

Angeleitetes Lernprojekt

# Reinforcement Learning für intelligente brownsche Dynamik

Paul A. Monderkamp

15. Mai 2025



Abb. 1: Algorithmus des Reinforcement Learning am lebenden Objekt: Der Hund Lasse gibt sein Pfötchen. Um diese Aktion zu bestärken, bekommt er eine Belohnung (reinforcement). In Zukunft wird er diese Aktion in Erwartung einer erneuten Belohnung häufiger ausführen. Wir belohnen diese Aktion jedes Mal, bis der Zusammenhang zwischen Aktion und Belohnung verinnerlicht ist.

# Inhaltsverzeichnis

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Einleitung</b>                            | <b>3</b>  |
| <b>2</b> | <b>Theorie</b>                               | <b>4</b>  |
| 2.1      | Random Walk . . . . .                        | 4         |
| 2.2      | Q-Learning . . . . .                         | 5         |
| <b>3</b> | <b>Korrektter Umgang mit Daten</b>           | <b>7</b>  |
| <b>4</b> | <b>Korrektter Umgang mit Programmcode</b>    | <b>7</b>  |
| <b>5</b> | <b>Aufgaben</b>                              | <b>8</b>  |
| 5.1      | Mittleres Verschiebungsquadrat . . . . .     | 8         |
| 5.2      | Implementation des Lernalgorithmus . . . . . | 9         |
| 5.3      | Wahl der Hyperparameter . . . . .            | 10        |
| 5.3.1    | Discount factor $\gamma$ . . . . .           | 10        |
| 5.3.2    | Learning rate $\alpha$ . . . . .             | 11        |
| 5.4      | Stochastisches Hindernis . . . . .           | 12        |
| <b>6</b> | <b>Abgabe</b>                                | <b>14</b> |
| <b>7</b> | <b>Zusammenfassung und Ausblick</b>          | <b>14</b> |

# 1 Einleitung

Dieses angeleitete Lernprojekt bietet eine Einführung in die Grundlagen des Reinforcement Learning am Beispiel des  $Q$ -Learning-Algorithmus. Dieser bildet die Grundlage für viele komplexe Machine Learning Algorithmen, die heutzutage in unterschiedlichen Bereichen Anwendung finden. Das Lernziel dieses Projektes besteht darin, die Grundlagen des  $Q$ -Learning zu verstehen, indem die/der Bearbeitende diesen Algorithmus auf verschiedene Probleme anwenden soll. Die Aufgaben, die in dieser Projektarbeit zu bearbeiten sind, sollen die Grundlagen des Reinforcement Learning festigen und bieten gleichzeitig einen Einstieg in objektorientiertes Programmieren in Python 3. Die Einbettung in den Kontext eines eindimensionalen Random Walk soll außerdem physikalische Intuition für die Erforschung von diffusiven Prozessen in der weichen Materie liefern.

In Kapitel 2 werden die erforderlichen Grundlagen zur Bearbeitung dieses Lernprojekts vermittelt. In Kapitel 2.1 werden der Random Walk und seine Analogie zur Diffusion in einer Dimension erläutert. In Kapitel 2.2 wird der Algorithmus erläutert, mit dessen Hilfe innerhalb dieser Umgebung intelligent navigiert werden kann.

Im Folgenden werden verschiedene Bezeichnungen für das Objekt verwendet, dessen Bewegung es in diesem Projekt zu verstehen gilt. In der Physik ist oft von *Teilchen* die Rede, in der kondensierten weichen Materie wird häufig die Bezeichnung *(Mikro-)Schwimmer* für Teilchen verwendet, die brownischer Dynamik unterliegen und über einen Eigenantrieb verfügen. Im Bereich maschineller Intelligenz wird geläufig die Bezeichnung *Agent* für das lernende Individuum verwendet. Wir werden an den entsprechenden Stellen die angemessene Terminologie verwenden, um einen Bezug zur existierenden Literatur herzustellen. Des Weiteren ist von *Lernzeit*, *Lernprozess*, *Trainingszeit*, *Simulation* oder Ähnlichem die Rede. All diese Begriffe beschreiben den selben Vorgang: einen Durchlauf des Machine Learning Programmes, um den Agenten vollständig zu trainieren.

Sollten Sie zu einem beliebigen Zeitpunkt Fragen oder Feedback haben oder sollte es Unklarheiten geben, freue ich mich über Feedback. Meine E-Mail-Adresse finden Sie auf der Homepage der *TP2 - HHU*. Ich hoffe, Sie lernen viel und haben Spaß bei der Bearbeitung.

Paul Monderkamp

## 2 Theorie

### 2.1 Random Walk

Ein beliebtes Modell für einen Random Walk (dt. zufällige Irrfahrt) ist der Random Walk auf dem Zahlenstrahl der ganzen Zahlen. Hierfür wird ein Teilchen auf einer beliebigen Position  $x_0$  des Zahlenstrahls initialisiert (bspw.  $x_0 = 0$ ). Im Folgenden werden in diskreten Zeitabständen zufällig Schritte in entweder positive oder negative  $x$ -Richtung durchgeführt ( $\Delta x = \pm 1$ ). Die zufälligen Schritte können z. B. durch einen Münzwurf festgelegt werden.

Die Wahrscheinlichkeit  $w_N(r)$ , dass das Teilchen von  $N$  Schritten genau  $r$  nach rechts und  $l$  nach links macht, berechnet sich aus der Binomialverteilung

$$w_N(r) = \frac{N!}{r!l!} p^r q^l = \frac{N!}{r!(N-r)!} p^r (1-p)^{N-r} . \quad (1)$$

Hierbei bezeichnet  $p$  die Wahrscheinlichkeit in einem einzelnen Schritt nach rechts zu gehen und  $q = 1 - p$  die Wahrscheinlichkeit nach links zu gehen.

