

BACHELORARBEIT

Abbildung von Learning-Algorithmen-Modellen in deren Reward-Funktion

David Müller

Entwurf vom 31. März 2021



BACHELORARBEIT

Abbildung von Learning-Algorithmen-Modellen in deren Reward-Funktion

David Müller

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien

Betreuer: Thomas Gabor
Thomy Phan

Abgabetermin: 1. Januar 2099



Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 1. Januar 2099

.....
(Unterschrift des Kandidaten)

Abstract

Für ML-Algorithmen gibt es eine Vielzahl von Modellen und Strategien, die genutzt werden können, um das Lernverhalten des Agenten zu kontrollieren und damit schlussendlich dessen Resultate zu verbessern. Bereits eine simple Erweiterung wie das Lernen auf einer Epsilon-Greedy-Policy fügt so schon neue Komponenten zum Lernalgorithmus hinzu. Wir untersuchen, inwieweit sich derartige Erweiterungen allein durch die Wahl der Reward-Funktion abbilden lassen und welche Auswirkungen dies auf das Lernverhalten sowie die Resultate des Agenten hat. Außerdem wird in diesem Zuge analysiert, welches eigentliche Ziel durch so eine Reward-Funktion umgesetzt wird. Für eine anschauliche Darstellung wird ein Landschaftsnavigationsproblem betrachtet, in dem der Agent in einem zufällig generierten Terrain den höchsten Gipfel finden soll.

Inhaltsverzeichnis

1 Einleitung	1
2 Navigations-Problem	3
2.1 Das Environment	3
2.2 Q-Learning-Experimente	5
2.2.1 Das Prinzip von Q-Learning	5
2.2.2 Der Ablauf von Q-Learning	8
2.2.3 Exploration vs Exploitation	9
2.2.4 Implementierung in Python	9
2.2.5 Experimente	12
2.3 Deep-Q-Learning Experimente	16
2.3.1 Das Prinzip von Deep-Q-Learning	16
2.3.2 Implementierung in Python	18
2.3.3 Erste Experimente	21
3 Luna-Lander	23
Abbildungsverzeichnis	25
Tabellenverzeichnis	27
Listings	29
Literaturverzeichnis	31

1 Einleitung

2 Navigations-Problem

Wir betrachten zunächst ein Landschaftsnavigationsproblem. Der Agent soll in einer zufällig generierten Landschaft unterschiedliche Aufgaben lösen.

2.1 Das Environment

Als Environment für diese Experimentreihe wollen wir eine Gebirgslandschaft erzeugen, über der ein Raster liegt, worauf sich der Agent bewegen kann. Hierbei soll jeder Punkt auf dem Raster eine Höhe besitzen. Außerdem soll die Landschaft zufällig generiert werden können.

Die simpelste Lösung hierfür wäre wohl, ein zweidimensionales Array mit zufälligen Zahlen zu füllen. Auf diese Weise erhält man ein für jede Koordinate eine zufällige Höhe. Wir wollen allerdings für die intuitive Auswertung der Experimente (zum Beispiel „Hat der Agent einen Berg gefunden?“) eine Landschaft erstellen, die organisch und natürlich aussieht.

Perlin Noise Um dieses Ziel zu erreichen verwenden wir *Perlin Noise* ([Par15]). Hierbei handelt es sich um eine Rauschfunktion, mit der sich sehr natürlich wirkende Texturen zufällig generieren lassen. Abbildung 2.1 zeigt eine simple Darstellung von zweidimensionalem Perlin Noise, bei der die generierten Werte über Farbwerten von Schwarz bis Weiß abgebildet werden.

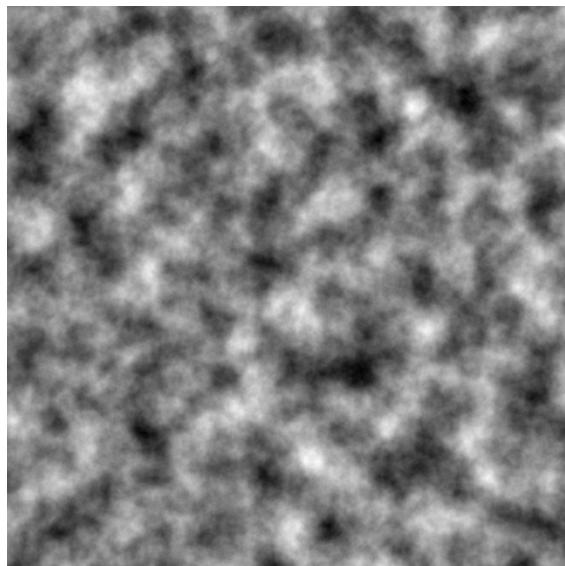


Abbildung 2.1: Visualisierung von zweidimensionaler Perlin Noise

Quelle: https://miro.medium.com/max/2400/1*vs239SecVBaB4HvLsZ805Q.png

2 Navigations-Problem

Perlin Noise ist nach [Par15] ein fundamentaler Algorithmus in der prozeduralen Generierung von Terrain und somit optimal geeignet, um unsere Umgebung zu erstellen. Wir verwenden eine modifizierte Implementierung von TODO, um ein zweidimensionales Array mit zufälligen Werten zwischen -1 und 1 zu erhalten, welche wir mit einer beliebigen Höhe multiplizieren können. Je nachdem, wie stark man in die Rauschfunktion „hereinzoomt“ erhält man unterschiedliche Verteilungen der Landschaft, wie man in Abbildung 2.2 erkennen kann.

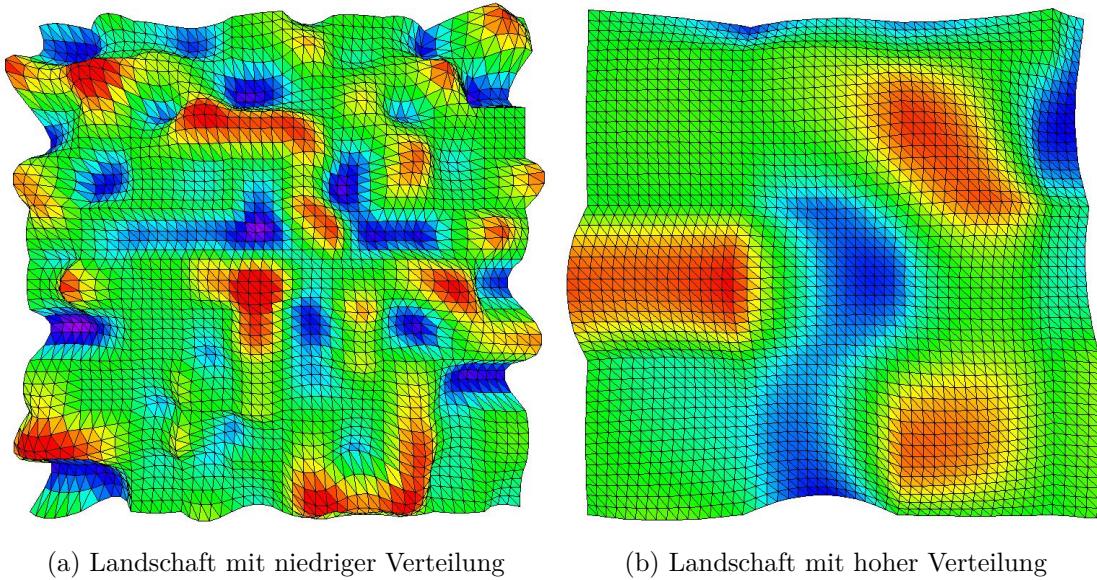


Abbildung 2.2: Mittels Perlin Noise zufällig generierte Landschaften

Wir werden nicht näher auf die Details der Funktion eingehen, da dies nicht Kern dieser Arbeit ist. Für weitere Ausführungen diesbezüglich verweisen wir auf [Arc11].

