# Vergleich verschiedener Verfahren zur Lösung des MountainCar-v0 Problems

Sebastian Strobel

Hochschule für Angewandte Wissenschaften Hamburg, 20099 Hamburg, Deutschland
Sebastian.Strobel@haw-hamburg.de

**Zusammenfassung.** In dieser Arbeit werden die Verfahren *State Diskretisierung*, *Tile Coding* und *Deep Q-Learning* im Bezug auf die Lösung des MountainCar-v0 Problem betrachtet. Dazu wird als Grundlage der Q-Learning Algorithmus genutzt und um diese Verfahren ergänzt. Ziel ist es, das MountainCar Problem zu lösen, ohne die Rewards der Umgebung zu verändern. Dafür wurde jedes dieser Verfahren implementiert und anhand mehrerer Test die Hyperparameter optimiert. Dabei wurde es bei allen drei Verfahren erfolgreich geschafft, das Ziel mit dem MountainCar zu erreichen. Am besten hat dabei das Tile Coding abgeschnitten, bei dem das MountainCar Problem „offiziell“ nach bereits circa 1000 Episoden mit einer greedy Policy gelöst werden konnte. Das zweitbeste Ergebnis konnte mit Deep Q-Learning erreicht werden, welches ein wenig besser als die State Diskretisierung im Bezug auf die Rewards abgeschnitten hat. Jedoch konnte bei der State Diskretisierung ein deutlich stabileres Trainings- und Testverhalten erreicht werden.

**Schlüsselwörter:** Reinforcement Learning · MountainCar-v0 · State Diskretisierung · Tile Coding · Deep Q-Learning

## 1 Einleitung

### 1.1 MountainCar-v0 Problem

Das Mountain Car Problem [1] ist ein bekanntes Problem im Bereich Reinforcement Learning. Es handelt sich um eine eindimensionale Strecke, die zwischen zwei Bergen platziert ist. Das Ziel ist es, die Steigung des rechten Berges hochzufahren. Jedoch ist der Motor des Autos zu schwach, um dies zu aus reiner Motorleistung bewältigen. Die einzige Möglichkeit ist es demnach, zuerst den linken Berg rückwärts ein Stück hochzufahren und dann mit dem geholten Schwung, zusätzlich zu der Motorleistung, die Steigung des rechten Berges zu überwinden. Die Umgebung besteht aus einem kontinuierlichen Zustandsraum und einem diskreten Aktionsraum. Der Zustand des Autos wird zu jedem Zeitpunkt von einen Vektor beschrieben, der die Position und Geschwindigkeit des Autos beinhaltet. Die Position kann Werte im Bereich von $[-1.2, 0.6]$, die Geschwindigkeit Werte im Bereich von $[-0.07, 0.07]$ annehmen. Die Position des Ziels in der Umgebung liegt bei 0.5. Wenn die Position des Autos $< 0.5$ ist, bekommt das MountainCar einen Reward von -1, wenn das Ziel erreicht wird einen Reward von 0. Die

Startposition des Autos befindet sich zufällig in dem Bereich von $[-0.6, -0.4]$, die Geschwindigkeit beträgt Anfangs immer 0. In jedem Schritt kann das Auto zwischen 3 Aktionen wählen: Nach links Beschleunigen (Aktion: 0), garnicht Beschleunigen (Aktion: 1) oder nach rechts Beschleunigen (Aktion 2). Eine Episode terminiert, wenn die Position des Autos $> 0.5$ (Ziel erreicht) ist oder 200 Schritte vergangen sind.

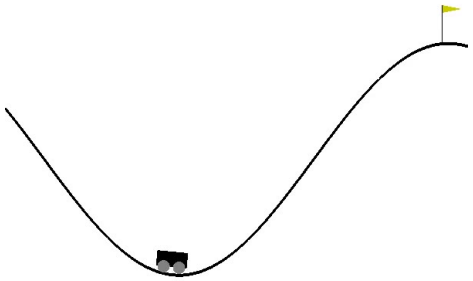


**Abb. 1.** Visualisierte MountainCar-v0 Umgebung

Die Schwierigkeit des Problems liegt unter anderm in den *sparse rewards*. Allgemein wird im Reinforcement Learning das Model abhängig vom Reward geupdatet. Wenn immer der gleiche Reward (-1) erhalten wird, ist nicht offensichtlich, wie die Parameter des Models geupdated werden sollen. Daher werden immer zufällige Aktionen ausgeführt, bis aus Zufall das Ziel erreicht wird und das MountainCar einen anderer Reward (0) bekommt, was unter Umständen lange dauern kann. Da der Reward von 0 anfangs sehr selten ist, kann es sein, dass bis zum Erreichen des Rewards viele Aktionen ausgeführt wurden. Daher ist zunächst nicht klar, welche Aktionen letztendlich für Erfolg verantwortlich waren, was zu einen erschwerten Trainingsprozess führt. Das MountainCar Problem gilt als gelöst, wenn in 100 aufeinanderfolgenden Episoden ein durchschnittlicher Reward von $>= -110$ erreicht wird.

### 1.2 Q-Learning Grundlagen

Q-Learning wurde erstmals 1989 von Watkins [2] vorgestellt. Es ist ein off-policy TD control Algorithmus mit dem Ziel, in jedem State die beste Action zu wählen. Q-Learning wird als Off-Policy bezeichnet, da sich die aktualisierte Policy von der Verhaltens-Policy unterscheidet. Dazu wird eine Action-Value Funktion gelernt, die eine Vorhersage des mit jeder Aktion $a$ in jedem State $s$ verbundenen Rewards beinhaltet [3]. Sie ist definiert als:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \qquad (1)$$

mit

- $t$: Diskreter Zeitschritt $t$ während einer Episode
- $S_t$: State zum Zeitpunkt $t$, typischerweise abhängig von $S_{t-1}$ und $A_{t-1}$
- $A_t$: Aktion zum Zeitpunkt $t$
- $\alpha$: Lernrate
- $R_{t+1}$: Reward $r$ zum Zeitpunkt $t+1$
- $\gamma$: Discount Paramter. Beschreibt, wie viel Einfluss die zukünftigen Rewards der Aktionen $a$ auf die Updates des relativen States $s$ haben
- $\max_a Q(S_{t+1}, a)$: Gibt maximalen Wert von Q für alle möglichen Aktionen $a$ im nächsten State $S_{t+1}$ zurück

Der Algorithmus sieht wie folgt aus:

**Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$**

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal

**Abb. 2.** Q-Learning Algorithmus. Übernommen von [4], Kapitel 6.5

Um eine Konvergenz zu erreichen, muss sichergestellt werden, dass alle State-Action Paare oft genug besucht werden und immer weiter geupdated werden [4].

