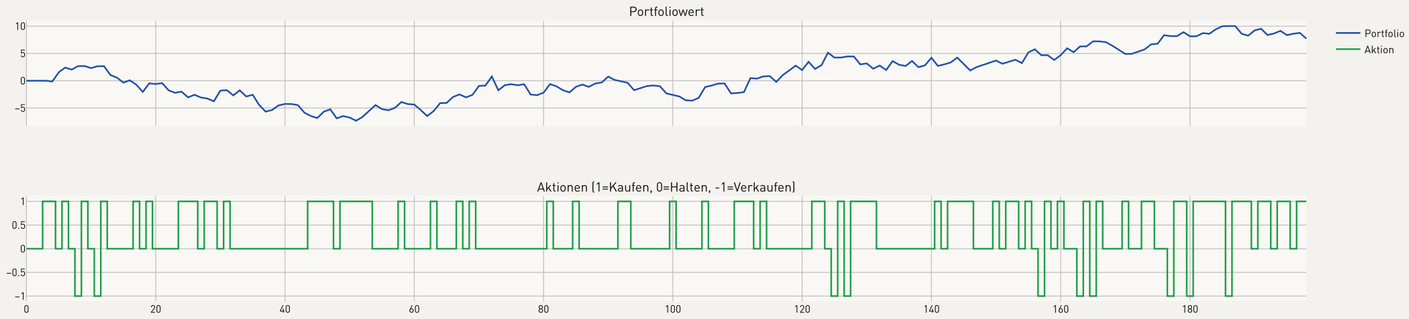
Auch hier zeigt sich, dass der Q-Learning Algorithmus auf diesem synthetischen Datensatz lernt, Kauf- und Verkaufssignale zuzuordnen. Zu beachten ist, dass dies ein vereinfachtes Spielzeugbeispiel ist: Die Kursdaten sind zufällig generiert, es gibt keine Transaktionskosten, und die Ergebnisse lassen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

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



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()
```



RL FRAMEWORKS

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

▶ Code

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.

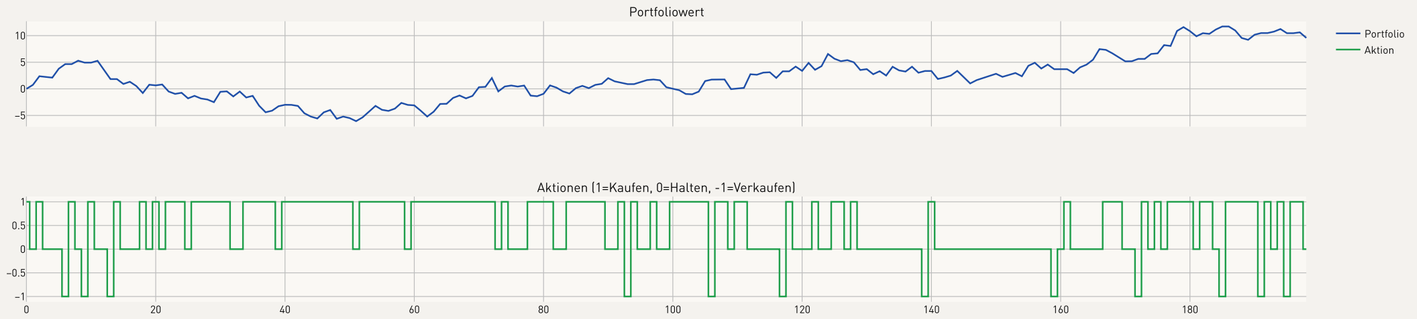
Wir nutzen hier den *Proximale Policy Optimization (PPO)* Algorithmus. Er ist ein iteratives Verfahren, welches zur Optimierung von Richtlinien in Agenten verwendet wird und instabile Updates der Policy vermeidet und ermöglicht so die effiziente Handhabung komplexer Aufgaben mit kontinuierlichen Aktionsräumen. 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.

▶ Code

```
Execution Time: 121.6568009853363
```

```
# 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)
```

```
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()
```



ROBOTIK

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.

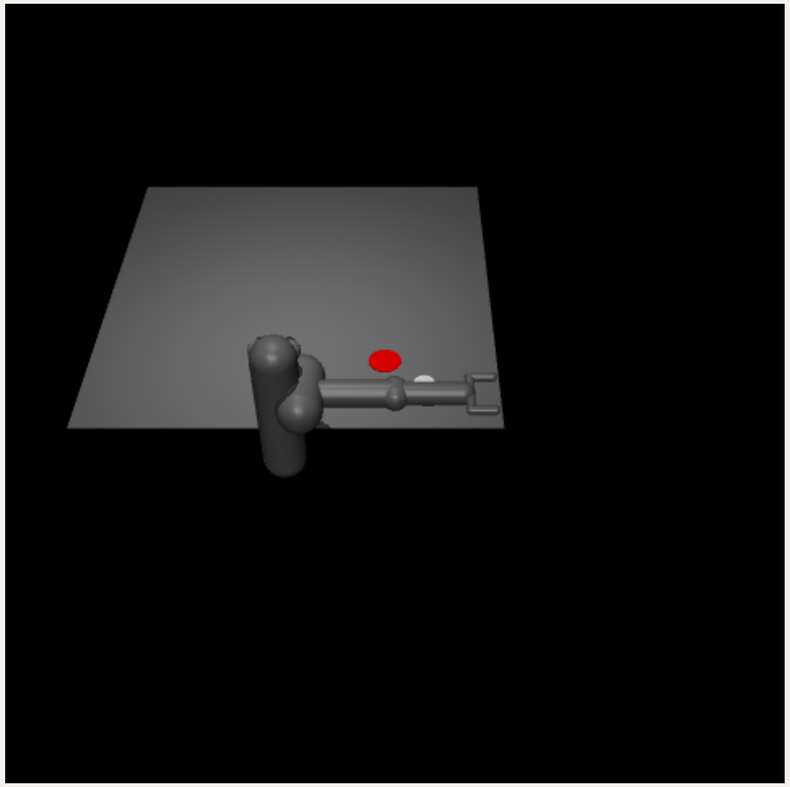
Robot Pusher

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.

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.

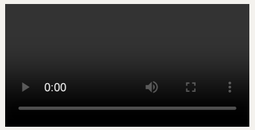
```
#| echo: false
from IPython.display import IFrame
IFrame(width="800", height="413", src="https://www.youtube.com/embed/_QmcH1TyNwg", title="Gymnasium - Pusher-v4, Test with different algorithms",
        frameborder="0", allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture; web-share", referrerpolicy="strict-origin-when-cross-origin", allowfullscreen=True)
```

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

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,
```

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



Referenzen

Bellman, R. (1957). A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5), 679–684.

QUESTIONS