Für die Visualisierung der Landschaft benutzen wir eine abgewandelte Form des Codes von TODO. Zur besseren Differenzierung werden Berge und Täler zusätzlich zur perspektivischen Unterscheidung rot bzw. blau dargestellt.

Wir besitzen nun die Möglichkeit, eine zufällige Landschaft zu generieren und diese visuell darzustellen. Um bei allen Experimenten die gleichen Voraussetzungen zu gewährleisten, werden wir im Folgenden das mittels der eben beschriebenen Methode zufällig generierten Terrain benutzen, der in Abbildung 2.3 zu sehen ist. Der höchste Punkt befindet sich bei dieser Landschaft auf dem Berg ganz oben in der Mitte.

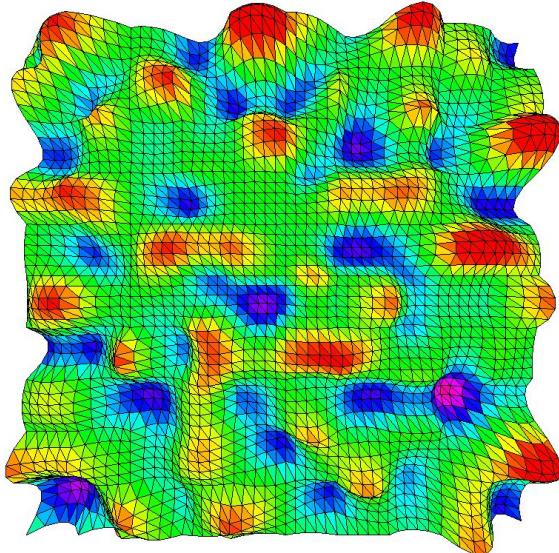


Abbildung 2.3: TODO Main Terrain

2.2 Q-Learning-Experimente

Um nun zu testen, ob die Landschaft für unsere Zwecke geeignet ist, werden wir einige Experimente durchführen. Wir wollen hierfür zunächst einen simplen Reinforcement-Learning-Agenten implementieren, welcher *Q-Learning* verwendet.

2.2.1 Das Prinzip von Q-Learning

Dieses Kapitel stützt sich zu einem Großteil auf das Buch *Reinforcement Learning: An Introduction, Second Edition* [SB20]. Falls nicht anders angegeben, wurden die Informationen hieraus entnommen.

Markov Decision Processes Alle folgenden Experimente zielen darauf ab, Probleminstanzen von *Markov Decision Processes* – oder kurz MDPs – zu lösen. In MDPs gibt es eine handelnde Instanz, den *Agenten*, welcher mit seinem Umfeld, der so genannten *Umgebung* interagiert. Diese Interaktion erfolgt Sequenziell in Zeitschritten $t = 0, 1, 2, 3, \dots$. Der Agent erhält in jedem Zeitschritt t eine Repräsentation seiner Umgebung, den Zustand S_t , und führt basierend darauf eine *Aktion* A_t aus. Für diese erhält er von der Umgebung eine Belohnung R_{t+1} , sowie einen Folgezustand S_{t+1} .

2 Navigations-Problem

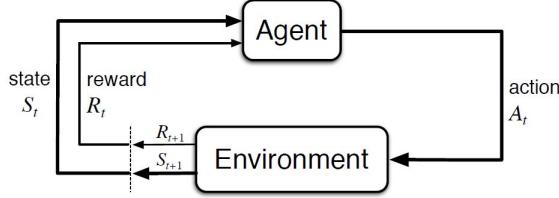


Abbildung 2.4: Interaktion zwischen Umgebung und Agent in einem MDP

Quelle: [SB20]

Ziel des Agenten ist es nun, seinen erwarteten Ertrag G_t zu maximieren. Im einfachsten Fall ist dieser die Summe aller Belohnungen:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T, \quad (2.1)$$

wobei T Zeitschritt einer Episode ist. In vielen Fällen ist die Interaktion zwischen Agent und Umgebung allerdings nicht endlich. Somit ist in diesen Fällen $T = \infty$ und der Ertrag, den der Agent maximieren soll, nach 2.1 unendlich. Wir führen deswegen das *discounting* ein. Für G_t ergibt sich hiermit:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= \sum_{k=0}^T \gamma^k R_{t+k+1}, \end{aligned} \quad (2.2)$$

wobei die *discount rate* γ ein Wert zwischen 0 und 1 ist. Eine Belohnung k Zeitschritte in der Zukunft ist also nur γ^{k-1} -mal so viel wert wie eine Belohnung, welche im aktuellen Zeitschritt erhalten wurde.

Policies Der Agent folgt zu jedem Zeitpunkt einer Policy π . Hierbei gibt $\pi(a|s)$ die Wahrscheinlichkeit dafür an, dass der Agent zum Zeitschritt t die Aktion $a \in A$ im Zustand $s \in S$ ausführt, also dass die Aktion $A_t = a$ wenn $S_t = s$. Hierbei ist S die Menge aller Zustände und A die Menge aller Aktionen. $\pi(a|s)$ ist also eine Wahrscheinlichkeitsverteilung über $a \in A(s)$ für jedes $s \in S$, wobei $A(s)$ alle möglichen Aktionen im Zustand s beschreibt.

State-Value Functions Wir benötigen nun eine Möglichkeit einzuschätzen, wie gut ein Zustand s ist, wenn wir der Policy π folgen. Hierfür nutzen wir die *state-value function* v_π . Diese beschreibt die erwartete Belohnung eines Zustands s unter der Policy π zum Zeitschritt t . Wir definieren $v_\pi(s)$ als

$$\begin{aligned} v_\pi(s) &\doteq E_\pi [G_t | S_t = s] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \end{aligned} \quad (2.3)$$

wobei E_π der Erwartungswert des Ertrags G_t nach 2.2 ist, wenn der Agent sich im Zustand $S_t = s$ befindet und der Policy π folgt.

Action-Value Functions Ähnlich hierzu gibt die *action-value function* q_π an, wie profitabel es für den Agenten ist, in einem gegebenen Zustand eine gewisse Aktion auszuführen, wenn der Agent der Policy π folgt.

Der Wert einer Aktion a im Zustand s unter der Policy π ist also die erwartete Belohnung, wenn man im Zustand s zum Zeitschritt t die Aktion a ausführt. Wir definieren $q_\pi(s, a)$ als

$$\begin{aligned} q_\pi(s, a) &\doteq E_\pi [G_t | S_t = s, A_t = a] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \end{aligned} \quad (2.4)$$

Die action-value function wird auch als Q-function bezeichnet, welche als Ergebnis für ein state-action Paar die Q-value liefert. Für die folgenden Implementierungen ist diese von großer Wichtigkeit.

Optimale Policies und Optimale Value Functions Das Ziel des Agenten ist, die optimale Policy π für ein Markov Decision Problem zu finden. Ist dieses Ziel erreicht so lässt sich sagen, dass die Reinforcement Learning Aufgabe erfüllt ist. Optimal ist hierbei die Policy, welche nach Aufsummieren der Belohnungen über alle Schritte einer Episode die beste gesamte Belohnung liefert. Eine Policy π ist also besser als Policy π' , wenn die erwartete Belohnung von π für **alle** Zustände $s \in S$ größer ist als die von π' . [SB20] verwendet die Formulierung

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in S. \quad (2.5)$$

Es gibt immer eine Policy, die besser als oder gleichwertig mit allen anderen Policies ist. Diese wird beziehungsweise werden als π_* bezeichnet. Die besten Policies besitzen die gleich state-value function, welche die *optimale state-value function* v_* genannt wird und definiert wird als

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s) \quad (2.6)$$

für alle $s \in S$.