Die Wahrscheinlichkeitsverteilung  $P_N(m)$  für die Position  $m$  nach  $N$  Schritten lässt sich daraus bestimmen, da  $m = -l + r$ , sodass  $r = (m + N)/2$ .  $P_N(m)$  ergibt sich in diesem Fall aus  $w_N((m + N)/2)$ :

$$P_N(m) = \frac{N!}{((m + N)/2)!(N - (m + N)/2)!} p^{\frac{m+N}{2}} (1-p)^{\frac{N-m}{2}} . \quad (2)$$

An dieser Stelle können Sie einen dreidimensionalen Random Walk durch die Universität unternehmen und zu jedem Zeitpunkt  $t$  werden Sie einen Wissenschaftler treffen, der Ihnen einen anderen Weg vorschlägt, aus den obigen Formeln herzuleiten, dass im Übergang zum kontinuierlichen Fall  $m \rightarrow x$  mit  $p = q = 1/2$  eine Gaußverteilung folgt. Sollten Sie Ihren Weg zurück zur *TP2 - Soft Matter* finden, wird man Ihnen verraten, dass der einfachste Weg die Anwendung der Stirling-Formel für große Fakultäten ist, woraus aus Gl. (2)

$$\mathcal{P}(x, t) = \frac{1}{\sqrt{4\pi Dt}} \exp\left(-\frac{x^2}{4Dt}\right) \quad (3)$$

folgt. Weitere Details hierzu sind im Kapitel 2.1.5 im Vorlesungsskript zur Statistischen Mechanik (Stand 28.3.2025). Für die Diffusionskonstante  $D$  gilt hier

$$D := \frac{a^2}{2\tau} , \quad (4)$$

wobei  $\tau$  die Zeitspanne zwischen zwei Schritten und  $a = |\Delta x|$  die Schrittweite im diskreten Fall bezeichnen.

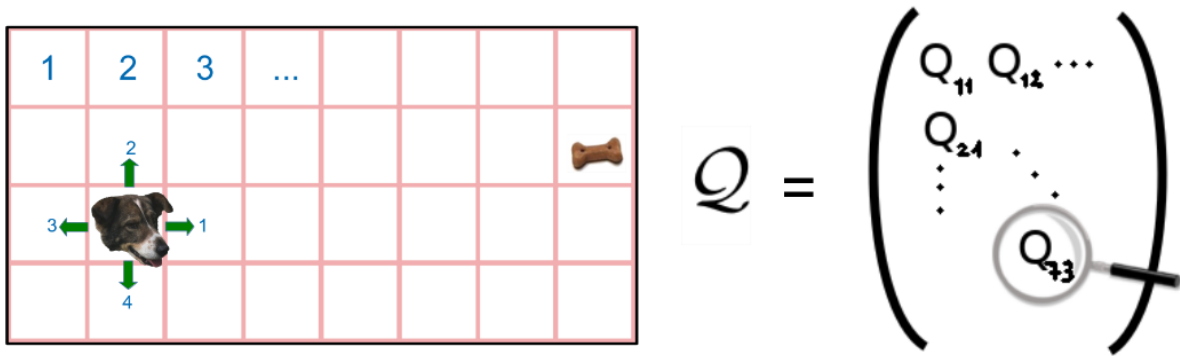


Abb. 2: Schema eines zweidimensionalen Pfad-Finde-Problems. Der Agent (Hund) soll lernen, zum Hundeknochen zu finden. **Links:** Diskretisierung des Raums und Definition der Aktionen. Der Konfigurationsraum wird diskretisiert. Jede Position bekommt einen eindeutigen Index. Jede Aktion bekommt einen eindeutigen Index. **Rechts:**  $Q$ -Matrix, welche die Strategie zur Bewältigung des Problems enthält. Zu jedem Zeitpunkt wird die Aktion ausgeführt, die dem Spaltenindex des Maximums der Zeile entspricht. Diese  $Q$ -Matrix wird im Laufe des Lernprozesses optimiert.

## 2.2 $Q$ -Learning

Es existieren verschiedene große Bereiche innerhalb der *künstlichen Intelligenz*. Darunter fallen unter anderem *Unsupervised Learning* (*unüberwachtes Lernen*), *Supervised Learning* (*überwachtes Lernen*) und *Reinforcement Learning* (*bestärkendes Lernen*). Ersteres dient in den meisten Fällen der Datenverarbeitung und Datensortierung großer Datenmengen. Clusteranalysen wie *k-Means* oder Dimensionsreduktion von hochdimensionalen Datensets werden i. A. zum *Unsupervised Learning* gezählt. *Supervised Learning* bezeichnet den Prozess einem Programm / Agenten die Ausführung einer Aufgabe anhand vorgegebener Beispieldaten beizubringen. So lässt sich bspw. einem Computer das Schach Spielen anhand menschlicher Partien beibringen. Dem *Reinforcement Learning* hingegen liegen keine fertigen Daten zugrunde. Der Agent generiert im Laufe der Trainingszeit anhand einer Strategie selber Daten. Des Weiteren wird dem Agenten eine Möglichkeit gegeben, seine eigene Leistung in der zu lernenden Aufgabe zu quantifizieren. Im Laufe der Simulation wird die Strategie des Agenten entsprechend optimiert. Dies geschieht in der Hoffnung, zum Ende der Lernzeit einen Agenten trainiert zu haben, der die zu bewältigende Aufgabe gemeistert hat.

Als Beispiel betrachten wir ein Navigationsproblem in zwei Dimensionen (siehe Abb. 2, links). Der Hundeagent soll darauf trainiert werden, den Hundeknochen unabhängig von seiner Startposition so schnell wie möglich zu erreichen. Die Position des Hundeknochens bleibt unverändert. Zunächst wird der Raum, in dem der Agent sich aufhalten kann, diskretisiert, sodass eine endliche Anzahl an Zuständen vorliegt. Jeder Zustand bekommt einen eindeutigen Index. Zu jedem Zeitpunkt der Simulation befindet sich der Agent in einem dieser Zustände. Als Nächstes werden Aktionen mit eindeutigen Indizes definiert, die der Agent in jedem Zustand ausführen kann. In diesem Beispiel sind die Aktionen diskrete Schritte zwischen den Zuständen ( $\rightarrow \uparrow \leftarrow \downarrow$ ).

