Masterarbeit

Reinforcement Learning zur Erstellung zustandsorientierter Putzstrategien in konzentrierenden solarthermischen Kraftwerken

Felix Terhag

3. Dezember 2018

Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät

Universität zu Köln

Betreuung: Prof. Dr. Oliver Schaudt

An dieser Stelle möchte ich mich beim Institut für Solarforschung des Deutschen Zentrums für Luft- und Raumfahrt bedanken – insbesondere ist hier Dr. Fabian Wolfertstetter für die inhaltliche und persönliche Unterstützung zu nennen. Schließlich möchte ich Prof. Dr. Oliver Schaudt für die mathematische Betreuung danken.

Inhaltsverzeichnis

Einleitung

Die globale Erwärmung ist eines der Hauptprobleme unserer Zeit. Durch die Unterzeichnung des *Paris Agreement* haben sich 197 Nationen dazu verpflichtet, die globale Erwärmung auf deutlich unter 2 °C gegenüber den vorindustriellen Werten zu begrenzen [UNF15], [Har17]. Dafür muss die CO_2 Emission spätestens 2020 ihren Höhepunkt erreichen und in den folgenden 30 Jahren um ungefähr 90% zurückgehen [Roc+17]. Damit diese Ziele zu erreicht werden können, ist eine weitreichende Umstellung auf erneuerbare Energien unvermeidbar.

Innerhalb der erneuerbaren Energien spielen konzentrierende solarthermische Kraftwerke (CSP)[1,2] eine besondere Rolle. Sie nutzen Spiegel, um die Direktnormalstrahlung der Sonne auf einen Receiver zu bündeln. Dieses Verfahren wird in unterschiedlichen Kraftwerktypen angewandt. Die beiden häufigsten sind Turm- und Parabolrinnenkraftwerke. In Turmkraftwerken gibt es einen zentralen Receiver, auf den alle Spiegel fokussiert sind. Im Gegensatz dazu werden in Parabolrinnenkraftwerken parabolförmige Spiegel in Reihen aufgestellt, in deren Brennpunkt sich ein Receiverrohr mit einem flüssigem Wärmeträgermedium befindet; siehe Abbildung 0.0.1. In dieser Arbeit werden ausschließlich Parabolrinnenkraftwerke betrachtet. Unabhängig von der Bauart wandelt der Receiver des CSP Kraftwerks Strahlung in Wärme um. Diese kann genutzt werden, um Dampf zu erzeugen und eine Turbine anzutreiben [Sol18b]. Der Vorteil dieser Technologie gegenüber anderen Formen der regenerativen Energien – wie zum Beispiel Photovoltaik (PV) oder Windkraft – ist, dass sich die so gewonnene Wärme leicht speichern lässt. Dafür wird sie in große Tanks mit geschmolzenem Salz oder einem anderen Wärmeträger geleitet. Mit Hilfe dieser gespeicherten Wärme kann ein CSP Kraftwerk auch nach Sonnenuntergang regelbaren Strom erzeugen. So können Lastspitzen abgefedert und das Kraftwerk Grundlastfähig werden. Diese Eigenschaft wird umso wichtiger, je größer der Anteil von nicht regelbaren erneuerbaren Energien wie Windenergie und Photovoltaik im Stromnetz ist [Sol18a], [Pie+14].

CSP Kraftwerke verursachen jedoch bislang im Gegensatz zu PV hohe Betriebs- und Wartungskosten. Ein signifikanter Anteil wird durch das Säubern der Kollektoren verursacht [Ba+17].

Bei der Reinigung muss ein Kompromiss gefunden werden: Während häufiges Putzen zu hoher Reflektivität und somit einem hohen Stromertrag führt, kann selteneres Putzen die

[1] Engl. *Concentrating Solar Power*.
[2] Siehe Notationsverzeichnis im Anhang A.

