

BACHELORARBEIT

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

David Müller

Entwurf vom 10. Mai 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: 11. Mai 2021



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

München, den 11. Mai 2021

.....
(Unterschrift des Kandidaten)

Abstract

Für Machine-Learning-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 Modifikation der Reward-Funktion abbilden lassen, welche Auswirkungen dies auf das Lernverhalten sowie die Resultate des Agenten hat und ob dieser Ansatz somit eine Da-seinsberechtigung hat. Für eine anschauliche Darstellung entwickeln wir eine Domäne, in der der Agent auf einem zufällig generierten Terrain Aufgaben lösen soll. Wir betrachten zusätzlich die Performance des Agenten auf der Domäne des Lunar Lander. Wir kommen zu dem Schluss, dass unser Ansatz überraschend stabil lernt, sich teilweise sogar ähnlich verhält wie Epsilon-Greedy und somit weiter erforscht werden sollte.

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Einleitung | 1 |
| 2 Grundlagen | 3 |
| 2.1 Q-Learning | 3 |
| 2.1.1 Das Prinzip von Q-Learning | 3 |
| 2.1.2 Der Ablauf von Q-Learning | 5 |
| 2.1.3 Exploration vs Exploitation | 6 |
| 2.1.4 Implementierung in Python | 7 |
| 2.2 Deep-Q-Learning | 9 |
| 2.2.1 Das Prinzip von Deep-Q-Learning | 9 |
| 2.2.2 Implementierung in Python | 11 |
| 3 Gebirgslandschafts-Domäne | 15 |
| 3.1 Das Environment | 15 |
| 3.2 Q-Learning-Experimente | 17 |
| 3.2.1 Einzelnes Experiment | 17 |
| 3.2.2 Vergleich des Trainings mit und ohne ϵ -greedy Strategie | 19 |
| 3.2.3 Erkundungsstrategie codiert im Reward | 20 |
| 3.3 Deep-Q-Learning Experimente | 22 |
| 3.3.1 Experimente zur Eignung des Lernziels | 22 |
| 3.3.2 Experimente mit unterschiedlichen Strategien | 28 |
| 3.3.3 Abbildung einer Erkundungsstrategie über die Modifikation des Rewards | 30 |
| 4 Lunar-Lander-Domäne | 41 |
| 4.1 Die Umgebung und die Aufgabe | 41 |
| 4.2 Experimente | 42 |
| 5 Related Work | 47 |
| 6 Ergebnisse und Fazit | 51 |
| Abbildungsverzeichnis | 53 |
| Tabellenverzeichnis | 55 |
| Listings | 57 |
| Literaturverzeichnis | 59 |

1 Einleitung

Eine der wohl herausstechendsten Fähigkeiten des Menschen ist es, durch das Sammeln von Erfahrungen immer neue Dinge zu lernen und auf diese Weise komplexe Zusammenhänge verstehen zu können. Es ist daher verständlich, dass uns die Idee der Entwicklung von künstlichen Intelligenzen, welche diese Fähigkeit ebenfalls besitzen, fasziniert. Bereits seit Ende des 20. Jahrhunderts haben sich viele Hollywood-Produktionen diese Faszination zu Nutzen gemacht. Wenn man erfolgreichen Filmreihen wie „Matrix“ oder „Terminator“ Glauben schenkt, so werden solche Maschinen früher oder später die Welt übernehmen. Stuart Russel, Professor an der University of California und Experte für künstliche Intelligenz, gibt hier zunächst Entwarnung: „Hollywood’s theory that spontaneously evil machine consciousness will drive armies of killer robots is just silly“ [Rus16].

Die Angst vor Maschinen, die sich gegen uns wenden, ist trotzdem nicht ganz unbegründet. Russel sieht das Problem eher darin, dass eine künstliche Intelligenz überaus gut darin werden kann, eine andere Aufgabe zu lösen als die, die wir vorgesehen haben. Der Mathematiker Norbert Wiener erkennt bereits im Jahr 1960: „If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere..., we had better be quite sure that the purpose put into the machine is the purpose which we really desire.“ ([Wie60], zitiert nach [Rus16]). Es ist daher wichtig sicherzustellen, dass die künstliche Intelligenz tatsächlich das von uns gewünschte Ziel verfolgt.

Bei Reinforcement Learning wird die Aufgabe der lernenden Instanz über ein Signal der Umgebung formalisiert, dem *Reward* [SB20]. Wir können der künstlichen Intelligenz nur indirekt über diese Belohnung vermitteln, was ihr Ziel ist. Der Reward spielt also eine sehr zentrale Rolle und ist eine mächtige Komponente im Bereich des Machine Learning, die mit Bedacht formuliert werden sollte.

Wir wollen in dieser Arbeit noch einen Schritt weiter denken und stellen uns die Frage, ob es möglich ist, auch komplexere Mechanismen ausschließlich im Reward zu codieren. Hierbei geht es um Strategien, die normalerweise im Trainings-Algorithmus implementiert werden und so eine zusätzliche Ebene darstellen die das, was der Agent eigentlich lernt, für uns verschleiert. Wir betrachten insbesondere die Erkundung des Environments.

Während der Drang Neues zu entdecken in der Natur des Menschen liegt, benötigt eine künstliche Intelligenz eine Strategie für das Erforschen des Umfelds, in dem sie sich befindet. Es existieren viele Strategien für die Erkundung der Umgebung des Agenten, doch eine der bekanntesten ist die ϵ -greedy-Strategie. Hierbei wählt der Agent meistens die beste ihm bekannte Aktion, probiert allerdings mit der Wahrscheinlichkeit ϵ eine zufällige Aktion aus [DOB20, SB20]. Auf diese Weise wird erreicht, dass er bessere Lösungswege finden kann, die er davor noch nicht kannte.

Zielsetzung Wir experimentieren in dieser Arbeit mit dem Ansatz, eine Erkundungsstrategie allein durch die Modifikation des von der Umgebung zurückgegebenen Rewards abzubilden. Wir vergleichen den Trainingsverlauf und die Resultate dieses Ansatzes mit der ϵ -greedy-Strategie und einem Agenten ohne Erkundungsstrategie. Letzteres soll zei-

1 Einleitung

gen, ob unsere Methode an sich einen Vorteil bringt. Wir experimentieren auf zwei sehr unterschiedlichen Domänen, um zu sehen, ob es hier Unterschiede in der Performance unserer Strategie gibt. Ziel ist es zu prüfen, ob unser Ansatz eine Daseinsberechtigung hat und ob es lohnenswert ist, in dieser Richtung weiterzuforschen.

Aufbau der Arbeit In Kapitel 2 erklären wir zunächst die Grundlagen von Q-Learning und Deep-Q-Learning. Außerdem beschreiben wir, wie wir diese in Python implementieren. In Kapitel 3 bauen wir zunächst eine eigene Domäne (Gebirgslandschaft), um maximale Kontrolle über das Environment zu haben. Im Anschluss führen wir auf dieser nach einigen einführenden Experimenten Versuche durch, bei denen wir eine Erkundungsstrategie nur durch die Modifikation des Rewards implementieren. Wir verwenden sowohl einen Q-Learning- als auch einen Deep-Q-Learning-Agenten und vergleichen unseren Ansatz mit der ϵ -greedy-Strategie und einem Agenten ohne Erkundungsstrategie. Unsere Methode zeigt hier ein ähnliches Lernverhalten wie ϵ -greedy, obwohl unser Agent immer greedy agiert. In Kapitel 4 nutzen wir unsere Erkenntnisse aus Kapitel 3, um unseren Ansatz auf der Domäne des Lunar Landers zu testen. Wie im Kapitel davor vergleichen wir unseren Ansatz mit der ϵ -greedy Strategie und einem Agenten ohne Erkundungsstrategie. Auch hier lässt sich die Tendenz erkennen, dass unsere Methode einen positiven Einfluss auf das Training hat. In Kapitel 5 betrachten wir verwandte Publikationen auf diesem Gebiet und vergleichen diese mit unserer Arbeit. Abschließend fassen wir in Kapitel 6 unsere Ergebnisse zusammen, ziehen daraus ein Fazit und verweisen auf mögliche zukünftige Entwicklungen.

2 Grundlagen

2.1 Q-Learning

Einer der ersten Durchbrüche im Bereich des Reinforcement Learning war die Entwicklung des Q-Learning-Algorithmus [SB20]. Diesen wollen wir im Folgenden erklären.

2.1.1 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 Experimente in dieser Arbeit zielen darauf ab, Probleminstanzen von *Markov Decision Processes* – oder kurz MDPs – zu lösen. In MDPs gibt es eine handelnde Instanz, den *Agenten*, welcher mit seiner *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 einen Reward R_{t+1} , sowie einen Folgezustand S_{t+1} . Abbildung 2.1 zeigt eine schematische Repräsentation dieses Ablaufs.

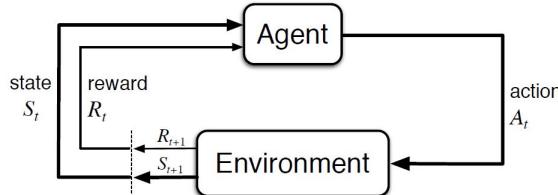


Abbildung 2.1: Interaktion zwischen Umgebung und Agent in einem MDP

Quelle: [SB20]

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

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

wobei T der letzte 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 Gleichung 2.1 ebenfalls potenziell

2 Grundlagen

unendlich. Wir führen deswegen das *discounting* ein. Für G_t ergibt sich hiermit:

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

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

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

wobei E_π der Erwartungswert vom Ertrag G_t nach Gleichung 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 und dann Policy π folgt. Wir definieren $q_\pi(s, a)$ als

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

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 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 – oder im Fall, dass es mehrere gibt werden – als π_* bezeichnet. Die besten Policies besitzen die gleiche 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:

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

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.

2.1.2 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 lernt.

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

2 Grundlagen

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 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 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 2.1.3 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 2.9 und 2.10 wegen lassen wir die Policy π hier weg, gemeint ist natürlich $q_\pi(s, a)$) 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.

2.1.3 Exploration vs Exploitation

In Kapitel 2.1.2 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 eventuell eine bessere Belohnung liefern könnten. Er nutzt nur die ihm bekannten aus (*exploitation*). Besser wäre es, wenn er ebenfalls Zeit in die Erkundung (*exploration*) der Umgebung stecken würde.

Dies kann realisiert werden, indem der Agent die meiste Zeit „gierig“ (*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. Diese Strategie wird ϵ -*greedy-Strategie* genannt und zählt nach [DOB20] auch heute noch bei der Erkundung der Umgebung zu den am meisten benutzten.

2.1.4 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 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

    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. In den beiden Variablen `rewards_all_episodes` und `max_rewards_all_episodes` werden die Belohnungen des Trainings abgelegt.

2 Grundlagen

Die train()-Methode Die `train()`-Methode ist das Herzstück des Algorithmus:

```
def train(self, width: int, length: int, params: Parameters,
          environment, ...):
```

`width` und `length` beschreiben die Breite und die Länge des Rasters. Dies wird in Kapitel 3.3 deutlich werden. Mit diesen Daten wird die Größe der Q-table bestimmt. Mit den `params` übergeben wir der Funktion die Hyperparameter. Das `environment` ist die Umgebung, in der der Agent operieren soll. 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 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 starke Pfadabhängigkeiten auf. In Kapitel 2.2.1 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 infolgedessen 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

```

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

```

Außerdem wird das neue ϵ berechnet. Wir verwenden hierfür eine exponentielle Funktion, damit ϵ am Anfang stark 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) *\ 
    np.exp(-params.exploration_decay_rate * episode)

params.rewards_all_episodes.append(rewards_current_episode)
params.max_rewards_all_episodes.append(max_reward_current_episode)

return q_table, params

```

2.2 Deep-Q-Learning

2.2.1 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 Neuronalem Netz und erhalten auf diese Weise ein so genanntes *Deep-Q-Network (DQN)*.

2 Grundlagen

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 bezieht sich auf das „Exploration versus Exploitation“-Dilemma und soll eine gleichmäßige Erkundung der Umgebung bewirken. Da wir uns hiermit bereits in Kapitel 2.1.3 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. Wir wollen allerdings kein zufälliges Vorgehen erlernen. 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 2.1.4 verwenden wir ebenfalls einen solchen replay buffer.

Target network Ein ähnliches Problem stellt der Zusammenhang zwischen benachbarten Schritten dar. Die Bellman equation aus Gleichung 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 nach [Lap18] 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
- vii. loss 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 wieder etwas verlernt hat. Wir ergänzen also:

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

2.2.2 Implementierung in Python

Wir wollen nun einen DQN-Agenten in Python implementieren. PyTorch ist ein beliebtes Deep-Learning-Framework, welches Benutzerfreundlichkeit und Leistung vereint [PGM⁺19]. Als Grundlage verwenden wir das Codebeispiel von 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 heraus gekürzt (gekennzeichnet mit ...). Außerdem haben wir die Einrückung auf Methodenebene entfernt. Die Schritte sind entsprechend der Liste aus Kapitel 2.2.1 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.
```

2 Grundlagen

```

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_
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)
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 haben wir 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]
# vii. 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)
loss.backward()
# v.i. Ende
self.policy_net.optimizer.step() # v.i.i.

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

```

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. PyTorch nutzt hierfür die Bezeichnung `Linear` layer. 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. In unserem Fall sind das `n_actions = 4`, also die vier möglichen Bewegungsrichtungen oben, rechts, unten und links.

2 Grundlagen

Die ReLU-activation-function wird im Moment als die mit der besten Performance angesehen [SAV20]. Trotzdem benutzen wir für unsere activation function Sigmoid, da diese in unserem Anwendungsfall bessere Ergebnisse zu erzielen scheint.

Die Hyperparameter werden in der Datenklasse `DeepQParameters` verwaltet. Diese enthält die gleichen Parameter wie `Parameters` aus Kapitel 2.1.4, 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 2.2.1 bereits erläutert.

3 Gebirgslandschafts-Domäne

Wir betrachten zunächst ein Landschaftsnavigationsproblem. Der Agent soll in einer zufällig generierten Landschaft unterschiedliche Aufgaben lösen. Wir erarbeiten Methoden, wie wir eine Erkundungsstrategie im Reward abbilden können und vergleichen diese mit der herkömmlichen ϵ -greedy-Strategien.

3.1 Das Environment

Als Environment für die folgenden Experimente wollen wir eine Gebirgslandschaft erzeugen. Die Umgebung bildet hierbei ein Raster, worauf sich der Agent bewegen kann. Jeder Punkt auf dem Raster soll eine Höhe besitzen. Die Landschaft soll zufällig generiert werden können.

Die simpelste Lösung hierfür wäre wohl, ein zweidimensionales Array mit zufälligen Zahlen zu füllen. Auf diese Weise erhält man für jede Koordinate eine zufällige Höhe. Es ist allerdings für die Lesbarkeit des Problems essenziell, dass benachbarte Punkte einen Zusammenhang haben (zum Beispiel „Bewegt sich der Agent gerade nach oben in die Richtung eines Gipfels oder nach unten in die Richtung eines Tals?“). Dazu kommt, dass die Ergebnisse der Experimente intuitiv sichtbar sein sollen (zum Beispiel „Hat der Agent einen Berg gefunden?“). Deshalb wollen wir 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 3.1 zeigt eine simple Darstellung von zweidimensionalem Perlin Noise, bei der die generierten Werte mit Farbwerten von Schwarz bis Weiß abgebildet werden. Perlin Noise ist nach [Par15] ein fundamentaler Algorithmus in der prozeduralen Generierung von Terrain und somit offenbar sehr gut geeignet, um unsere Umgebung zu erstellen. Wir verwenden ihn um ein zweidimensionales Array mit zufälligen Werten zwischen -1 und 1 zu erzeugen, 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 3.2 erkennen kann. 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 <https://github.com/hnhaefliger/PyEngine3D> (Zugriff am 25.04.2021). Wir nutzen diese vergleichsweise simple 3D-Engine, damit wir alle Aspekte des Environments und dessen Visualisierung kontrollieren und für unseren Zweck anpassen können. So werden zum Beispiel zur besseren Differenzierung Berge und Täler – zusätzlich zur perspektivischen Abhebung – rot bzw. blau dargestellt.