Nun wird eine Matrix definiert, deren Zeilen die Indizes der Zustände und Spalten die Indizes der Aktionen bezeichnen. Die Strategie, welche nun das  $Q$ -Learning auszeichnet, ist  $Q$  gemäß

$$\text{Aktion in Zustand } i = \arg \max_j (Q_{ij}) \quad (5)$$

als Entscheidungsbasis zu verwenden. Das Ziel des Lernalgorithmus besteht nun darin, die Matrix  $Q$  so zu optimieren, dass die Entscheidungen auf Basis von  $Q$  den Agenten auf dem möglichst schnellsten Weg zum Hundeknochen führen. Hierzu führt der Agent die gewünschte Aufgabe (die Suche nach dem Hundeknochen)  $N$ -mal durch. Eine einzelne solche Durchführung wird als eine Epoche oder Episode bezeichnet. Jede Episode beginnt mit der Initialisierung des Agenten

in einem beliebigen Zustand. Der Agent führt Aktionen aus, bis er das Ziel erreicht hat. Mit Wahrscheinlichkeit  $\epsilon$  wählt er eine zufällige Aktion und mit Wahrscheinlichkeit  $1 - \epsilon$  wählt der Agent eine Aktion auf Basis von  $Q$  (nach Gl. (5)).

Nach jeder ausgeführten Aktion wird  $Q$  gemäß

$$Q_{ij}^{\text{neu}} = Q_{ij}^{\text{alt}} + \alpha \left( \mathcal{R} + \gamma \max_k (Q_{i'k}) - Q_{ij}^{\text{alt}} \right) \quad (6)$$

aktualisiert. Hier bezeichnet  $i$  den Zustand vor der zuletzt ausgeführten Aktion und  $j$  den Index der zuletzt ausgeführten Aktion.  $Q_{ij}^{\text{alt}}$  ist der entsprechende Wert in  $Q$  **vor** der Aktualisierung,  $Q_{ij}^{\text{neu}}$  ist der entsprechende Wert **nach** der Aktualisierung. Weitere Parameter sind die Lernrate  $\alpha$  (*learning rate*) und der Diskontierungsfaktor  $\gamma$  (*discount factor*).  $\mathcal{R}$  bezeichnet die Belohnung (*reward*) für das Durchführen der aktuellen Aktion.  $\max_j (Q_{i'j})$  ist der maximale Wert in Zeile  $i'$ , dem Zustand **nach** Ausführung der Aktion. Abhängig vom zugrunde liegenden Problem muss im Allgemeinen  $i'$  nicht von  $i$  verschieden sein. Es ist nämlich auch möglich, dass eine Aktion durchgeführt wird, die den Zustand nicht ändert. Der Term  $\gamma \max_k (Q_{i'k})$  bildet eine Abschätzung im aktuellen Zustand an die mögliche Belohnung in der Zukunft und führt dazu, dass Belohnungen in bestimmten Zuständen auch die Wahl der Aktion anderer Zustände beeinflussen (mehr zu diesen Parametern lernen wir in Aufgabe 5.3).

### 3 Korrekter Umgang mit Daten

Um gute Arbeitsethik mit wissenschaftlichen Daten zu lernen, wollen wir dies zu einem zentralen Bestandteil der Bearbeitung dieses Lernprojektes machen. Sollten Sie im späteren Verlauf Ihrer Arbeit mit größeren Datenmengen arbeiten, die auf viele Dateien aufgeteilt sind und alle den Namen `data.txt` o. Ä. tragen, sind diese Daten für Sie als Wissenschaftler häufig quasi unbenutzbar.

Achten Sie daher darauf, dass die Dateien, die Sie im Laufe der Bearbeitung anlegen, die entsprechenden Dateinamen tragen, sodass sie am Ende entsprechend identifiziert werden können. Achten Sie weiterhin darauf, dass jede Datei deskriptiv benannt ist und das Datum und die Uhrzeit Ihrer Erstellung trägt. Hierfür gibt es auch entsprechende Bibliotheken in Python (wie `datetime`). Leerzeichen sollten in Dateinamen grundsätzlich vermieden werden. Optional können Sie Ihren Namen hinzufügen.

Bsp.: **Aufgabe\_1\_QMATRIX\_Paul\_Monderkamp\_2022-06-10T10\_17\_00.txt**

Der Zeitstempel ist im obigen Beispiel gemäß der **ISO 8601** angegeben.

Achten Sie bei allen Diagrammen auf eine angemessene Schriftgröße und Achsenbeschriftung.

### 4 Korrekter Umgang mit Programmcode

Um Abhängigkeiten von Code zu vermeiden, der unabhängige Aufgaben ausführt, sollte der Code der verschiedenen Aufgaben voneinander getrennt werden. Da die verschiedenen Aufgaben jedoch aufeinander aufbauen, werden Sie aufgefordert, für die Bearbeitung der entsprechenden Aufgaben den Code der letzten Bearbeitung in das entsprechende nächste Verzeichnis zu kopieren, um dort weiterzuarbeiten. Machen Sie Gebrauch vom `pass`-Kommando innerhalb leerer, noch nicht bearbeiteter Funktionen, damit Sie den Code trotz leerer Funktionen ausführen können, um Funktionen zu testen, an denen Sie arbeiten.

Wenn Sie keine Infrastruktur wie *Jupyter-Notebooks* verwenden, welche es Ihnen ermöglicht, Code-Blöcke auszuführen, machen Sie außerdem Gebrauch vom `exit()`-Kommando, um die vorgefertigten Code-Blöcke für spätere Aufgaben nicht auszuführen.

Ich empfehle außerdem ausführlichen Gebrauch von *GitHub* zu machen, um eine Versionskontrolle zu gewährleisten. Ein routinierter Gebrauch von Versionskontrolle wird Ihnen im späteren Verlauf Ihrer Arbeit, z. B. bei größeren Projekten wie Abschlussarbeiten, mindestens Tage oder Wochen an Arbeitszeit ersparen. Vertrauen Sie mir einfach in diesem Punkt. Bei Fragen zu GitHub wenden Sie sich gerne an mich.