### 1.3 Policys

Q-Learning ist ein modellfreier Algorithmus. Das bedeutet, dass der Agent die Umgebung erforscht und von den Rewards der Aktionen lernt, ohne ein internes Modell zu erlernen. Der Agent kennt zu Beginn den Aktions- und Zustandsraum und lernt dann durch die Erkundung der Umgebung. Es gibt verschiedene Ansätze, die Umgebung zu erkunden und eine Balance aus *exploration* (eine zufällige Aktion ausführen, um neues zu entdecken) und *exploitation* (das aktuelle Wissen ausnutzen) herzustellen.

**Greedy Policy** Eine greedy Policy wählt die nächste Aktion immer anhand des bereits erlernten Wissens. Dabei wird immer die Aktion mit dem höchsten erwarteten Reward gewählt. Das Problem dieser Policy ist, dass der Agent in

einem suboptimalen Zustand stecken bleiben kann, da er die bestmögliche Aktion noch nie gesehen hat und diese somit auch nicht mehr durch *exploration* finden wird. Außerdem kann auf eine Änderung der Umgebung nach gewisser Zeit nicht gut reagiert werden [5]. Eine greedy Policy kann wie folgt beschrieben werden [4]:

$$A_t \doteq \arg\max_a Q_t(a) \tag{2}$$

**$\epsilon$-greedy Policy** Um eine Balance zwischen *exploration* und *exploitation* herzustellen, kann eine $\epsilon$-greedy Policy genutzt werden. Mit einer über die Zeit sinkenden Wahrscheinlichkeit $\epsilon$ entscheiden wir, wie häufig eine zufällige Aktion ausgeführt werden soll, um die Umgebung weiter zu entdecken. Anfangs ist der Wert von $\epsilon$ hoch, beispielsweise 0.9. Daraufhin wird eine zufällige Zahl in dem Bereich [0, 1] ausgewählt. Ist diese Zahl größer als $\epsilon$, wird die bestmögliche Aktion ausgewählt, also einer greedy Policy gefolgt. Ist die Zahl kleiner, wird eine zufällige Aktion des Aktionsraums ausgewählt [6]. Die suboptimalen Aktionen sind bei dieser Auswahl gleich gewichtet. Der Wert für $\epsilon$ wird mit der Zeit immer kleiner, weshalb immer häufiger einer greedy Policy gefolgt wird. Im Gegensatz zu der reinen greedy Policy wird hier den Zustandsraum viel besser erkundet, weshalb dieses Verhalten zu besserem Erfolg führt. Wenn nach einer gewissen Zeit $\epsilon = 0$ gilt, handelt es sich um eine reine greedy Policy. Um dies zu verhindern, wird ein kleiner $\epsilon\_min$ Wert gesetzt, z.B. 0.01, damit immer eine kleine Wahrscheinlichkeit vorhanden ist, die Umgebung weiter zu erkunden. Dies ist besonders von Vorteil, wenn sich die Umgebung mit der Zeit ändern sollte. Ein Beispel für eine $\epsilon$-greedy Aktionsauswahl ist in Abbildung 3 zu sehen.

**Algorithm 2:** Epsilon-Greedy Action Selection

```
Data: Q: Q-table generated so far, : a small number, S: current
      state
Result: Selected action
Function SELECT-ACTION(Q, S, ϵ) is
    n ← uniform random number between 0 and 1;
    if n < ϵ then
        A ← random action from the action space;
    else
        A ← maxQ(S,.);
    end
    return selected action A;
end
```

**Abb. 3.** Beispel einer $\epsilon$-greedy Aktionsauswahl, Übernommen von [5]

**Boltzmann exploration Policy** Anstatt immer die beste oder eine zufällige Aktion zu wählen, wird bei der Boltzmann exploration eine Aktion mit einer gewichteten Wahrscheinlichkeit gewählt. Um dies zu erreichen, wird eine softmax

Funktion für die vorhergesagten Werte eines Netzwerkes genutzt. Sie garantiert, dass die Output-Werte in Summe den Wert „1“ ergeben, wobei die Output-Werte unterschiedlich gewichtet sind, und damit unterschiedliche Wahrscheinlichkeiten haben. Daher wird die beste Aktion mit einer höheren Wahrscheinlichkeit, aber nicht garantiert ausgewählt. Somit wird die *exploration* garantiert. Im Gegensatz zu $\epsilon$-greedy werden hier die suboptimalen Aktionen nicht gleich stark, sondern anhand ihrer Werte gewichtet. Mit diesem Vorgehen kann die Wahrscheinlichkeit verringert werden, dass schlechtere Aktionen zufällig ausgeführt werden, was ein Vorteil gegenüber $\epsilon$-greedy ist.

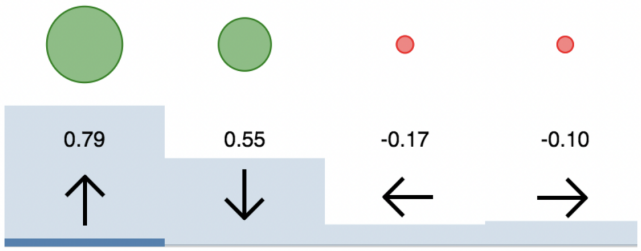



**Abb. 4.** Beispel einer Boltzmann Verteilung, Übernommen von [7]. Die Zahlen beschreiben den Q-Value einer Aktion an einem Punkt des Zustandsraums. Die Höhe der hellblauen Balken beschreibt die Wahrscheinlichkeit, dass diese Aktion gewählt wird.

### 1.4 Ziel der Arbeit

Aufgrund der in Unterabschnitt 1.1 genannten Eigenschaften der Umgebung ist es schwer, das MountainCar zu trainieren. Ziel dieser Arbeit ist es, der Umgebung bei Verwendung von Q-Learning (Unterabschnitt 1.2), genau ein Verfahren hinzuzufügen, um den Trainingsprozess zu erleichtern. Dazu wird bei allen Verfahren die Standardumgebung aus dem Open-AI Gym importiert. Diese hinzugefügten Verfahren sind Tile Coding, State Diskretisierung und Deep Q-Learning. Mit diesen Verfahren soll das MountainCar-v0 Problem gelöst werden, ohne die Rewards der Umgebung zu ändern. Dazu werden diese Verfahren implementiert und die Ergebnisse miteinander verglichen.