Wir besitzen nun die Möglichkeit, eine zufällige Landschaft zu generieren und diese

3 Gebirgslandschafts-Domäne

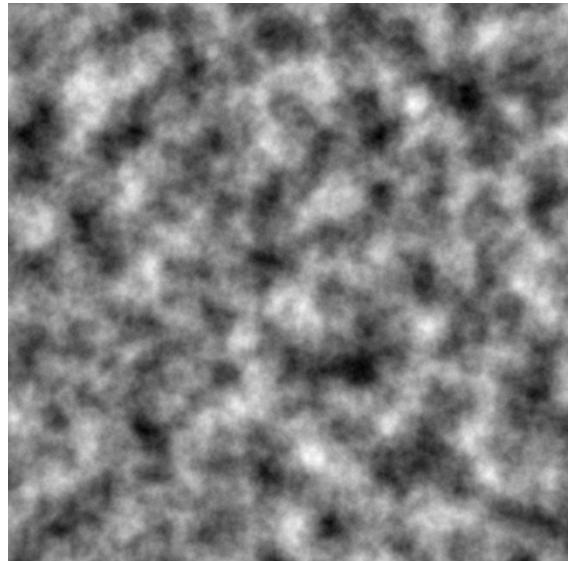
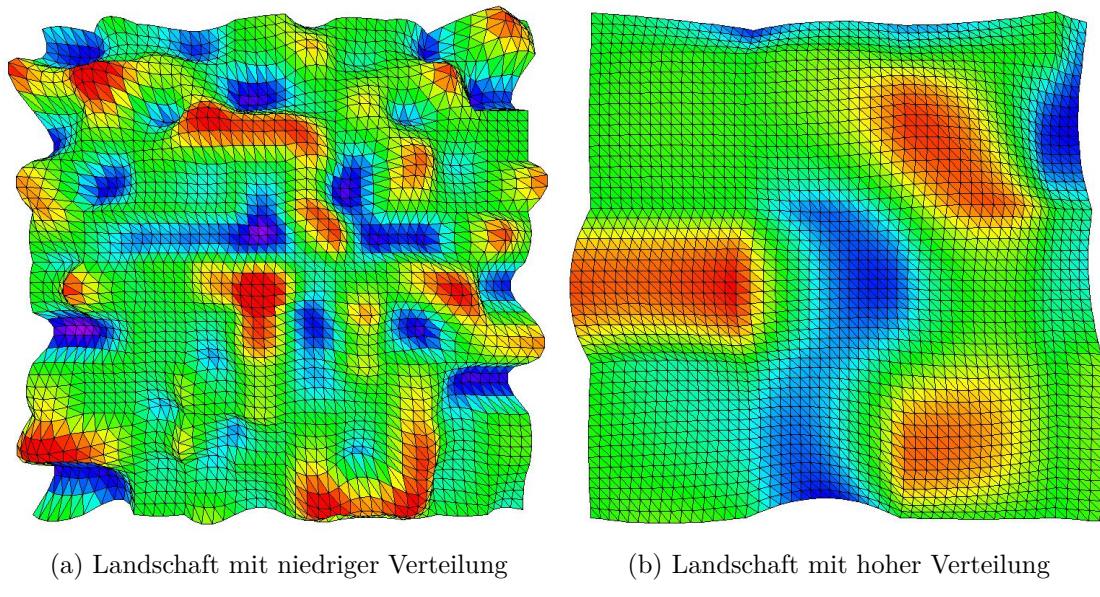


Abbildung 3.1: Visualisierung von zweidimensionalem Perlin Noise

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



(a) Landschaft mit niedriger Verteilung

(b) Landschaft mit hoher Verteilung

Abbildung 3.2: Mittels Perlin Noise zufällig generierte Landschaften

visuell darzustellen. Um bei allen Experimenten die gleichen Voraussetzungen zu gewährleisten, generieren wir mit der eben beschriebenen Methode zufällig ein Terrain, das allen folgenden Experimenten als Environment dient. Dieses ist in Abbildung 3.3 dargestellt. Der höchste Punkt befindet sich bei dieser Landschaft auf dem Berg ganz oben in der Mitte.

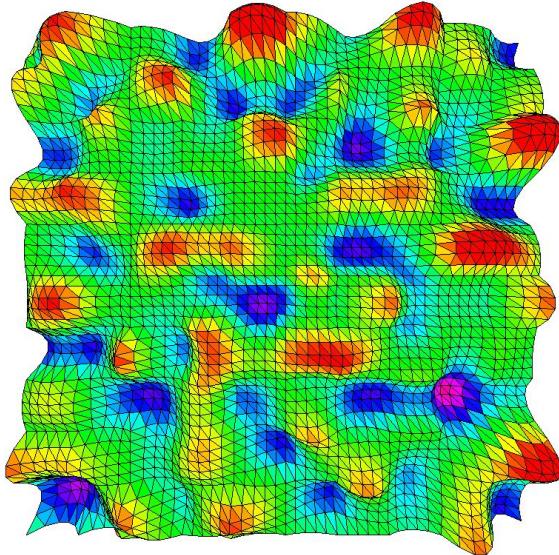


Abbildung 3.3: Landschaft, auf der die Experimente durchgeführt werden

3.2 Q-Learning-Experimente

Um nun zu testen, ob die Landschaft für unsere Zwecke geeignet ist, werden wir einige Experimente durchführen. Wir nutzen hierfür die Implementierung des Q-Learning-Agenten, die in Kapitel 2.1.4 beschrieben wurde. Ziel ist es, dass der Agent zum höchsten Gipfel läuft. 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 . Die Summe aller Rewards ist also dann am größten, wenn er auf den höchsten Berg läuft.

3.2.1 Einzelnes Experiment

Nach einigem Ausprobieren haben sich die folgenden Hyperparameter als passend für unsere Domäne erwiesen:

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

Wir stellen die Ergebnisse in einem Graph dar. Nach einem Trainingdurchlauf erhält man die in Abbildung 3.4a dargestellte Ausgabe. Die x-Achse stellt die aktuelle Episode

3 Gebirgslandschafts-Domäne

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 Rewards aus allen Zeitschritten für jede Episode an. Die orange Linie ist der Durchschnitt der letzten 100 Gesamtbelohnungen 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 in jeder Episode. Die Achsenbeschriftung 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 Abbildung 3.4b in weiß eingezeichneten 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.

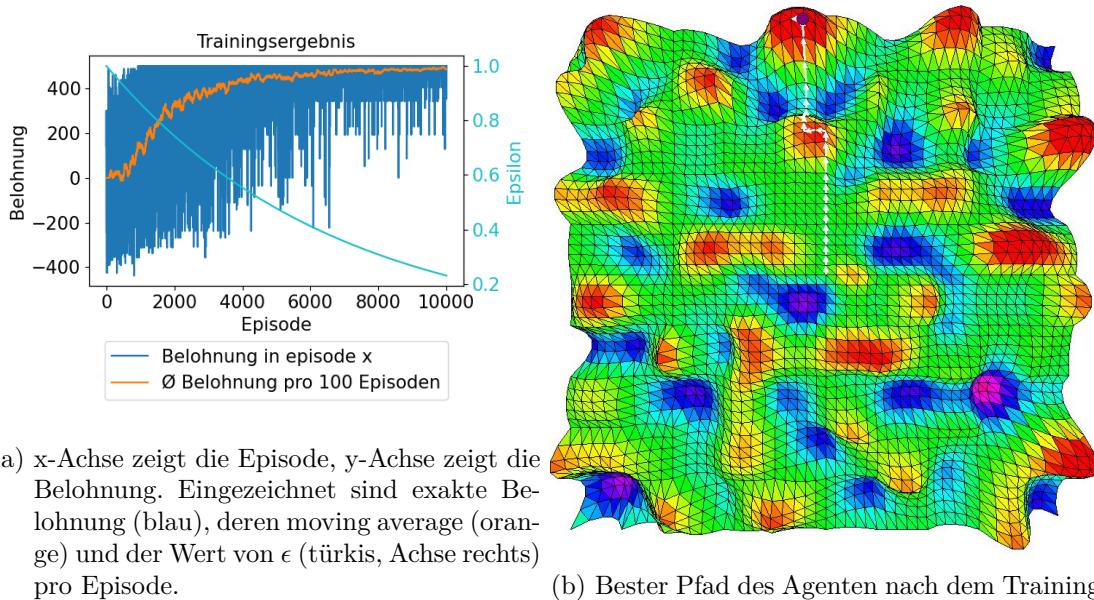


Abbildung 3.4: Trainingsergebnisse des ersten Experiments

Ein 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 ersteres 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.

3.2.2 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 Trainingsergebnis zeigen.

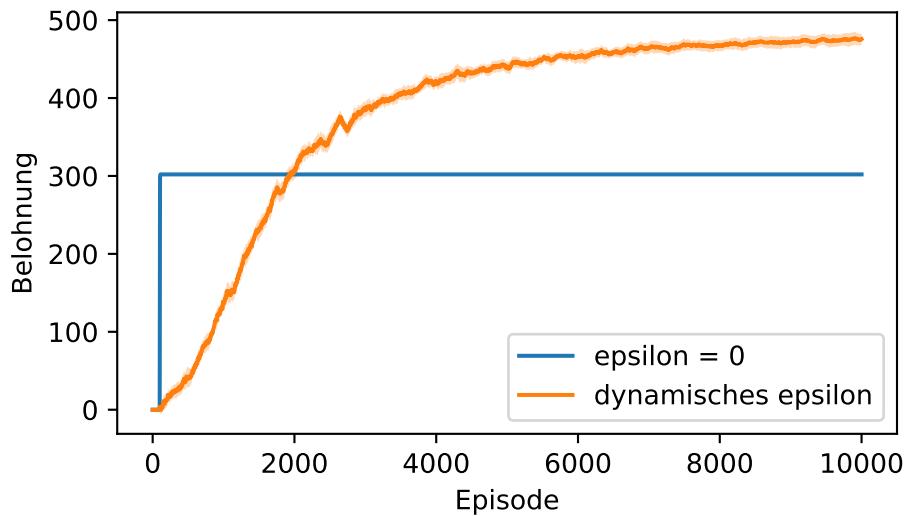


Abbildung 3.5: Vergleich der Trainingsverläufe mit (orange) und ohne (blau) ϵ -greedy Strategie nach jeweils 20 Wiederholungen. Der Graph zeigt den durchschnittlichen moving average und dessen Standardabweichung pro Episode.

Die Achsen vom Graph in Abbildung 3.5 sind bis auf das Fehlen der ϵ -Achse identisch mit dem aus Abbildung 3.4a. Die beiden Linien zeigen jeweils den Durchschnitt der moving average Werte aller 20 Experimentiterationen. Der leicht transparente Bereich um die Linien herum ist die Standardabweichung in der jeweiligen Episode. Es lässt sich hier sehr deutlich erkennen, dass der Agent ohne eine Erkundungsstrategie wie ϵ -greedy 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 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 Agent ohne Erkundungsstrategie gerade einmal 1.1% sind.

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

3.2.3 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:

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

In Abbildung 3.5 ist zu erkennen, dass die maximale erreichte Belohnung etwas unter 500 liegt. Aufgeteilt auf die 300 Steps pro Episode bedeutet das, dass der trainierte Agent in jedem Zeitschritt eine durchschnittliche Belohnung von circa 1,7 bekommt. Da sich dieser Wert nicht signifikant von der maximalen `exploration_rate` von 1 unterscheidet, verzichten wir an dieser Stelle auf eine Normierung des `actual_reward`. Die `exploration_rate` verhält sich genau so wie beim Experiment davor. Die 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. Am Anfang wird der `actual_reward` komplett ignoriert. Nach und nach wird die 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:

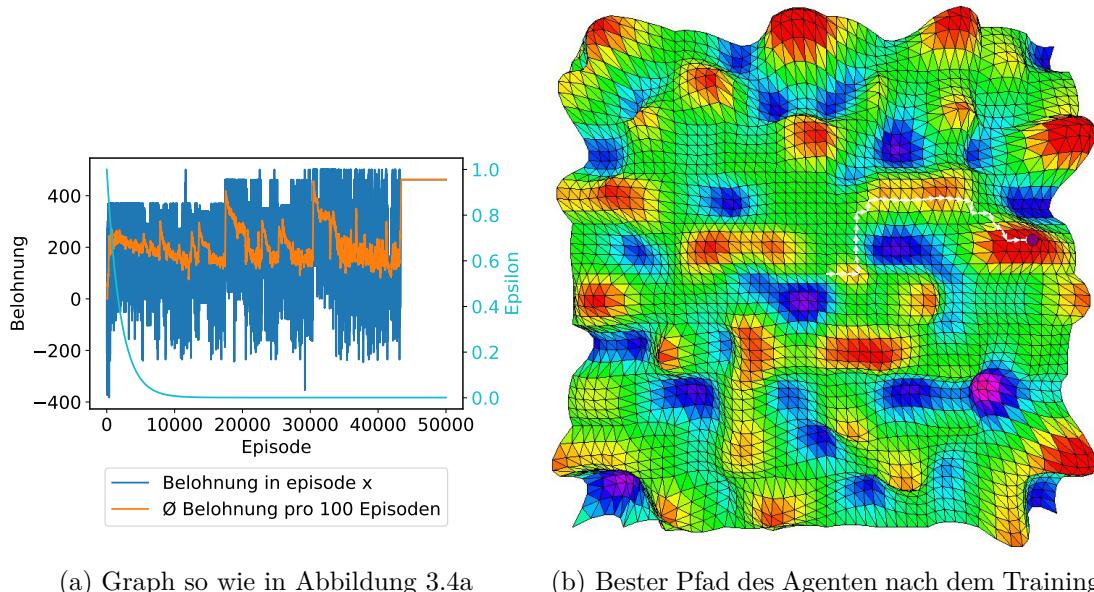


Abbildung 3.6: Trainingsergebnisse mit Erkundungsstrategie codiert im Reward

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 Abbildung 3.4a. Wir erhalten keine saubere Lernkurve. Trotzdem erreicht der Agent einen maximalen moving average von etwas über 462. Zum Vergleich: Der maximale moving average in Abbildung 3.4a liegt bei etwas über 497. Die Q-table dieses Experiments enthält 850 von 10000 mit Null belegte Einträge.

Abbildung 3.6b 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 Durchschnitt mit Standardabweichung berechnen, der in Abbildung 3.7 zu sehen ist.

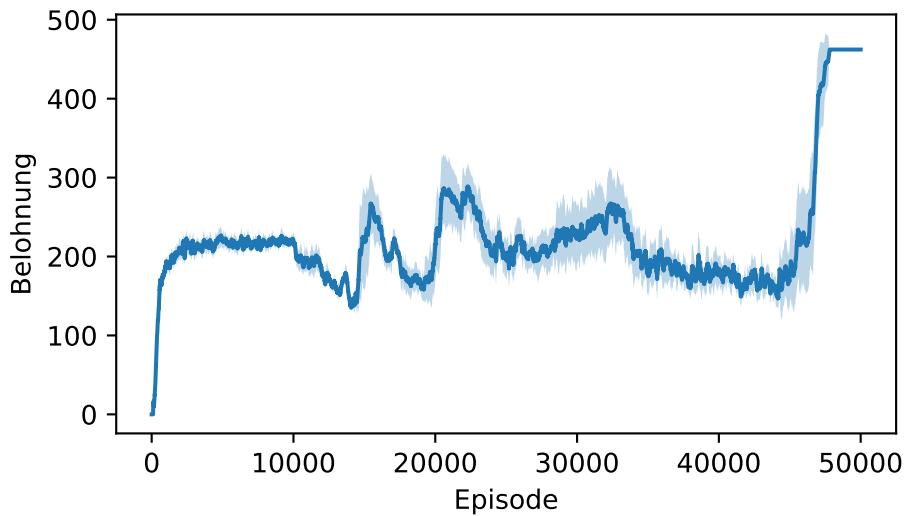


Abbildung 3.7: Trainingsergebnisse mit Erkundungsstrategie codiert im Reward nach 20 Wiederholungen. Graph so wie in Abbildung 3.5.

Der Agent benötigt mit 50000 Episoden sehr lange, um seinen Höchstwert zu erreichen. Die `exploration_rate` liegt circa ab Episode 10000 bei 0. Ab hier erhält der Agent also den tatsächlichen, unverfälschten Reward. Da er immer greedy agiert, findet er anscheinend nach jeder Episode wieder einen besseren Pfad. Dies hat dann zur Folge, dass andere Felder anders bewertet werden, was wiederum einen neuen Pfad eröffnet. Das wiederholt sich so lange, bis sich kein lukrativerer Pfad mehr auftut. Dies scheint hier circa ab Episode 47000 der Fall zu sein. Zu diesem Zeitpunkt existiert kein besserer ihm bekannter Pfad mehr, wodurch der Agent immer denselben Weg läuft, was den starken Anstieg der durchschnittlichen Belohnung erklärt. Dieser Ablauf ist allerdings lediglich eine Vermutung von uns, welche nicht belegt ist. Verglichen mit dem Agenten ohne Erkundungsstrategie lässt sich festhalten, dass die Modifikation der Belohnung in diesem Fall einen 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.

3.3 Deep-Q-Learning Experimente

Wir wollen die Implementierung aus Kapitel 2.2.2 nun für einige Experimente mit Deep-Q-Agenten nutzen. Wir betrachten zunächst einige Experimente mit Varianten des Ziels einen hohen Gipfel zu finden und argumentieren, dass diese für unseren Vergleich nicht geeignet sind. Stattdessen formulieren wir daraufhin ein neues Ziel, bei dem der Agent möglichst viel Fläche abdecken soll, während er aber gleichzeitig nahe an seiner Startposition bleibt. Diese Aufgabe besitzt für unsere Zwecke eine passende Komplexität und bildet daher die Grundlage für die anschließenden Vergleiche unterschiedlicher Lernstrategien.

3.3.1 Experimente zur Eignung des Lernziels

Ausgangsexperiment Das DQN erhält als Eingabe die aktuelle Position des Agenten in Form einer x- und einer y-Koordinate. Diese beschreiben in diesem Experiment den Zustand des Agenten. Die Mitte der Landschaft hat die Koordinaten (0, 0). Dies ist ebenfalls der Startpunkt des Agenten. Als Belohnung erhält der Agent wie in Kapitel 3.2 die Differenz der Höhe des Folgezustands und des aktuellen Zustands. Die Hyperparameter werden wie folgt belegt:

```
params = DeepQParameters(
    num_episodes=10000,
    max_steps_per_episode=100,
    replay_buffer_size=20000,
    batch_size=32,
    learning_rate=0.001,
    discount_rate=0.999,
    target_update=25,
    start_exploration_rate=1,
    max_exploration_rate=1,
    min_exploration_rate=0.001,
    exploration_decay_rate=0.001,
    # ... Rest wird erst während des Trainings belegt
)
```

Damit einzelne Beobachtungen nicht zu einer völligen Veränderung der Gewichte im DQN führen, ist die `learning_rate` im Vergleich zum Training mit der Q-table sehr klein.