Optimale Policies teilen sich ebenfalls die gleiche *optimale action-value function* q_* , welche definiert ist als

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a) \quad (2.7)$$

für alle $s \in S$ und $a \in A$. q_* liefert also für jedes state-action Paar den größtmöglichen erwarteten Ertrag, den irgendeine Policy erreichen kann.

Bellman Optimality Equation Die optimale action-value function q_* muss die folgende Gleichung erfüllen:

$$q_*(s, a) = E \left[R_{t+1} + \gamma \max_{a'} q_*(s', a') \right] \quad (2.8)$$

2 Navigations-Problem

Diese Gleichung wird *Bellman optimality equation* für q_* genannt und besagt, dass der beste erwartete Ertrag für jedes state-action Paar (s, a) zum Zeitpunkt t der Summe aus der direkten Belohnung R_{t+1} der Aktion a und dem **maximalen** erwarteten Ertrag, der von einem der nächsten state-action Paare (s', a') erreicht werden kann entsprechen muss. Hierbei ist s' der Folgezustand S_{t+1} und a' die Aktion $A_{t+1} \in A(s')$, welche den meisten Ertrag bringt.

Das folgende Kapitel beschreibt, wie die Bellman equation verwendet wird, um q_* zu finden, was uns wiederum die optimale Policy liefern soll.

2.2.2 Der Ablauf von Q-Learning

Das Ziel von Q-Learning ist, die optimale Policy zu finden, indem der Agent die optimalen Q-values für jedes state-action Paar erlernt.

Der Q-Learning Algorithmus benutzt die Bellman equation als Update-Regel, um nach und nach die Q-values für jedes state-action Paar anzunähern. Dieses Verfahren nennt man *value iteration*.

Bei überschaubaren Umgebungen ist es möglich, die Werte für jedes state-action Paar in einer Tabelle, der so genannten *Q-table* zu speichern. Zu Beginn weiß der Agent nichts über eine Umgebung. Die Q-table ist dementsprechend leer beziehungsweise ist der Wert jedes state-action Paars 0. Der Agent operiert nun eine vorbestimmte Anzahl von *Episoden* in der Umgebung und produziert im Laufe der Zeit neue Q-values, mit denen die Q-table aktualisiert wird.

Zu Beginn jedes Schritts – auch *step* genannt – wählt der Agent eine Aktion für den aktuellen Zustand aus. Intuitiv macht es Sinn, die beste bisher bekannte Aktion zu wählen, um die Belohnung zu maximieren. Dieses Vorgehen ist allerdings nicht zielführend, da der Agent ja am Anfang nichts über seine Umgebung weiß. Er benötigt also für die Wahl seiner Aktionen eine bessere Strategie. Auf dieses Problem gehen wir in Kapitel 2.2.3 näher ein.

Nehmen wir an, der Agent hat im Zustand s zum Zeitschritt t eine Aktion a ausgewählt. Nach der Bellman equation 2.8 ist dann die Q-value $q(s, a)$ (der Übersicht in Gleichung 2.10 wegen wird die Policy π hier weggelassen) die für die Aktion erhaltene Belohnung R_{t+1} plus der maximale erwartete Ertrag eines folgenden state-action Paars, also

$$q(s, a) = R_{t+1} + \gamma \max_{a'} q(s', a'). \quad (2.9)$$

Dies berücksichtigt allerdings nicht, dass der Agent in einem früheren Zeitschritt oder in einer anderen Episode vielleicht bereits einen Wert $q(s, a)$ für dieses state-action Paar berechnet und in der Q-table gespeichert hat. So wird bei jeder Berechnung eventuell ein alter Wert überschrieben und vergangene Erkenntnisse haben keinen Einfluss auf die aktuelle Berechnung.

Ein besserer Ansatz ist die Verwendung einer *learning rate*. Die learning rate ist ein Wert zwischen 0 und 1, der festlegt, wie schnell der Agent vergangene Q-values aus der Q-table verwirft. Anders gesagt legt sie fest, wie viel Information aus vorherigen Berechnungen bei einem Update einer Q-value erhalten bleibt. Wir verwenden für die learning rate das Symbol α .

Für die Berechnung der neuen Q-value für das state-action Paar (s, a) zum Zeitpunkt

t ergibt sich dann

$$q_{\text{neu}}(s, a) = (1 - \alpha)q(s, a) + \alpha \left(R_{t+1} + \gamma \max_{a'} q(s', a') \right). \quad (2.10)$$

Bei einer learning rate von $\alpha = 0.6$ bleiben so 40% des alten Wertes erhalten, während der neu erlernte Wert mit 60% gewichtet wird.

2.2.3 Exploration vs Exploitation

In Kapitel 2.2.2 sind wir auf die Notwendigkeit einer Strategie, mit der der Agent seine nächste Aktion auswählt, gestoßen. Wie dort bereits erwähnt ist eine sehr simple Methode die Auswahl der Aktion mit der größten erwarteten Belohnung. Eine solche Aktion wird *greedy* Aktion genannt. Gibt es mehrere greedy Aktionen mit demselben erwarteten Ertrag, so wird eine davon zum Beispiel per Zufall ausgewählt.

Diese Strategie klingt auf den ersten Blick sinnvoll, ist aber nicht so zielführend wie es scheint. Der Agent versäumt es andere Aktionen auszuprobieren, die eine bessere Belohnung liefern könnten. Er nutzt nur die ihm bekannten aus (engl. *exploitation*). Besser wäre es, wenn er ebenfalls Zeit in die Erkundung (engl. *exploration* der Umgebung stecken würde.

Dies kann realisiert werden, indem der Agent die meiste Zeit „gierig“ (engl. *greedy*) agiert und die Aktion mit dem besten geschätzten Ertrag wählt, mit einer Wahrscheinlichkeit von ϵ allerdings ab und zu zufällig eine von allen verfügbaren Aktionen auswählt. ϵ ist hierbei ein Wert zwischen 0 und 1, der entweder statisch oder dynamisch definiert wird. Auf diese Weise wird erreicht, dass der Agent auch Aktionen ausprobieren kann, welche er zuvor noch nicht gesehen hat. Methoden, welche nach diesem Schema agieren, werden ϵ -*greedy* Methoden genannt und zählen nach [DOB20] auch heute noch bei der Erkundung der Umgebung zu den am meisten benutzten.

2.2.4 Implementierung in Python

Mit diesem Wissen werden wir nun einen Q-Learning Algorithmus in Python implementieren.

Hyperparameter In den vorherigen Kapiteln haben wir einige Variablen eingeführt, von denen uns manche als so genannte *Hyperparameter* dienen werden [Rav18]. Diese steuern das Verhalten des Agenten und sollten für den optimalen Lernerfolg angepasst werden. Wir verwenden hierfür eine selbst definierte Datenklasse, um alle Hyperparameter an zentraler Stelle verwalten zu können:

```
@dataclass
class Parameters:
    num_episodes: int
    max_steps_per_episode: int

    learning_rate: float
    discount_rate: float
```

2 Navigations-Problem

```
start_exploration_rate: float
max_exploration_rate: float
min_exploration_rate: float
exploration_decay_rate: float

rewards_all_episodes: list
max_rewards_all_episodes: list
```

`num_episodes` gibt die Anzahl der Episoden an, die der Agent trainieren soll, `max_steps_per_episode` die Schritte pro Episode. `learning_rate` und `discount_rate` sind selbsterklärend. Die folgenden vier Werte beziehen sich auf die ϵ -greedy Strategie. Wir wollen die Möglichkeit haben, unser ϵ dynamisch anzupassen. Hierfür initialisieren wir die `start_exploration_rate` als unser Anfangs- ϵ , die `max_exploration_rate` als Absicherung und eventuelle Variable für die Zukunft (ist normalerweise identisch mit der `start_exploration_rate`), die `min_exploration_rate` als minimales ϵ und die `exploration_decay_rate` als Größe die festlegt, wie schnell ϵ schrumpfen soll. Hierzu in den folgenden Kapiteln (TODO) mehr. In den beiden Variablen `rewards_all_episodes` und `max_rewards_all_episodes` werden die Belohnungen des Trainings abgelegt.