## 5 Aufgaben

### 5.1 Mittleres Verschiebungsquadrat

In dieser Aufgabe widmen wir uns der Abhängigkeit der Position des diffundierenden Teilchens von der Diffusionskonstante  $D$  und der Zeit  $t$  (siehe Gl. (4)). Bei einem diffundierenden Teilchen betrachtet man typischerweise die Verschiebung zur Startposition. Da es sich in diesem Fall um eine statistische Größe handelt, wird über verschiedene Trajektorien gemittelt. Aufgrund der Symmetrie von positiver und negativer Verschiebung, wird über das Quadrat gemittelt. Die Größe  $\langle (x - x_0)^2 \rangle$  trägt den Namen *mittleres Verschiebungsquadrat* (*mean squared displacement* oder kurz: MSD). In Gl. (4) haben wir die Diffusionskonstante eingeführt. Da in unserem Problem sowohl die Bewegung als auch die Zeit diskretisiert sind, haben wir bei einem Diffusionsschritt zu jedem Zeitpunkt immer eine Diffusionskonstante von  $1/2$ . Um die Diffusionskonstante in diesem Problem variieren zu können, versehen wir das Problem mit einer Wahrscheinlichkeit, dass überhaupt ein Diffusionsschritt zu einem beliebigen diskreten Zeitpunkt stattfindet.

Der Code ist in zwei Dateien aufgeteilt:

In der Datei `fp_classes.py` werden die Objekt-Klassen `environment` und `agent` definiert.

Im Script `fp_reinforcement_learning_main.py` wird die Datei `fp_classes.py` importiert und die Klassen werden verwendet. Zu Beginn von `fp_reinforcement_learning_main.py` werden zwei Objekte `learner` vom Typ `agent` und `env` vom Typ `environment` definiert.

Variablen und Funktionen, die innerhalb der Klassen definiert werden, bezeichnet man als *member variables* und *member functions*. Jedes Objekt einer Klasse hat seine eigenen Variablen. Variablen innerhalb des Codes der Klassen werden als `self.x` aufgerufen und bezeichnen die Variablen des Objektes, welche die jeweilige *member function* aufruft. Außerhalb des Codes der Klassen werden diese mit dem Namen des Objektes, z. B. als `learner.x`, aufgerufen. Hier bedeutet `learner.x` die *member variable*  $x$  vom Objekt `learner` der Klasse `agent`.

1. Sie finden die zur Bearbeitung nötige Vorlage für den Code im Ordner zur Aufgabe 1. Öffnen Sie den Code in einer Entwicklungsumgebung Ihrer Wahl.

In Gl. (4) bezeichnet  $D$  die Diffusionskonstante für einen Random Walk mit Schrittweite  $a$  und Zeit zwischen Schrittereignissen  $\tau$ . Da in diesem Projekt der Zeitschritt ( $\tau = 1$ ) und die Schrittweite ( $a = \Delta x = 1$ ) konstant sind, ist auch  $D$  theoretisch festgelegt. Wir können die effektive Diffusionskonstante kontinuierlich skalieren, wenn wir in jedem Zeitschritt nur mit einer gewissen Wahrscheinlichkeit einen Diffusionsschritt durchführen und damit die mittlere Zeit zwischen zwei Diffusionsschritten verändern.

2. Definieren Sie innerhalb der Funktion `__init__` der Klasse `agent` in `fp_classes.py` die Variable `self.P_diffstep`. Setzen Sie diese Variable auf den entsprechenden Wert in Abhängigkeit des vorher definierten `self.D` (gemäß Gl. (4)).
3. Vervollständigen Sie innerhalb der Funktion `random_step(...)` den Code, sodass `self.x` mit einer Wahrscheinlichkeit von `self.P_diffstep` entweder um 1 oder um  $-1$  verändert wird.  
Zufällige  $z \in \mathbb{Z}$  im Intervall  $[n_0, n_1)$  lassen sich mit `np.random.randint( $n_0, n_1$ )` generieren.  
Zufällige  $z \in \mathbb{R}$  im Intervall  $[0., 1.)$  lassen sich mit `np.random.rand()` generieren.
4. Fügen Sie innerhalb der fertigen *for-loops* das Kommando `learner.random_step()` ein, um die Funktion als *member function* der Klasseninstanz `learner` des Typs `agent` auszuführen.

In Gl. (3) können Sie ablesen, dass das zweite Moment  $\langle x^2 \rangle$  der Verteilung den Wert  $2Dt$  trägt. Wenn Sie nun an die zuvor generierten Daten einen linearen Fit anlegen, lässt sich die effektive Diffusionskonstante aus der Steigung ablesen.



5. Setzen Sie `agent.N_episodes` auf 20000 und `agent.tmax_MSD` auf 100.
6. Löschen Sie nun das `exit()`-command hinter den obigen *for-loops*.  
Der Code berechnet nun bei der Ausführung  $\langle x^2(t) \rangle$ , führt einen linearen Fit an die Daten durch und schreibt die Koeffizienten in `p`. Aus dem ersten Eintrag in `p` können Sie die Steigung ablesen. Bestimmen Sie hieraus  $D$ .
7. Testen Sie dies nun mit drei verschiedenen Diffusionskonstanten, die Sie in `agent` manuell ändern können. Gibt es mögliche Grenzen für die Diffusionskonstante in dieser Implementation? Falls ja, welche?
8. Speichern Sie die drei entsprechenden Bilder unter den Namen **Aufgabe\_1\_MSD\_D0\_D1\_NAME\_ZEIT.png** in dem Verzeichnis für Aufgabe 1. D0 bezeichnet das manuell eingestellte  $D$ . D1 bezeichnet das vom linearen Fit gemessene  $D$ .  
Mit dem Befehl `fig.savefig(filename, bbox_inches='tight')` können Sie optional die Figure als Bild direkt im aktuellen Verzeichnis speichern. Für `filename` sollten Sie einen sinnvollen Dateinamen wählen. Der zu Beginn des Scripts definierte String `now` enthält Datum und Uhrzeit im ISO-Format. Für D0, D1 machen Sie sich mit String Formatting (`f'foo-{...}'`) und String Concatenation vertraut.