Der Agent scheint mit diesen Informationen noch nicht viel anfangen zu können. Der Graph in Abbildung 3.8a zeigt, dass der moving average Wert (orange) über die komplette Trainingszeit sehr inkonsistent ist. Außerdem liegt er größtenteils deutlich unter dem möglichen Höchstwert. Dies lässt sich daran erkennen, dass die blaue Linie – also die Belohnung der einzelnen Episoden – teilweise fast bis 400 geht, der moving average diesen aber nur wenige Male fast erreicht. Außerdem wissen wir aus Kapitel 3.2, dass definitiv ein Wert von 497 möglich ist. In Abbildung 3.8b ist zu erkennen, welchen Pfad der Agent unter Verwendung des aus dem Training resultierenden Netzes zurücklegt. Er bewegt sich auf einen Gipfel, welcher sich in der Nähe des Startzustands befindet. Optimal wäre jedoch der Gipfel ganz oben in der Mitte.

Das DQN liefert hier also kein sonderlich gutes Ergebnis. Dies könnte daran liegen, dass der Agent keinerlei Information über die Höhe seiner Zustände hat, von denen seine erhaltene Belohnung und die Erfüllung der Aufgabe ja stark abhängt.

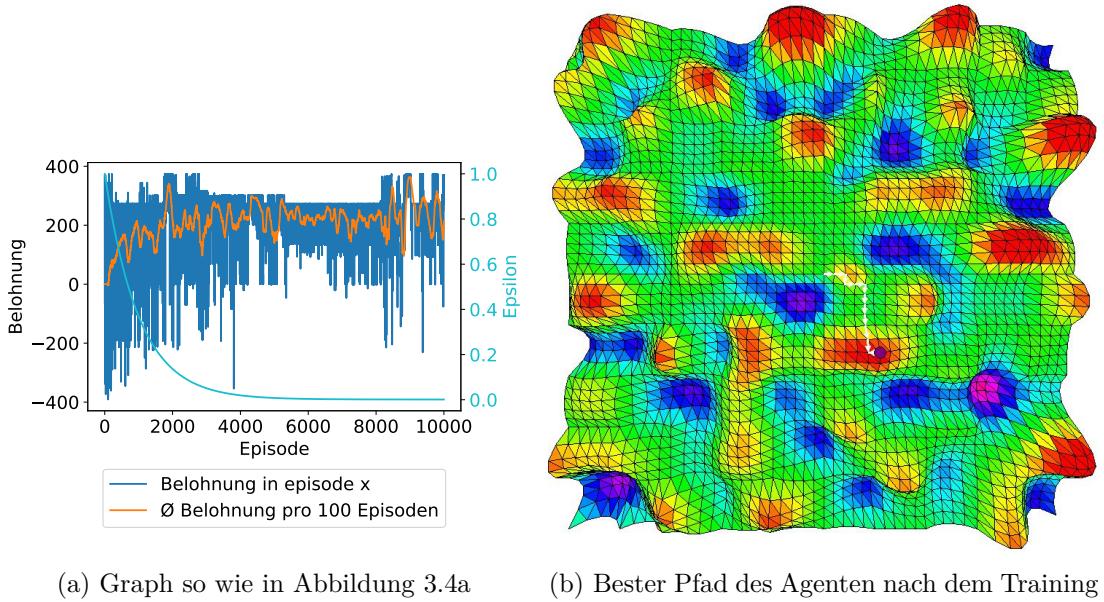


Abbildung 3.8: Ergebnisse des ersten Experiments mit DQN

Anpassung der Parameter Wir passen also die Werte an, die einen Zustand beschreiben und fügen die Höhe des aktuellen Zustands, sowie die der umliegenden Zustände hinzu. Das DQN erhält also nun als Eingabe sieben Werte (x- und y-Koordinate, eigene Höhe und die Höhe der vier umliegenden Felder).

Das letzte Training hat für die 10000 Episoden etwas über eineinhalb Stunden gedauert¹. Wir suchen eine Aufgabe, die für den Vergleich unterschiedlicher Lernstrategien genutzt werden soll und wollen für jede Strategie eine Experimentreihe durchlaufen, um eine statistische Auswertung zu ermöglichen. Diese sollten in zumutbarer Zeit durchführbar sein. Daher ist es wichtig, die Trainingszeit für einzelne Experimente zu reduzieren.

Wir reduzieren daher die Episodenanzahl `num_episodes` auf 1500. Dementsprechend muss auch die `exploration_decay_rate` angepasst werden. Wir setzen diese auf 0.005. In Abbildung 3.9b lässt sich schnell erkennen, dass das Trainingsergebnis auch hier nicht zufriedenstellend ist. Der Agent bewegt sich nur ein paar Felder weit zu einem nahe gelegenen, sehr kleinen Hügel. Der moving average in Abbildung 3.9a zeigt auch keinen gewünschten Trainingsverlauf wie beispielsweise in Abbildung 3.4a. Lediglich die Trainingsdauer hat sich wie erwartet verringert.

Zufällige Startposition Wir werden daher unseren Ansatz etwas verändern. Die Startposition wird zu Beginn jeder Episode zufällig gewählt. Das DQN erhält außerdem statt der absoluten Position im Grid die relative Position zum Startpunkt. Das bedeutet, dass dieser immer die Koordinaten (0, 0) besitzt. Dies soll die Abhängigkeit von einem immer gleichen Startzustand aufbrechen und die Aufgabe interessanter machen. Der Agent läuft nun von seiner Startposition aus so lange in die Richtung eines höheren Punktes, bis es nicht mehr weiter nach oben geht. Einer dieser möglichen Pfade ist in Abbildung 3.10a zu sehen. Das ist schon mal ein Schritt in die richtige Richtung. Die Aufgabe ist

¹Die Experimente wurden auf einer Nvidia RTX 2060 durchgeführt

3 Gebirgslandschafts-Domäne

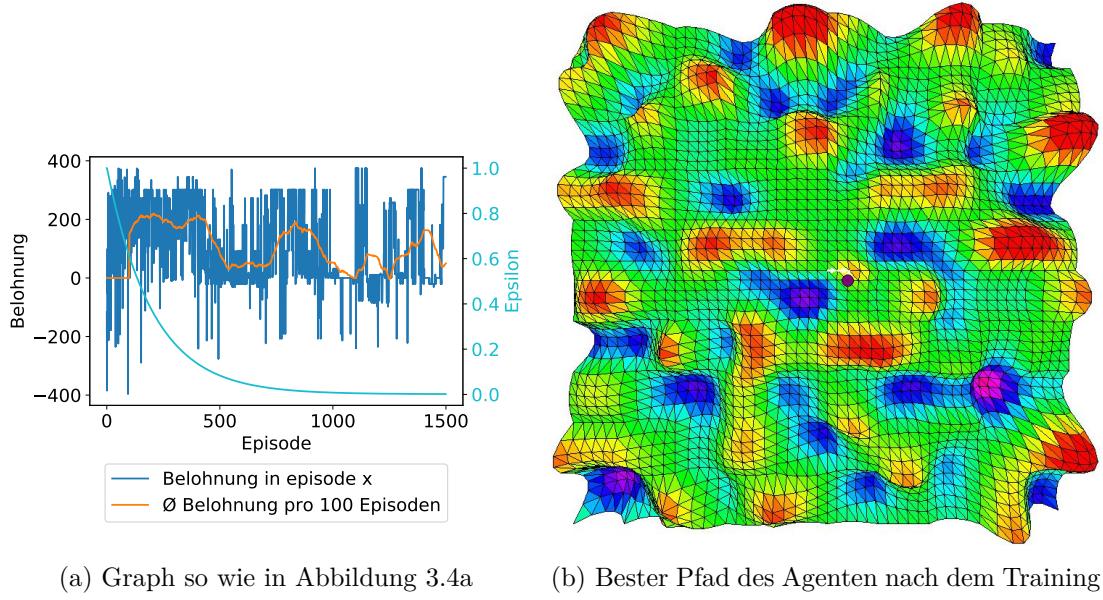


Abbildung 3.9: Ergebnisse mit angepassten Parametern

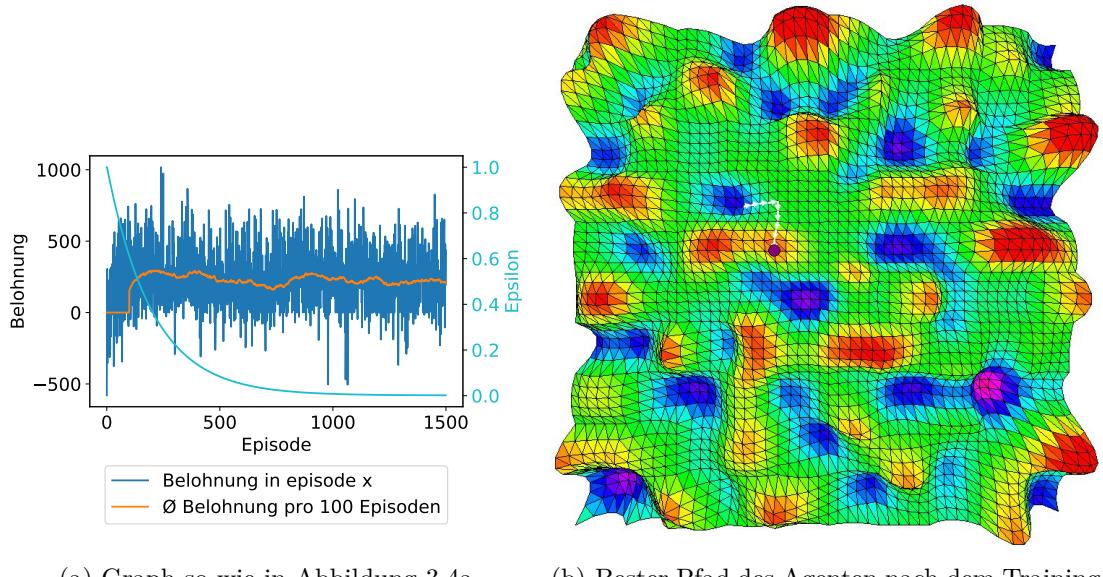


Abbildung 3.10: Ergebnisse mit zufälliger Startposition

allerdings zu einfach. Im Graph in Abbildung 3.10a lässt sich erkennen, dass der Großteil des Lernvorgangs bereits vor der 100-Episoden-Marke passiert. Dies ist nicht optimal für unsere Zwecke, da wir erst ab Episode 100 den moving average und damit unsere Hauptvergleichsquelle verfolgen können.

Das neue Ziel soll daher sein, dass der Agent nicht nur nach oben läuft, sondern einen möglichst hohen Punkt in der Nähe des Startpunktes findet. Zu diesem Zweck erweitern wir die Eingaben, die das DQN bekommt. Wir übergeben nun die Höhe und die relative Position des in diesem Zeitschritt bisher höchsten besuchten Feldes. Außerdem wird die

Anzahl der übrigen und der maximalen Zeitschritte angefügt. Dies soll in der Theorie dazu führen, dass der Agent seine Umgebung erkundet, solange noch genügend Zeit ist. Gegen Ende der Episode sollte er dann zum bisher höchsten bekannten Gipfel laufen.

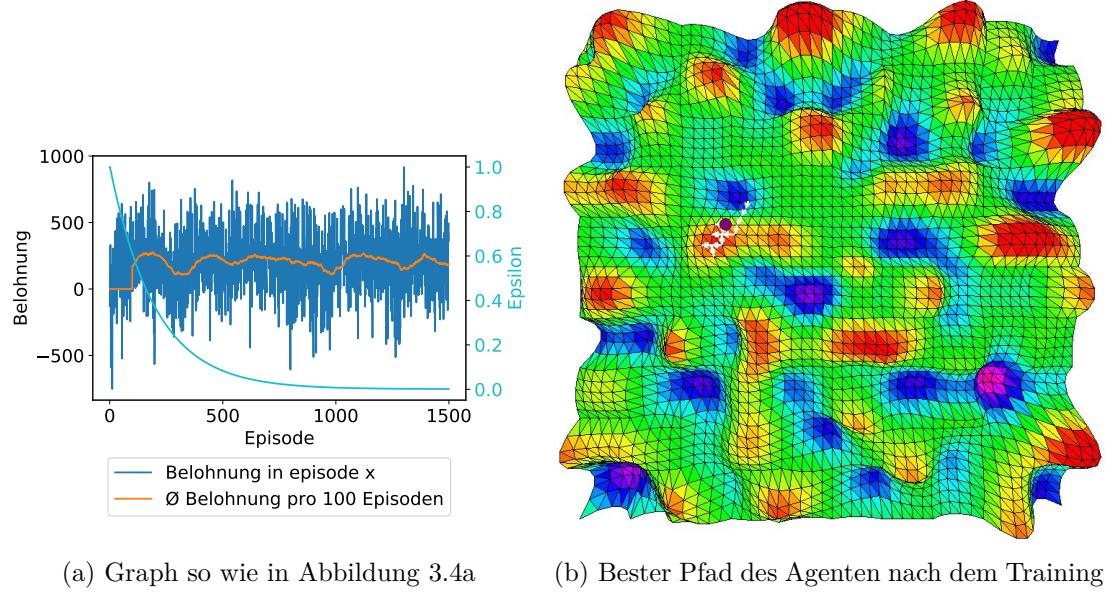


Abbildung 3.11: Ergebnisse mit zusätzlichen Inputs

Das Ergebnis ist ähnlich wie das des vorangegangenen Experiments. Zum besseren Vergleich haben wir für die Darstellung in Abbildung 3.11b denselben Startpunkt gewählt wie in Abbildung 3.10b. Der Agent orientiert sich tatsächlich hin zum etwas höheren Gipfel und scheint in einem sehr kleinen Umkreis die Umgebung zu erkunden, versäumt es aber am Ende auf dem höchsten Punkt aufzuhören. Dies kann daran liegen, dass sich der Agent in jedem Zeitschritt bewegen muss und auf diese Weise nicht die Möglichkeit besitzt, genau auf dem höchsten Feld aufzuhören. Die Farbcodierung der Landschaft verrät uns allerdings, dass das Feld links oder unterhalb des vorletzten Schritts höher gelegen ist als das gewählte obere Feld. Viel wichtiger ist jedoch, dass der Graph in Abbildung 3.11a keinen wirklichen Trainingsfortschritt zeigt. Der moving average pendelt hier grob um denselben Wert und zeigt keinerlei Verbesserung des Agenten über längere Zeit. Auch dieser Ansatz ist also für unsere Zwecke nicht geeignet.

Neudeinition des Lernziels Aufgrund der Misserfolge mit dem Finden von hohen Gipfeln wollen wir nun versuchen, ein anderes Ziel zu erarbeiten. Die neue Idee ist, dass der Agent so viele Felder wie möglich besuchen soll, sich dabei aber so wenig wie möglich vom Startfeld entfernt. Wir erhoffen uns hiervon, dass die quasi gegensätzlichen Ziele zu interessanten Ergebnissen führen und für ein DQN ein angemessenes Problem darstellen.

Um dieses Verhalten zu erreichen, werden sowohl die Beschreibung eines Zustands als auch die Belohnung stark angepasst. Die Belohnung setzt sich aus den beiden Aufgaben zusammen und sieht in etwa so aus:

```
reward = (NEW_POINT_REWARD if is_new_point else 0) -\
(DISTANCE_MULTIPLIER * distance_from_spawn)
```

3 Gebirgslandschafts-Domäne

Der erste Part gibt den fixen Belohnungswert `NEW_POINT_REWARD` aus, falls das Feld vom Agenten noch nicht besucht wurde, ansonsten null. Davon wird dann die Distanz vom Startzustand abgezogen. Auf diese Weise erhält der Agent höhere Strafen je weiter er sich von diesem entfernt. Die Distanz wird davor noch mit einem ebenfalls fixen Wert `DISTANCE_MULTIPLIER` multipliziert. Die beiden fixen Werte (im Folgenden *Belohnungsparameter* genannt) sollen als Stellschrauben dienen, um die richtige Gewichtung der beiden Ziele zu finden.

Als nächstes passen wir die Werte an, die einen Zustand beschreiben. Nicht mehr benötigt werden die Daten über die Höhe des eigenen und der umliegenden Felder. Stattdessen wird für jede angrenzende Position ein positiver Wert geliefert, wenn der Agent diese noch nicht besucht hat. Hat er das Feld schon besucht, so entspricht dieser Wert 0. Befindet sich der Agent am Rand der Landschaft und eines der angrenzenden Felder somit außerhalb des Grids, so wird der entsprechende Wert mit einer negativen Zahl belegt. Die Höhe der positiven und der negativen Wert lässt sich ebenfalls über einen fixen Parameter festlegen.

