Angeleitetes Lernprojekt:
Reinforcement learning für intelligente Brownsche Dynamik

Paul A. Monderkamp

14. Juni 2022

Abbildung 1: Reinforcement learning Algorithmus am lebenden Objekt: Der Hund Lasse gibt sein Pfötchen. Dafür bekommt er eine Belohnung um diese Aktion zu bestärken (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

1 Einleitung

Dieses angeleitete Lernprojekt bietet eine Einführung in die Grundlagen des reinforcement learning am Beispiel des $\mathcal{Q}$-learning algorithmus. Dieser bildet die Grundlage für viele komplexe machine learning Algorithmen, die heutzutage in verschiedenen Bereichen Anwendung finden. Das Lernziel dieses Projektes besteht darin die Grundlagen des $\mathcal{Q}$-learning zu verstehen, die/der Bearbeitende diesen Algorithmus auf verschiedene Probleme anwenden lernen 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 python3. Die Einbettung in den Kontext eines eindimensionalen random-walks 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 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 häufig von *Teilchen* die Rede, in der kondensierten weichen Materie wird häufig die Bezeichnung *(Mikro-)Schwimmer* verwendet, für Teilchen, die Brownscher 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*, *Trainigszeit*, *Simulation* oder ähnlichem die Rede. Alle diese Begriffe beschreiben den selben Vorgang: einen Durchlauf des Machine Learning Programmes um den Agenten vollstänfig 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 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 bspw. durch Münzwurf festlegt werden.

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

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

Hierbei bezeichnet p die Wahrscheinlichkeit in einem einzelnen Schritt nach rechts zu gehen, $q = 1 - p$ 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 $w_N((m+N)/2)$:

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

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 kontiuerlichen Fall $m \to x$ mit $p = q = 0.5$ 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) folgt:

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

Siehe hierzu Kapitel 2.5.1 (Stand 9.6.22) im script zur Statistischen Mechanik. D bezeichnet hier die Diffusionskonstante

$$D := \frac{a^2}{2\tau}, \tag{4}$$

wobei τ die Zeitspanne zwischen zwei Schritten, und $a = |\Delta x|$ die Schrittweite, im diskreten Fall bezeichnet.

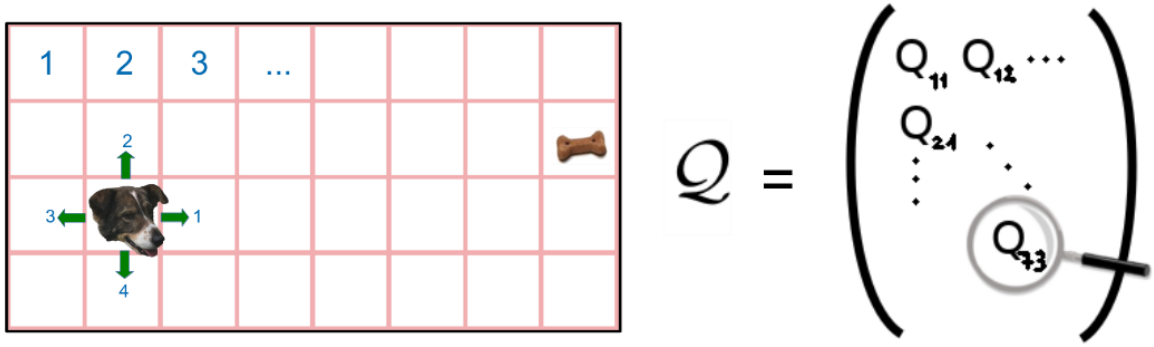

Abbildung 2: Schema eines zwei-dimensionalen Pfad-Finde Problems. Der Agent (Hund) soll lernen zum Hundeknochen zu finden. **Links**: Diskretisierung des Raums und Definition der Aktionen. Der Konfigurationsraum wird diskretisiert. Jede Position bekommt einen eindeutigen Index. Jede Aktion bekommt einen Index. **Rechts**: $\mathcal{Q}$-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 $\mathcal{Q}$-Matrix wird im Laufe des Lernprozesses optimiert.