## 5.2 Implementation des Lernalgorithmus

In dieser Aufgabe werden wir uns der Implementation des Lernalgorithmus widmen. In der Klasse `environment` stehen die Parameter für die Umgebung, in der sich der Agent bewegt. `N_states` bezeichnet die Anzahl der Zustände, `target_position` die Position des Ziels und `starting_position` die Startposition des Agenten. Das Problem soll mit periodischen Randbedingungen umgesetzt werden. Mögliche Aktionen sind *nach links gehen* ( $\leftarrow$ ), *nach rechts gehen* ( $\rightarrow$ ) und *auf der aktuellen Position verbleiben* ( $\downarrow$ ). Das Ziel ist erreicht, wenn der Agent ein Mal auf der Zielposition verblieben ist.

1. Kopieren Sie Ihren Code aus Aufgabe 5.1 in den entsprechenden Ordner dieser Aufgabe.
2. Kommentieren Sie den Code-Block zur Untersuchung des MSD aus oder löschen Sie ihn, damit er nicht bei jeder Ausführung des Lernalgorithmus mit ausgeführt wird.
3. Setzen Sie die Diffusionskonstante auf einen Wert, sodass Sie im Durchschnitt in jedem vierten Schritt einen zufälligen Diffusionsschritt erwarten.
4. Setzen Sie `agent.N_episodes` auf  $10^4$ ,  $\alpha$  auf 0.01 und  $\gamma$  auf 0.9.
5. Die Variable `agent.zero_fraction` bezeichnet den Zeitpunkt, an dem  $\epsilon$  auf 0 abgefallen ist. Schreiben Sie die Funktion `adjust_epsilon` entsprechend, dass  $\epsilon$  (`agent.epsilon`) bei Episode 0 den Wert 1 trägt und bei Episode `agent.zero_fraction`  $\times$  `agent.N_episodes` den Wert 0. Dazwischen soll  $\epsilon$  linear abfallen.
6. `agent.x` bezeichnet die aktuelle Position / den aktuellen Zustand des Agenten. Schreiben Sie die Funktion `choose_action`, sodass die Variable `self.chosen_action` mit einer Wahrscheinlichkeit von  $\epsilon$  auf einen zugelassenen Zufallswert gesetzt wird ( $\leftarrow = 0$ ,  $\downarrow = 1$ ,  $\rightarrow = 2$ ). Andernfalls soll `self.chosen_action` entsprechend Gl. (5) festgelegt werden. Im Falle zweier Maxima soll auch eine zufällige Aktion gewählt werden.
7. Schreiben Sie die Funktion `perform_action`. Gemäß `self.chosen_action` soll die entsprechende Aktion ausgeführt werden. Bewegt sich der Agent aus dem Intervall hinaus, soll er auf der anderen Seite wieder hineinkommen (periodische Randbedingungen).

8. Schreiben Sie die Funktion `update_Q`, sodass der entsprechende Wert  $Q_{ij}$  entsprechend Gl. (6) aktualisiert wird. Bedenken Sie, auf welche Zustände sich  $i$  und  $i'$  beziehen. Speichern Sie den Zustand des *Agenten* zu Beginn der Episode in der Variable `x_old`, um diese bei der Aktualisierung von  $Q$  zu verwenden. Die Belohnung für das Verbleiben auf dem Ziel ( $\downarrow$ , s. o.) ist in der Variable `target_reward` gespeichert.
9. Fügen Sie die geschriebenen Funktionen in dieser Reihenfolge in der *for-loop* für die Episoden im Main Script ein: `adjust_epsilon`, `choose_action`, `random_step`, `perform_action`, `update_Q`.
10. Fügen Sie dem Ende des Scripts die drei Befehle ein:
  - `f = open('Q_MATRIX_' + now + '.txt', 'w')`
  - `f.write(str(learner.Q))`
  - `f.close()`
11. Am Ende des Main Scripts befindet sich Code, der mit Hilfe der `celluloid`-Bibliothek ein Film im *.gif*-Format in dem aktuellen Ordner ausgibt. Betrachten Sie dieses *.gif*, um zu beurteilen, ob der Agent lernt, sich sinnvoll in Richtung des Ziels zu bewegen.
12. Diese Befehle schreiben am Ende der Simulation die Matrix  $Q$  in eine Datei im aktuellen Ordner. Öffnen Sie die Datei und untersuchen Sie, ob der Lernprozess erfolgreich war. In diesem Fall sollte in der Zeile des Zielzustands  $\downarrow$  die bevorzugte Aktion darstellen. Direkt links davon sollte der Agent  $\rightarrow$  ausführen, direkt rechts  $\leftarrow$ . Aufgrund der periodischen Randbedingungen sollte etwa `env.N_states/2` Zustände weiter die weit entfernteste Position liegen. Links davon sollte der Agent gelernt haben nach links, rechts davon nach rechts zu gehen.

### 5.3 Wahl der Hyperparameter

Häufig ist der aufwändigste Arbeitsschritt jeder Arbeit zum Thema *Machine Learning* eine angemessene Wahl der Parameter  $\gamma$  und  $\alpha$ . Eine angemessene Wahl dieser Hyperparameter bestimmt die Konvergenz des Algorithmus gegen eine sinnvolle Strategie. In den meisten Problemen wird unabhängig davon, wie gut Ihr Problem algorithmisch modelliert ist, d.h., wie die Zustände gewählt sind und welche Aktionen erlaubt sind, eine schlechte Wahl der Hyperparameter dazu führen, dass der Lernprozess scheitert. Daher wollen wir uns in diesem Teil des Lernprojektes mit einigen sinnvollen Methoden zum Verständnis der Hyperparameter vertraut machen.