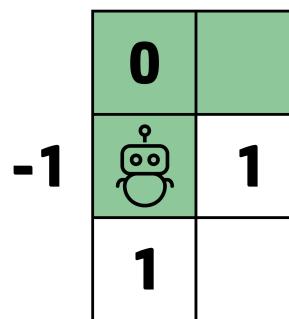


Abbildung 3.12: Visualisierung der Inputs des Agenten abhängig von den angrenzenden Positionen im Raster

Abbildung 3.12 zeigt ein Beispiel, wie die entsprechenden Werte ohne Multiplikator aussehen können. Das Raster stellt einen kleinen Ausschnitt des Landschaft-Grids dar. Die grünen Felder hat der Agent bereits besucht. Das DQN erhält für das obere, bereits gesehene Feld also den Wert 0. Die beiden unerforschten Positionen rechts und unten liefern jeweils einen positiven Wert – in diesem Fall ohne einen Multiplikator also den Wert 1. Da die Position links des Agenten außerhalb des Grids liegt, ist diese mit dem Wert -1 belegt.

Die Werte bezüglich den Zeitschritten aus dem vorherigen Experiment sind ebenfalls enthalten. Insgesamt besteht ein Zustand jetzt also aus 8 Werten: Die relative Position (2), die Information über die anliegenden Positionen (4) und die Werte der übrigen beziehungsweise maximalen Zeitschritte (2). Der Graph in Abbildung 3.13a zeigt wieder die Trainingsergebnisse der 1500 Episoden an. Am moving average lässt sich diemal ein sauberer Lernfortschritt erkennen. Die durchschnittliche Belohnung steigt bis circa Episode 400 stark an und flacht dann langsam ab. Dieser Verlauf ist zunächst zufriedenstellend. Weniger zufriedenstellend ist allerdings das Verhalten, welches der Agent unter Verwendung des erlernten DQNs zeigt. Wie in Abbildung 3.13b zu sehen ist, bewegt sich der Agent von seiner Startposition aus lediglich ein Feld in jede Richtung und springt dann nur noch hin und her. An dieser Stelle kommen unsere Belohnungsparameter ins

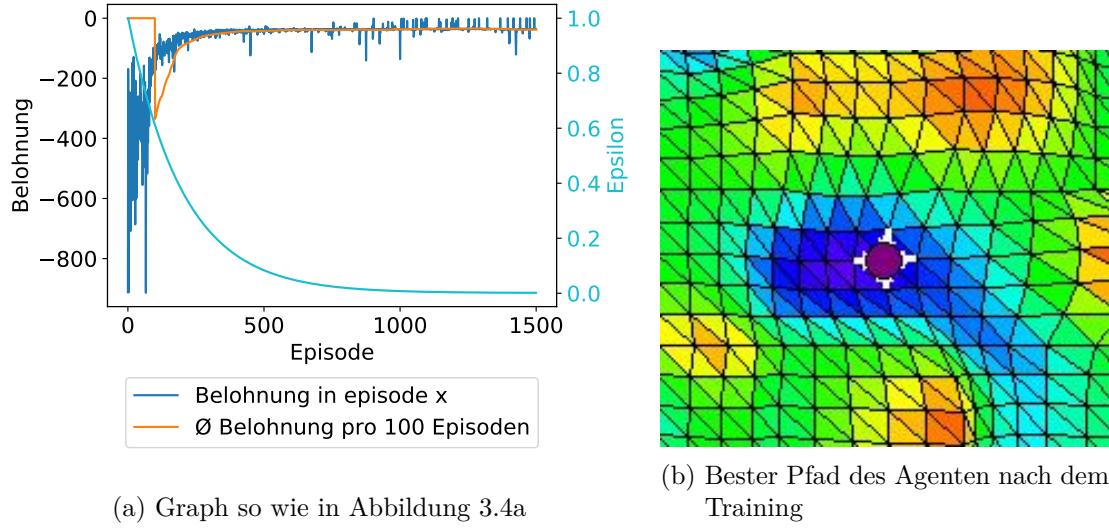
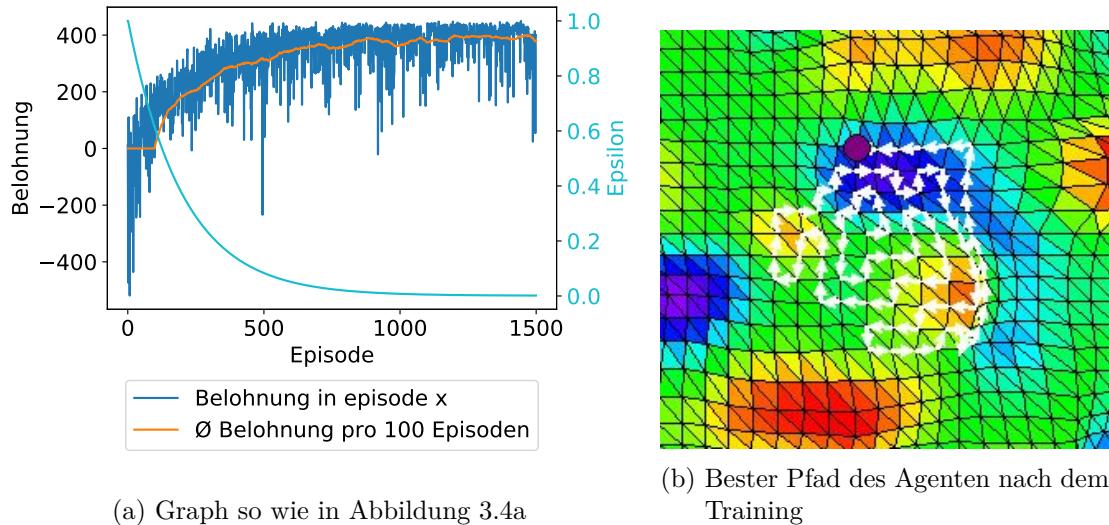


Abbildung 3.13: Ergebnisse nach Neudeinition des Lernziels

Spiel. Da es dem Agenten aktuell wichtiger zu sein scheint, sich so wenig wie möglich vom Startpunkt zu entfernen, setzen wir den `NEW_POINT_REWARD` auf 10. Dies hat einen sehr


 Abbildung 3.14: Ergebnisse mit größerem `NEW_POINT_REWARD`

positiven Einfluss auf das Ergebnis. Abbildung 3.14b stellt wieder einen Pfad des Agenten nach dem Training dar. Die Startposition liegt hier etwa in der Mitte der abgelaufenen Fläche und der Agent läuft quasi spiralförmig immer weiter von diesem Punkt weg. Für das Lösen der Aufgabe ist das eine sehr gute Strategie, da viele neue Felder besucht werden und die Distanz zum Startpunkt gleichzeitig so gering wie möglich gehalten wird.

Der Graph in Abbildung 3.14a zeigt wie im Experiment davor eine schöne Lernkurve, welche zu Beginn des Trainings stark ansteigt und dann langsam abflacht. Aufgrund der höheren Belohnung für neu besuchte Felder verläuft der moving average nun im positiven

3 Gebirgslandschafts-Domäne

Bereich.

Aufgrund dieser positiven Resultate mit dem Experiment werden wir diese Aufgabe für die Durchführung der weiteren Experimente nutzen und die Performance der unterschiedlichen Strategien vergleichen.

3.3.2 Experimente mit unterschiedlichen Strategien

In Abbildung 3.14a erkennt man, dass sich der moving average gegen Ende kaum noch verändert. Er steigt allerdings bis kurz davor noch minimal an, weswegen wir die Episodenlänge – also die Zeitschritte pro Episode – auf 2000 erhöhen. So soll sich auch bei langsameren Lernkurven der moving average am Ende bei einem Wert eingependelt haben. Es ist außerdem wichtig zu erwähnen, dass für die zufällige Wahl der Startposition für alle Experimentreihen der gleiche Seed benutzt wird. Die zufälligen Spawnpunkte und deren Reihenfolge sind also für alle Experimente gleich und somit besitzen alle die gleichen Voraussetzungen.

Experimentreihe mit den erarbeiteten Parametern Wir führen zunächst eine Experimentreihe mit den in Kapitel 3.3.1 erarbeiteten Parametern durch. Das Experiment wird wie in Kapitel 3.2 20 Mal wiederholt. Bei den folgenden Graphen handelt es sich – falls nicht anders angegeben – immer um den durchschnittlichen moving average Wert und dessen Standardabweichung, welche als leicht transparenten Bereich um die Linie dargestellt wird.

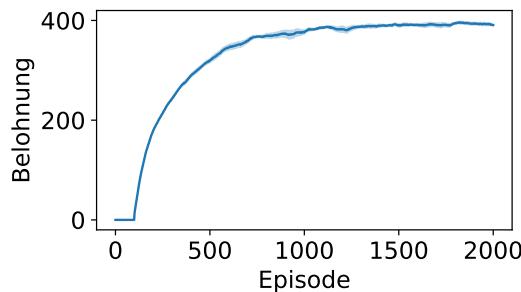


Abbildung 3.15: Trainingsergebnisse mit Experiment wie in Abbildung 3.14a nach 20 Wiederholungen. Graph so wie in Abbildung 3.5.

Der Lernfortschritt scheint auch über mehrere Experimente hinweg sehr konsistent zu sein. Die Lernkurve verläuft in Abbildung 3.15 wie gewünscht anfangs steil und flacht gegen Ende ab. Die Standardabweichung ist zu jedem Zeitpunkt sehr gering, die Ergebnisse der Einzelexperimente unterscheiden sich also nicht stark voneinander.

Agent ohne Erkundungsstrategie Anders verhält es sich beim Training ohne Erkundungsstrategie. Wir setzen hierfür unser ϵ konstant auf 0. Der Agent agiert also wieder nur greedy. Auch dieses Experiment wiederholen wir 20 Mal. Der Graph in Abbildung 3.16 zeigt das Resultat dieses Experiments (orange) und nochmals zum Vergleich das Resultat aus Abbildung 3.15 (blaue). Es fällt auf, dass die orange Linie wesentlich weiter unten

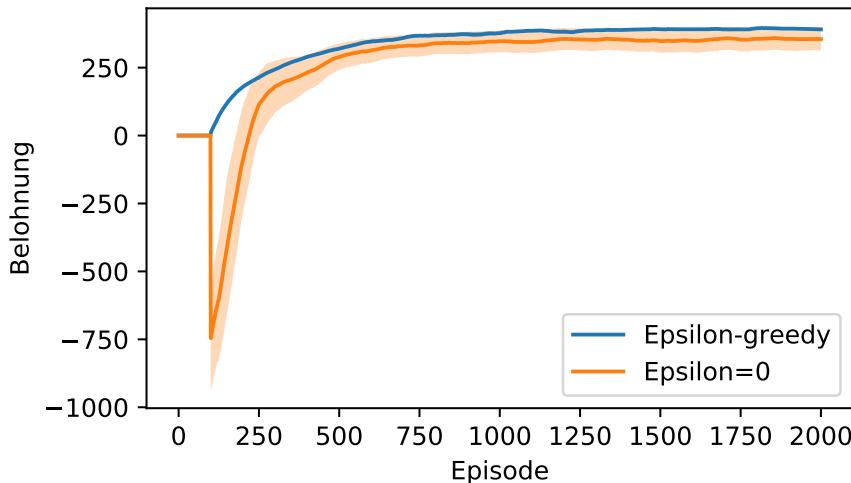


Abbildung 3.16: Vergleich der Trainingsverläufe mit und ohne ϵ -greedy Strategie nach jeweils 20 Wiederholungen. Graph so wie in Abbildung 3.5.

beginnt als die blaue. Das bedeutet, dass der Agent ohne die ϵ -greedy Strategie langsamer lernt. Außerdem erreicht dieser nicht das gleiche Belohnungsmaximum. Dazu kommt, dass die Standardabweichung auffällig größer ist. Der Lernerfolg ist demnach zusätzlich weniger zuverlässig. So zeigt sich erneut, dass die Verwendung einer Erkundungsstrategie die Performance des Agenten deutlich verbessert.

Konstante ϵ -Werte Als zusätzlichen Vergleich führen wir zwei weitere Experimente mit konstantem ϵ durch. Wir setzen hierfür $\epsilon = 0.2$ und $\epsilon = 0.5$.

In Abbildung 3.17 scheint ein Agent ohne Erkundungsstrategie (orange) auf den ersten Blick bessere Ergebnisse zu erzielen als die Agenten mit konstanten ϵ -Werten (grün und rot). Man darf hierbei allerdings nicht vergessen, dass diese ihr ϵ nicht „loswerden“, sondern bis zum Ende immer mit der Wahrscheinlichkeit ϵ eine zufällige Aktion wählen. Da das willkürliche Vorgehen natürlich keine gute Strategie ist, ist in Abbildung 3.17 auch der moving average der Experimente mit größerem, konstanten ϵ geringer. Die tatsächliche Performance des Agenten wird quasi von dem fixen ϵ sabotiert. Wir führen daher eine weitere Form der Datendarstellung ein: sogenannte Boxplots.

Die Boxplots sollen das Verhalten des Agenten unter Anwendung des erlernten DQNs beschreiben. Die y-Achse in den Plots in Abbildung 3.18 zeigt wieder die Summe der Belohnungen innerhalb einer Episode an. Die x-Achse beschreibt, um welches Experiment es sich handelt, also in diesem Fall welches ϵ benutzt wurde. Von links nach rechts sind das in diesem Fall unser dynamisches ϵ gefolgt von den drei fixen Werten 0.0, 0.2 und 0.5. Jedes Experiment wurde bisher 20 Mal durchgeführt. Das bedeutet, dass wir für jede Experimentenreihe 20 trainierte DQNs besitzen. Der Agent durchläuft nun mehrere Iterationen, wobei er jedes dieser Netze 5 Mal in der Umgebung anwendet. Pro Experimentreihe erhalten wir also 100 Belohnungssummen, welche wir in einem Boxplot darstellen. Der Seed für die Startposition wird zu Beginn jedes Experiments zurückgesetzt, um gleiche Voraussetzungen zu gewährleisten. Der Graph in Abbildung 3.18a zeigt die resultierenden Boxplots mit ihren Ausreißern. In Abbildung 3.18b wurden diese für

3 Gebirgslandschafts-Domäne

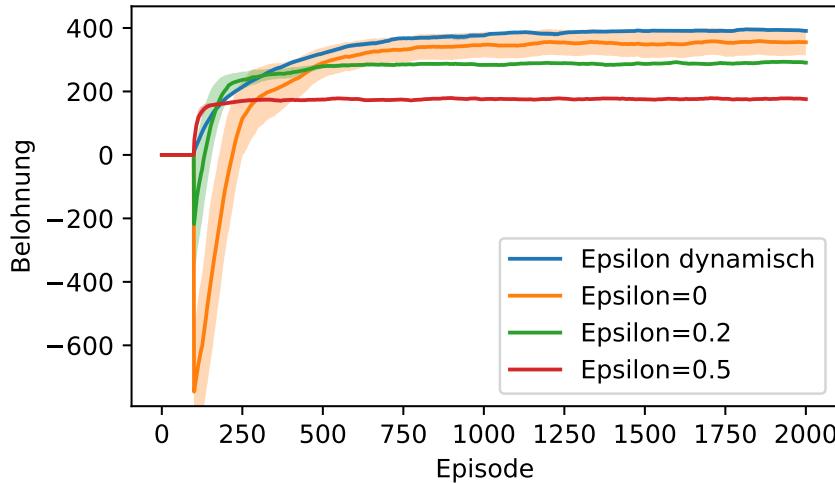


Abbildung 3.17: Vergleich der Trainingsverläufe mit dynamischem ϵ und unterschiedlichen statischen Werten für ϵ nach jeweils 20 Wiederholungen. Graph so wie in Abbildung 3.5.

die bessere Lesbarkeit der Boxplots abgeschnitten.

Es lässt sich zunächst feststellen, dass der Median beim Boxplot des dynamischen ϵ den höchsten Wert hat. Ebenso liegen aber auch die anderen Werte, also das 1. Quartil, das 3. Quartil, das Minimum und das Maximum, bei diesem Experiment am höchsten. Es lässt sich also klar sagen, dass ein über Zeit schrumpfendes Epsilon – also die klassische ϵ -greedy Strategie – in der Anwendung bei uns die beste Performance liefert.

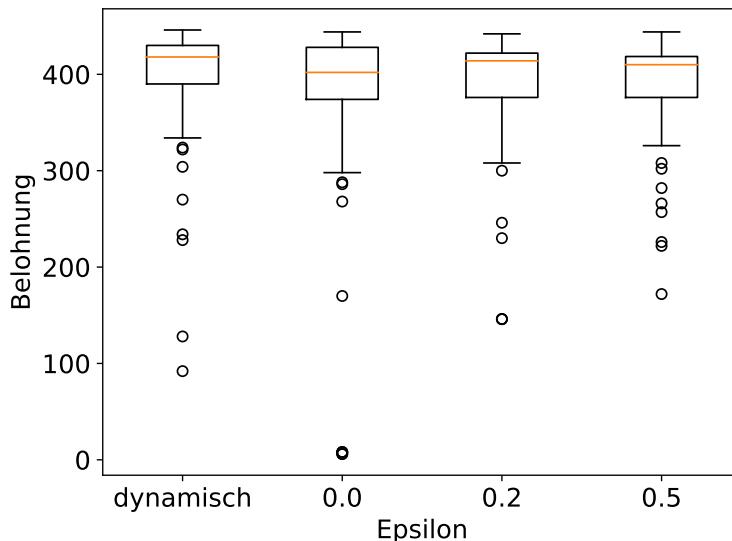
Bei den Experimenten mit fixem Epsilon fällt als Erstes auf, dass das Minimum mit steigendem Epsilon immer größer wird. Wir interpretieren dies so, dass die Agenten mit einem größeren Epsilon mehr von ihrer Umgebung erkundet haben und daher für mehr Anwendungsfälle eine bessere Strategie haben. Dass der Median beim Epsilon von 0.5 wieder leicht niedriger ist als bei 0.2 kann eventuell bedeuten, dass ein zu großes Epsilon dazu führt, dass die bereits erkundeten Pfade weniger stark perfektioniert werden. Die Änderung ist allerdings relativ gering. Um hier eine eindeutige Aussage zu treffen bräuchten wir mehr Daten.