## 2 Verfahren

Um die Ergebnisse vergleichbar zu machen, werden die Action-Value Funktionen der Verfahren trainiert und mit Testepisoden, die dem Umfang von 10% der Trainigsepisoden entsprechen, validiert. Bei allen Verfahren werden die Rewards der Umgebung nicht geändert. Da das Ergebnis eines Trainings abhängig von den Initialwerten der Q-Tabelle und den Gewichten des Netzes sein kann, werden

alle Verfahren zehn mal trainiert und getestet. In den Graphen der Auswertung ist der Durchschnitt dieser zehn Läufe abgebildet. Für eine übersichtliche Darstellung der Ergebnisse werden diese in einem einheitlichen Format visualisiert. Außerdem wurde bei allen Graphen eine horizontale Linie bei $y = -110$ eingefügt die den Wert repräsentiert, beim dem das Problem als gelöst gilt.

### 2.1 State Diskretisierung

In Unterabschnitt 1.2 wurde erwähnt, dass der Q-Learning Algorithmus konvergiert, wenn jedes State-Action Paar oft genug besucht wurde. Die MountainCar-v0 Umgebung hat einen kontinuierlichen Zustandsraum, was bedeutet, dass es theoretisch unendlich viele State-Action Paare gibt. Daher ist es unmöglich diese Bedingung zu erfüllen. Eine Möglichkeit dieses Problem zu lösen ist, den Zustandsraum zu diskretisieren. Dazu kann das erste Elements des Zustandsvektors auf das nächste Zehntel (z.B. 0.1) und das zweite Element auf das nächste hundertstel (z.B. 0.01) gerundet werden. Darauf hin kann dann das erste Element mit dem Faktor 10 und das zweite Element mit dem Faktor 100 multipliziert werden. Dieses Vorgehen reduziert die Anzahl möglicher State-Action Paare von unendlich vielen auf 855, weshalb es jetzt möglich ist, die Voraussetzung für die Konvergenz zu erfüllen [8].

Die Umsetzung im Code sieht wie folgt aus:

**Listing 1.1.** State Diskretisierung

```
# Discretize state
state_adj = (state - env.observation_space.low)
            * np.array([10, 100])
state_adj = np.round(state_adj, 0).astype(int)
.
.
# Discretize state2
state2_adj = (state2 - env.observation_space.low)
             * np.array([10, 100])
state2_adj = np.round(state2_adj, 0).astype(int)
```

### 2.2 Tile Coding

Bei kontinuierlichen Zustandsräumen wird für das Lernen der Wertefunktion eine Form der Funktionsannäherung benötigt. Dafür kann Tile Coding verwendet werden, was bereits in mehreren Reinforcement Learning Systemen erfolgreich eingesetzt wurde [[9], [10], [11]]. Dabei wird der kontinuierliche Zustandsraum in sogenannte „Tiles“ partitioniert. Jede Partition wird „Tiling“, und jedes Element der Partition „Tile“ genannt. Die einfachste Form von Tiling in einem zweidimensionalen Zustandsraum ist ein gleichmäßiges Gitter, wie in Abbildung 5 aufgezeigt.

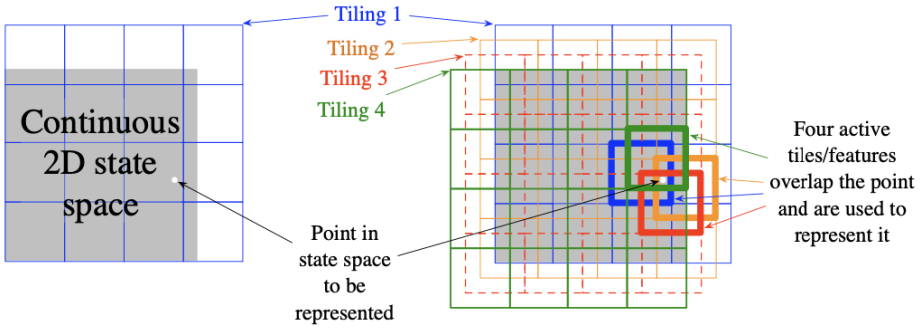



**Abb. 5.** Mehrere, sich überschneidende Tilings in einem begrenzten zweidimensionalen Raum. Diese Kacheln sind in jeder Dimension um einen einheitlichen Betrag voneinander entfernt. Übernommen von [4], Figure 9.9

Abbildung 5 ist ein einfaches Beispiel für vier Tilings. Der weiße Punkt fällt genau in ein Tile in jedem der vier Tilings. Diese Tilings entspechen den vier Features die aktiv werden, wenn dieser Zustand auftritt. Der feature Vektor $x(s)$ hat eine Komponente für jedes Tile in jedem Tiling. In diesem Beispiel gibt es 4 x 4 x 4 = 64 Features von denen alle Null sind, außer die vier, die dem Zustand $s$ entsprechen [4]. Aus Abbildung 5 kann erkannt werden, dass Tile Coding mit nur einem Tile eine einfache Aggregation des Zustands ist, bei dem ein Feature von dem Tile repräsentiert wird in welches es fällt. Bei der Nutzung mehrerer Tilings wird diese Idee erweitert, indem ein Feature durch mehrere Tiles von mehreren Tilings dargestellt wird. Die Anzahl der aktiven Features entspricht somit der Anzahl der Tiles. Dies ist leistungsfähiger, da nun jedes Feature mehrere Tiles teilen und gleichzeitig zu mehreren Tiles gehören kann [12]. Tile Coding erlaubt somit eine Form der Diskretisierung, wobei die überlappenden Tilings einen bestimmen Grad der Generalisierung garantieren. Der geschätzte Wert eines Features wird durch die Addition der Gewichte der Tiles ermittelt, in denen er enthalten ist [13]. Tile Coding hat durch die Verwendung von binären Feature-Vektoren auch rechnerische Vorteile. Da jedes Feature entweder Null oder Eins ist, ist die geschätzte Wertefunktion fast trivial zu berechnen [4].