#### 5.3.1 Discount factor $\gamma$

Betrachten wir ein vereinfachtes Modellproblem des oben beschriebenen zweidimensionalen Pfad-Finde-Problems: ein Pfad-Finde-Problem in einer Dimension. Zunächst kann der Agent sich in einer von vier Positionen befinden. Die Zustände tragen die Indizes 0, 1, 2, 3. Die erlaubten Aktionen sind das Wechseln in einen der benachbarten Zustände (nach links gehen = Aktion 0 oder nach rechts gehen = Aktion 1). Der Agent startet **immer** in Zustand 0. Das Ziel befindet sich in Zustand 3. Sobald der Agent sich auf das Ziel bewegt, endet die Epoche. Die Belohnung für diese Aktion ist  $\mathcal{R}$ . Sollte der Agent in Position 0 einen Schritt nach links wählen, wird die Aktion als durchgeführt betrachtet, die Position / der Zustand des Agenten ändert sich jedoch nicht.  $Q$  wird entsprechend aktualisiert.

1. Welche Form hat  $Q$  in diesem Modell?
2. Zu Beginn des Lernalgorithmus ist  $\epsilon \approx 1$ . Für diese Aufgabe ist der genaue Wert unerheblich. Berechnen Sie  $Q$  **von Hand** nach dem Ende der erste Epoche mit einer beliebigen Aktionsfolge in Abhängigkeit von  $\mathcal{R}$ ,  $\gamma$  und  $\alpha$ .

3. Berechnen Sie  $Q$  **von Hand** nach der zweiten Epoche nach der Aktionsfolge  $\rightarrow \leftarrow \rightarrow \rightarrow$  in Abhängigkeit von  $\mathcal{R}$ ,  $\gamma$  und  $\alpha$ .
4. Berechnen Sie  $Q$  **von Hand** während der dritten Epoche nach der Aktion  $\rightarrow$  in Abhängigkeit von  $\mathcal{R}$ ,  $\gamma$  und  $\alpha$ .
5. Wie erklären Sie, dass der Agent nach dem Lernprozess schon in Zustand 0 von dem Ziel in Zustand 3 gelernt hat? Welche Aktion wird der Agent für diesen Zustand gelernt haben?
6. Kann der Agent in diesem Modell lernen in einer Aktion nach links zu laufen?

Im Allgemeinen wird der *discount factor*  $\gamma$  auf Werte  $0.5 \lesssim \gamma \lesssim 1.0$  gesetzt. Wie Sie oben gesehen haben, propagiert die Belohnung durch benachbarte Zustände und wird bei jeder Fortpflanzung mit einem Faktor  $\gamma$  versehen. Für  $\gamma \approx 1$  ist zu erwarten, dass über viele Zustände hinweg die Belohnungen der anderen Zustände einen Einfluss haben. Für  $\gamma \lesssim 0.5$  spielen die Belohnungen über wenige Zustände hinweg keine bedeutende Rolle mehr. Die Wahl des *discount factor*  $\gamma$  sollte sich daran anpassen, wie ähnlich das erlernte Verhalten in den verschiedenen Zuständen zu erwarten ist. Im oben vorgestellten Problem ist das Verhalten in allen Zuständen identisch. Die Wahl von  $\gamma$  ist daher beliebig.

### 5.3.2 Learning rate $\alpha$

Ähnlich dem Parameter  $\Delta t$  in der numerischen Lösung einer Differentialgleichung bestimmt die *learning rate*  $\alpha$  die Geschwindigkeit der Konvergenz des Algorithmus. In den vielen Fällen führt ein zu groß gewähltes  $\alpha$  jedoch dazu, dass der Algorithmus überhaupt nicht konvergiert.

Nachdem Sie sich in der letzten Aufgabe mit der Wahl des *discount factor*  $\gamma$  vertraut gemacht haben, werden wir in dieser Aufgabe den Lernalgorithmus implementieren und das  $\alpha$  festlegen.

1. Kopieren Sie Ihre aktuelle Version des Codes in den Ordner dieser Aufgabe.
2. Setzen Sie die Diffusionskonstante temporär auf 0.
3. Schreiben Sie in das Main Script entsprechend Code, dass für jede Episode die Anzahl der Schritte gezählt wird, bis das Ziel erreicht ist. Speichern Sie die Anzahl der Schritte und jeweilige Episode in einer Liste. Lassen Sie das Script die Anzahl der Schritte im Laufe der Simulation automatisch nach Ausführung plotten. Lassen Sie außerdem das Script die Figure als Bild in dem aktuellen Ordner speichern (siehe Aufgabe 5.1, 8.). Nutzen Sie als plot-Kommando `matplotlib.pyplot.semilogy()`. Wir bezeichnen diese Darstellung der Performance gegen die Simulationszeit als Lernkurve (*learning curve*). Sie sehen, dass die benötigte Anzahl der Schritte etwa exponentiell abfällt. Für `episode  $\geq$  agent.zero_fraction  $\times$  env.N_episodes` sehen Sie einen konstanten Wert in der Lernkurve.
4. Modifizieren Sie den Code nun so, dass zu Beginn jeder Episode die Position des Agenten zufällig in einem beliebigen Zustand initialisiert wird. Nutzen Sie hierfür `env.N_states` und `np.random.randint(...)`. Lassen Sie sich erneut die Lernkurve ausgeben.
5. Wie Sie sehen, konvergiert die Lernkurve hier nicht gegen einen konstanten Wert für `agent.zero_fraction  $\times$  env.N_episodes`. Dies liegt selbstverständlich daran, dass auch nach Konvergenz des Algorithmus die Startposition noch zufällig ist. Ein geeigneteres Maß für die Leistung des Agenten ist es zu prüfen, ob der Agent tatsächlich die minimale Anzahl von Aktionen benötigt hat. Berechnen Sie die minimal mögliche Anzahl von Aktionen zu Beginn jeder Episode. Bedenken Sie hierbei die periodischen Randbedingungen und die Tatsache, dass die Episode endet, nachdem der Agent ein Mal auf dem Ziel verweilt.

