

BACHELORARBEIT

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

David Müller

Entwurf vom 16. März 2021



BACHELORARBEIT

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

David Müller

Aufgabensteller: Prof. Dr. Claudia Linnhoff-Popien

Betreuer: Thomas Gabor
Thomy Phan

Abgabetermin: 1. Januar 2099



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

München, den 1. Januar 2099

.....
(Unterschrift des Kandidaten)

Abstract

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

Inhaltsverzeichnis

1 Einleitung	1
2 Navigations-Problem	3
2.1 Das Environment	3
2.2 Q-Learning	4
2.2.1 WORKING TITLE: Theory	4
2.2.2 Implementierung in Python	7
3 Luna-Lander	9
Abbildungsverzeichnis	11
Tabellenverzeichnis	13
Listings	15
Literaturverzeichnis	17

1 Einleitung

2 Navigations-Problem

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

2.1 Das Environment

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

Die simpelste Lösung hierfür wäre wohl, ein zweidimensionales Array mit zufälligen Zahlen zu füllen. Auf diese Weise erhält man ein für jede Koordinate eine zufällige Höhe. Wir wollen allerdings eine Landschaft erstellen, die organisch und natürlich aussieht.

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

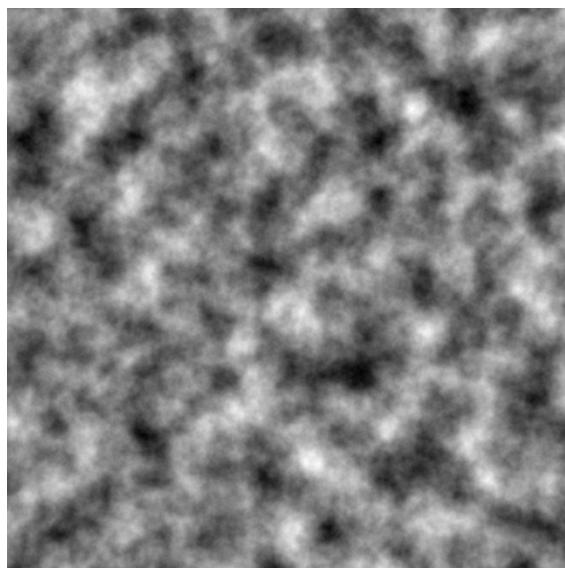


Abbildung 2.1: Visualisierung von zweidimensionaler Perlin Noise

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

Perlin Noise ist nach [Par15] ein fundamentaler Algorithmus in der prozeduralen Generierung von Terrain und somit optimal geeignet, um unsere Umgebung zu erstellen.

2 Navigations-Problem

Wir verwenden eine modifizierte Implementierung von TODO, um ein zweidimensionales Array mit zufälligen Werten zwischen -1 und 1 zu erhalten, welche wir mit einer beliebigen Höhe multiplizieren können. Je nachdem, wie stark man in die Rauschfunktion „hereinzoomt“ erhält man unterschiedliche Verteilungen der Landschaft, wie man in Abbildung 2.2 erkennen kann.