Die `train()`-Methode (TODO Code reference). Die `train()`-Methode ist das Herzstück des Algorithmus. Sie besitzt die folgenden Parameter:

```
def train(self, width: int, length: int, params: Parameters,
          environment, visualize=False, plot=False, plot_interval=1,
          plot_moving_avg_period=100):
```

`width` und `length` beschreiben die Breite und die Länge des Rasters aus 2.3, sprich die Größe der Landschaft. Mit diesen Daten wird die Größe der Q-table bestimmt. Mit den `params` übergeben wir der Funktion die Hyperparameter. Das `environment` ist die Umgebung des Agenten (TODO genauer). Die restlichen Parameter sind optional und beziehen sich auf die Visualisierung der Ergebnisse während des Trainings.

Wir verwenden für die Implementierung der Q-table *NumPy*, die primäre Bibliothek für die Array-Programmierung in Python [HMvdW⁺20]. Wir erzeugen ein zweidimensionales NumPy-Array, das für jeden Zustand unserer Umgebung eine Zeile und für jede Aktion (in unserem Fall die Bewegung nach oben, rechts, unten und links) eine Spalte enthält. Alle Elemente werden zunächst mit 0 initialisiert. Außerdem setzen wir unser ϵ auf die in den Hyperparametern festgelegte `start_exploration_rate`. Wir erstellen außerdem einen Buffer, welcher die Tupel aus Zustand, Aktion, Belohnung und Folgezustand enthält. Dieser wird am Ende jeder Episode gemischt und dann abgearbeitet. Dieses Verfahren löst nach TODO starke Pfadabhängigkeiten auf. In Kapitel 2.3.1 gehen wir hierauf näher ein.

```
q_table = np.zeros((width * length, 4))
exploration_rate = params.start_exploration_rate
buffer = []
```

Zu Beginn jeder Episode setzen wir den Zustand auf den Startzustand der Umgebung und erzeugen die beiden Variablen, die die Belohnungen der Episode speichern:

```

for episode in range(params.num_episodes):
    state = environment.reset_agent()
    rewards_current_episode = 0
    max_reward_current_episode = 0

```

In jedem Zeitschritt wenden wir für die Wahl der Aktion unsere ϵ -greedy Strategie an. Hierfür erzeugen wir eine zufällige Zahl zwischen 0 und 1. Falls diese größer ist als unser aktuelles ϵ , wählt der Agent die beste bekannte Aktion, ansonsten wird aus den möglichen Aktionen zufällig eine ausgewählt. Die Umgebung liefert uns infogedessen den Folgezustand und die erhaltene Belohnung. Anschließend speichern wir das Tupel im Buffer, aktualisieren den Zustand, speichern die Belohnungen und zeigen ggf. die Position des Agenten an:

```

for step in range(params.max_steps_per_episode):
    exploration_rate_threshold = random.uniform(0, 1)
    if exploration_rate_threshold > exploration_rate:
        action = np.argmax(q_table[state, :])
    else:
        action = random.choice(
            environment.get_agent_possible_actions())
    new_state, reward, _ = environment.agent_perform_action(action)
    sars = (state, action, reward, new_state)
    buffer.append(sars)

    q_table[state, action] = (1 - params.learning_rate) * \
        q_table[state, action] + params.learning_rate * (reward + \
        params.discount_rate * np.max(q_table[new_state, :]))

    state = new_state
    rewards_current_episode += reward
    if max_reward_current_episode < reward:
        max_reward_current_episode = reward

    if visualize:
        environment.redraw_agent()
        time.sleep(0.04)

```

Am Ende jeder Episode aktualisieren wir die entsprechenden Einträge in der Q-table mit den Daten aus dem Buffer. Hierfür wird die Gleichung für die Berechnung der Q-value 2.10 angewendet:

```

random.shuffle(buffer)
while len(buffer) > 0:
    (state, action, reward, new_state) = buffer.pop(0)
    q_table[state, action] = (1 - params.learning_rate) * \
        q_table[state, action] + params.learning_rate * (reward + \
        params.discount_rate * np.max(q_table[new_state, :]))

```

2 Navigations-Problem

Außerdem wird das neue ϵ berechnet. Wir verwenden hierfür eine exponentielle Funktion, damit ϵ am Anfang start abfällt und gegen Ende langsamer. Zuletzt werden die Belohnungen in den Params gespeichert und ggf. als Graph angezeigt.

```
exploration_rate = params.min_exploration_rate +\
    (params.max_exploration_rate - params.min_exploration_rate) *\n    np.exp(-params.exploration_decay_rate * episode)\n\n    params.rewards_all_episodes.append(rewards_current_episode)\n    params.max_rewards_all_episodes.append(max_reward_current_episode)\n    if plot and episode % plot_interval == 0:\n        plot_progress(params.rewards_all_episodes, exploration_rate, plot_moving_avg\n\nreturn q_table, params
```

2.2.5 Experimente

Nachdem der Agent implementiert ist, wollen wir diesen in unserer Umgebung testen. Wir verwenden die zuvor beschriebene Landschaft 2.3. Ziel ist es, dass der Agent den höchsten Gipfel erreicht. Zu diesem Zweck liefert die Umgebung als Belohnung die Differenz der Höhe des alten und neuen Zustands. Wenn sich der Agent also von einem Feld mit der Höhe 2.3 in ein Feld mit der Höhe 1.8 bewegt erhält er als Belohnung -0.5 .

Einzelnes Experiment Nach einigem Ausprobieren haben sich die folgenden Hyperparameter als solche erwiesen, die gute Ergebnisse erzielen:

```
params = Parameters(\n    num_episodes=10000,\n    max_steps_per_episode=300,\n    learning_rate=0.6,\n    discount_rate=0.99,\n    start_exploration_rate=1,\n    max_exploration_rate=1,\n    min_exploration_rate=0.01,\n    exploration_decay_rate=0.00015,\n    # ... Rest wird erst während des Trainings belegt\n)
```

Wir stellen die Ergebnisse in einem Graph da. Nach einem Trainingdurchlauf erhält man die in 2.5a dargestellte Ausgabe. Die x-Achse stellt die aktuelle Episode dar, während die y-Achse die erhaltene Belohnung, bzw. für die türkise Linie das ϵ angibt. Die blaue Linie, welche aufgrund der großen Menge an unterschiedlichen Werten kaum mehr als solche zu erkennen ist, zeigt die Summe der Belohnungen aus allen Zeitschritten für jede Episode an. Die orange Linie ist der Durchschnitt der letzten 100 Gesamtbewertungen pro Episode. Dieser Wert wird auch als *moving average* bezeichnet. Die Werte von Episode 0 bis 99 sind hier mit 0 belegt. Die türkise Linie zeigt das ϵ zu jeder Episode. Die Beschriftung hierfür befindet sich auf der rechten Seite des Graphen.

Es lässt sich an der orangen Linie gut erkennen, wie der Agent mit der Zeit immer bessere Belohnungen erhält.

Lässt man den Agenten nun die im Training erzeugt Q-table verwenden, um die beste Aktion für jeden Zeitschritt auszuwählen, so folgt er dem in 2.5b sichtbaren Pfad. Er findet also den höchsten Berg in der gegebenen Landschaft, obwohl dieser weit entfernt vom Startpunkt in der Mitte und hinter einem Graben liegt.