6. Modifizieren Sie den Code für die Lernkurve so, dass der Plot nun das Verhältnis der ausgeführten Schritte zur minimal möglichen Anzahl der Schritte zeigt.
7. Führen Sie den Code erneut aus und betrachten Sie die Lernkurve. Diese sollte nun gegen 1 konvergieren. Ist dies nicht der Fall, reduzieren Sie  $\alpha$ .
8. Setzen Sie  $\alpha$  nun auf 0.999999. Lassen Sie sich die Lernkurve erneut ausgeben. Sehen Sie, wie die Konvergenz des Algorithmus nun schlechter wird und für `agent.zero_fraction × env.N_episodes` nun kein konstanter Wert erreicht wird. Aufgrund der Einfachheit dieses Problems, benötigt es extreme Werte für  $\alpha$ , um eine Nicht-Konvergenz zu erreichen. Sollten Sie entgegen der Erwartungen bei diesen Werten keinen Unterschied beobachten, vergleichen Sie die verschiedenen Werte von  $\alpha$  bei geringerer Anzahl der Episoden. Üblicherweise ist  $\alpha \ll 1$ .  
Sollten Sie mit dem obigen  $\alpha = 0.999999$  keine Verschlechterung erzielen, überlegen Sie, was Sie bei  $\alpha = 1$  erwarten. Nähern Sie  $\alpha$  so weit 1 an, bis dies sichtbar wird.
9. Setzen Sie  $\alpha$  auf einen sinnvollen Wert zurück. Setzen Sie die Diffusionskonstante außerdem auf den vorherigen Wert.

In vielen Problemen und Modellen ist es sinnvoll, die Werte von  $Q$  und deren Zeitentwicklung explizit zu visualisieren. Dies würde in unserem Beispiel wie unten angegeben aussehen. In diesem Modell sind die Plots aufgrund der Einfachheit des Problems nicht nötig. Die folgenden Schritte sind optional.

- Sie definieren eine Liste vor Beginn der Lernschleife bspw. mit dem Namen `Q_VALUES_OVER_TIME`.
- Sie definieren in dem Konstruktor `__init__` der `agent`-Klasse eine Variable mit dem Namen `self.output_state = 30`.
- Sie fügen im Main Script bei jedem Update von  $Q$  der Liste `Q_VALUES_OVER_TIME` die Zeile von  $Q$  mit dem Index `self.output_state` hinzu (hierzu nutzen Sie `list.append(...)`).

Typischerweise wollen Sie, dass Ihr  $\alpha$  so groß wie möglich, aber so klein wie nötig ist, da Sie für zu kleine  $\alpha$  mehr Lernzeit zur Konvergenz benötigen. Ähnliches gilt für die Anzahl der Episoden: so wenige wie möglich, so viele wie nötig. Entsprechend sind beides Parameter, welche gegeneinander abgewogen werden müssen. Häufig ist der limitierende Faktor die Rechenzeit, die der Algorithmus benötigt. Gleichzeitig ist es das Ziel, mit der entsprechenden Wahl der Parameter die maximale Lerneffizienz in geringstmöglicher Zeit zu erreichen. Es ist außerdem üblich  $\alpha$  im Lauf der Simulation, ähnlich wie  $\epsilon$  abfallen zu lassen. Die Idee dahinter ist, dass der Algorithmus sich mit fortschreitender Lernzeit auf eine Strategie festlegt und die Konvergenz verbessert. Bei der funktionalen Abhängigkeit  $\alpha(\text{episode})$  vom Fortschritt der Simulation sind alle obigen Faktoren zu berücksichtigen.

## 5.4 Stochastisches Hindernis

Bisher haben wir Navigationsprobleme gelöst, deren Lösung offensichtlich ist: Der Agent lernt, auf direktem Weg in Richtung des Ziels zu navigieren. Zum Abschluss dieses Lernprojektes werden wir sehen, dass der Agent auch in komplexeren Landschaften, bei denen die Lösung ggf. nicht sofort ersichtlich ist, lernt, den schnellsten Weg zum Ziel zu finden.

Zu diesem Zweck werden wir ein Hindernis für den Agenten implementieren, welches ihn mit einer gewissen Wahrscheinlichkeit verschiebt und so einen Widerstand für dessen Bewegung darstellt.

1. Kopieren Sie die aktuelle Version Ihres Codes in das Verzeichnis der aktuellen Aufgabe.
2. Damit die Rechenzeit in dieser Aufgabe schneller ist, setzen Sie die Anzahl der Zustände `env.N_states` auf 15.
3. In der `environment`-Klasse finden Sie die Variablen `obstacle_interval` und `self.P_obstacle`. Solange die Position des Agenten im `obstacle_interval` ist, soll er sich in jedem Schritt mit einer Wahrscheinlichkeit `P_obstacle` nach links bewegen. Schreiben Sie in der *member function* `stoch_obstacle` den entsprechenden Code und fügen Sie die Funktion hinter `random_step` im Main Script ein.
4. Sofern nicht der Fall, setzen Sie `obstacle_interval` auf `np.arange(9,12)`.
5. Setzen Sie `target_position` in `environment` auf 12 und `starting_position` auf 8.
6. Modifizieren Sie den Main Code, sodass die Position des Agenten in jeder Episode bei `env.starting_position` initialisiert wird (anstatt zufällig wie zuvor).
7. Definieren Sie **vor** der *for-loop* über den Episoden die Variable `total_action_displacement`.
8. Hinter `perform_action`: Fügen Sie ein Kommando zum Addieren der Verschiebung der aktuellen Aktion auf `total_action_displacement` für den Fall, dass  $\epsilon == 0$  (`total_action_displacement += (learner.chosen_action-1)`). Dieses Kommando bestimmt die gesamte Verschiebung durch Aktionen für die Episoden, nachdem der Agent fertig trainiert ist.
9. **Nach** der *for-loop* für das gesamte Training: Fügen Sie ein Kommando ein, welches `total_action_displacement/np.abs(total_action_displacement) =:  $\kappa$`  in die Konsole ausgibt.
10. Wenn diese Zahl  $-1$  ist, macht der ausgelernte Agent Schritte nach links, bei  $+1$  macht er Schritte nach rechts. Bei welchem `env.P_obstacle =  $\mathcal{P}_0$`  erwarten Sie ungefähr den Übergang von  $\kappa$ ? Warum?
11. Bestimmen Sie die Position des Übergangs numerisch, indem Sie das gesamte Script mit einer *for-loop* einhüllen, welche den gesamten Lernprozess für einige Werte von `P_obstacle` um das erwartete  $\mathcal{P}_0$  herum ausführt und die Werte für `P_obstacle` und  $\kappa$  speichert. Achten Sie darauf, dass die Definition von `env` und `learner` innerhalb dieser *for-loop* geschieht, sodass die Lernprozesse mit unabhängigen  $\mathcal{Q}$  starten.
12. Kommentieren Sie den Plotting Code aus, der für die Lösung dieser Aufgabe nicht nötig ist.
13. Plotten Sie  $\kappa$  als Funktion von `P_obstacle` und lesen Sie den Übergang aus dem Plot (mit `plt.show()`) ab. Nähern Sie sich ggf. dem Übergang an, indem Sie das Intervall sukzessiv verkleinern. Beachten Sie, dass der Algorithmus statistischen Fluktuationen unterliegt und Ergebnisse für den Übergang abweichen können. Um diesen Übergang genau zu bestimmen, ist eine größere Menge an Daten nötig, welche für dieses Lernprojekt nicht zielführend ist.
14. Setzen Sie  $\alpha$  erneut auf 0.999999. Führen Sie das selbe Script für einige Werte von `env.P_obstacle` zwischen 0 und 1 aus. Beobachten Sie, dass der Algorithmus gegen eine sehr schlechte Strategie konvergiert. Sollte dies nicht sichtbar sein, verfahren Sie wie oben (siehe Aufgabe 5.3.2, 8).