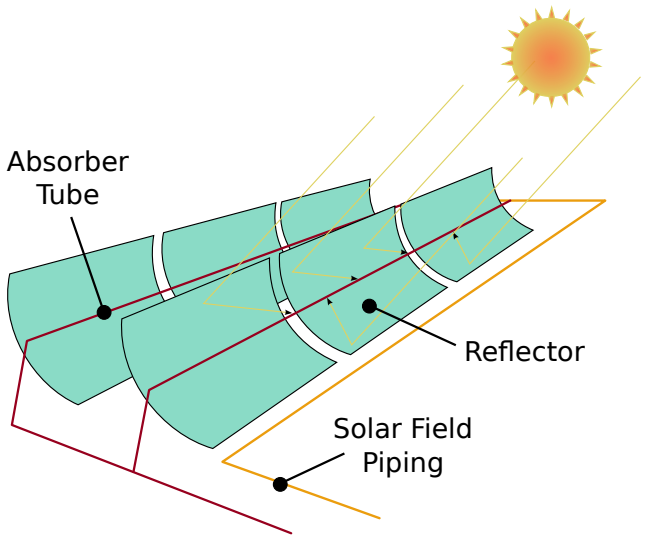


Abbildung 0.0.1: Schematische Darstellung eines Parabolrinnenkraftwerks.[3]

Betriebskosten senken. Dieser Konflikt wird dadurch verstärkt, dass CSP Kraftwerke häufig in Regionen mit hohem Staubaufkommen und gleichzeitiger Wasserknappheit errichtet werden. Gesucht wird also eine Strategie, die das Putzverhalten steuert und zwischen dem Sparen von Wasser- beziehungsweise Putzkosten auf der einen Seite und dem Erhalten einer hohen Reflektivität auf der anderen Seite abwägt.

Die Putzsteuerung sollte auf Basis des jeweils aktuellen Zustands des Solarfeldes geschehen. Dieser kann durch verschiedene Parameter beschrieben werden: So hilft es wahrscheinlich die aktuelle Sauberkeit der Spiegel zu kennen, darüber hinaus kann es auch sinnvoll sein verschiedene Wetterparameter wie Verschmutzungsrate oder Einstrahlung in die Putzentscheidung mit einzubeziehen. Um zu untersuchen welche Parameter für die Strategie von Nutzen sind, muss für verschiedene Parameterkombinationen eine gute Strategie gefunden werden. Vergleicht man zum Beispiel eine Strategie für die nur die aktuelle Feldsauberkeit bekannt ist, mit einer Strategie die außerdem noch Zugriff auf die vorausgesagte Verschmutzungsrate hat, kann abgeschätzt werden wie groß der Mehrgewinn durch die Vorhersage der Verschmutzungsrate ist.

Als geeignete Methode für dieses Problem erscheint *Reinforcement Learning*. Hier findet ein Agent eine Strategie, die eine Belohnung auf Basis von Eingabeparametern maximiert. Der Vorteil dieser Methode ist, dass das Vorgehen bei unterschiedlicher Anzahl von Eingabeparametern gleich bleibt. Das heißt, das Verfahren findet für beliebige sichtbare Zustandsparameter eine gute Lösung. Ein weiterer Vorteil ist, dass Reinforcement Learning

[3]Quelle: By AndrewBuck - Own work, CC BY-SA 4.0, `https://commons.wikimedia.org/w/index.php?curid=3569495`. Aufgerufen am 28.11.2018.

schon erfolgreich auf mehrere Kontrollprobleme angewendet wurde: Zum Beispiel zeigt ein Team um Yuanlong Li, dass mit Hilfe von Reinforcement Learning Methoden der Energieverbrauch eines Datenzentrums gegenüber einer Referenzstrategie um 15% gesenkt werden kann [Li+17].