### 2.3 Deep Q-Learing

Ein Grund für die Entwicklung des Deep Q-Learnings war es mit Umgebungen, welche einen kontinuierlichen Zustandsraum haben, umgehen zu können. Bei Deep Q-Learning wird ein neuronales Netz genutzt, um die Q-Value Funktion zu approximieren. Dabei ist der State die Eingabe des Netzes und der Q-Value aller möglichen Aktionen die Ausgabe. Deep Q-Learning ersetzt somit die Q Tabelle mit einem neuronalen Netz. Anstatt ein State-Action Paar einen Value in der Q-Tabelle zuzuweisen, werden hier die Input-States zu (Action, Q-Value)

Paaren vom Netz zugewiesen. Ein Vergleich zum Q-Learning ist in Abbildung 6 abgebildet.

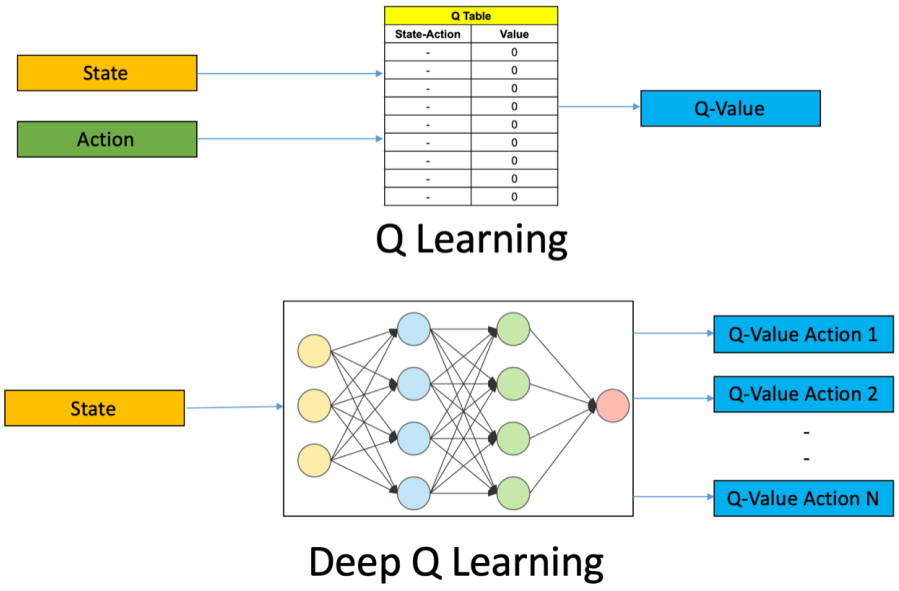



**Abb. 6.** Q-learning vs. Deep Q-learning, Übernommen von [14]

Da das selbe Netzwerk sowohl den Zielwert (*target value*) als auch den vorhergesagten Wert (*predicted value*) berechnet, kann es bei diesen Werten zu großen Abweichungen kommen. Daher werden zwei neuronale Netze benutzt, das Main-Netzwerk und das Target-Netzwerk. Diese haben die gleiche Architektur, jedoch sind die Werte von dem Target-Netzwerk eingefroren [14]. Alle $N$ Schritte werden die Gewichte von Main- zu dem Target-Netzwerk kopiert. Dies führt zu einem stabileren Trainingsprozess und einem effektiveren Lernen [15]. Des Weiteren wird ein Memory Buffer implementiert, in dem die bisherigen Erkenntnisse gespeichert werden. Dieses wird benötigt, um beim Training genug Diversität für das Netz zu garantieren. Dabei wird ein Batch zufällig aus dem Buffer erstellt, mit dem das Netz trainiert wird. Dies ist besonders bei Änderungen in der Umgebung relevant. Die Loss-Funktion ist der Mean Squared Error (MSE) vom vorhergesagten Q-Value und dem Ziel Q-Value minus dem tatsächlichen Wert Q*. Da es sich jedoch um ein RL Problem handelt, kennen wir den Ziel und tatsächlichen Wert nicht. Ausgehend von der Bellmann Gleichung erhalten wir:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{Target} - \underbrace{Q(S_t, A_t)}_{Prediction}] \quad (3)$$

wobei der rot eingerahmte Term das target repräsentiert. Für die Werte von $Q(S_{t+1}, a)$ werden die Werte aus dem Target-Netzwerk genutzt. Da $R$ der echte Reward ohne bias ist, wird das Netzwerk mit Hilfe der backpropagation konvergieren [14].

# 3 Optimierung der Verfahren

## 3.1 State Diskretisierung

Als Orientierung wurde der Code von Hayes [8] genommen. Für die Optimierung werden verschiedene Lernraten und $\epsilon - greedy$ Faktoren getestet. Zuerst werden für drei Lernraten verschiedene $\epsilon - greedy$ Faktoren getestet, um den besten Faktor zu ermitteln. Für die Reduktion von $\epsilon$ werden die Werte 0.05, 0.005, 0.0005, decay function und episodic decay getestet mit:

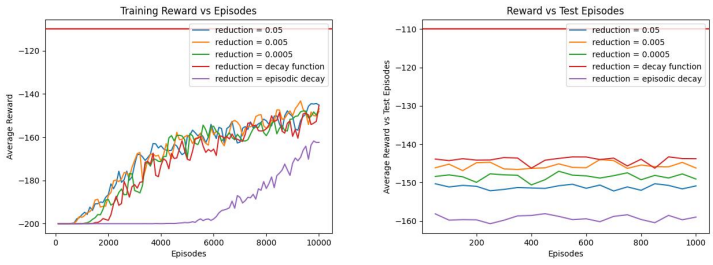
$$episodicdecay = (\epsilon\_start - \epsilon\_min)/Episodes \tag{4}$$

$$decayfunction = (\epsilon\_start * (annealing\_stop - \min(episode, annealing\_stop)) + (\epsilon\_min * \min(episode, annealing\_stop))/annealing\_stop \tag{5}$$

und den Hyperparametern:

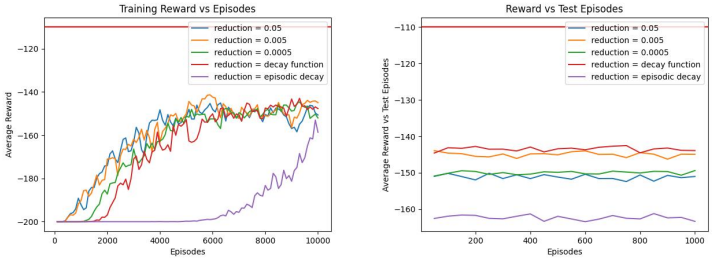
| | |
|---|---|
| Episodes | 10000 |
| $\epsilon\_start$ | 0.9 |
| $\epsilon\_min$ | 0.01 |
| annealing_stop | Episodes / 4 |
| $\gamma$ | 0.99 |