2.2 $\mathcal{Q}$-learning

Es existieren verschiedene große Bereiche innerhalb der Wissenschaft der *künstlichen Intelligenz*. Darunter fallen unter anderem *unsupervised machine learning*, *supervised machine learning* und *reinforcement learning*. Ersteres dient in den meißten Fällen der Datenverarbeitung und Datensortierung großer Datenmengen. Clusteranalysen wie *k-Means* oder Dimensionsreduktion von hochdimensionalen Datensets werden i.A. zum *unsupervised machine learning* gezählt. *supervised machine 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 zu Grunde. Der Agent generiert im Laufe der Trainingszeit anhand einer Strategie Daten selber. Des weiteren wird dem Agenten eine Möglichkeit gegeben die Leistung in der zu lernenden Aufgabe zu quantifizieren. Im Laufe der Simulation wird die Strategie des Agenten entsprechend optimiert, in der Hoffnung zu 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. Fig. (2, links). Der Hundeagent soll darauf trainiert werden den Hundeknochen, unabhaengig von seiner Startposition, so schnell wie möglich zu erreichen. Die Position des Hundeknochens bleibt unverändert. Zunächst wird der Raum, in dem der Agent sich auffhalten 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 bezeichnet. Die Strategie, welche nun das $\mathcal{Q}$-learning auszeichnet, ist $\mathcal{Q}$ als Entscheidungsbasis zu verwenden, gemäß:

$$\text{Aktion in Zustand i} = \max_j(\mathcal{Q}_{ij}). \tag{5}$$

Das Ziel des Lern-Algorithmus besteht nun darin, die Matrix $\mathcal{Q}$ so zu optimieren, dass die Entscheidungen auf Basis von $\mathcal{Q}$ 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 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 ϵ wählt er eine zufällige Aktion und mit Wahrscheinlichkeit $1-\epsilon$ wählt der Agent eine Aktionauf Basis von $\mathcal{Q}$ (siehe Gl (5)).

Nach jeder ausgeführten Aktion wird $\mathcal{Q}$ aktualisiert gemäß

$$\mathcal{Q}_{ij}^{neu} = \mathcal{Q}_{ij}^{alt} + \alpha \left(\mathcal{R}_{ij} + \gamma \max_j (\mathcal{Q}_{i'j}) - \mathcal{Q}_{ij}^{alt} \right). \tag{6}$$

i bezeichnet hier den Zustand vor der aktuellen Aktion und j bezeichnet die Aktion, nach deren Durchführung $\mathcal{Q}$ jetzt aktualisiert wird. $\mathcal{Q}_{ij}^{alt}$ bezeichnet den entsprechenden Wert in $\mathcal{Q}$ **vor** der Aktualisierung, $\mathcal{Q}_{ij}^{neu}$ bezeichnet den entsprechenden Wert **nach** der Aktualisierung. α bezeichnet die *learning rate*. γ bezeichnet den *discount factor*. $\mathcal{R}_{ij}$ bezeichnet die Belohnung/reward für das Durchführen der Aktion j **in Zustand** i. $\max_j(\mathcal{Q}_{i'j})$ bezeichnet den maximalen Wert in Zeile i'. Dies bezeichnet den Zustand **nach** Ausführung der Aktion. Im Allgemeinen muss, abhängig vom zugrunde liegenden Problem, i' nicht von i verschieden sein, sollten eine Aktion durchgeführt werden, welche den Zustand nicht ändert. Der Term $\gamma \max_j(\mathcal{Q}_{i'j})$ bildet eine Abschätzung in aktuellen Zustand an die mögliche Belohnung in der Zukunft und führt dazu, dass Belohnungen in speziellen Zuständen auch die Wahl der Aktion anderer Zustände beeinflussen (Mehr zu diesen Parametern lernen wir in Aufgabe 3.3).

3 Aufgaben

Um gute Arbeitsethik mit wissenschaftlichen Daten zu lernen, wollen wir dies zu einem zetralen Bestandteil der Bearbeitung dieses Lernprojektes machen. Sollten Sie im späteren Verlauf Ihrer (Forschungs-)Arbeit mit größeren Datenmengen Arbeiten, welche auf viele Dateien aufgeteilt ist, die alle bspw. den Namen *data.txt* tragen, sind diese Daten für Sie als Wissenschaftler häufig quasi un- und nur sehr erschwert benutzbar.