Wir haben allerdings einmal mehr gezeigt, dass sich die Erkundungsstrategie auf das Verhalten des Agenten auswirkt.

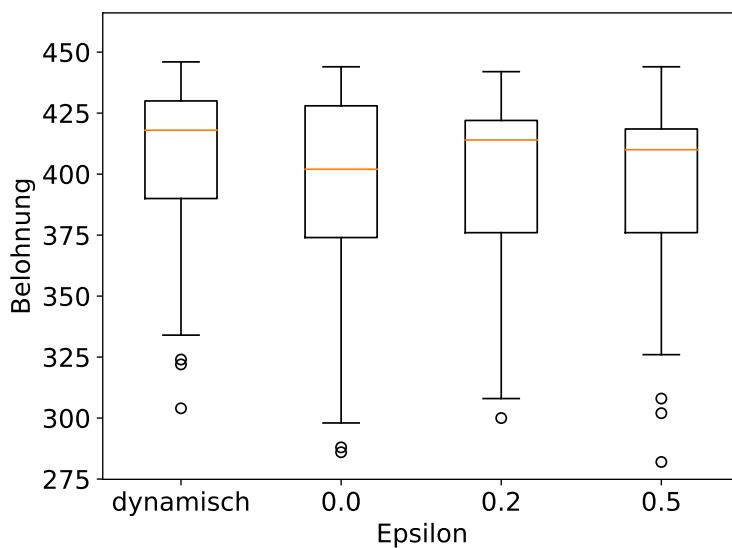
3.3.3 Abbildung einer Erkundungsstrategie über die Modifikation des Rewards

Im Folgenden soll die Erkundungsstrategie nur über die Reward-Funktion abgebildet werden. Wir modifizieren unseren Agenten also zunächst dahingehend, dass er in jedem Fall greedy agiert. Die Hyperparameter für Epsilon haben also keinen direkten Einfluss mehr auf die Wahl der Aktion. Wir nutzen für die Experimente, falls nicht anders angegeben, die folgenden Hyperparameter:

```
params = DeepQParameters(
    num_episodes=2000,
```



(a) Kompletter Plot



(b) Ausschnitt ohne die unteren Ausreißer

Abbildung 3.18: Boxplots der Belohnungssummen nach jeweils 100 Durchläufen mit den trainierten DQNs (100 pro Experimentreihe, also bei 20 Iterationen somit 5 Durchläufe pro DQN). y-Achse zeigt die Belohnung, x-Achse beschreibt das betrachtete Experiment. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, $\epsilon = 0.2$, $\epsilon = 0.5$

```
max_steps_per_episode=80,
replay_buffer_size=20000,
batch_size=32,
learning_rate=0.001,
```

3 Gebirgslandschafts-Domäne

```
discount_rate=0.999,  
target_update=25,  
start_exploration_rate=1,  
max_exploration_rate=1,  
min_exploration_rate=0.001,  
exploration_decay_rate=0.005,  
# ... Rest wird erst während des Trainings belegt  
)
```

Modifizierter Reward mit fixem Faktor In Abbildung 3.17 lässt sich erkennen, dass die Belohnungssumme in einer Episode maximal circa 400 beträgt. Da eine Episode 80 Zeitschritte enthält, bekommt der Agent pro Zeitschritt eine durchschnittliche Belohnung von ungefähr 5. Wir nutzen dieses Wissen, um eine neue Belohnungs-Funktion zu formulieren. Da die durchschnittliche Belohnung diesmal stärker von der maximalen `exploration_rate` abweicht, gewichten wir die `exploration_rate` mit eben diesem Durchschnitt:

```
modified_reward = (1 - exploration_rate) * reward - exploration_rate * 5
```

Ähnlich wie beim Q-Learning-Experiment soll der Agent so zu Beginn negative Belohnungen erhalten, damit er andere Pfade erkundet. Wir lassen den Agenten mit dieser Strategie ebenfalls 20 Mal trainieren. Da bei dieser Belohnungs-Funktion mit den aktuellen Hyperparametern am Anfang nichts von der tatsächlichen Belohnung übrig bleibt und der Agent auf diese Weise eventuell in den ersten Zeitschritten nichts lernt, führen wir noch ein weiteres Experiment durch, dessen Hyperparameter mit `start_exploration_rate=0.5` und `max_exploration_rate=0.5` belegt werden.

Die Graphen in Abbildung 3.19 zeigen wieder den durchschnittlichen moving average und dessen Standardabweichung für alle Episoden. Zum Vergleich sind hier noch die Werte für das klassische Epsilon (blau) und dem fixen Epsilon 0.0 (orange) eingetragen. Der Graph in Abbildung 3.19c enthält alle Werte. Der Graph in Abbildung 3.19a zeigt für einen detaillierten Vergleich des ersten Trainingsviertels die ersten 600 Episoden. Um die Unterschiede nach dem ersten Trainingsviertel besser erkennen zu können, zeigt der Graph in Abbildung 3.19b einen Ausschnitt der Belohnungswerte von 250 bis 400.

Die grüne und die rote Linie zeigt den Trainingsverlauf unter Anwendung oder oben beschriebenen, modifizierten Reward-Funktion, wobei letztere das Experiment mit `start_exploration_rate=0.5` und `max_exploration_rate=0.5` beschreibt. Beide liefern im ersten Viertel des Trainings schlechtere Belohnungswerte als das Training ohne Erkundungsstrategie. Danach überholen sie dieses allerdings und liegen am Ende zwischen dem Training mit $\text{Epsilon}=0$ und der klassischen ϵ -greedy Strategie. Zudem scheinen die Ergebnisse zuverlässiger zu sein als bei $\text{Epsilon}=0$, wie man an der geringeren Standardabweichung – vor allem gegen Ende – erkennen kann.

Der Start mit einer geringeren `start_exploration_rate` führt wie erwartet am Anfang zu schnelleren Ergebnissen, wird allerdings von der Strategie, bei der die `exploratrion_rate` bei 1 startet, überholt und scheint insgesamt etwas inkonsistenter Belohnungen zu liefern, was uns die Standardabweichung verrät.

Um zu beurteilen, wie sich der Agent nach dem Training verhält, sehen wir uns wieder die Boxplots an.

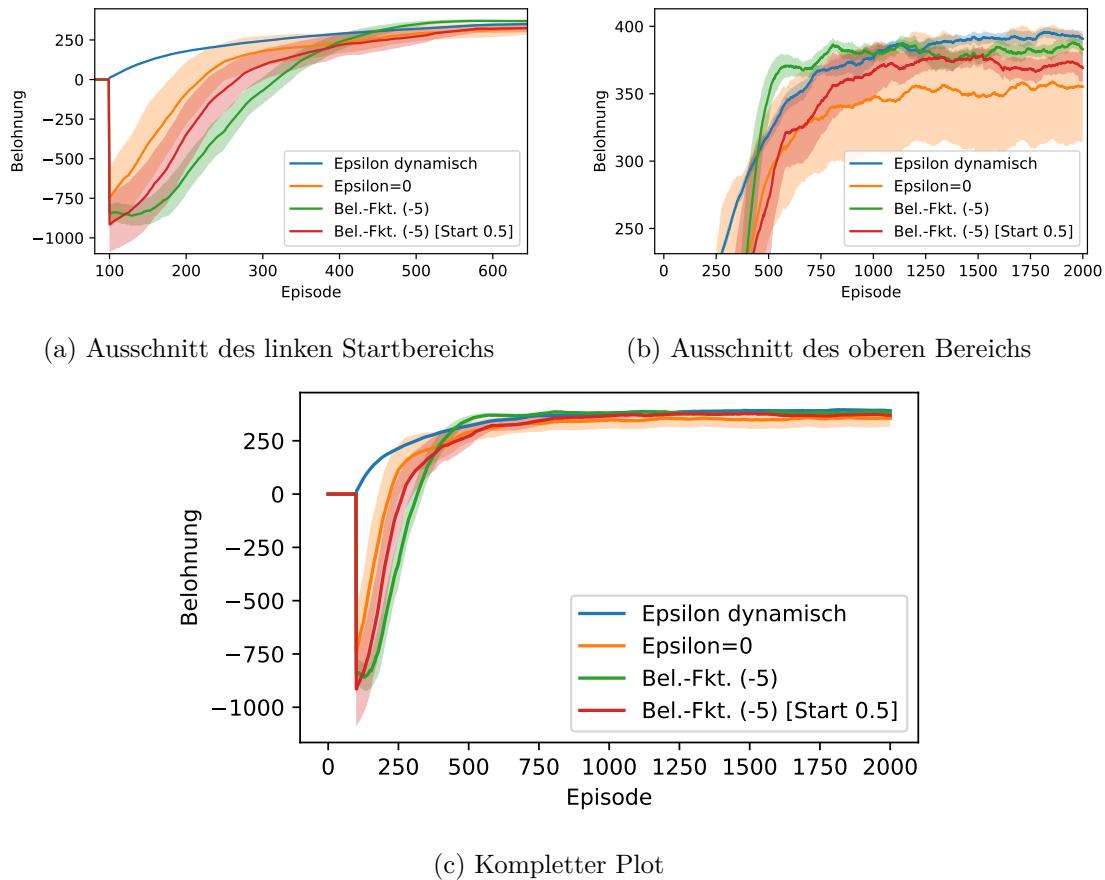
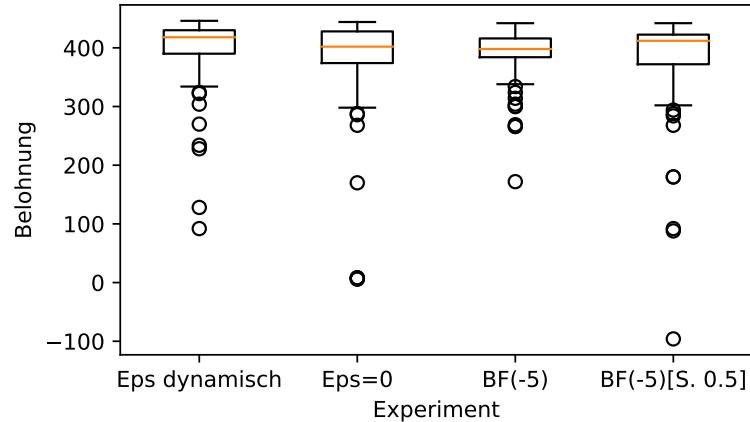


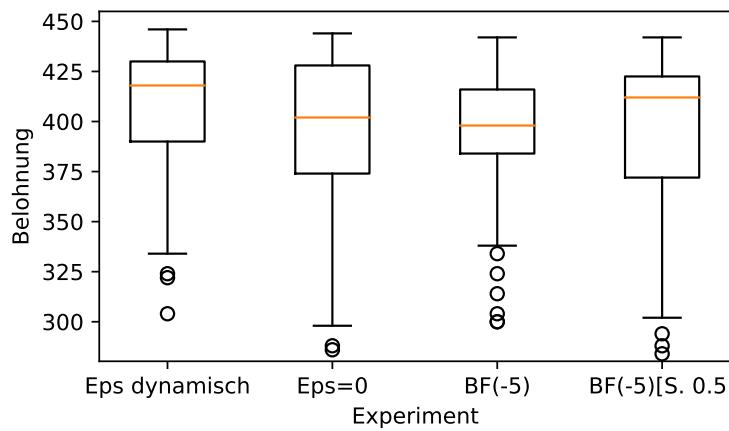
Abbildung 3.19: Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 5 (grün) und modifiziertem Reward mit Faktor 5 mit `start_exploration_rate=0.5` (rot) nach jeweils 20 Wiederholungen.

Die Graphen in Abbildung 3.20 sind genauso aufgebaut wie die in Abbildung 3.18. Vergleichen wir nun die beiden Experimente mit der modifizierten Reward-Funktion miteinander. BF(-5) zeigt die Belohnungen mit der `start_exploration_rate=1`, BF(-5)[S. 0.5] beschreibt dementsprechend die Versuche mit `start_exploration_rate=0.5`. Bei letzterem liegt der Median etwas höher. Dies ist vermutlich wieder darauf zurückzuführen, dass die erkundeten Pfade aufgrund der niedrigen `exploration_rate` öfter durchlaufen wurden und für diese eine optimalere Strategie gefunden wurde. Allerdings ist das Minimum hier wieder niedriger. Wir denken auch hier, dass dies an der geringeren Quantität der durchlaufenden Pfade liegt und der Agent so für weniger Zustände eine Strategie entwickelt hat. Der Interquartilsabstand spiegelt gewissermaßen die Standardabweichung aus Abbildung 3.19 wider. Der Agent mit `start_exploration_rate=1` liefert konsistenter Resultate. Sein Boxplot deckt zudem in Bezug auf Minimum und Maximum einen ähnlichen Bereich ab wie die des klassischen ϵ -greedy Agenten. Die Quartile von letzterem liegen allerdings weiterhin weiter oben, was dessen höheren moving average am Ende vom Graph in Abbildung 3.19b erklärt.

3 Gebirgslandschafts-Domäne



(a) Kompletter Plot



(b) Ausschnitt ohne die unteren Ausreißer

Abbildung 3.20: Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor 5 mit `start_exploration_rate=0.5`

Modifizierter Reward mit zufälligem Faktor Wir wollen nun versuchen, den Schwerpunkt noch etwas mehr auf die zufällige Erkundung der Umgebung zu setzen. Wir passen hierfür unsere Belohnungs-Funktion an:

```
modified_reward = (1 - exploration_rate) * reward -\
    exploration_rate * random.uniform(0.0, 5.0)
```

Die `exploration_rate` wird nun nicht mehr direkt mit unserem errechneten Wert 5 multipliziert, sondern mit einem zufälligen Wert zwischen 0 und 5. Dies soll die zufällige Wahl der Aktionen und damit die zufällige Erkundung der Umgebung gewissermaßen über die Belohnung abbilden. Wir belassen es diesmal bei einem Experiment mit `start_exploration_rate=1` und `max_exploration_rate=1`, da diese Parameter im letzten Experiment konsistenter Werte und ein höheres Endergebnis geliefert haben. Wir vergleichen das Resultat wieder mit der klassischen ϵ -greedy Strategie (blau) und dem fi-

xen Epsilon 0.0 (orange). Außerdem plotten wir das Resultat des letzten Experiments mit `start_exploration_rate=1` (grün). Wir stellen diese Daten so wie in Abbildung 3.19 dar.