Bisher wurde der Einfluss von Verschmutzung und Putzstrategien in CSP Kraftwerken nur selten untersucht. Frühe Studien haben sich auf zeitabhängige Putzstrategien mit einer angenommenen durchschnittlichen Verschmutzungsrate beschränkt [DGS86]. In [KVH12] wird unter der Annahme einer konstanten Verschmutzung ein Zielsauberkeitswert für die Reflektoren festgelegt, welcher nicht unterschritten werden sollte. Ba et al. führen in [Ba+17] eine zustandsorientierte Putzstrategie ein. Diese werden nicht nur auf die Verschmutzungsrate, sondern auch auf die Strompreise angepasst. Die Studie zeigt, dass die Putzkosten um 5-30% verringert werden konnten, wenn ein Schwellenwert für die Sauberkeit festgelegt wurde. Diese Ergebnisse basieren auf einem hypothetischen Kraftwerk mit künstlich erstellten Wetterdaten.

Als Grundlage für diese Masterarbeit ist vor allem die Arbeit von Wolfertstetter et al. zu nennen, die den Einfluss einer Schwellenwert basierten Putzstrategie auf den Gewinn eines Parabolrinnenkraftwerks untersucht und mit einer konstanten Referenzstrategie vergleicht. Hierzu wurden Messdaten genutzt, welche über einen Zeitraum von drei Jahren in Südspanien aufgenommen wurden. Die Studie hat ergeben, dass der Gewinn eines Kraftwerks mit Hilfe einer Strategie, welche einen Schwellenwert für die aktuelle Feldsauberkeit definiert, gegenüber der Referenzstrategie um 0,8% gesteigert werden konnte [Wol+18].

Darauf aufbauend soll in dieser Arbeit Reinforcement Learning auf das Problem der Putzstrategien Optimierung angewendet werden, so dass sich die Strategie auf beliebige Zustandsparameter anpasst. Außerdem können so komplexere Strategien gelernt werden als mit einfachen Schwellenwerten für die Feldsauberkeit. Mit Hilfe dieser Methode soll außerdem die Wichtigkeit verschiedener Parameter für die Putzstrategie abgeschätzt werden.

Der Hauptteil dieser Arbeit beschäftigt sich zunächst mit den theoretischen Grundlagen von Reinforcement Learning und künstlichen neuronalen Netzen. Dabei wird der Schwerpunkt auf die im weiteren Verlauf genutzten Methoden gelegt.

Im Anschluss wird die praktische Anwendung der Reinforcement Learning Methoden auf das Problem der Putzsteuerung in CSP Kraftwerken beschrieben. Dafür wird zuerst das Problem genauer spezifiziert und beschrieben wie der Algorithmus auf das Problem angewandt wird. Daraufhin wird die Implementierung dargestellt, um schließlich den Algorithmus in zwei Schritten auszuwerten.

Im ersten Schritt wird der Algorithmus untersucht: Hier wird zum einen betrachtet wie robust der Algorithmus gegenüber der zufälligen Initialisierung ist, zum anderen wird untersucht wie gut die gefundene Lösung ist, indem die Ergebnisse in einem leichten Fall mit einer optimalen Lösung verglichen und in einem komplexeren Fall die Putzentscheidungen beobachtet und einer Plausibilitätsprüfung unterzogen werden. Im zweiten Abschnitt der Auswertung wird mit Hilfe des Algorithmus das zugrundeliegende Problem der Putzstrategien-Findung analysiert. Dafür wird zum einen untersucht, welchen Nutzen die Verfügbarkeit verschiedener Vorhersagehorizonte der Verschmutzungsraten für die Putzstrategie hat, zum anderen wird die Umwelt verändert: Es werden mehrere starke Verschmutzungsereignisse hinzugefügt oder der Wasserpreis erhöht. Hierbei soll beobachtet werden, wie die Strategie auf diese Veränderung angepasst wird und welche Schlüsse sich daraus für andere Standorte ergeben. Abschließend sollen die Schwierigkeiten der vorgestellten Methode diskutiert werden.