Achten Sie daher darauf, dass die Dateien, welche Sie im Laufe der Bearbeitung anlegen die entsprechenden Dateinamen tragen, sodass sie am Ende entsprechend gefunden werden können. Achten Sie weiterhin darauf, dass jede Datei deskriptiv benannt ist und das Datum und die Uhrzeit Ihrer letzten Änderung trägt. White Spaces sollten in Dateinamen grundsätzlich vermieden werden. Optional können Sie Ihren Namen hinzufügen.
Bspw.: **Aufgabe_1_QMATRIX_Paul_Monderkamp_2022-06-10T10_17_00.txt**
Der Zeitstempel ist im obigen Beispiel bzgl. der **ISO 8601** angegeben.
Achten Sie bei allen Diagrammen auf eine angemessene Schriftgröße und Achsenbeschriftung.
Um Abhängingkeiten von Code zu vermeiden, der unabhängige Aufgaben ausführt, sollte der Code der verschiedenen Aufgaben von einander getrennt werden. Da die verschiedenen Aufgaben jedoch aufeinander aufbauen, werden Sie aufgefordert für die Bearbeitung der ensprechenden Aufgaben den Code der letzten Bearbeitung in das entsprechende nächste Verzeichnis zu kopieren um dort weiter zu arbeiten. 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 deren Sie arbeiten.
Wenn sie keine Infrastruktur wie, Jupyter-Notebook verwenden, welche es Ihnen ermöglicht code-Blöcke auszuführen, machen Sie außerdem Gebraucht vom *exit()*-Kommando um die vorgefertigten code-Blöcke für spätere Aufgaben nicht auszuführen.

3.1 Mittleres Verschiebungsquadrat

In dieser Aufgabe widmen wir uns der Abhängigkeit der Position des diffundierenden Teilchens von Diffusionskonstante D und Zeit (vgl. 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 üeber das Quadrat gemittelt: $\langle \Delta x^2 \rangle$. Diese Größe trägt den Namen *mean squared displacement* (kurz: MSD). In Gleichung (4) haben wir die Diffusionskonstante eingeführt. Da in unserem Problem sowohl die Bewegung, als auch die Zeit diskretisiert ist, haben wir bei einem Diffusionsschritt zu jedem Zeitpunkt immer eine Diffusionskonstante von $1/2$. τ in Gl. (4) bezeichnet den charakteristischen Zeitabstand zweier Zufallsschritte. Um die Diffusionskonstante in diesem Problem variieren zu können versehen wir das Problem mit einer Wahrscheinlichkeit das üeberhaupt ein Diffusionsschritt zu einem beliebigen diskreten Zeitpunkt statt findet.

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/member functions*. Jedes Objekt einer Klasse hat seine eigenen Variablen. Variablen innerhalb des codes der Klassen werden bsow. 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 Objekts, bspw. als *learner.x*, aufgerufen. Hier bezeichnet *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 zu Aufgabe 1. 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 Abhängingkeit des vorher definierten *self.D*. (gemäß Gl. (4))

2. Schreiben Sie innerhalb der Funktion *random_step(...)* code, sodass *self.x* mit einer Wahrscheinlichkeit von *self.P_diffstep* um ± 1 verändert wird.
   Zufällige $\mathbb{Z}$ im Interval $[n_0, n_1)$ lassen sich mit *np.random.randint*(n_0,n_1) generieren. Zufällige $\mathbb{R}$ im Interval $[0., 1.)$ lassen sich mit *np.random.rand()* generieren.

3. Fügen Sie innerhalb der fertigen *for-loops* das Kommando textitlearner.random_step() ein, um die Funktion als member-Funktion der Klassesinstanz *learner* des Typs *agent* auszuführen.

In Gleichung (3) können Sie ablesen, dass das zweite Moment der Verteilung $\langle x^2 \rangle$ 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 Ausführung $\langle x^2(t) \rangle$, führt einen linaeren 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* von Hand ändern können.

8. Speichern Sie die drei entsprechenden Bilder unter den Namen:
   Aufgabe_1_MSD_D0_D1_NAME_ZEIT.txt in dem Verzeichnis für Aufgabe 1. D0 bezeichnet das eingestellte D. D1 bezeichnet das 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 (*ffoo_*{...}") und string concatenation vertraut.

3.2 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* bezeichnet die Position des Ziels. *starting_position* bezeichnet 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 3.1 in den enstsprechenden 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, α auf 0.01 und γ auf 0.9.
5. Die Variable *agent.zero_fraction* bezeichnet den Zeitpunkt, zu dem ϵ auf 0 abgefallen ist. Schreiben Sie die Funktion *adjust_epsilon* entsprechend, dass ϵ (*agent.epsilon*) bei Episode 0 den Wert 1 trägt und bei Episode *agent.zero_fraction* $\times$ *agent.N_episodes* den Wert 0. Dazwischen soll ϵ 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 ϵ 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 Aktion ausgeführt werden. Bewegt sich der agent aus dem Intervall hinaus, soll er auf der anderen Seite wieder hinein kommen (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 eine Variable *x_old*, um diese bei der Aktualisierung von $\mathcal{Q}$ zu verwenden.
9. Fügen Sie die geschriebenen Funktionen in dieser Reihenfolge in dem *for-loop* für die Episoden im main scipt ein: *adjust_epsilon*, *choose_action*, *perform_action*, *perform_action*, *random_step*, *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. Diese Befehle schreiben am Ende der Simulation $\mathcal{Q}$ 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.