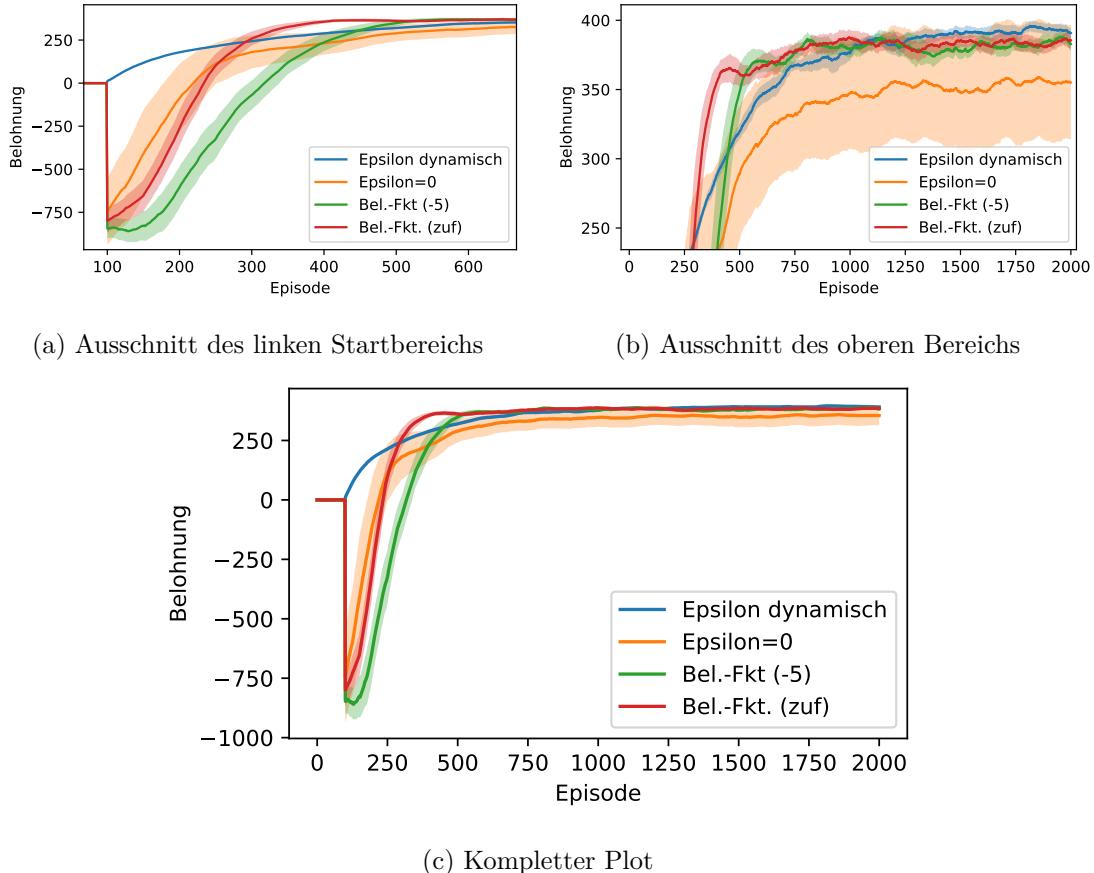


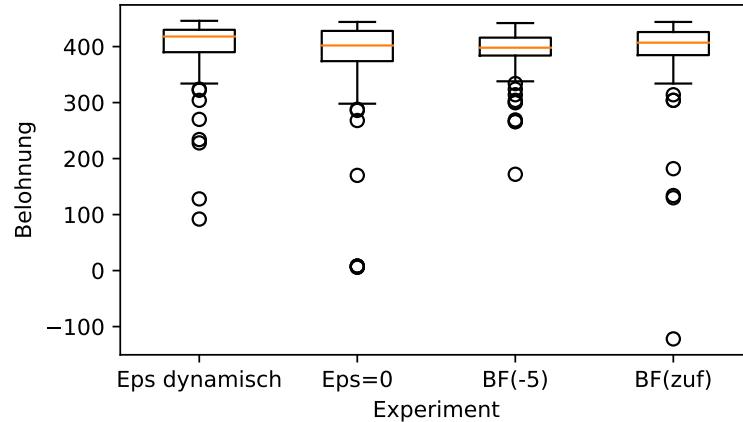
Abbildung 3.21: Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 5 (grün) und modifiziertem Reward mit Faktor zufällig zwischen 0 und 5 (rot) nach jeweils 20 Wiederholungen.

Die Daten des neuen Experiments sind in Abbildung 3.21 rot eingezeichnet. Die Form der Kurve dessen ist ähnlich mit der des vorangegangenen Experiments, allerdings verläuft sie leicht höher, gut erkennbar in Abbildung 3.21a. Eine weitere Ähnlichkeit ist der Belohnungswert, bei dem sich beide gegen Ende des Trainings einpendeln, wie in Abbildung 3.21b zu sehen ist. Allerdings kommt der Agent mit dem Zufallsfaktor in seiner Belohnung etwas schneller bei diesem Wert an, was als eine direkte Verbesserung zum letzten Experiment angesehen werden kann. Im Punkt der Konsistenz stimmen die Experimente augenscheinlich ebenfalls überein, da sich die Standardabweichung der beiden kaum unterscheidet.

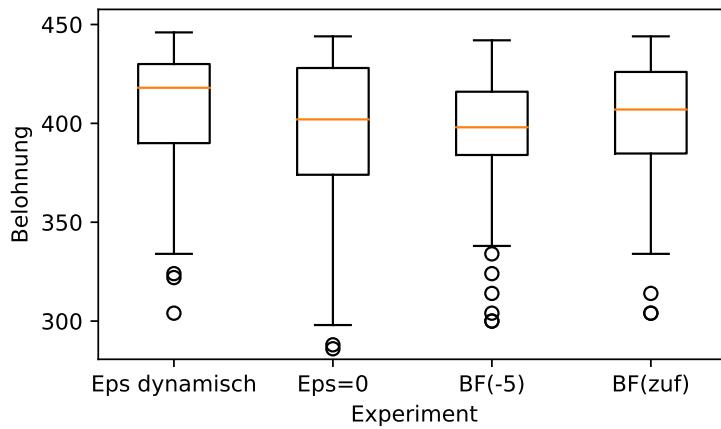
Beide liegen dementsprechend am Ende über dem Wert des Trainings ohne Erkundungsstrategie, allerdings weiterhin unter der klassischen ϵ -greedy Strategie.

Betrachten wir auch für diesen Vergleich die Boxplots der Experimente. Die Graphen in Abbildung 3.22 sind genauso aufgebaut wie in Abbildung 3.20. Auch hier lässt sich erken-

3 Gebirgslandschafts-Domäne



(a) Kompletter Plot



(b) Ausschnitt ohne die unteren Ausreißer

Abbildung 3.22: Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor zufällig zwischen 0 und 5

nen, dass sich die Boxplots der beiden Agenten mit modifizierter Belohnungs-Funktion sehr ähneln. Lediglich der Median vom neuen Experiment (BF(zuf)) ist ein wenig höher als der des letzten Versuchs (BF(-5)). Das Resultat des neuen Experiments ist zudem schon sehr ähnlich zu dem des klassischen ϵ -greedy Agenten (Eps dynamisch). Auch hier scheint nur der Median weiter oben zu liegen. Im Vergleich zu dem des Agenten ohne Erkundungsstrategie liegt vor allem das Minimum aller anderen Experimente ein gutes Stück höher.

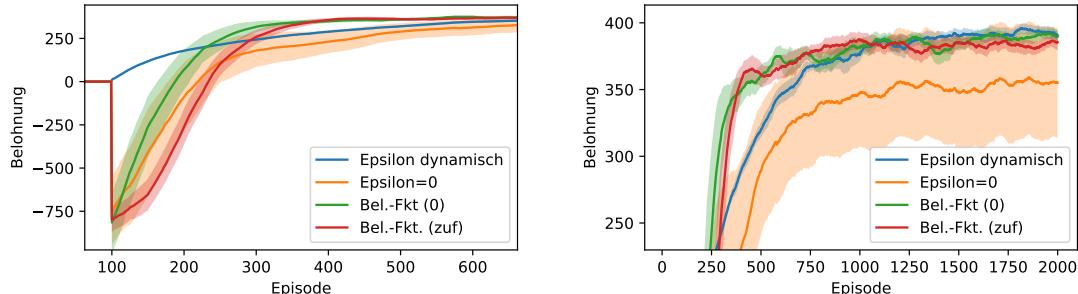
Modifizierter Reward mit Faktor 0 Wir haben im letzten Experiment gesehen, dass das Austauschen des fixen Multiplikators 5 in der Belohnungs-Funktion mit einem zufälligen Wert zwischen 0 und 5 ein schnelleres Ergebnis liefert. Dies könnte bedeuten, dass Faktoren kleiner als 5 grundsätzlich besser funktionieren. Wir werden daher für unser nächstes Experiment diesen Faktor auf 0 setzen:

```
modified_reward = (1 - exploration_rate) * reward - exploration_rate * 0
```

Daher bleibt lediglich der reduzierte Wert der ursprünglichen Belohnung übrig:

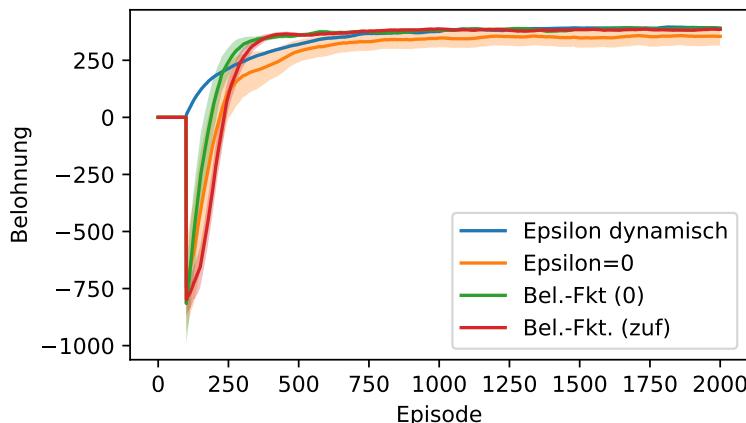
```
modified_reward = (1 - exploration_rate) * reward
```

Wir betrachten erneut die durchschnittliche Belohnung der ϵ -greedy Strategie, Epsilon=0, die des letzten Experiments und natürlich die des aktuellen Experiments. Abbildung



(a) Ausschnitt des linken Startbereichs

(b) Ausschnitt des oberen Bereichs



(c) Kompletter Plot

Abbildung 3.23: Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 0 (grün) und modifiziertem Reward mit Faktor zufällig zwischen 0 und 5 (rot) nach jeweils 20 Wiederholungen.

3.23 ist hierbei wieder ebenso aufgebaut wie Abbildung 3.19. In Abbildung 3.23a fällt auf, dass die Kurve des Experiments ohne zusätzlichen Faktor (Bel.-Fkt. (0)) zu Beginn schneller steigt als die mit zufälligem Faktor und auch die ohne Erkundungsstrategie. Der Graph in Abbildung 3.23b zeigt uns, dass sie circa ab Episode 270 sehr ähnlich zu der des Experiments mit zufälligem Faktor in der Belohnung verläuft. Es lässt sich also festhalten, dass die Strategie dieses Experiments am Anfang des Trainings etwas schneller ist als unsere bisherigen Ansätze.

Sehen wir uns nun noch die Boxplots zu diesem Experiment an. Abbildung 3.24 ist hier wieder so aufgebaut wie Abbildung 3.20. Im Graphen in Abbildung 3.24b sieht man,

3 Gebirgslandschafts-Domäne

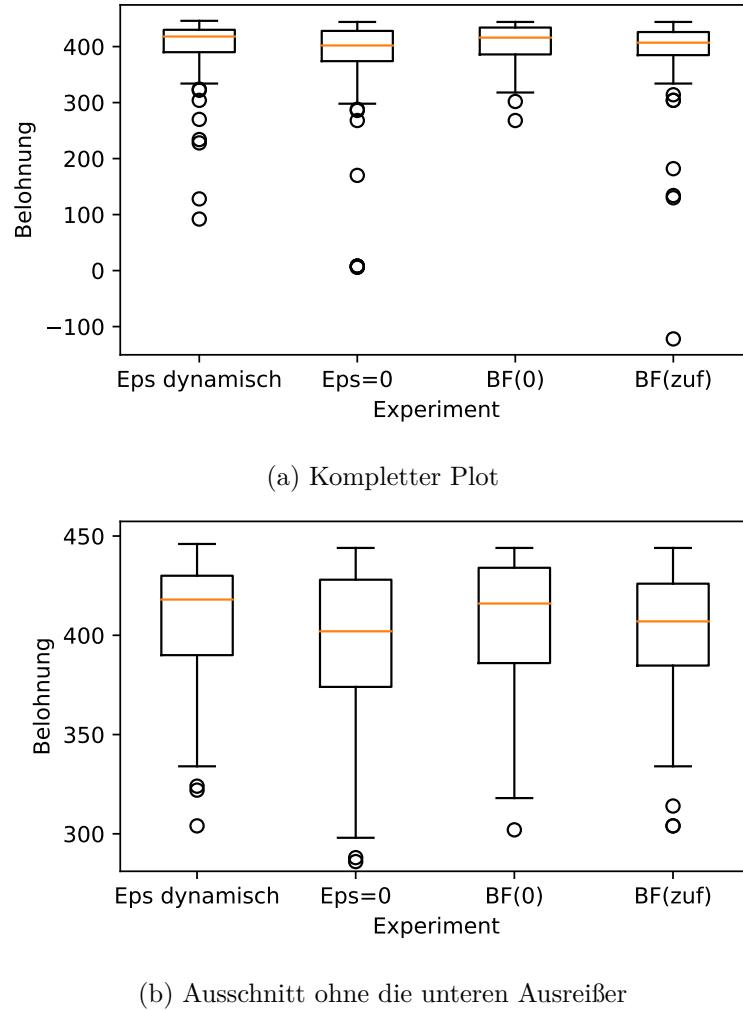


Abbildung 3.24: Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 0, modifizierter Reward mit Faktor zufällig zwischen 0 und 5.

dass das neue Minimum (BF(0)) im Vergleich zum letzten Experiment (BF(zuf)) wieder etwas tiefer liegt. Allerdings ist dafür der Median größer und augenscheinlich fast auf einer Ebene mit dem des klassischen Ansatzes (Eps dynamisch). Außerdem zeigt der Graph in Abbildung 3.24a, dass wir mit dem neuesten Ansatz am wenigsten Ausreißer haben. Dieser Faktor ist eventuell nicht sehr wichtig, fällt aber im Vergleich zu allen anderen bisherigen Boxplots doch auf. Eventuell gleicht diese Tatsache auch zusammen mit dem größeren Median das geringere Minimum aus. Dafür spricht die Beobachtung aus Abbildung 3.23b, dass sich der Verlauf sowie die Standardabweichung der beiden Experimente sehr ähneln.

Aufgrund dessen und vor allem aufgrund der besseren Ergebnisse zu Beginn des Trainings, welche diesmal merklich über denen des Agenten ohne Erkundungsstrategie liegen, kommen wir zu dem Schluss, dass dies für diese Domäne der beste von den von uns getesteten Ansätzen für die Abbildung einer Erkundungsstrategie im Reward ist. Der Agent

3.3 Deep-Q-Learning Experimente

lernt schneller, konsistenter und besser als ohne eine Erkundungsstrategie und die Lernresultate können sich laut Boxplots sogar fast mit denen der ϵ -greedy-Strategie messen.

4 Lunar-Lander-Domäne

Wir wollen nun untersuchen, wie sich ein Agent mit unserer Strategie in einer anderen Domäne verhält.

4.1 Die Umgebung und die Aufgabe

Wir nutzen für die folgenden Experimente das Toolkit *OpenAi Gym* [BCP⁺16]. OpenAi Gym ist eine Kollektion von Environments, die für das Trainieren und Vergleichen von diversen Machine-Learning-Algorithmen gedacht sind. Ein wichtiges Kriterium für die Entwickler ist, die Umgebungen so zugänglich wie möglich zu machen.

Wir verwenden für unsere Versuche das Environment „LunarLander-v2“. Der Agent soll hierbei lernen, eine kleine, zweidimensionale Raumkapsel auf einer Landeplattform (markiert mit gelben Fahnen) zu landen. Hierbei ist es wichtig, dass das Raumschiff sachte auf dem Boden aufsetzt, ohne dass seine Landefüße beschädigt werden. Eine visuelle Repräsentation des Environments ist in Abbildung 4.1 dargestellt.

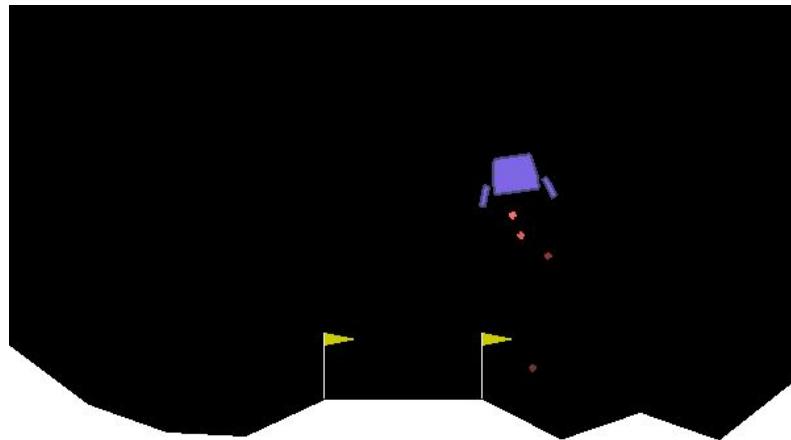


Abbildung 4.1: OpenAi Gym’s „LunarLander-v2“

Quelle: Screenshot vom laufenden Programm des „LunarLander-v2“ aus [BCP⁺16]

Zu Beginn jeder Episode spawnt der Lander am oberen Bildschirmrand mit einem zufälligen Winkel und einer leicht variierenden Position. Sein Zustand wird mit sieben Attributen beschrieben: Die horizontale und vertikale Koordinate, die horizontale und vertikale Geschwindigkeit, der Winkel und die Winkelgeschwindigkeit der Kapsel und ob die einzelnen Landefüße (2 Stück) den Boden berühren. Die Belohnung hängt von mehreren Komponenten ab: Ob sich der Lander der Landezone nähert oder sich davon

entfernt, ob er zum Stillstand kommt oder crasht, ob seine Füße den Boden berühren und natürlich ob er sich in der Landezone befindet. Außerdem erhält er eine kleine Strafe jedes Mal, wenn er das Haupttriebwerk benutzt. Dies ist neben nichts tun, linkes Triebwerk benutzen und rechtes Triebwerk benutzen eine der vier möglichen Aktionen.

Der in diesem Kapitel verwendete Lernalgorithmus ist bis auf kleine Anpassungen mit dem aus Kapitel 2.2.2 identisch. Der Hauptunterschied ist, dass dieses Mal ReLu als Activation-Function der Hidden-Layers verwendet wird.