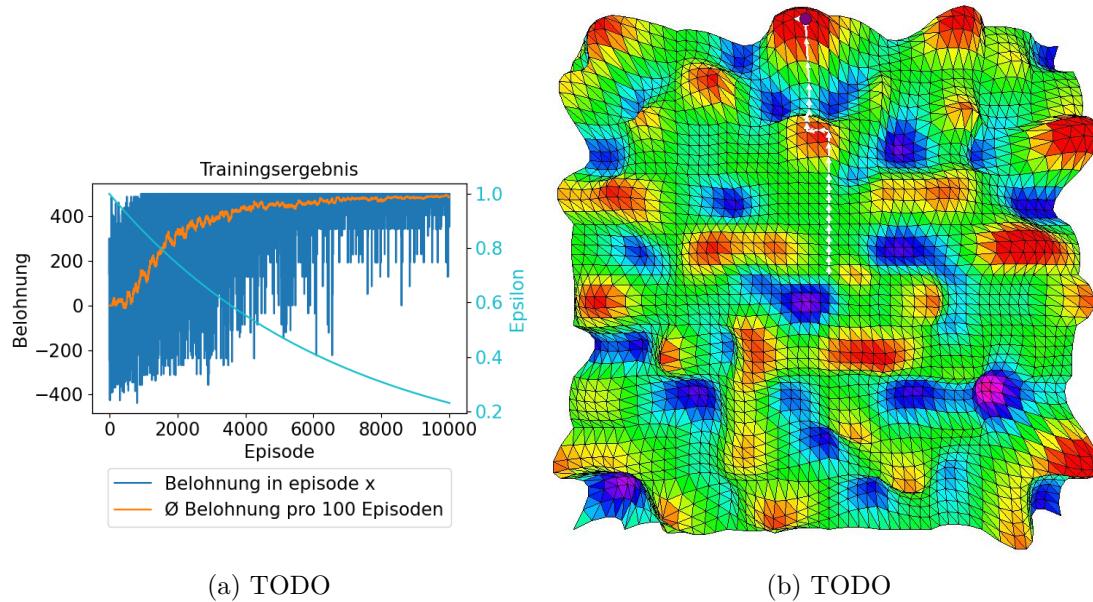


Abbildung 2.5: Ergebnisse des ersten Experiments

Eine weiter interessanter Wert ist die Anzahl der mit 0 belegten Einträge in der Q-table. Diese besitzen entweder zufällig den errechneten Q-value 0 oder wurden vom Agenten nicht berechnet. Da die meisten Einträge im 14-Stelligen Nachkommabereich liegen, ist letzteres relativ unwahrscheinlich und so lässt sich sagen, dass die Summe der mit 0 belegten Einträge ungefähr der Summe der nicht erkundeten Zustände entspricht. In Fall des aktuellen Experiments sind 850 der 10000 Einträge mit 0 belegt. Der Agent hat also ungefähr 91.5% der Umgebung erkundet.

Dies ist nur ein einzelnes Experiment und hat natürlich keine statistische Aussagekraft. Es diente lediglich der Demonstration und der Erklärung der Visualisierung. Wir werden im Folgenden testen, welche Auswirkung die Verwendung der ϵ -greedy Strategie auf den Lernprozess hat.

Vergleich des Trainings mit und ohne ϵ -greedy Strategie Um eine aussagenkräftigere Datengrundlage zu erhalten, werden wir die folgenden Experimente jeweils 20 mal wiederholen. Diese Zahl hat sich als ein gutes Mittelmaß zwischen einer ausreichenden Menge an Daten für die Statistik und der Berechenbarkeit in zumutbarer Zeit erwiesen.

Die erste Experimentreihe erfolgt mit den gleichen Parametern wie im vorherigen Experiment. Für die zweite Experimentreihe setzen wir lediglich ϵ auf 0. Das kommt dem Weglassen der ϵ -greedy Strategie gleich und bedeutet, dass der Agent in jedem Fall greedy agiert und die beste Aktion wählt. Dies soll die Notwendigkeit von ϵ für ein besseres

2 Navigations-Problem

Trainingsergebnis zeigen.

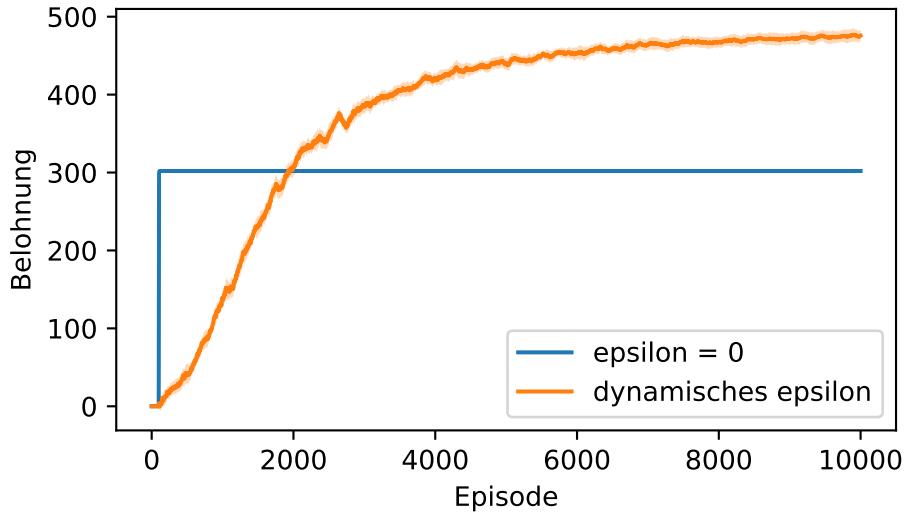


Abbildung 2.6: Vergleich des Lernerfolgs mit und ohne ϵ -greedy Strategie

Die Achsen von Graph 2.6 sind bis auf das Fehlen der ϵ -Achse identisch mit dem aus 2.5a. Die beiden Linien zeigen jeweils den Durchschnitt der moving average Werte aller 20 Experimentiterationen. Der leicht transparente Bereich um die Linien herum ist die Standartabweichung in der jeweiligen Episode. Es lässt sich hier sehr deutlich erkennen, dass der Agent ohne eine Erkundungsstrategie wie ϵ -greedy (blaue Linie) zu Beginn einen relativ lukrativen Pfad findet, diesen aber dann auch nicht mehr verlässt, um andere Pfade zu erkunden und so immer die gleiche Belohnung bekommt. Er wird schließlich vom ϵ -greedy Agenten (orange Linie) überholt, da dieser seine Umgebung erkundet. Dieser erhält am Ende des Trainings wesentlich höhere Belohnungen.

Betrachten wir den Durchschnitt der Anzahl der mit 0 belegten Einträge der Q-tables beider Experimentreihen lässt sich abschätzen, dass der ϵ -greedy Agent im Schnitt 89.9% der Umgebung erkundet hat, während es beim Agenten ohne Erkundungsstrategie gerade einmal 1.1% sind.

Dies zeigt, dass eine Erkundungsstrategie für den Erfolg des Agenten sehr wichtig ist.

Erkundungsstrategie codiert im Reward Für das nächste Experiment lassen wir der Agenten ebenfalls in jedem Zeitschritt greedy agieren. Diesmal erreichen wir dies, indem wir unabhängig vom aktuellen ϵ immer die beste Aktion auswählen. Der Agent soll allein durch die Veränderung der Belohnung dazu gebracht werden, seine Umgebung besser zu erkunden und trotzdem einen möglichst hohen Punkt zu finden.