**Tabelle 1.** Feste Hyperparameter State Diskretisierung



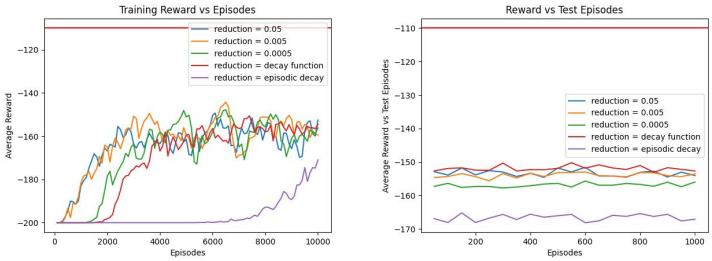

(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 7.** Trainings- und Testverlauf für Lernrate 0.05




(a) Average Trainigsrewards (b) Average Testrewards

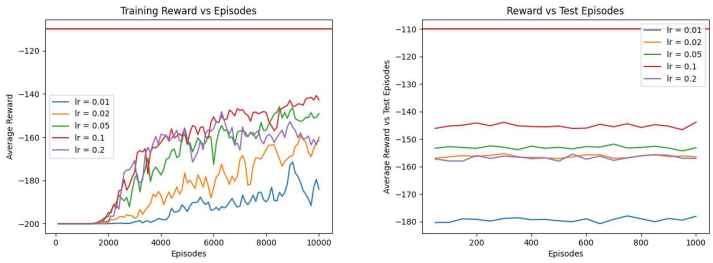
**Abb. 8.** Trainings- und Testverlauf für Lernrate 0.1




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 9.** Trainings- und Testverlauf für Lernrate 0.2

Anhand der Graphen aus Abbildung 7 -Abbildung 9 kann erkannt werden, dass der rote Graph ($\epsilon - decay = decayfunction$) über 10 Trainings- und Testzyklen im Durchschnitt bei allen drei Lernraten das beste Ergebnis liefert. Ausgehend von dieser Erkenntnis werden dann verschiedene Lernraten mit dem $\epsilon - decay$ als decay Faktor getestet.




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 10.** Trainings- und Testverlauf bei $\epsilon - decay = decayfunction$

Ausgehend von diesen Ergebnissen konnte festgestellt werden, dass eine Lernrate von 0.1 in Kombination mit einer $\epsilon - greedy$ Policy mit $\epsilon - decay = decayfunction$ das beste Ergebnis für dieses Verfahren bei einem Training von zehntausend Episoden liefert. Diese Kombination wird in Abschnitt 4 mit den beiden anderen Verfahren verglichen.

### 3.2 Tile Coding

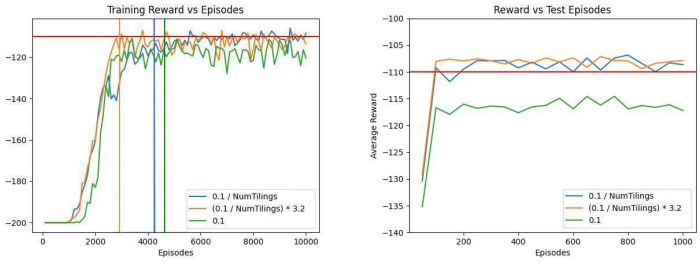
Als Grundlage für das Tile Coding wurde der Code aus einem Github repository [16] genommen. Hier werden ebenfalls verschiedene Hyperparameter auf eine Verbesserung der Ergebnisse getestet. Als Grundlage für alle Tests werden folgende Parameter genutzt:

| | |
|---|---|
| Tiles | 7 |
| Tiles per Tile | 14 |
| Number of Tiles | 1372 |
| Episodes | 10000 |
| $\epsilon\_start$ | 0.9 |
| $\epsilon\_min$ | 0.01 |
| Boltzmann temperature | np.logspace(-1, 1, Episodes) |
| annealing_stop | Episodes / 4 |
| $\gamma$ | 0.99 |

**Tabelle 2.** Feste Hyperparameter Tile Coding

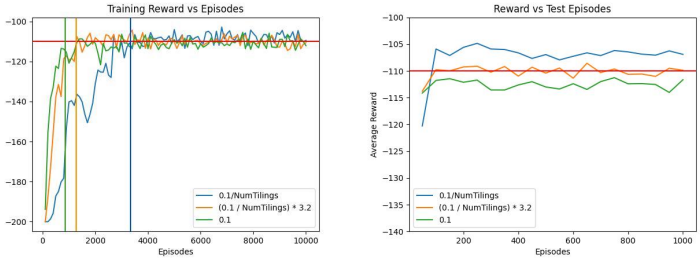
Für die Optimierung werden dabei drei verschiedene Step-sizes und drei verschiedene Policys getestet. Als Lernraten werden dabei zum einen die Lernrate aus dem Code von [16] und 2 typische Lernraten für das Tile Coding getestet. Diese sind $(0.1/NumTilings) * 3.2$ [16], $0.1/NumTilings$ [17], und 0.1 [18]. Für die verschiedenen Policys werden die greedy Policy, die $\epsilon$-greedy Policy und die

Boltzmann exploration Policy verwendet. Die vertikalen Linien in den Graphen beschreiben die durchschnittliche Episode, an der das MountainCar Problem gelöst wurde.




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 11.** Trainings- und Testverlauf für $\epsilon$-greedy policy

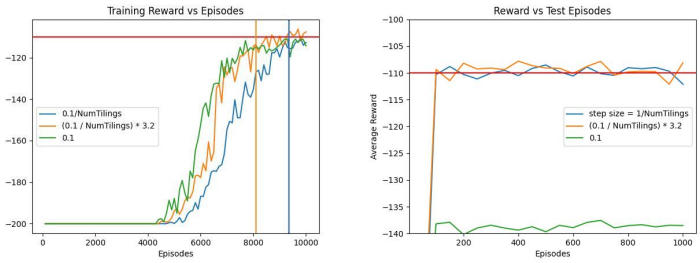



(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 12.** Trainings- und Testverlauf für greedy Policy

Aus den Ergebnissen von Abbildung 11 - Abbildung 13 ist ersichtlich, dass Tile Coding es mit jeder Policy und Step-size geschafft hat, das MountainCar Problem zu lösen (außer bei Boltzmann mit Step-size = 0.1). Bei Boltzmann konnte das Problem bei mehreren Tests immer nur zwischen drei und sieben mal gelöst werden, weshalb kein Durchschnitt über die zehn Tests berechnet werden konnte. Außerdem ist auffällig, dass die Boltzmann exploration mit Lernrate 0.1 trotz eines vielversprechenden Trainingsverlaufes ein schlechtes Testergebnis gegenüber den anderen Lernraten liefert. Dieses Verhalten konnte bei mehreren Tests repliziert werden.