4.2 Experimente

Klassisches ϵ -greedy Wir beginnen wieder mit dem klassischen ϵ -greedy Ansatz. Nach einigen Vorabtests wählen wir die Hyperparameter wie folgt:

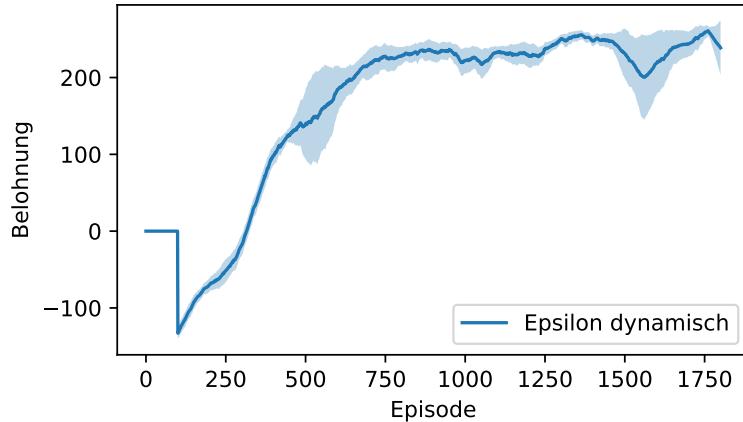
```
params = DeepQParameters(
    num_episodes=2500,
    max_steps_per_episode=0,
    replay_buffer_size=20000,
    batch_size=32,
    learning_rate=0.001,
    discount_rate=0.99,
    target_update=10,
    start_exploration_rate=1,
    max_exploration_rate=1,
    min_exploration_rate=0,
    exploration_decay_rate=0.005,
    # ... Rest wird erst während des Trainings belegt
)
```

Hierbei fällt auf, dass die `max_steps_per_episode` auf 0 gesetzt sind. Dies liegt daran, dass die maximale Schrittanzahl in der Umgebung intern auf 1000 festgelegt ist. Eine Episode endet außerdem, sobald der Lander das Bild verlässt, crasht oder zum Stillstand kommt. Aufgrund der sehr langen Trainingszeit beschränken wir die Anzahl der Iterationen pro Experiment auf 10.

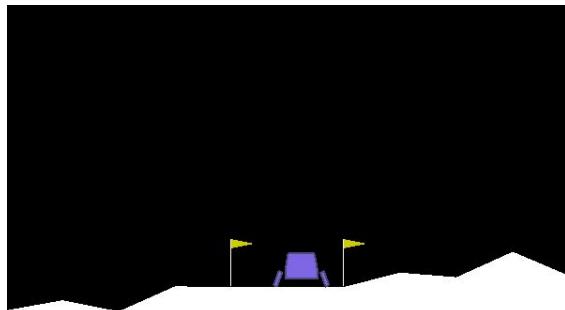
Abbildung 4.2 zeigt die Ergebnisse des ersten Experiments. Der Graph in Abbildung 4.2a zeigt wieder den durchschnittlichen moving average mitsamt seiner Standardabweichung. Die Belohnung steigt bis Episode 750 stetig an und pendelt sich dann bei einem Wert von ungefähr 230 ein. Gegen Ende circa bei Episode 1500 bricht der Plot nochmal leicht ein, steigt aber danach wieder an. Die Standardabweichung ist an den meisten Stellen gering. Die Lernkurve verläuft also weitestgehend wie gewünscht. Auch das Resultat ist sehr positiv, da der Agent unter Anwendung der trainierten DQNs den Lander in den meisten Fällen schnell aber sanft in der Zielzone landet.

Training ohne Erkundungsstrategie Wir wollen nun wieder die unterschiedlichen Ansätze miteinander vergleichen. Um zunächst zu sehen, wie das Training ohne eine Erkundungsstrategie verläuft, setzen wir unser ϵ wieder konstant auf 0.

Wie erwartet zeigt der Graph in Abbildung 4.3, dass das Training ohne Erkundungsstrategie schlechtere Ergebnisse liefert. Die durchschnittliche Belohnung steigt nicht so



(a) Graph so wie in Abbildung 3.5.



(b) Gelandeter Lander unter Verwendung des trainierten DQNs

Abbildung 4.2: Trainingsergebnisse mit dynamischem ϵ nach 10 Wiederholungen.

schnell an wie im letzten Experiment und die Standardabweichung ist größer. Der Lernfortschritt ist also langsamer und inkonsistenter als zuvor, was auch hier klar für die Verwendung der ϵ -greedy Strategie spricht.

Training mit modifiziertem Reward Zuletzt bilden wir die Erkundungsstrategie wieder ausschließlich im Reward ab. Wir nutzen hierfür die Ergebnisse aus Kapitel 3.3.3 und multiplizieren die Belohnung, die die Umgebung zurückgibt, mit `(1 - exploration_rate)`:

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

Damit der Agent zu Beginn überhaupt eine Belohnung bekommt starten wir mit einer `exploration_rate` von 0.5, die `start_exploration_rate` und `max_exploration_rate` werden also auf 0.5 gesetzt. Der Agent agiert für dieses Experiment wieder ausschließlich greedy.

Wir vergleichen den Verlauf des Trainings mit denen der vorangegangenen Experimente. Für den Vergleich haben wir in diesem Versuch selbstverständlich wieder den echten Reward der Umgebung genommen. Betrachten wir zunächst die ersten 800 Episoden,

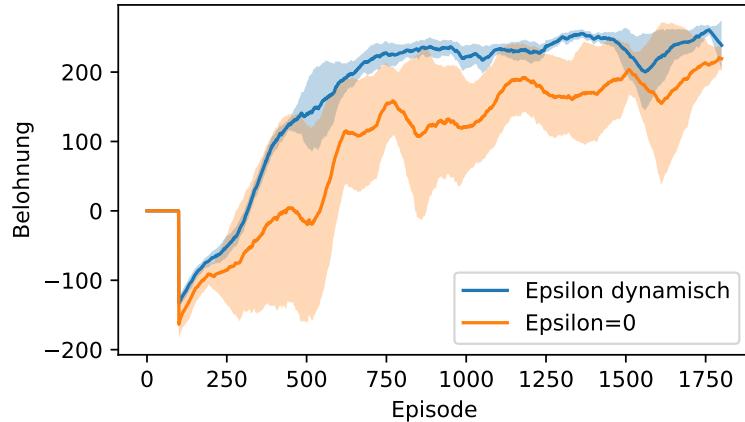


Abbildung 4.3: Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau) und statischem $\epsilon = 0.0$ bzw. keiner Erkundungsstrategie (gelb).

dargestellt in Abbildung 4.4a. Die Belohnung steigt etwas schneller an als bei $\epsilon = 0$ und die Standardabweichung ist geringer. In Episode 800 liegt der Belohnungswert etwa auf gleicher Höhe mit dem der klassischen ϵ -greedy Strategie. Es sieht so aus, als würde sich unsere Strategie hier besser schlagen als ein Agent ohne Erkundungsstrategie. Betrachten wir nun allerdings den Verlauf nach Episode 800, gut zu erkennen in Abbildung 4.4b, so fällt auf, dass der moving average sogar unter den Wert vom Experiment mit $\epsilon = 0$ fällt. Die Standardabweichung bricht hier ebenfalls sehr stark aus. Beides bessert sich erst gegen Ende des Trainings wieder. Diese Tatsache lässt unsere Strategie hier nicht ganz so gut aussehen. Es lässt sich deswegen keine absolut klare Aussage über einen definitiven Vorteil unserer Strategie gegenüber dem Nichtverwenden einer Erkundungsstrategie treffen.

Betrachten wir noch die Boxplots aus Abbildung 4.5 für die Experimente nach jeweils 100 Durchläufen mit den jeweiligen DQNs. Wie erwartet ist der Median beim klassischen ϵ -greedy Ansatz am größten, wenn auch nicht um viel. Vergleichen wir die letzten beiden Versuche miteinander, so lässt sich erkennen, dass alle Quantile unseres Experiments über denen des Versuchs ohne Erkundungsstrategie liegen. Unser Ansatz scheint etwas bessere Ergebnisse zu liefern als ein Training ohne Erkundungsstrategie und der Boxplot kann dem der ϵ -greedy-Strategie fast das Wasser reichen. Allerdings lässt sich anhand dieser Daten schwer eine endgültige Aussage hierüber treffen. Erforderlich wären eine größere Episodenanzahl, wesentlich mehr Iteration pro Experiment und weitere Durchläufe zum Anpassen der Hyperparameter. Hierfür fehlen uns für diese Domäne im Zuge dieser Arbeit die erforderlichen Rechenressourcen.

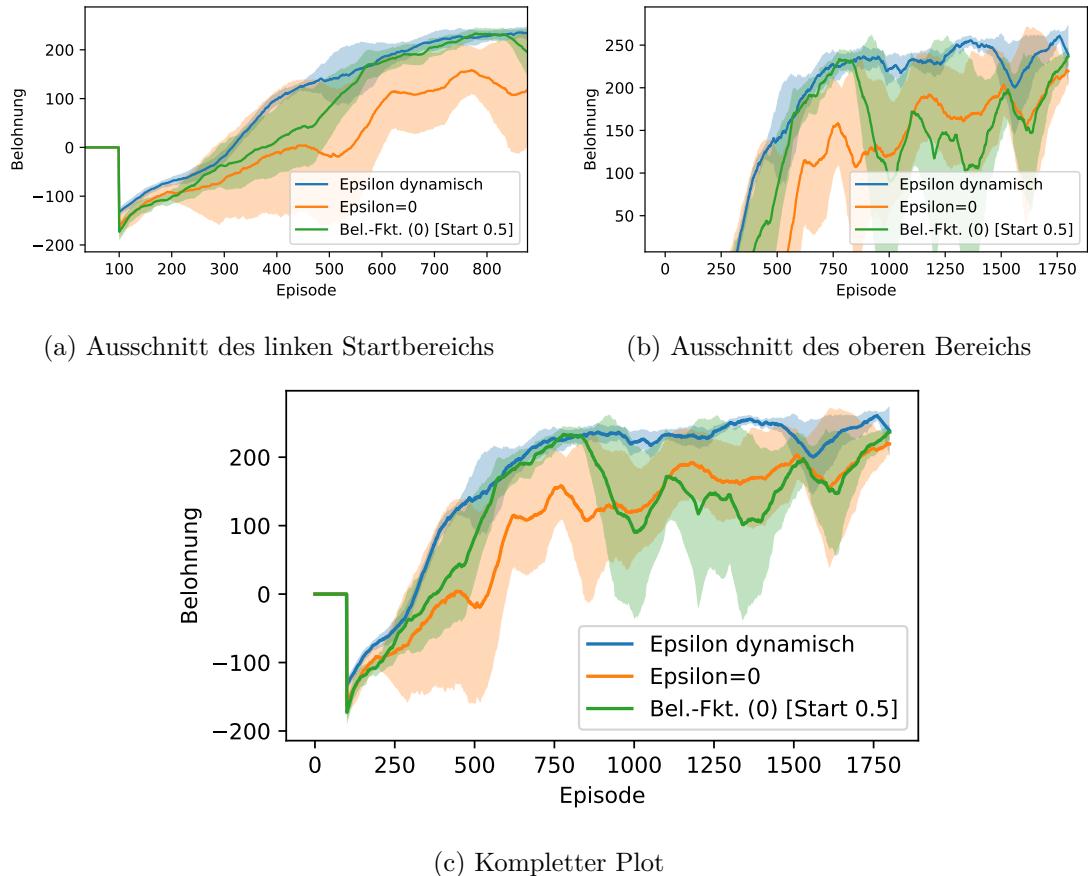
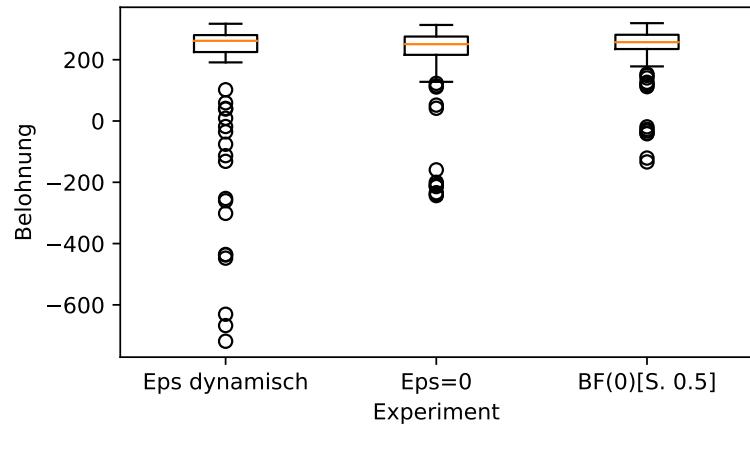
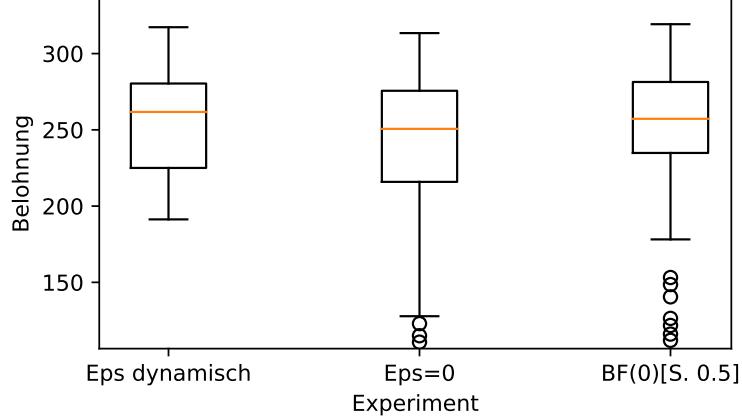


Abbildung 4.4: Graphen so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb) und modifiziertem Reward mit Faktor 0 mit `start_exploration_rate=0.5` (grün) nach jeweils 20 Wiederholungen.



(a) Kompletter Plot



(b) Ausschnitt ohne die unteren Ausreißer

Abbildung 4.5: Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor 0 mit `start_exploration_rate=0.5`.

5 Related Work

Das Entwickeln von alternativen Erkundungsstrategien stellt für viele Forscher ein interessantes Gebiet dar. Bisher gibt es noch keinen Ansatz, der eine Erkundungsstrategie nur durch die Modifikation des Rewards während des Trainings implementiert. Es existieren allerdings ein paar ähnliche Strategien:

In [SB06] wird für die optimale Erkundung der Umgebung ein eigener Markov Decision Process (MDP) formuliert, der sogenannte *derived MDP*. Wenn der Agent nach der optimalen Policy des derived MDPs agiert, führt er eine für das Erlernen einer optimalen Policy für den eigentlichen MDP optimale Erkundung der Umgebung aus. Abbildung 5.1 zeigt eine schematische Repräsentation des Algorithmus. Der externe Zustand sowie die Belohnung, welche von der Umgebung zurückgegeben werden, werden wie gewohnt für das Aktualisieren der Task Value Function des MDPs, also die der eigentlichen Aufgabe, verwendet. Die Wahl der Aktion hängt allerdings von der Behavior Value Function, also der des derived MDPs, ab. Diese nutzt einen intrinsischen Reward, welcher von der Entwicklung der Task Value Function abhängt. Hier wird also neben der externen Belohnung eine hiervon abhängige weitere Belohnung erzeugt, welche Auswirkungen auf das Lernverhalten des Agenten hat. Es lässt sich argumentieren, dass diese zweite Belohnung Ähnlichkeiten mit unserer modifizierten Belohnung hat.

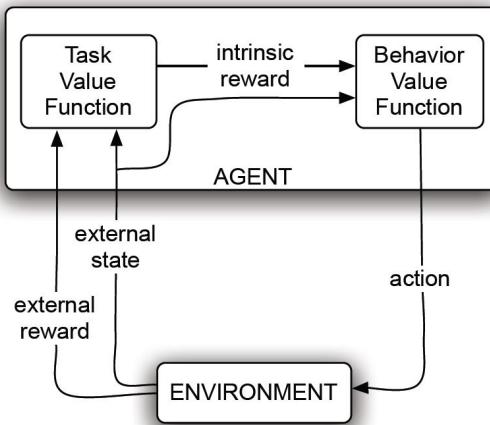


Abbildung 5.1: Schematische Repräsentation des Algorithmus

Quelle: [SB06]

Das Konzept von intrinsischen Rewards ist natürlich nicht neu. Der bekannte KI-Forscher Jürgen Schmidhuber erklärt in [Sch09], dass *curiosity* dazu genutzt werden kann, den Agenten zur aktiven Erkundung der Umgebung anzuregen. Der Agent soll hierfür die beobachteten Daten aufgrund von auftretenden Regularitäten zusammenzufassen. Die komprimierte Version der Daten kann als deren vereinfachte Erklärung aufgefasst wer-

5 Related Work

den. Daten, welche sich nicht in bisher bekannte Regelmäßigkeiten einordnen lassen, sollen für den Agenten interessanter sein, da diese zum Erlernen einer besseren Komprimierung aller Daten beitragen. Deshalb erhält der Agent zusätzlich einen internen Reward, wenn er lernt die bisher gesammelten Daten mit weniger Bits darzustellen, welcher zusammen mit dem externen Reward maximiert werden soll. Diese Methode soll auch in Umgebungen mit spärlicher Belohnung dazu führen, dass der Agent diese erkundet und lernt wie sie funktioniert. Dieser intrinsische Reward, welchen Schmidhuber als *curiosity reward* bezeichnet, hat also ebenfalls starke Auswirkung auf das Erkundungs- und Lernverhalten, weswegen wir diesen Ansatz hier erwähnen.