Grundlagen

1. Reinforcement Learning

Reinforcement Learning beschreibt sowohl eine Problemstellung, als auch eine Klasse von Verfahren zur Lösung eben dieser Problemstellung. Verkürzt behandelt Reinforcement Learning die Frage wie optimal in einer Umwelt agiert werden kann: So will man beispielsweise einem Roboter das Gehen beibringen oder mit einem Computer das Brettspiel Go und Atari Computerspiele auf übermenschlichem Niveau spielen; vergleiche [Sze10, S.3] und [SB18, S.1].
Auf die in dieser Arbeit behandelte Problematik der Steuerung der Putzaktionen wird das *Strategie Gradient Verfahren*[4] aus der Klasse der Reinforcement Learning Methoden angewendet. Um dieses Verfahren anzuwenden, werden in diesem Kapitel die Grundlagen von Reinforcement Learning behandelt. Zunächst werden die Grundidee, sowie Begriffe und Funktionsweisen eingeführt, um dieses Problem genauer zu verstehen und in eine mathematische Formulierung zu überführen. Daraufhin werden die Verfahren zur Lösung dieses Problems eingeführt, um im Anschluss genauer auf das, im Verlauf der Arbeit angewandte, Strategie Gradienten Verfahren einzugehen. Dieses Kapitel basiert hauptsächlich auf den Büchern [Sze10], [SB18] und der Vorlesung [Sil15].

1.1. Grundidee

Abbildung 1.1.1 zeigt die Grundstruktur eines Reinforcement Learning Problems. Wir nennen die Einheit, welche Entscheidungen trifft und ausführt *Agent*. Alles außerhalb dieses Agenten nennen wir *Umwelt*. Die Umwelt befindet sich in einem *Zustand* s. Dieser Zustand ist für den Agenten ganz oder in Teilen sichtbar. Daraufhin wirkt der Agent, indem er eine *Aktion* a ausführt.[5] Nach jeder Aktion a erhält der Agent von der Umwelt einen neuen Zustand $\tilde{s}$ und eine skalare *Belohnung* r. Über die diskreten Zeitpunkte $t = 0, 1, 2, \ldots$ entsteht nun die Folge

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots$$

[4] Engl: *Policy Gradient Method*.
[5] Wir verwenden hier, wie in [SB18], die im Reinforcement Learning geläufigeren Ausdrücke: *Agent, Umwelt* und *Aktion*, statt der, in der Regelungstheorie vorkommenden, vergleichbaren Ausdrücke: *Regler, Regelstrecke* und *Signal*.

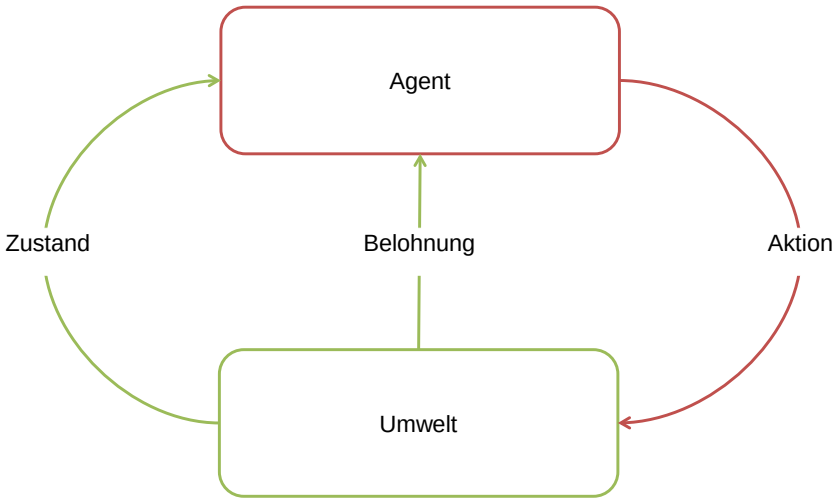


Abbildung 1.1.1: Schematische Darstellung eines Reinforcement Learning Problems. Vergleichbare Darstellungen finden sich zum Beispiel in [SB18, S. 48 Figure 3.1] oder in [Sze10, S. 3 Figure 1].

eine *Trajektorie*. Das Ziel des Agenten ist es, die kumulierte Belohnung über eine solche Trajektorie zu maximieren; vergleiche dazu [SB18, S.47]. Dies lässt sich in der *Belohnungs-Hypothese* zusammenfassen:

Definition 1.1.1. *Alle Ziele eines Reinforcement Learning Agents können beschrieben werden als die Maximierung des Erwartungswerts eines kumulierten skalaren Signals; die sogenannte Belohnung.*

Vergleiche hierzu die Definition in Lecture 1 der Vorlesung [Sil15] und [SB18, S. 53]. Die Beschränkung auf eine skalare Belohnung für die Formulierung des Ziels ist eines der Alleinstellungsmerkmale von Reinforcement Learning. Die Belohnungssignale müssen derart beschaffen sein, dass die Maximierung der Belohnung gleichbedeutend mit dem Erreichen des gewünschten Ziels ist. Allerdings ist es wichtig, dass in der Belohnung nicht vorgegeben ist, wie das Ziel erreicht wird. So sollte eine Belohnung für einen Schach lernenden Agenten nur daraus bestehen, ob das Spiel gewonnen, verloren oder in einem Remis endet. Keine Belohnung sollte das Erreichen von Zwischenzielen wie das Schlagen von Figuren oder das Besetzen der Mitte hervorrufen. Wenn solche Zwischenziele belohnt werden, kann es sein, dass der Agent einen Weg findet diese zu erreichen, ohne zum eigentlich Ziel zu gelangen; vergleiche [SB18, S. 54].
Auch wenn das Formulieren der Ziele über die skalaren Belohnungen auf den ersten Blick einschränkend wirkt, hat es sich in der Praxis als vielfältig einsetzbar bewährt. Am besten wird dies durch einige Anwendungsbeispiele deutlich, indem gezeigt wird wie dort die Belohnung ein-

gesetzt wird; vergleiche [SB18, S. 53] und [Sil15] Lecture 1 und 2. Für das Meistern von Atari Spielen nutzen Mnih et al. in [Mni+15] direkt den Punktestand in den jeweiligen Spielen als Belohnung. In [Hee+17] lernt ein simulierter humanoider Körper sich durch verschiedene Umgebungen zu bewegen. Als Belohnung wird hauptsächlich die Geschwindigkeit in die gewünschte Richtung plus einen kleinen Term, welcher zu hohen Drehmomente penaltisiert, genutzt. Außerdem erhält der Agent eine positive Belohnung für jeden Zeitpunkt, indem er nicht umgefallen ist. Das Programm, mit dem ein Team von Google Mitarbeitern um David Silver den Weltmeister des chinesische Brettspiel Go geschlagen hat, nutzt als Belohnung lediglich $+1$ für ein gewonnenes und -1 für ein verlorenes Spiel; vergleiche das Paper zu der neuesten Version des Programms [Sil+17].

Bei der Betrachtung dieser Beispiele werden die Schwierigkeiten eines Reinforcement Learning Problems offensichtlich. Zum einen wird nur ein Belohnungssignal gegeben, es wird nicht überwacht, wie das Ziel erreicht wird. Zum anderen kann das Belohnungssignal stark verzögert sein. Zum Beispiel bekommt man erst am Ende eines Go Spiels eine Belohnung. Es lässt sich aber nicht sagen, welche Züge die entscheidenden Züge waren, die zu diesem Ergebnis geführt haben. Dieses letzte Problem wird auch *Belohnungs-Zuordnungsproblem*[6] genannt; vergleiche [Sil15].

1.2. Begriffe und Funktionsweisen

Nachdem im vorherigen Kapitel die Grundidee des Reinforcement Learnings beschrieben wurde, soll nun diese Idee in mathematische Formalitäten übersetzt werden. Dazu werden die Begriffe eingeführt, welche dafür notwendig sind.