3.3 Wahl der Hyperparameter

Häufig ist der aufwändigste Arbeitsschritt jeder Arbeit zum Thema *machine learning* eine angemessene Wahl der Parameter γ, α. Eine angemessene Wahl dieser Hyperparameter bestimmt die Konvergenz des Algorithmus gegen eine sinnvolle Strategie. In den meißten 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.

3.3.1 Discount factor γ

Betrachten wir ein vereinfachtes Modellproblem des oben beschriebenen zwei-dimensionalen Pfad-Finde Problems: Ein Pfad-Finde Problem in einer Dimension. Zunächst kann der Agent sich in einem vier Positionen befinden. Die Zustände tragen die Indizes 0,1,2,3. Die erlaubten Aktionen ist das Wechseln in einen der benachbarten Zustände (nach links = Aktion 0, oder rechts = Aktion 1 gehen). 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 Belohng $\mathcal{R}_{21}$ für diese Aktion ist $\mathcal{R}_{21} = \mathcal{R}$. 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 $\mathcal{Q}$ in diesem Modell?
2. Zu Beginn des Lernalgorithmus ist $\epsilon \approx 1$. Für diese Aufgabe ist der genaue Wert unerheblich. Berechnen Sie $\mathcal{Q}$ **von Hand** nach dem Ende der erste Epoche mit einer Beliebigen Aktionsfolge in Abhängigkeit von $\mathcal{R}, \gamma, \epsilon$.
3. Berechnen Sie $\mathcal{Q}$ **von Hand** nach der zweiten Epoche nach der Aktionsfolge $\rightarrow\leftarrow\rightarrow\rightarrow\rightarrow$ in Abhängigkeit von $\mathcal{R}, \gamma, \epsilon$.
4. Berechnen Sie $\mathcal{Q}$ **von Hand** während der dritten Epoche nach der einer Aktion $\rightarrow$ in Abhängigkeit von $\mathcal{R}, \gamma, \epsilon$.
5. Wie erklären Sie, dass der Agent nach dem Lernprozess schon in Zustand 0 von dem Reward 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* γ 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 γ versehen. Häufig erhält der Agent jedoch nicht nur einem Zustand eine Belohnung. 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 üeber wenige Zustände hinweg keine bedeutende Rolle mehr. Die Wahl des *discount factors* γ sollte sich daran anpassen, wie ähnlich das erlernte Verhalten in den verschiedenen Zuständen zu erwarten ist. Im oben vorstellten Problem ist das Verhalten in allen Zuständen identisch. Die wahl von γ ist daher beliebig.

3.3.2 Learning rate α

Ähnlich dem Parameter Δt in der numerischen Lösung einer Differenzialgleichung, bestimmt die *learning rate* α die Geschwindigkeit der Konvergenz des Algorithmus. In den meißten Fällen führ ein zu groß gewähltes α jedoch dazu, dass der Algorithmus nicht Konvergiert.
Nachdem Sie sich in der letzten Aufgabe mit der Wahl des *discount factors* γ vertraut gemacht haben, werden wir in dieser Aufgabe den Lernalgorithmus implementieren und das α 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 eine Liste. Lassen Sie das script die Anzahl der Schritte im Lauf der Simulation automatisch nach Ausführung plotten und die Figure als Bild in den aktuellen Ordner speichern (siehe Aufg. 3.1, 8.). Nutzen Sie als plot-Kommando *matplotlib.pyplot.semilogy()*.
   Wir nennen bezeichnen diese Darstellung der performance gegen die Simulationszeit als Lernkurve (learning curve). Sie Sehen, dass die benötigte Anzahl Schritte etwa exponentiell abfällt. Für *episode >= agent.zero_fraction* sehen sie einen Konstanten Wert in der learning curve.

4. Modifizieren Die 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 learning curve ausgeben.

5. Wie Sie sehen, konvergiert die learning curve hier nicht gegen einen konstanten Wert für *episode >= agent.zero_fraction*. Dies liegt selbstverständlich daran, dass auch nach Konvergenz des Algorithmus die Startposition noch zufällig ist. Ein geeigneteres Maß der Leistung des Agenten ist zu prüfen, ob der Agent tatsächlich die minimale Anzahl Aktionen benötigt hat.
   Berechnen Sie zu Beginn jeder Episode die minimale Anzahl Aktionen. Bedenken Sie hierbei die periodischen Randbedingungen und die Tatsache, dass die Episode endet, nachdem der Agent ein mal auf dem Ziel verweilt.