Im Gegensatz zu unserer Idee, den Reward in Bezug auf ϵ zu modifizieren, existiert außerdem der gegenteilige Ansatz, den Wert von ϵ während des Trainings je nach erhaltenem Reward zu kontrollieren. [dSMdR17] nennt dies *adaptive ϵ -greedy* und geht hierbei so vor, dass nach einer gewissen Anzahl von zufällig gewählten Aktionen überprüft wird, ob die durchschnittliche Belohnung größer ist als beim letzten Mal. Falls ja wird ϵ entsprechend angepasst, ist sie geringer, so wird ϵ aus 0.5 gesetzt. In Abbildung 5.2 ist diesen Algorithmus dargestellt. [dSMdR17] zeigt, dass die adaptive ϵ -greedy-Methode in den

Algorithm 1 Adaptive ϵ -greedy

```

1:  $max_{prev} \leftarrow 0$                                 8:   if  $\Delta > 0$  then                               15:    $max_{prev} \leftarrow max_{curr}$ 
2:  $k \leftarrow 0$                                          9:      $\epsilon \leftarrow sigmoid(\Delta)$                          16:    $k \leftarrow 0$ 
3: if normal distribution  $\leq \varepsilon$  then                10:   else                                         17:   end if
4:    $max_{curr} \leftarrow Q_t(A_t^*)$                    11:     if  $\Delta < 0$  then                           18:   randomly selects an action
5:    $k \leftarrow k + 1$                                  12:        $\epsilon \leftarrow 0.5$                            19:   else
6:   if  $k = l$  then                                13:     end if                                20:   selects  $A_t^*$ 
7:      $\Delta \leftarrow (max_{curr} - max_{prev}) * f$     14:   end if                                21:   end if

```

Abbildung 5.2: Adaptiver ϵ -greedy-Algorithmus

Quelle: [dSMdR17]

durchgeführten Experimenten zu besseren Ergebnissen führt als die ϵ -greedy Methode. Man muss allerdings erwähnen, dass das ϵ bei letzterem einen statischen Wert hat.

Eine weitere Lernmethode, die ohne einen Reward auskommt, ist das in [EGIL18] beschriebene Erlernen von Fähigkeiten mittels *Diversity*. In einer unüberwachten Phase soll der Agent zunächst eine Menge von Fähigkeiten erlernen, die sich stark voneinander unterscheiden. Hierzu folgt der Agent dem DIAYN-Algorithmus („Diversity is all you need“). Die Belohnung für die einzelnen Aktionen hängt in dieser Phase von der Unterscheidbarkeit der Fähigkeiten ab. Anders gesagt sollen sich die besuchten Zustände so unterschiedlich wie möglich sein. In Abbildung 5.3 ist das Ergebnis dieser Phase für eine simple 2D Navigation dargestellt. Der Agent startet hier in der Mitte und kann sich im Raum bewegen. Es lässt sich erkennen, dass die erlernten Fähigkeiten, welche hier als farbige Pfade dargestellt sind, aufgrund ihrer Ungleichheit einen Großteil des Zustandsraums abdecken. Wenn nun die Aufgabe in der nächsten Phase lautet, dass der Agent nach unten links gehen soll, so ist der lila Pfad ein sehr vielversprechender Kandidat. Die Tatsache, dass die erlernten Fähigkeiten im Idealfall so unterschiedlich wie möglich sind, stellt nach [EGIL18] sicher, dass einige brauchbare Fähigkeiten dabei sind. Soll der Agent nun ein tatsächliches Problem in der Domäne lösen, werden die gefundenen Fähigkeiten als mögliche Aktionen genutzt. [EGIL18] zeigt anhand von einigen Experimenten, dass die DIAYN-Methode zu schnelleren Lernerfolgen führt.

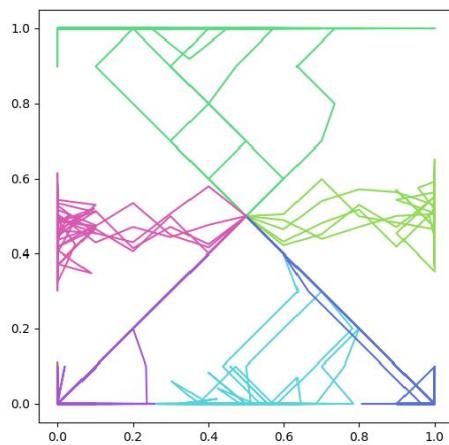


Abbildung 5.3: Mittels DIAYN erlernte Skills in einem 2D-Environment

Quelle: [EGIL18]

6 Ergebnisse und Fazit

Der Reward ist die zentrale Komponente beim Reinforcement Learning. Wir konnten durch die Experimente in dieser Arbeit ein Gefühl dafür entwickeln, inwieweit er sich für die Abbildung einer Erkundungsstrategie eignet.

Zu Beginn der Arbeit haben wir in Kapitel 2 die Grundlagen von Q-Learning und Deep-Q-Learning, sowie Exploration und Exploitation erklärt. Auf dieser Basis haben wir im Anschluss in Kapitel 3 auf unserer eigens für diese Arbeit entwickelten Domäne gezeigt, dass die Modifikation des Rewards beim Q-Learning eine größere durchschnittliche Belohnung erzielt als das Nichtbenutzen einer Erkundungsstrategie. Anschließend haben wir über die Änderung des Rewards, welcher von der Umgebung zurückgegeben wird, das Ziel des Agenten so angepasst, dass es für unsere Experimente mit Deep-Q-Learning geeignet ist. Wir sind hier zu dem Schluss gekommen, dass die Multiplikation des Rewards mit dem Faktor (`1 - exploration_rate`) sowohl in Hinblick auf Trainingsverlauf und -konsistenz als auch auf das Resultat einen Vorteil gegenüber einem Agenten ohne Erkundungsstrategie bringt und sogar teilweise ähnlich gut performt wie ϵ -greedy. Die ϵ -greedy-Strategie war allerdings wie erwartet in beiden Fällen in jeder Hinsicht die beste Strategie. In Kapitel 4 haben wir die erarbeitete Strategie auf der Domäne des Lunar Landers getestet und konnten feststellen, dass unsere Strategie auch hier Potenzial besitzt. Die Unterschiede sind hier nicht so deutlich wie bei unserer Domäne, dennoch führt die Modifikation des Rewards zu einem im Großen und Ganzen besseren Ergebnis als beim Agenten ohne Erkundungsstrategie. Im Vergleich lernt unser Ansatz zu Beginn des Trainings etwas schneller und liefert etwas bessere Resultate, allerdings stürzt der durchschnittliche Belohnungswert während des Trainings bis unter den des Vergleichsagenten ab und die Standardabweichung bricht aus. Für ein eindeutiges Ergebnis sind auf dieser Domäne mehr Daten erforderlich. In Kapitel 5 haben wir verwandte Arbeiten und deren Vorgehen betrachtet.

Auf unserer eigenen Domäne hat sich somit gezeigt, dass der von uns erarbeitete Ansatz sowohl beim Q-Learning als auch beim Deep-Q-Learning bessere Ergebnisse liefert als ein Agent ohne Erkundungsstrategie. Die Trainingsresultate waren bei den Deep-Q-Learning-Experimenten sogar ähnlich gut wie bei der Verwendung der ϵ -greedy-Strategie. Beim Lunar Lander war das Ergebnis nicht ganz so eindeutig. Unsere Versuche haben zwar gezeigt, dass unsere Strategie auch hier das Training positiv beeinflusst, lässt aber einige Fragen offen. So ist beispielsweise unklar, warum der Belohnungsdurchschnitt nach seinem Höhepunkt wieder sehr stark abfällt. Um hier eine klare Aussage treffen zu können, benötigen wir mehr Daten, welche wir aufgrund des Mangels an nötigen Rechenressourcen im Zuge dieser Arbeit nicht erlangen konnten.

Wir sind uns außerdem darüber im Klaren, dass die 10 Iterationen beim Lunar Lander keine starke statistische Aussagekraft haben. Die Ergebnisse dieser Domäne stellen also eher eine Tendenz dar. Es steht ebenfalls fest, dass die erfolgreiche Anwendung in ein bis zwei Domänen nicht für das Funktionieren unserer Methode im Allgemeinen spricht.

Wir konnten allerdings die zugrunde liegende Fragestellung dieser Arbeit, ob das Abbil-

6 Ergebnisse und Fazit

den einer Erkundungsstrategie über den Reward eine Daseinsberechtigung hat, beantworten. Die gesammelten Ergebnisse zeigen, dass unser Ansatz dahingehend funktioniert, die Performance im Vergleich zu einem Trainieren ohne Erkundungsstrategie zu verbessern und er dementsprechend Potenzial hat. Wir haben Reward-Funktionen gefunden, die wir rein greedy lernen, die sich aber ähnlich verhalten wie ϵ -greedy und deren Lernverhalten trotz der Modifikation des Rewards eine sehr hohe Stabilität besitzen. Verlagert man möglichst viele Komponenten des Lernalgorithmus in die Reward-Funktion, so muss man am Ende nur deren Korrektheit überprüfen und läuft weniger Gefahr eine künstliche Intelligenz zu schaffen, die etwas anderes lernt als das, was wir vorgesehen haben. Im besten Fall haben wir so die Möglichkeit genau aufzuschreiben, was der Agent versucht zu tun.

Für eine weitere Forschungsarbeit wäre es interessant, unsere Strategie zum einen ausgiebiger auf der Domäne des Lunar Lander zu testen, um hier eine aussagekräftige Statistik erstellen zu können und die offenen Fragen zu klären. Zudem könnte man die Experimente auf weitere bekannte Domänen ausweiten und mit bisherigen Forschungsergebnissen vergleichen. Es wäre außerdem spannend zu erforschen, wie sich eine reaktive Modifikation des Rewards auf den Verlauf und die Resultate des Trainings auswirkt, die Anpassung der Belohnung also unter Berücksichtigung des aktuellen Zustands und des bisherigen Trainingsverlaufs stattfindet.

Abbildungsverzeichnis

| | | |
|------|--|----|
| 2.1 | Interaktion zwischen Umgebung und Agent in einem MDP | 3 |
| 3.1 | Visualisierung von zweidimensionalem Perlin Noise | 16 |
| 3.2 | Mittels Perlin Noise zufällig generierte Landschaften | 16 |
| 3.3 | Landschaft, auf der die Experimente durchgeführt werden | 17 |
| 3.4 | Trainingsergebnisse des ersten Experiments | 18 |
| 3.5 | Vergleich der Trainingsverläufe mit (orange) und ohne (blau) ϵ -greedy Strategie nach jeweils 20 Wiederholungen. Der Graph zeigt den durchschnittlichen moving average und dessen Standardabweichung pro Episode. | 19 |
| 3.6 | Trainingsergebnisse mit Erkundungsstrategie codiert im Reward | 20 |
| 3.7 | Trainingsergebnisse mit Erkundungsstrategie codiert im Reward nach 20 Wiederholungen. Graph so wie in Abbildung 3.5. | 21 |
| 3.8 | Ergebnisse des ersten Experiments mit DQN | 23 |
| 3.9 | Ergebnisse mit angepassten Parametern | 24 |
| 3.10 | Ergebnisse mit zufälliger Startposition | 24 |
| 3.11 | Ergebnisse mit zusätzlichen Inputs | 25 |
| 3.12 | Visualisierung der Inputs des Agenten abhängig von den angrenzenden Positionen im Raster | 26 |
| 3.13 | Ergebnisse nach Neudefinition des Lernziels | 27 |
| 3.14 | Ergebnisse mit größerem NEW_POINT_REWARD | 27 |
| 3.15 | Trainingsergebnisse mit Experiment wie in Abbildung 3.14a nach 20 Wiederholungen. Graph so wie in Abbildung 3.5. | 28 |
| 3.16 | Vergleich der Trainingsverläufe mit und ohne ϵ -greedy Strategie nach jeweils 20 Wiederholungen. Graph so wie in Abbildung 3.5. | 29 |
| 3.17 | Vergleich der Trainingsverläufe mit dynamischem ϵ und unterschiedlichen statischen Werten für ϵ nach jeweils 20 Wiederholungen. Graph so wie in Abbildung 3.5. | 30 |
| 3.18 | Boxplots der Belohnungssummen nach jeweils 100 Durchläufen mit den trainierten DQNs (100 pro Experimentreihe, also bei 20 Iterationen somit 5 Durchläufe pro DQN). y-Achse zeigt die Belohnung, x-Achse beschreibt das betrachtete Experiment. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0, \epsilon = 0.2, \epsilon = 0.5$ | 31 |
| 3.19 | Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 5 (grün) und modifiziertem Reward mit Faktor 5 mit <code>start_exploration_rate=0.5</code> (rot) nach jeweils 20 Wiederholungen. | 33 |
| 3.20 | Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor 5 mit <code>start_exploration_rate=0.5</code> | 34 |

Abbildungsverzeichnis

| | | |
|------|--|----|
| 3.21 | Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 5 (grün) und modifiziertem Reward mit Faktor zufällig zwischen 0 und 5 (rot) nach jeweils 20 Wiederholungen. | 35 |
| 3.22 | Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor zufällig zwischen 0 und 5 | 36 |
| 3.23 | Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb), modifiziertem Reward mit Faktor 0 (grün) und modifiziertem Reward mit Faktor zufällig zwischen 0 und 5 (rot) nach jeweils 20 Wiederholungen. | 37 |
| 3.24 | Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 0, modifizierter Reward mit Faktor zufällig zwischen 0 und 5. | 38 |
| 4.1 | OpenAi Gym's „LunarLander-v2“ | 41 |
| 4.2 | Trainingsergebnisse mit dynamischem ϵ nach 10 Wiederholungen. | 43 |
| 4.3 | Graph so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau) und statischem $\epsilon = 0.0$ bzw. keiner Erkundungsstrategie (gelb). | 44 |
| 4.4 | Graphen so wie in Abbildung 3.5. Vergleich der Trainingsverläufe mit dynamischem ϵ (blau), statischem $\epsilon = 0.0$ (gelb) und modifiziertem Reward mit Faktor 0 mit <code>start_exploration_rate=0.5</code> (grün) nach jeweils 20 Wiederholungen. | 45 |
| 4.5 | Boxplots so wie in Abbildung 3.18. Experimente von links nach rechts: ϵ dynamisch, $\epsilon = 0.0$, modifizierter Reward mit Faktor 5, modifizierter Reward mit Faktor 0 mit <code>start_exploration_rate=0.5</code> | 46 |
| 5.1 | Schematische Repräsentation des Algorithmus | 47 |
| 5.2 | Adaptiver ϵ -greedy-Algorithmus | 48 |
| 5.3 | Mittels DIAYN erlernte Skills in einem 2D-Environment | 49 |

Tabellenverzeichnis

Listings

Literaturverzeichnis

- [Arc11] ARCHER, TRAVIS: *Procedurally generating terrain*. In: *44th annual mid-west instruction and computing symposium, Duluth*, Seiten 378–393, 2011.
- [BCP⁺16] BROCKMAN, GREG, VICKI CHEUNG, LUDWIG PETTERSSON, JONAS SCHNEIDER, JOHN SCHULMAN, JIE TANG und WOJCIECH ZAREMBA: *OpenAI Gym*. CoRR, abs/1606.01540, 2016.
- [DOB20] DABNEY, WILL, GEORG OSTROVSKI und ANDRÉ BARRETO: *Temporally-Extended ϵ -Greedy Exploration*, 2020.
- [dSMdR17] SANTOS MIGNON, ALEXANDRE DOS und RICARDO LUIS DE AZEVEDO DA ROCHA: *An Adaptive Implementation of ϵ -Greedy in Reinforcement Learning*. Procedia Computer Science, 109:1146–1151, 2017.
- [EGIL18] EYSENBACH, BENJAMIN, ABHISHEK GUPTA, JULIAN IBARZ und SERGEY LEVINE: *Diversity is All You Need: Learning Skills without a Reward Function*, 2018.
- [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, VOLODYMYR, 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, VOLODYMYR, 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.

Literaturverzeichnis

- [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. BEYGELZIMER, F. d'ALCHÉ-BUC, E. FOX und R. GARNETT (Herausgeber): *Advances in Neural Information Processing Systems 32*, Seiten 8024–8035. Curran Associates, Inc., 2019.
- [Rav18] RAVICHANDIRAN, SUDHARSAN: *Hands-on Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow*. Packt Publishing Ltd, 2018.
- [Rus16] RUSSELL, STUART: *Should we fear supersmart robots?* Scientific American, 314(6):58–59, 2016.
- [SAV20] STEVENS, E., L. ANTIGA und T. VIEHMANN: *Deep Learning with PyTorch*. Manning Publications, 2020.
- [SB06] ŞİMŞEK, ÖZGÜR und ANDREW G BARTO: *An intrinsic reward mechanism for efficient exploration*. In: *Proceedings of the 23rd international conference on Machine learning*, Seiten 833–840, 2006.
- [SB20] SUTTON, RICHARD S und ANDREW G BARTO: *Reinforcement learning: An introduction, second edition*. MIT press, Cambridge, 2020.
- [Sch09] SCHMIDHUBER, JUERGEN: *Driven by Compression Progress: A Simple Principle Explains Essential Aspects of Subjective Beauty, Novelty, Surprise, Interestingness, Attention, Curiosity, Creativity, Art, Science, Music, Jokes*, 2009.
- [Wie60] WIENER, NORBERT: *Some moral and technical consequences of automation*. Science, 131(3410):1355–1358, 1960.