Das Problem konnte am schnellsten mit der greedy Policy gelöst werden und in den Tests hat die greedy Policy ebenfalls am besten abgeschnitten. Des




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 13.** Trainings- und Testverlauf für Boltzmann exploration

Weiteren ist einheitlich zu erkennen, dass die Step-size von 0.1 bei allen Policys im Test am schlechtesten abgeschnitten hat. Die anderen gesteteten Step-sizes haben ähnliche Ergebnisse erzielen können. Das beste Testergebnis wurde dabei mit der greedy Policy und Step-size von $0.1/NumTilings$ erreicht.

### 3.3 Deep Q-Learing

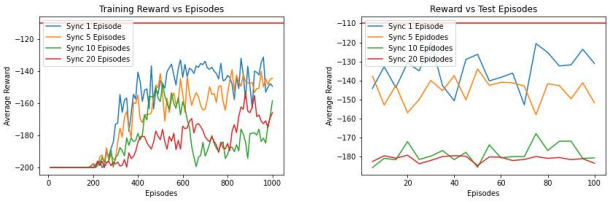
Für dir Optimierung des Deep Q-Learnings werden folgende Hyperparameter genutzt:

| | |
|---|---|
| Layer 1 | size: 24, activation: Relu |
| Layer 2 | size: 48, activation: Relu |
| Layer 3 | size: 3, activation: linear |
| Batch Size | 32 |
| Memory Buffer Size | 10000 |
| Optimizer | Adam |
| Episodes | 1000 |
| $\epsilon\_start$ | 0.9 |
| $\epsilon\_min$ | 0.01 |
| Boltzmann temperature | np.logspace(-1, 1, Episodes) |
| annealing_stop | Episodes / 4 |
| $\gamma$ | 0.99 |

**Tabelle 3.** Feste Hyperparameter Deep Q-Learning

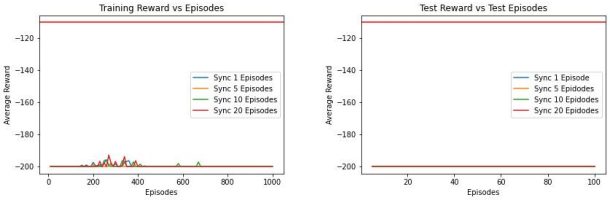
Ausgehend von diesen Parametern werden verschiedene Lernraten und Policys getestet. Als erste Lernrate wurde 0.001 getestet, welche die empfohlene Lernrate für den Adam Optimizer ist [19]. Beim Training eines Modells kann es jedoch von Vorteil für den Erfolg sein, die Lernrate über die Zeit zu reduzieren. Daher werden zwei unterschiedliche Werte $(0.5, 0.9)$ für ein *expotential Decay* [20] der Lernrate untersucht, bei dem die aktuelle Lernrate mit dem Faktor

*decay_rate* multipliziert wird. Durch die Reduzierung der Lernrate während des Traningsprozesses wurde neben der Lernrate von 0.001 eine größere Lernrate von 0.01 getestet. Diese Tests werden in Verbindung mit einer $\epsilon$-greedy Policy durchgeführt. Zuletzt wurde noch überprüft, ob bei der Verwendung einer Bolzmann exploration Policy eine Verbesserung gegenüber der $\epsilon$-greedy Policy erreicht werden kann.



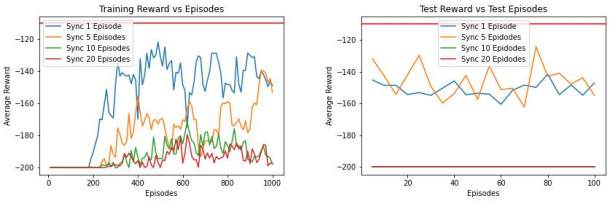

(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 14.** Trainings- und Testverlauf für lr = 0.001



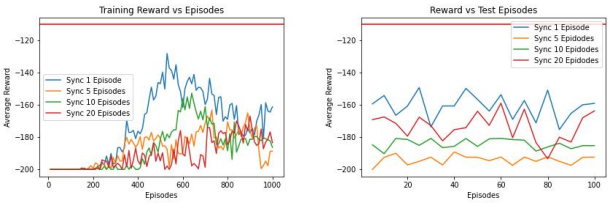

(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 15.** Trainings- und Testverlauf für lr = 0.01 mit ExponentialDecay (decay_steps = 10000, decay_rate = 0.5)



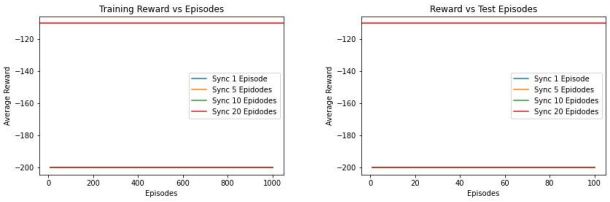

(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 16.** Trainings- und Testverlauf für lr = 0.01 mit ExponentialDecay (decay_steps = 10000, decay_rate = 0.9)




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 17.** Trainings- und Testverlauf für lr = 0.001 mit ExponentialDecay (decay_steps = 10000, decay_rate = 0.9)




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 18.** Trainings- und Testverlauf für Boltzmann exploration Policy