6. Führen Sie den Code erneut aus und Betrachten Sie die *learning curve*. Diese sollte nun geen 1 konvergieren. Ist dies nicht der Fall, reduzieren sie α

7. Setzen Sie α nun auf 0.999999. Lassen Sie sich die learning curve erneut ausgeben. Sehen Sie, wie die Konvergenz des Algorithmus nun schlechter wird, und für *episode >= agent.zero_fraction* nun kein konstanter Wert erreicht wird (aufgrund die Einfachheit dieses Problems, benötigt es extreme Werte für α um eine Nicht-Konvergenz zu erreichen. Sollten Sie entgegen der Erwartungen bei diesen Werten keinen Unterschied beobachten, vergleichen Die die verschiedenen Werte von α bei geringerer Anzahl der Episoden) Üblicherweise ist $\alpha \ll 1$.

8. Setzen sie α zurück auf einen sinnvollen Wert. Setzen Sie die Diffusionskonstante außerdem auf den vorherigen Wert.

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

- Sie definieren eine Liste vor Beginn der Lernschleife bsow. 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 $\mathcal{Q}$ der Liste *Q_VALUES_OVER_TIME* die Zeile von $\mathcal{Q}$ mit dem Index *self.output_state* hinzu (hierzu nutzen Sie *list.append(...)*).

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

3.4 Stochastisches Hindernis

Bisher haben wir Navitationsprobleme 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 in *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 enstprechenden 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** dem *for-loop* über die 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** dem *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? Warum?

11. Bestimmen Sie die Postition des Übergangs numerisch, indem Sie das gesamte script mit einem *for-loop* einhüllen, welche den gesamten Lernprozess für einige Werte von *P_obstacle* in um das erwartete $\mathcal{P}_0$ herum ausführt und die Werte für *P_obstacle* und κ speichert. Achten Sie darauf, dass die Definition, von *env* und *learner* innerhalb dieses *for-loops* 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 κ vs *P_obstacle* und lesen sie den Übergang aus dem Plot (mit *plt.show()*) ab. Nähern Sie sich ggf. 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 sinnvoll zu bestimmen, ist eine größere Menge an Daten nötig, welche für dieses Lernprojekt nicht zielführend ist.

14. Setzen Sie α 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.

4 Zusammenfassung/ Ausblick

In diesem Lernprojekt haben Sie die Grundzüge von *reinforcement learning* am Beispiel eines Navitationsproblems in einer Dimension kennen gelernt. Sie haben außerdem Simulationscode zu einem Diskreten Diffusionsproblem implementiert, die linearität des mittleren Verschiebungsquadrats in der Zeit kennen gelernt und effektive Diffusionskonstanten bestimmt. Sie haben nahezu vollständig einen *reinforcement learning* Algorithmus implementiert und sich mit den verschiedenen Datenformen und Befehlen und deren Umgang in Python vertraut gemacht. Darüber hinaus haben Sie den Umgang von Klassen und Objekten in Python gefestigt. In Aufgabe 3.3 haben Sie gelernt, wie ein *Agent* lernt zu einer diskreten/singulären Belohnung in der Ferne zu nagivieren, ohne diese von Anfang an wahrzunehmen. Außerdem haben Sie mit der *learning curve* und dem plot der Einträge von $\mathcal{Q}$ zwei Werkzeuge gelernt die Hyperparameter quantitativ festzulegen. In Aufgabe 3.4 haben Sie gesehen, dass der Algorithmus fähig ist sich mit unterschiedliche Strategien speziellen Umgebungsparametern anzupassen.

Obwohl das Problem und *reinforcement learning*-Modell in diesem angeleiteten Lernprojekt relativ simpel erscheint, bildet es die Basis für das Verständnis einer Vielzahl an Modellen und Algorithmen und bereits dieses Modell kann mit geringfügiger Modifikation Probleme lösen, deren Inhalt Abschlussarbeiten und aktueller Forschung würdig ist.
Einige denkbare Modifikationen dieses Modells beinhalten bspw. 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. Aufgrund des $\alpha \neq 0$ selbst nach Ende der Lernzeit passt der Algorithmus sich an bspw. langsam beweglichen 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 dieser Algorithmus sich auf *deep reinforcement learning* verallgemeinern, welches Anzahlen von Zuständen beinhaltet, die das Fassungsvermögen des menschlichen bei weitem übersteigen.