Wir modifizieren hierfür die nach jeder Aktion von der Umgebung erhaltene Belohnung wie folgt:

```
new_state, actual_reward, _ = environment.agent_perform_action(action)

reward = ((1 - exploration_rate) * actual_reward) - exploration_rate
```

```
sars = (state, action, reward, new_state)
buffer.append(sars)
```

Die `exploration_rate` verhält sich hierbei genau so wie beim Experiment davor. Diese Formel soll bewirken, dass der Agent zu Beginn bei einer hohen `exploration_rate` alle besuchten Felder mit einem negativen Wert belegt, sodass er beim nächsten mal andere Felder besucht und so seine Umgebung erkundet. Nach und nach wird diese Belegung dann immer mehr mit den mittels korrekter Belohnungen ermittelten Q-values ersetzt, wodurch sich der Agent auf die besten Zustände einpendeln soll. Wir setzen die learning rate auf 1, damit der Agent nicht an den zu Beginn verfälschten Belohnungen festhält. Nach 50000 Episoden erhält man folgendes Ergebnis:

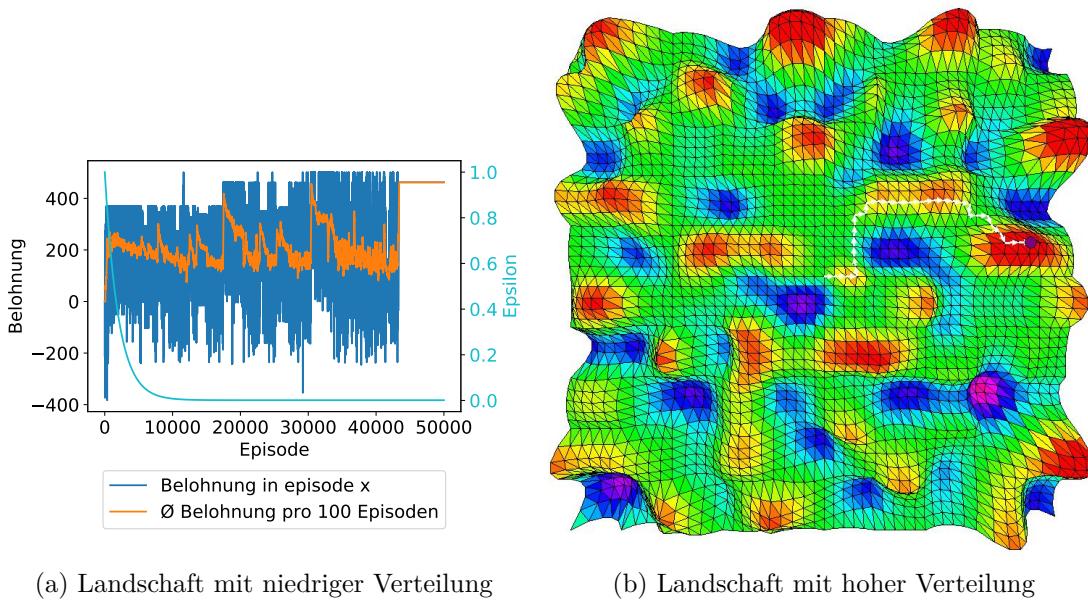


Abbildung 2.7: Ergebnisse des ersten Experiments

Wichtig ist es an dieser Stelle zu erwähnen, dass der Graph 2.7a die unverfälschte, von der Umgebung gelieferte Belohnung vor der Modifikation zeigt, da uns der tatsächliche Lernfortschritt des Agenten interessiert und man diesen sonst nicht mit den Ergebnissen anderen Experimente vergleichen könnte. Es fällt deutlich auf, dass der Lernprozess hier anders verläuft als in 2.5a. Wir erhalten keine saubere Lernkurve. Trotzdem erreicht der Agent einen maximalen moving average von etwas über 462. Zum Vergleich: Der maximale moving average von 2.5a liegt bei etwas über 497. Die Q-table dieses Experiments enthält 850 von 10000 mit Null belegte Einträge.

Abbildung 2.7b zeigt den Pfad des Agenten bei Verwendung der erzeugten Q-table. Er findet zwar nicht den höchsten Punkt, erklimmt aber dennoch einen hohen Berg, welcher sich nicht in unmittelbarer Nähe des Startzustands befindet. Die Strategie hat also zur besseren Erkundung der Umgebung beigetragen.

Wiederholt man das Experiment 20 mal, so lässt sich der folgende Durchschnitt mit Standardabweichung berechnen:

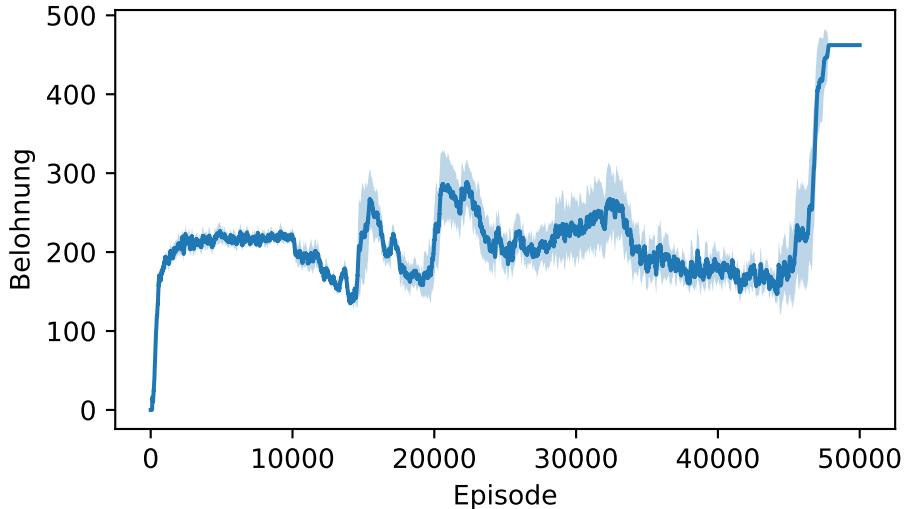


Abbildung 2.8: TODO

Der Agent benötigt mit 50000 Episoden sehr lange, um seinen Höchstwert zu erreichen. Dieser scheint sich auch nach circa Episode 47 nicht mehr zu verändern, was vermutlich darauf zurückzuführen ist, dass der Agent immer greedy agiert und zu diesem Zeitpunkt kein besserer ihm bekannter Pfad mehr existiert. Verglichen mit dem Agenten ohne Erkundungsstrategie lässt sich festhalten, dass die Modifikation der Belohnung in diesem Fall eine höheren Ertrag sowie eine bessere Erkundung der Umgebung bewirkt hat. Diese Strategie dauert allerdings deutlich länger und liefert etwas weniger Ertrag als die klassische ϵ -greedy Strategie.

2.3 Deep-Q-Learning Experimente

2.3.1 Das Prinzip von Deep-Q-Learning

Bisher haben wir alle Q-Values in einer Q-table gespeichert. Dieses Vorgehen ist relativ simpel, stößt aber nach [Lap18] bei großen Zustands- bzw. Aktionsräumen schnell an seine Grenzen. Als Beispiel wird hier Q-Learning bei Atari-Spielen aufgeführt, bei denen die Pixel als Zustände benutzt werden. Es leuchtet ein, dass in so einem Fall aufgrund der großen Menge an state-action Paaren die Speicherung deren Q-values schwierig, wenn nicht sogar unmöglich ist.

Nach [Lap18] ist die Benutzung von Neuronalen Netzen eine der populärsten Methoden, um mit diesem Problem umzugehen. Wir kombinieren Q-Learning mit einem Neuronalen Netz und erhalten auf diese Weise ein so genanntes *Deep-Q-Network (DQN)*.