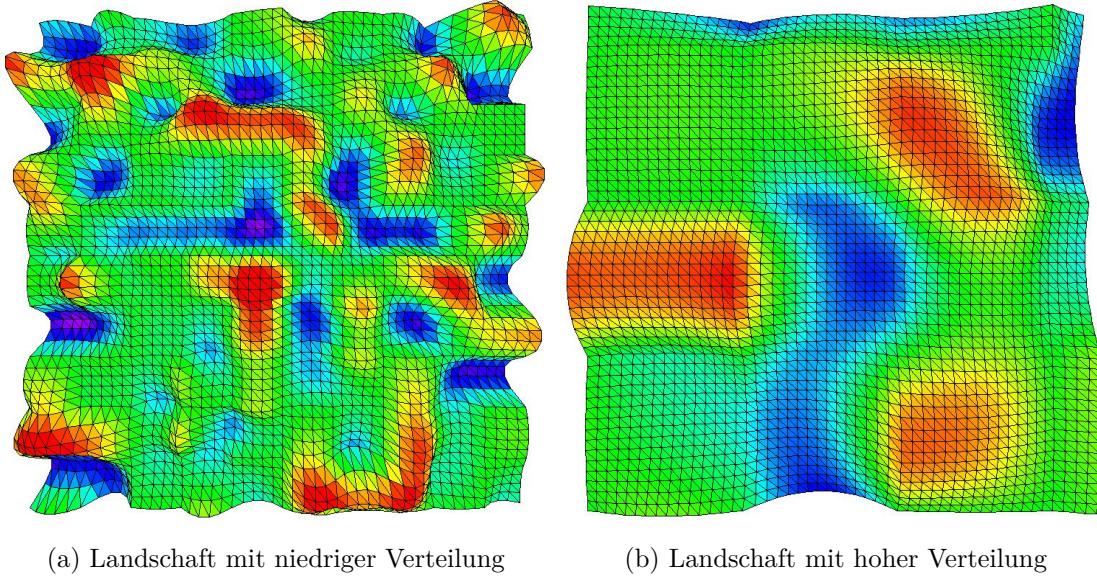


Abbildung 2.2: Mittels Perlin Noise zufällig generierte Landschaften

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

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

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

2.2 Q-Learning

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

2.2.1 WORKING TITLE: Theory

Dieses Kapitel stützt sich zu einem Großteil auf das Buch *Reinforcement Learning: An Introduction, Second Edition* von Sutton u.a. [SB20]. Falls nicht anders angegeben, wurden

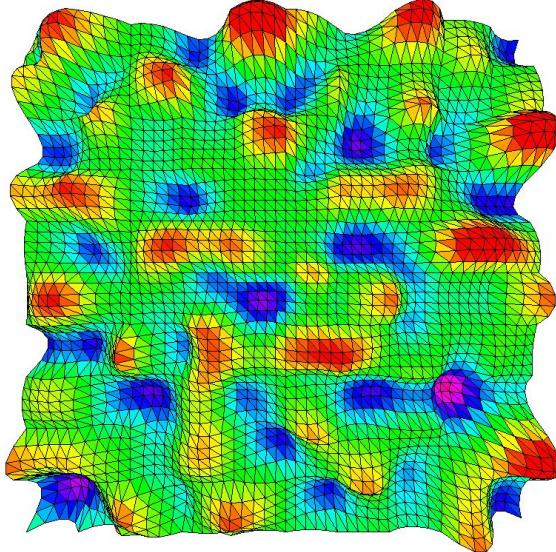


Abbildung 2.3: Visualisierung von zweidimensionaler Perlin Noise

die Informationen hieraus entnommen.

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 $A_t = a$ 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 $s \in S$.

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

$$\begin{aligned} v_\pi(s) &\doteq E_\pi [G_t | S_t = s] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \end{aligned} \tag{2.1}$$

wobei E_π der Erwartungswert einer Zufallsvariable ist, gegeben der Agent folgt der Policy π .

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

Der Wert einer Aktion a im Zustand s unter der Policy π ist also die erwartete Belohnung, wenn man im Zustand s zum Zeitschritt t die Aktion a ausführt. Wir definieren

2 Navigations-Problem

$q_\pi(s, a)$ als

$$\begin{aligned} q_\pi(s, a) &\doteq E_\pi [G_t | S_t = s, A_t = a] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \end{aligned} \quad (2.2)$$

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

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

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in S. \quad (2.3)$$

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

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad (2.4)$$

für alle $s \in S$.

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

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \quad (2.5)$$

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:

$$q_*(s, a) = E \left[R_{t+1} + \gamma \max_{a'} q_*(s', a') \right] \quad (2.6)$$

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, mit discount γ entsprechen muss.

Wir werden die Bellman equation verwenden, um q_* zu finden. q_* wiederum liefert uns die optimale Policy.

2.2.2 Implementierung in Python

3 Luna-Lander

Abbildungsverzeichnis

2.1	Visualisierung von zweidimensionaler Perlin Noise	3
2.2	Mittels Perlin Noise zufällig generierte Landschaften	4
2.3	Visualisierung von zweidimensionaler Perlin Noise	5

Tabellenverzeichnis

Listings

Literaturverzeichnis

- [Arc11] ARCHER, TRAVIS: *Procedurally generating terrain*. In: *44th annual midwest instruction and computing symposium, Duluth*, Seiten 378–393, 2011.
- [Par15] PARBERRY, IAN: *Modeling real-world terrain with exponentially distributed noise*. Journal of Computer Graphics Techniques, 4(2):1–9, 2015.
- [SB20] SUTTON, RICHARD S und ANDREW G BARTO: *Reinforcement learning: An introduction, second edition*. MIT press, Cambridge, 2020.