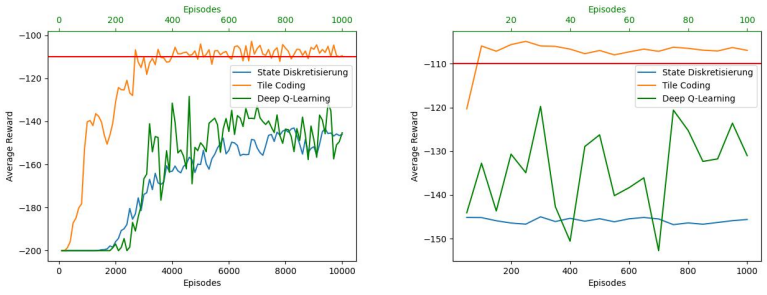
Sowohl bei einem zu kleinen decay Faktor als auch der Boltzmann exploration war es nicht möglich, das MountainCar zu trainieren. Bei der Boltzmann exploration konnten keine unterschiedlichen gewichteten Wahrscheinlichkeiten erreicht werden. Über den Zeitraum der 1000 Trainingsepochen lag die Wahrscheinlichkeit jeder Aktion immer zwischen 32% und 34%, wodurch kein erfolgreicher Trainingsprozess entstehen konnte. Dies lässt vermuten, dass die Boltzmann exploration nicht für sparse Rewards in Verbindung mit DQN's geeignet ist. Da die gleiche Umsetzung bei dem Tile Coding (Unterabschnitt 3.2) und einem DQN für die Lösung für das „CartPole-v1" erfolgreich funktioniert hat, wird dies für wahrscheinlich gehalten. Des Weiteren konnte kein stabiler Trainingsverlauf hergestellt werden. Dies liegt unter anderem an dem Auftreten des „Vergessens", worauf in Abschnitt 6 genauer eingegangen wird. Das beste Testergebnis, mit einen Durchschnittswert von $-134.3$, konnte mit der Lernrate 0.001 und einer Sychronisierung in jeder Episode erreicht werden.

## 4 Vergleich der Ergebnisse

In diesem Kapitel werden die in Abschnitt 3 optimierten Verfahren miteinander verglichen. Aus den Optimierungen ergeben sich folgende Parameter:

| Verfahren | $\epsilon$_start | $\epsilon$_min | $\gamma$ | $\alpha$ | Policy | Eigenschaften |
|---|---|---|---|---|---|---|
| State Diskretisierung | 0.9 | 0.01 | 0.99 | 0.1 | $\epsilon$-greedy | 855 State-Aktion Paare |
| Tile Coding | 0.9 | 0.01 | 0.99 | 0.1 / NumTilings | greedy | 7 Tiles, 14 Tiles per Tiling |
| Deep Q-Learning | 0.9 | 0.01 | 0.99 | 0.001 | $\epsilon$-greedy | 3 Layer (24, 48, 3) |

**Tabelle 4.** Hyperparameter der Optimierten Verfahren




(a) Average Trainigsrewards (b) Average Testrewards

**Abb. 19.** Trainings- und Testverlauf der optimierten Verfahren. Die untere x-Achse mit dem Bereich [0, 10000] in (a) bzw. [0, 1000] in (b) gilt für die State Diskretisierung und das Tile Coding, die obere x-Achse mit dem Bereich [0, 1000] in (a), bzw. [0, 100] in (b) gilt für das Deep Q-Learning

Anhand von Abbildung 19 kann erkannt werden, dass das Tile Coding aus den getesteten Verfahren am besten abgeschnitten hat. Es konnte sowohl beim Training als auch bei den Tests ein durchschnittlicher Reward von $> -110$ erreicht werden. Das zweitbeste Resultat konnte mit dem Deep Q-Learning erreicht werden, obwohl dieses bei den Tests einen instabilen Verlauf hat. Dennoch konnte hier ein bester Reward von $-119.75$ über zehn Episoden und ein durchschnittlicher Reward von $-134.3$ über 100 Episoden erreicht werden. Am schlechtesten hat die State Diskretisierung mit einem durchschnittlichen Reward von $-145.9$ abgeschnitten.

Da Q-Learning einen diskreten Zustandsraum benötigt wird bei der State Diskretisierung der Zustandsraum diskretisiert. Das schlechte Abschneiden des Verfahrens könnte daher kommen, dass entweder die Umgebung zu sehr verallgemeinert wurde oder zehntausend Episoden nicht ausgereicht haben, um jedes State-Action Paar ausreichend oft zu besuchen und ein optimales Verhalten zu erzeugen.

Das Deep Q-Learning hat es in zehn Prozent der Epochen der State Diskretisierung geschafft, ein besseres Ergebnis zu erreichen, obwohl die Umgebung nicht vereinfacht wurde. Ein Grund dafür ist die Verwendung eines Memory Buffers bei dem DQN, aus dem der Agent lernt. Wenn das Netz nur aus aufeinanderfolgenden Erfahrungswerten lernt, die nacheinander in der Umgebung aufgetreten sind, wären die Werte zeitlich korreliert, was zu ineffizienten lernen führen könnte. Die Nutzung zufälliger Batches aus dem Memory Buffer ermöglicht effizienteres Lernen und bricht diese zeitliche Korrelation [21]. In Zusammenhang mit dem Target-Netzwerk, welches das Training zusätzlich stabilisiert, kann das DQN ein besseres Ergebnis als das Q-Learning mit einem diskretisierten Zustandsraum erzielen.

Das Tile Coding ist im Grunde eine Form der Aggregation der Zustände, in der mehrere Grids sich überlappen. Die Vorteile hierbei sind eine bessere Diskriminierung (weniger bias) ohne Verlust der Generalisierung (weniger Varianz) durch die Abdeckung von mehr States mit weniger Features. Diese Eigenschaften helfen dem Tile Coding bei dem Umgang mit sparse Rewards.

## 5 Fazit

In dieser Arbeit wurden drei verschiedene Verfahren vorgestellt, mit denen es gelungen ist, das MountainCar so zu trainieren, dass es trotz der sparse Rewards das Ziel erreicht. Diese Verfahren wurden implementiert und durch die Anpassung verschiedener Hyperparameter optimiert. Mit den Optimierungen konnten die Ergebnisse der einzelnen Verfahren verbessert werden. Am besten hat dabei das Tile Coding mit einem durchschnittlichen Reward von $> -110$ abgeschnitten. Am zweitbesten das Deep Q-Learning mit einem Reward von $-134.3$ und am schlechtesten die State Diskretisierung mit einem durchschnittlichen Reward von $-145.9$. Besonders beim Deep Q-Learning gab es Konvergenzprobleme, die zwar durch die Optimierungen verbessert, aber nicht behoben werden konnten.

Lösungen für dieses Problem und noch weitere Möglichkeiten, zukünftig Verbesserungen zu erreichen, werden in Abschnitt 6 vorgestellt.