1.2.1. Markow Entscheidungsproblem

Markow Entscheidungsprobleme beschreiben formell die Umwelt in Reinforcement Learning Problemen. Beinahe alle Reinforcement Learning Probleme können als Markow Entscheidungsproblem geschrieben werden; vergleiche [Sil15] Lecture 2. Daher sollen in diesem Unterkapitel ein Markow Entscheidungsproblem formal definiert werden. Der Einfachheit halber beschränken wir uns auf diskrete Zeitpunkte wie in [Sil15], [SB18] und [Sze10]. Es lassen sich allerdings viele Ergebnisse auf stetige Zeit erweitern. Es sei dafür, wie in [SB18], exemplarisch auf [Doy96] und [BT95] verwiesen.

Wir betrachten im Folgenden einen endlichen, nicht leeren Zustandsraum $\mathcal{S} = \{s_0, s_1, s_2, \ldots, s_n\}$

[6] Engl. *credit assignment problem.*

und eine Folge von Zufallsvariablen S_t an diskreten Zeitpunkten $t \in \mathbb{N}_0$.[7] Zunächst wird die *Markow-Eigenschaft* eingeführt, da diese elementar für die Definition eines Markow Entscheidungsroblems ist. Sie beschreibt Zustände bei denen die Zukunft unabhängig von der Vergangenheit ist. Stattdessen hängt die Zukunft lediglich von der Gegenwart ab.

Definition 1.2.1. *Eine Reihe von Zufallsvariablen $S_{t=0,1,2,\dots}$ erfüllt die Markow-Eigenschaft genau dann, wenn für jede Menge an Indices $j_0, j_1, \dots, j_t$*

$$\mathbb{P}\left[S_{t+1} = s_{j_{t+1}} \mid S_t = s_{j_t}\right] = \mathbb{P}\left[S_{t+1} = s_{j_{t+1}} \mid S_1 = s_{j_1}, \dots, S_t = s_{j_t}\right]$$

gilt.

Das bedeutet, dass der aktuelle Zustand alle relevanten Informationen der Vergangenheit enthält. Die Übergangswahrscheinlichkeiten lassen sich dann in einer Übergangsmatrix $\mathcal{P}$ zusammenfassen, mit

$$\mathcal{P}_{ij} = \mathbb{P}\left[S_{t+1} = s_j \mid S_t = s_i\right].$$

Bemerkung 1.2.2. *Das Tupel $(\mathcal{S}, \mathcal{P})$ aus (endlichem) Zustandsraum $\mathcal{S}$ und zugehöriger Übergangsmatrix $\mathcal{P}$ nennt man Markow-Kette.*

Für die Beschreibung eines Reinforcement Learning Problems benötigen wir die Möglichkeit Aktionen durchzuführen. Dafür definieren wir einen endlichen, nicht leeren Aktionsraum $\mathcal{A} = \{a_0, a_1, \dots, a_m\}$. Da Aktionen auch Einfluss auf die Zustände haben, hängt der folgende Zustand nicht mehr nur von dem vorausgehenden Zustand, sondern auch von der gewählten Aktion ab. Um das zu berücksichtigen, muss die Übergangsmatrix angepasst werden. Es sei nun

$$\mathcal{P}_{ij}^{a_k} = \mathbb{P}\left[S_{t+1} = s_j \mid S_t = s_i, A_t = a_k\right] \tag{1.2.1}$$

die Wahrscheinlichkeit das zum Zeitpunkt $t+1$ der Zustand s_j angenommen wird – unter der Bedingung, dass zum Zeitpunkt t der Zustand s_i angenommen und Aktion a_k ausgeführt wurde. Für ein Reinforcement Learning Problem wird neben der Möglichkeit eine Aktion durchzuführen auch eine Belohnung benötigt. Dafür wird die Belohnungsfunktion $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ definiert. Diese ordnet dem Übergang von einem Zustand in den nächsten, unter einer bestimmten Aktion, einen skalaren Wert zu. Mit Hilfe der bis hier eingeführten Definitionen lässt sich nun ein Markow Entscheidungsproblem definieren.

[7] Notation: Sowohl bei Zuständen, als auch bei den im Folgenden eingeführten Aktionen und Belohnungen, steht die Majuskel (S_t, A_t oder R_t) für die Zufallsvariable und die Minuskel (s_t, a_t oder r_t) für das Ereignis.