## 6 Abgabe

1. Zur vollständigen Bearbeitung dieses Lernprojekts gehört die Erstellung eines Versuchsprotokolls, in dem Sie die Ergebnisse der einzelnen Aufgaben präsentieren und diskutieren. Orientieren Sie sich bei der Strukturierung Ihres Protokolls an der Struktur der Aufgaben. Gehen Sie bei der Beschreibung der Aufgaben davon aus, dass Ihr Protokoll ohne Zuhilfenahme dieser Versuchsanleitung verstanden werden soll.
2. Darüber hinaus geht der Code, der während der Bearbeitung der einzelnen Aufgaben erstellt wurde, in die Bewertung ein. Stellen Sie daher sicher, dass Ihr Code in den verschiedenen Unterordnern ausführbar und funktionsfähig ist. Er sollte Ergebnisse generieren können, wie sie in Ihrem Protokoll diskutiert werden.
3. Stellen Sie sicher, dass der Code verständlich lesbar ist. Eine ausführliche Kommentierung des Codes ist an dieser Stelle nicht notwendig, da Ihr Betreuer mit der Codestruktur vertraut ist.
4. Die ursprüngliche Version dieser Versuchsbeschreibung sieht keine spezielle Frist für die Bearbeitung vor. Stellen Sie jedoch sicher, mit Ihrem Betreuer über dessen Vorstellungen zu einer sinnvollen Bearbeitungszeit zu sprechen, sofern dies noch nicht geschehen ist.
5. Sobald Sie bereit für die Abgabe sind, erstellen Sie aus der Ordnerstruktur eine *.zip*-Datei, welche Sie Ihrem Betreuer zukommen lassen. Denken Sie bei der Benennung daran, dass die *.zip*-Datei einen sinnvollen Namen trägt und Ihnen zugeordnet werden kann.

## 7 Zusammenfassung und Ausblick

In diesem Lernprojekt haben Sie die Grundzüge des *Reinforcement Learning* am Beispiel eines Navigationsproblems in einer Dimension kennengelernt. Sie haben außerdem Simulationscode zu einem diskreten Diffusionsproblem erstellt, die Linearität des mittleren Verschiebungsquadrats in der Zeit gezeigt und effektive Diffusionskonstanten bestimmt. Sie haben nahezu vollständig einen Reinforcement-Learning-Algorithmus implementiert und sich mit den verschiedenen Datenformen und Befehlen sowie deren Umgang in Python vertraut gemacht. Darüber hinaus haben Sie den Umgang von Klassen und Objekten in Python gefestigt. In Aufgabe 5.3 haben Sie gelernt, wie ein *Agent* lernt, zu einer diskreten/singulären Belohnung in der Ferne zu navigieren, ohne diese von Anfang an wahrzunehmen. Außerdem haben Sie mit der *Lernkurve* und der Darstellung der Einträge von  $Q$  zwei Werkzeuge kennengelernt, die Hyperparameter quantitativ festzulegen. In Aufgabe 5.4 haben Sie gesehen, dass der Algorithmus fähig ist, sich mit unterschiedliche Strategien speziellen Umgebungsparametern anzupassen.

Obwohl das Problem und das Reinforcement-Learning-Modell in diesem angeleiteten Lernprojekt relativ simpel erscheinen, bildet es die Basis für das Verständnis einer Vielzahl an Modellen und Algorithmen. Bereits dieses Modell kann mit geringfügiger Modifikation Probleme lösen, deren Inhalt Abschlussarbeiten und aktueller Forschung würdig ist.

Einige denkbare Modifikationen dieses Modells beinhalten z. B. das Problem quasi-kontinuierlich zu machen, sodass die Position des Agenten im Rahmen numerischer Genauigkeit reell ist und anstatt eines diskreten Schrittes in eine Richtung eine gedämpfte Beschleunigung eine Aktion darstellt. Da selbst nach Ende der Lernzeit  $\alpha \neq 0$  ist, passt der Algorithmus sich an bspw. langsam bewegliche Ziele an und die Strategie ändert sich. Denkbar sind auch Navigationsprobleme mit räumlich variierender Schrittweite/Geschwindigkeit in höheren Dimensionen, sodass der Agent lernt, die schnellste Route zu finden und Bereiche niedriger Geschwindigkeit zu umgehen. Diese Probleme werden analytisch schnell quasi unlösbar, sodass das Machine Learning einen wichtigen wissenschaftlichen Beitrag leistet. Auch lässt sich dieser Algorithmus auf *Deep Reinforcement Learning* verallgemeinern, welches Anzahlen von Zuständen beinhaltet, die das menschliche Fassungsvermögen bei weitem übersteigen.