Für das stabile und effiziente Trainieren eines DQNs gibt es nach [Lap18] einige Techniken und Tricks. Die Informationen diesbezüglich wurden – falls nicht anders angegeben – aus [Lap18] entnommen.

ϵ -greedy Die ϵ -greedy Strategie löst das Exploration versus Exploitation Dilemma. Da wir uns hiermit bereits in Kapitel 2.2.3 befasst haben, halten wir uns an dieser Stelle

nicht weiter damit auf.

Replay buffer Die Daten, die der Agent im Laufe des Trainings sammelt, sind nicht unabhängig voneinander. Sie liegen höchst wahrscheinlich sehr nahe beieinander, da sie meist zur selben Episode gehören. Dazu kommt, dass die Verteilung der Daten durch die aktuelle Policy, beziehungsweise bei der Verwendung von ϵ -greedy teilweise zufällig bestimmt wird. Wünschenswert wäre eine Verteilung der Trainingsdaten identisch zu Stichproben unter Verwendung der optimalen Policy, die wir erlernen wollen.

Um dem entgegenzuwirken verwenden wir einen großen Speicher, welcher unsere vergangenen Beobachtungen enthält. Anstatt nun mit den letzten Beobachtungen zu trainieren, entnehmen wir zufällig Daten aus diesem Speicher. Diese Methode nennt man *replay buffer*. In Kapitel 2.2.4 verwenden wir ebenfalls einen solchen replay buffer.

Target network Ein ähnliches Problem stellt der Zusammenhang zwischen benachbarten Schritten dar. Die Bellman equation aus 2.8 besagt, dass wir den Wert von $q_\pi(s, a)$ über $q_\pi(s', a')$ berechnen können. Die Zustände s und s' sind allerdings nur einen Schritt voneinander entfernt. Sie sind also sehr ähnlich und vom Neuronalen Netz schwer zu unterscheiden. Das hat zur Folge, dass wir bei einem Update der Parameter des Netzes für die Annäherung von $q_\pi(s, a)$ an das gewünschte Resultat indirekt auch den Wert für $q_\pi(s', a')$ verändern können. Dies kann das Training sehr instabil machen.

Deshalb verwenden wir ein so genanntes *target network*. Das target network ist eine Kopie des training networks – bei uns später auch policy network genannt –, welches lediglich alle N Schritte oder Episoden synchronisiert wird. N ist hierbei ein weiterer Hyperparameter, den wir bei später als `target_update` bezeichnen. Mit dieser Kopie besitzen wir nun fixierte Werte für $q_\pi(s', a')$, was das Training wesentlich stabiler machen sollte.

Der Trainingsablauf von DQN Wir betrachten nun einen klassischen Algorithmus für DQN. [Lap18] entnimmt dessen Schritte den bekannten Papern *Playing Atari with Deep Reinforcement Learning* [MKS⁺13] und *Human-Level Control Through Deep Reinforcement Learning* [MKS⁺15]. Sinngemäß wiedergegeben ist der Ablauf nach [MKS⁺15] wie folgt:

1. Initialisieren der replay memory Kapazität
2. Initialisieren des Hauptnetzes mit zufälligen Parametern
3. Kopie des Hauptnetzes anlegen, das target network
4. *For each Episode do*
 - a) Startzustand initialisieren
 - b) *For each Zeitschritt do*
 - i. Auswahl einer Aktion via ϵ -greedy
 - ii. Ausführen der Aktion in einem Emulator
 - iii. Beobachten der Belohnung und des Folgezustands
 - iv. Speichern der Beobachtung im replay memory
 - v. Zufällige Auswahl einer Reihe von Beobachtungen (batch) aus dem replay memory
 - vi. Vorverarbeitung der Zustände

2 Navigations-Problem

- vii. loss (TODO) zwischen Q-values und Ziel-Q-values berechnen (Benutzung des target networks für den Folgezustand)
- viii. Aktualisieren der Gewichte im Netz, um den loss zu minimieren
- ix. Alle N Episoden wird das target network mit dem Hauptnetz synchronisiert

In unserem Fall kommt noch ein weiterer Schritt hinzu, in dem wir das aktuelle Netz kopieren und als `best_net` speichern, wenn der moving average einen neuen Höchstwert erreicht. Dieses Netz wird dann am Ende des Trainings zurückgegeben. Dies soll sicherstellen, dass zum Schluss das beste Trainingsergebnis ausgegeben wird, auch wenn der Agent im Laufe der Zeit Sachen wieder verlernt hat. Wir ergänzen also:

4. (Fortsetzung)
 - c) Bei neuer Höchstleistung des Trainings Synchronisation des Ausgabenetzes mit dem Hauptnetzes

2.3.2 Implementierung in Python

Wir wollen nun einen DQN-Agenten in Python implementieren. PyTorch ist eine beliebtes Deep-Learning-Framework, welches Benutzerfreundlichkeit und Leistung vereint [PGM⁺19]. Als Grundlage verwenden wir das Codebeispiel unter https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/DeepQLearning/torch_deep_q_model.py (Zugriff am 30.03.2021), welches für unsere Zwecke stark modifiziert wird. Das Zentrum des Geschehens ist wieder die `train()`-Methode. Der besseren Übersicht wegen sind hier einige weniger relevante Codeausschnitte herausgekürzt (gekennzeichnet mit ...). Außerdem wurde die Einrückung auf der Ebene der Methode entfernt. Die Schritte sind entsprechend der Liste aus 2.3.1 nummeriert:

```
def train(width: int, length: int, params, environment, ...):  
    agent = Agent( ... ) # 1. bis 3.  
    scores, eps_history = [], []  
    max_average = -99999  
    for episode in range(params.num_episodes): # 4.  
        score = 0  
        environment.reset_agent() # a)  
        observation = environment.get_state_for_deep_q(step=0, ... ) # a)  
        for step in range(params.max_steps_per_episode): # b)  
            action = agent.choose_action(observation) # i.  
            state, reward, done = environment.agent_perform_action(  
                action, ... )  
            # ii. und iii.  
            observation_ = environment.get_state_for_deep_q(step=step, ... )  
            # iii.  
            score += reward  
            agent.store_transition(  
                observation, action, reward, observation_, done  
            ) # iv.
```

```

agent.learn(episode) # v. bis ix.)

observation = observation_
# ... falls gewünscht Position des Agenten anzeigen
# ... falls gewünscht Pfad des Agenten anzeigen
scores.append(score)
agent.exploration_rate = params.min_exploration_rate +\
    (params.max_exploration_rate - params.min_exploration_rate) *\ 
    np.exp(-params.exploration_decay_rate * episode)
# ... falls gewünscht Trainingsfortschritt als Graph ausgeben
current_average = get_current_average( ... )
if max_average < current_average or episode == plot_moving_avg_period:
    max_average = current_average
    agent.best_net.load_state_dict(agent.policy_net.state_dict())# c)
params.rewards_all_episodes = scores
params.max_reward_average = max_average
return agent.best_net, params

```

Die Schritte 1. bis 3. passieren bei der Initialisierung des Agenten und sind relativ unspektakulär. Interessanter ist die `learn()`-Methode des Agenten, die in jedem Zeitschritt einmal aufgerufen wird und die Schritte v. bis ix. abdeckt. Wir werfen daher einen blick auf deren Code. Auch hier wurde aus Platzgründen die Einrückung auf der Ebene der Methode entfernt:

```

def learn(self, episode):
    if self.memory_counter < self.batch_size:
        return # return, falls noch nicht genügend Beobachtungen existieren
    self.policy_net.optimizer.zero_grad()

    max_mem = min(self.memory_counter, self.mem_size)
    batch = np.random.choice(max_mem, self.batch_size, replace=False) # v.

    batch_index = np.arange(self.batch_size, dtype=np.int32)
    state_batch =\
        T.tensor(self.state_memory[batch]).to(self.policy_net.device)
    new_state_batch =\
        T.tensor(self.new_state_memory[batch]).to(self.policy_net.device)
    reward_batch =\
        T.tensor(self.reward_memory[batch]).to(self.policy_net.device)
    terminal_batch =\
        T.tensor(self.terminal_memory[batch]).to(self.policy_net.device)
    action_batch = self.action_memory[batch]
    # vvi. Start

    q_eval = self.policy_net.forward(state_batch)[batch_index, action_batch]
    q_next = self.target_net.forward(new_state_batch)
    q_next[terminal_batch] = 0.0
    q_target = reward_batch + self.gamma * T.max(q_next, dim=1)[0]
    loss = self.policy_net.loss(q_target, q_eval).to(self.policy_net.device)

```

2 Navigations-Problem

```

loss.backward()
# vvi. Ende
self.policy_net.optimizer.step() # viii.

if episode % self.target_update == 0:
    self.target_net.load_state_dict(self.policy_net.state_dict()) # ix.