## 6 Ausblick

Obwohl das MountainCar mit allen Verfahren nach der Optimierung der Parameter gelöst werden konnte, gibt es noch andere Optimierungsansätze, die zu einer Verbesserung der Ergebnisse führen könnte. Bei der State Diskretisierung könnte versucht werden, eine Methode zu entwickeln, welche die Anzahl der State-Action Paare noch weiter reduziert, um das Training zu erleichtern. Dabei müsste darauf geachtet werden, dass die Anzahl der Paare nicht zu sehr reduziert wird, da sonst die Umgebung zu wenig Informationen beinhalten könnte und keine optimale Q-Tabelle mehr erstellt werden könnte. Das Auto würde immer noch lernen das Ziel zu erreichen, jedoch könnte dies Aufgrung von zu wenig State-Action Paaren auf eine suboptimale Art und Weise erreicht werden, was zu einem schlechteren Reward führen könnte.

Beim Tile Coding könnte getestet werden, ob eine andere Anzahl von Tilings oder eine Erhöhung der Tiles pro Tile eine Verbesserung bringt. Außerdem könnte versucht werden, die Tilings asynchron zueinander zu verschieben.

Unterabschnitt 3.3 zeigt, dass bei dem DQN bei keiner der Kombinationen ein stabiler Trainingsverlauf erreicht werden konnte, obwohl bereits Durchschnittswerte ausgewertet wurden. Des Weiteren ist ersichtlich, dass das Phänomen des „Vergessens“ auftritt, bei dem das Netz im Training wieder schlechter wird, anstatt zu konvergieren. Eine Möglichkeit, das Konvergenzproblem zu lösen ist das Hinzufügen eines festen Buffers Bufferanteils innerhalb des Memory Buffers. Dabei wird eine bestimmter Prozentteil des Memory Buffers, z.B. 10%, nicht aus dem Buffer entfernt, damit eine Diversität mit älteren Daten garantiert wird. Eine weitere Möglichkeit ältere Daten zu berücksichtigen, wäre eine Erhöhung der Memory Buffer Größe. Ist die Größe des Buffers zu groß oder zu klein, kann das Training sich verschlechtern. Eine Verdoppelung der Größe hat in dieser Ausarbeitung keine Verbesserung gebracht. Liu und Zou [21] haben in ihrer Veröffentlichung verschiedene Memory Buffer Größen getestet, wobei bei einer Buffer Größe von 60000-80000 das beste Ergebnis erzielt werden konnte. Allerdings wurde dort das MountainCar doppelt so lange trainiert. Außerdem könnte versucht werden, eine Policy zu nutzen, die mit den sparse Rewards der Umgebung besser umgehen kann, wie beispielsweise die *Bayesian Linear Regression* [[22], [23]].

Zuletzt könnten noch weitere Verfahren, wie beispielsweise die Zeit Diskretisierung getestet werden. Das MountainCar Problem kann nur gelöst werden, wenn das Auto lernt zuerst den hinteren Berg hochzufahren und dann Vollgas den vorderen Berg hochzufahren um die Steigung zu bewältigen. Daher ist es nicht optimal, in kurzer Zeit viele unterschiedliche Aktionen auszuführen. Um dem Auto zu helfen, dies schneller zu lernen, kann eine Zeit-Diskretisierung implementiert werden, sodass es nur alle X Epochen seine Aktion ändert.

## Literatur


1. OpenAI, "Mountaincar-v0," Abgerufen am 10.02.2022, von https://gym.openai.com/envs/MountainCar-v0/.
2. C. J. C. H. Watkins, "Learning from delayed rewards," 1989.
3. G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. Citeseer, 1994, vol. 37.
4. R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
5. Baeldung, "Epsilon-greedy q-learning," Abgerufen am 26.02.2022, von https://www.baeldung.com/cs/epsilon-greedy-q-learning, 01 2021.
6. S. Zychlinski, "The complete reinforcement learning dictionary," Abgerufen am 26.02.2022, von https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e, 02 2019.
7. A. Juliani, "Simple reinforcement learning with tensorflow part 7: Action-selection strategies for exploration," Abgerufen am 26.02.2022, von https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf, 11 2016.
8. G. Hayes, "Getting started with reinforcement learning and open ai gym," Abgerufen am 10.02.2022, von https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f, 02 2019.
9. J. C. Santamaria, R. S. Sutton, and A. Ram, "Experiments with reinforcement learning in problems with continuous state and action spaces," *Adaptive behavior*, vol. 6, no. 2, pp. 163–217, 1997.
10. P. Stone and R. S. Sutton, "Scaling reinforcement learning toward robocup soccer," in *Icml*, vol. 1, 2001, pp. 537–544.
11. R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in neural information processing systems*, vol. 8, 1995.
12. J. Zhang, "Reinforcement learning — tile coding implementation," Abgerufen am 10.02.2022, von https://towardsdatascience.com/reinforcement-learning-tile-coding-implementation-7974b600762b, 07 2019.
13. A. A. Sherstov and P. Stone, "On continuous-action q-learning via tile coding function approximation," *Under Review*, 2004.
14. A. Choudhary, "A hands-on introduction to deep q-learning using openai gym in python," Abgerufen am 10.02.2022, von https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, 04 2019.
15. M. Wang, "Deep q-learning tutorial: mindqn," Abgerufen am 12.02.2022, von https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc, 11 2020.
16. wagonhelm, "Tilecoding rl," Abgerufen am 12.02.2022, von https://github.com/wagonhelm/Tilecoder/blob/master/mountainCarQ.py, 11 2017.
17. R. Sutton, "Tile coding software – reference manual, version 2.1," Abgerufen am 26.02.2022, von http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/RLtoolkit/tiles.html.
18. A. A. Sherstov and P. Stone, "Function approximation via tile coding: Automating parameter choice," in *International Symposium on Abstraction, Reformulation, and Approximation*. Springer, 2005, pp. 194–205.

19. D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
20. Keras, "Exponentialdecay," Abgerufen am 26.02.2022, von https://keras.io/api/optimizers/learning_rate_schedules/exponential_decay/.
21. R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," in *2018 56th annual allerton conference on communication, control, and computing (Allerton)*. IEEE, 2018, pp. 478–485.
22. P. Morere and F. Ramos, "Bayesian rl for goal-only rewards," in *Conference on Robot Learning*. PMLR, 2018, pp. 386–398.
23. T. Wang, D. Lizotte, M. Bowling, and D. Schuurmans, "Bayesian sparse sampling for on-line reward optimization," in *Proceedings of the 22nd International Conference on Machine Learning*, ser. ICML '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 956–963. [Online]. Available: https://doi.org/10.1145/1102351.1102472