```

Zuletzt betrachten wir noch die Klasse DeepQNetwork, welche das DQN modelliert:

```

class DeepQNetwork(BasicNetwork):
    def __init__(self, learning_rate, input_dims, fc1_dims, fc2_dims,
                 n_actions):
        super(DeepQNetwork, self).__init__()
        self.learning_rate = learning_rate
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions

        self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims)
        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        self.out = nn.Linear(self.fc2_dims, self.n_actions)

        self.optimizer = optim.Adam(self.parameters(),
                                   lr=learning_rate)
        self.loss = nn.MSELoss()

        self.device = \
            T.device('cuda' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = T.sigmoid(self.fc1(state))
        x = T.sigmoid(self.fc2(x))
        actions = self.out(x)
        return actions

```

Wir verwenden in unserem DQN zwei sogenannte Fully-connected hidden Layers. Das bedeutet, dass alle Neuronen einer Schicht mit allen Neuronen der nächsten Schicht verknüpft sind. Die erste Ebene nimmt Eingaben mit den Dimensionen `input_dims` entgegen. Wir legen die Dimensionen der beiden hidden Layers für die folgenden Experimente auf `fc1_dims = 256` und `fc2_dims = 256` fest. Die Anzahl an Outputs der Ausgabeebene entspricht der Anzahl der Aktionen, die dem Agenten zur Verfügung stehen. Im Moment sind das `n_actions = 4`, also die vier möglichen Bewegungsrichtungen oben, rechts, unten und links.

Die Hyperparameter werden in der Datenklasse `DeepQParameters` verwaltet. Diese enthält die gleichen Parameter wie `Parameters` aus Kapitel 2.2.4, wird aber noch um folgende ergänzt:

```
@dataclass
class DeepQParameters:
    # ... wie in Parameters
    replay_buffer_size: int
    batch_size: int
    target_update: int
```

Deren Funktion wurde im Kapitel 2.3.1 bereits erläutert.

2.3.3 Erste Experimente

Wir wollen diese Implementierung nun für einige Experimente nutzen. Ziel ist es zunächst, eine geeignete Aufgabe für den Agenten zu finden, welche anschließend für den Vergleich unterschiedlicher Lernstrategien dienen soll.

TODO Das DQN erhält als Eingabe

3 Luna-Lander

Abbildungsverzeichnis

2.1	Visualisierung von zweidimensionaler Perlin Noise	3
2.2	Mittels Perlin Noise zufällig generierte Landschaften	4
2.3	TODO Main Terrain	5
2.4	Interaktion zwischen Umgebung und Agent in einem MDP	6
2.5	Ergebnisse des ersten Experiments	13
2.6	Vergleich des Lernerfolgs mit und ohne ϵ -greedy Strategie	14
2.7	Ergebnisse des ersten Experiments	15
2.8	TODO	16

Tabellenverzeichnis

Listings

Literaturverzeichnis

- [Arc11] ARCHER, TRAVIS: *Procedurally generating terrain*. In: *44th annual midwest instruction and computing symposium, Duluth*, Seiten 378–393, 2011.
- [DOB20] DABNEY, WILL, GEORG OSTROVSKI und ANDRÉ BARRETO: *Temporally-Extended ϵ -Greedy Exploration*, 2020.
- [HMvdW⁺20] HARRIS, CHARLES R., K. JARROD MILLMAN, ST'EFAN J. VAN DER WALT, RALF GOMMERS, PAULI VIRTANEN, DAVID COURNAPEAU, ERIC WIESER, JULIAN TAYLOR, SEBASTIAN BERG, NATHANIEL J. SMITH, ROBERT KERN, MATTI PICUS, STEPHAN HOYER, MARTEN H. VAN KERKWIJK, MATTHEW BRETT, ALLAN HALDANE, JAIME FERN'ANDEZ DEL R'IO, MARK WIEBE, PEARU PETERSON, PIERRE G'ERARD-MARCHANT, KEVIN SHEPPARD, TYLER REDDY, WARREN WECKESSER, HAMEER ABBASI, CHRISTOPH GOHLKE und TRAVIS E. OLIPHANT: *Array programming with NumPy*. Nature, 585(7825):357–362, September 2020.
- [Lap18] LAPAN, MAXIM: *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Packt Publishing, 2018.
- [MKS⁺13] MNIIH, VOLOODYMYR, KORAY KAVUKCUOGLU, DAVID SILVER, ALEX GRAVES, IOANNIS ANTONOGLOU, DAAN WIERSTRA und MARTIN RIEDMILLER: *Playing Atari with Deep Reinforcement Learning*. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.
- [MKS⁺15] MNIIH, VOLOODYMYR, KORAY KAVUKCUOGLU, DAVID SILVER, ANDREI A. RUSU, JOEL VENESS, MARC G. BELLEMARE, ALEX GRAVES, MARTIN RIEDMILLER, ANDREAS K. FIDJELAND, GEORG OSTROVSKI, STIG PETERSEN, CHARLES BEATTIE, AMIR SADIK, IOANNIS ANTONOGLOU, HELEN KING, DHARSHAN KUMARAN, DAAN WIERSTRA, SHANE LEGG und DEMIS HASSABIS: *Human-level control through deep reinforcement learning*. Nature, 518(7540):529–533, Februar 2015.
- [Par15] PARBERRY, IAN: *Modeling real-world terrain with exponentially distributed noise*. Journal of Computer Graphics Techniques, 4(2):1–9, 2015.
- [PGM⁺19] PASZKE, ADAM, SAM GROSS, FRANCISCO MASSA, ADAM LERER, JAMES BRADBURY, GREGORY CHANAN, TREVOR KILLEEN, ZEMING LIN, NATALIA GIMELSHEIN, LUCA ANTIGA, ALBAN DESMAISON, ANDREAS KOPF, EDWARD YANG, ZACHARY DEVITO, MARTIN RAISON, ALYKHAN TEJANI, SASANK CHILAMKURTHY, BENOIT STEINER, LU FANG, JUNJIE BAI und SOUMITH CHINTALA: *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In: WALLACH, H., H. LAROCHELLE, A. BEY-GELZIMER, F. d'ALCHÉ-BUC, E. FOX und R. GARNETT (Herausgeber): *Advances in Neural Information Processing Systems 32*, Seiten 8024–8035. Curran Associates, Inc., 2019.

Literaturverzeichnis

- [Rav18] RAVICHANDIRAN, SUDHARSHAN: *Hands-on Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow*. Packt Publishing Ltd, 2018.
- [SB20] SUTTON, RICHARD S und ANDREW G BARTO: *Reinforcement learning: An introduction, second edition*. MIT press, Cambridge, 2020.