Definition 1.2.3. *(Vergleiche hierzu mit anderer Notation [Hel13], Folie 6)*
Ein Markow Entscheidungsproblem ist ein 5-Tupel $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, S_0)$ *mit*

- $\mathcal{S}$ *einer endlichen Menge von Zuständen,*
- $\mathcal{A}$ *einer endlichen Menge von Aktionen,*
- $\mathcal{P}$ *einer Übergangsmatrix, wie in* (1.2.1) *definiert,*
- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ *eine Belohnungsfunktion und*
- $S_0 \in \mathcal{S}$ *einem Anfangszustand.*

1.2.2. Gewinn und Strategie

Das Ziel des Agenten ist es, zu jedem Zeitpunkt, den zukünftigen *Gewinn* zu maximieren. Der Gewinn $G_{t,\gamma}$ ist die Summe der Belohnungen ab Zeitpunkt t

$$G_{t,\gamma} = \sum_{i=0}^{H} \gamma^i \cdot R_{t+1+i}. \tag{1.2.2}$$

Hier ist R_i die Belohnung zu dem Zeitpunkt i. Der *Horizont* $H \in (0, \infty]$ beschreibt den Endzeitpunkt des Problems. Ist der *Diskontfaktor* $\gamma \in (0, 1]$ kleiner als eins, werden sofortige Belohnungen bestärkt. Je näher der Wert an null heranreicht, desto kurzsichtiger wird die Planung. Es ist oft Sinnvoll $\gamma < 1$ zu wählen, da es bei kurzfristigen Belohnungen hilft das *Belohnungs-Zuordnungsproblem* zu vereinfachen. Auch lehrt ein Blick in die Natur, dass sowohl Tiere als auch Menschen dazu tendieren kurzfristige gegenüber langfristigen Belohnungen zu präferieren; vergleiche [Ros+07]. Mathematisch verhindert ein Diskontfaktor $\gamma < 1$, dass die Reihe divergiert, wenn der Horizont $H = \infty$ ist.
Die *Strategie* π beschreibt, wie sich der Agent in bestimmten Zuständen verhält. Das bedeutet π ist eine Abbildung von Zuständen auf Aktionen. Wir werden im Folgenden nur stochastische Strategien betrachten. Das heißt in unserem Fall ist $\pi(a \mid s)$ eine Wahrscheinlichkeitsverteilung. Dass die Strategie nicht deterministisch ist hilft, wenn während des Trainings die Strategie angepasst werden soll: Wird durch eine Aktion eine positive Belohnung hervorgerufen, müsste bei einer deterministischen Strategie im Folgenden immer genau diese Aktion gewählt werden. Ist die Strategie stochastisch, so kann sie so angepasst werden, dass die Aktion im Folgenden etwas wahrscheinlicher ist. Auf diese Art wird das Optimierungsproblem geglättet.

1.3. Strategie Gradienten Verfahren

Nachdem bis hierhin das Reinforcement Learning Problem beschrieben wurde und die wichtigsten Begriffe eingeführt wurden, soll nun einem Verfahren zur Lösung solcher Probleme beschrieben werden. In dieser Arbeit soll ein *Strategie Gradienten Verfahren* genutzt werden. Der Vorteil von Strategie Gradienten Verfahren ist, dass diese kein Modell der Umwelt benötigen, um über dieses die Strategie anzupassen. Stattdessen passen diese Verfahren direkt die Strategie an die erfahrene Belohnung an, hierfür wird eine parametrisierte Strategie benötigt. Das heißt, die Strategie ist eine Wahrscheinlichkeitsverteilung, welche von Paramtern $\theta \in \mathbb{R}^d$ abhängt. So lässt sich die Beschreibung der Strategie in Kapitel 1.2.2 modifizieren zu

$$\pi\left(a \mid s, \theta\right) = \mathbb{P}\left\{A_t = a \mid S_t = s, \Theta_t = \theta\right\},$$

vergleiche [SB18]. Hierbei bezeichnet Θ_t die Folge der Zufallsvariablen der Strategieparameter zu diskreten Zeitpunkten $t \in \mathbb{N}$. Die Parameter sollen so angepasst werden, dass die bestmögliche Strategie gefunden wird. Dafür ist es notwendig die Qualität der Strategie zu bewerten. Zu diesem Zweck lassen sich verschiedene Kennwerte definieren:

Definition 1.3.1. *Sei $H \in \mathbb{N}$ ein endlicher fester Horizont. Dann sei der Startwert[8] J definiert als die erwartete Summe der Belohnungen vom Startzeitpunkt $t = 0$ bis zum Horizont H. Es sei also*

$$J\left(\theta\right) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} R_t\right].$$

Hier, wie auch im Folgenden bezeichnet $\mathbb{E}_\pi$ den Erwartungswert einer Zufallsvariable unter der Bedingung, dass der Agent der Strategie π folgt. Mit der Definition des Gewinns in (1.2.2), lässt sich der Startwert als erwarteter Gewinn zum Startzeitpunkt $\mathbb{E}_\pi\left[G_{0,1}\right]$ schreiben. Hierbei ist der Diskontfaktor $\gamma = 1$ gewählt. In anderen Fällen macht es Sinn einen diskontierten Startwert

$$J_{\text{disk}}\left(\theta\right) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right],$$

mit $\gamma < 1$ oder den durchschnittlichen Gewinn pro Zeitschritt

$$J_{\text{avg}}\left(\theta\right) = \lim_{H\to\infty} \frac{1}{H}\, \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} R_t\right]$$

[8] Die Bezeichnung *Startwert* bezieht sich darauf, dass die Belohnungen ab dem Startzeitpunkt $t = 0$ aufsummiert werden.

als Zielfunktion zu definieren. Allerdings soll im weiteren Verlauf ausschließlich auf den Startwert $J(\theta)$ als Qualitätskriterium eingegangen werden, da dieser aufgrund des festen endlichen Horizonts am besten zu dem, in dieser Arbeit behandelten, Problem passt. Sowohl der im Folgenden beschriebene Algorithmus als auch das Haupttheorem kann für die anderen Zielfunktionen analog beschrieben werden. Hierzu sei auf [Sch17], [SB18] und [Abb17] verwiesen.
Nun soll θ so gewählt werden, dass J maximiert wird. Es ergibt sich also das Problem,

$$\max_{\theta} J(\theta) \tag{1.3.1}$$

beziehungsweise

$$\max_{\theta} \left\{ \mathbb{E}_{\pi_\theta} \left[\sum_{t=0}^{H} R_t \right] \right\}.$$

Bei dieser Formulierung des Problems ist es nicht offensichtlich, wie die Parameter θ verändert werden können, dass sich die Strategie verbessert. Die Schwierigkeit liegt darin, dass die Belohnung, die ein Agent erhält sowohl von der Auswahl der Aktionen, als auch von den Zuständen, in welchen die Auswahl getroffen wird, abhängt. Beides wird durch eine Veränderung der Parameter θ beeinflusst. In den meisten Fällen ist der Einfluss der Parameter auf die Wahl der Aktionen bekannt. Allerdings ist typischerweise unbekannt, wie eine Veränderung der Parameter auf die Verteilung der Zustände wirkt, da es sich hierbei um eine Funktion der Umgebung handelt.[9]
Es soll also θ so gewählt werden, dass der Erwartungswert des Gewinns maximiert wird. Eine Möglichkeit zur Lösung dieses Problems ist zum Beispiel das *Downhill-Simplex-Verfahren* [NM65] oder andere Optimierungsmethoden, wie zum Beispiel genetische Algorithmen, die keinen Zugriff auf den Gradienten der Zielfunktion $\nabla_\theta J(\theta)$ benötigen. Auch kann der Gradient mit Hilfe von finiten Differenzen approximiert werden. Dabei wird die Strategie d-malig ausgeführt. Bei jeder Ausführung ist ein Eintrag von $\theta \in \mathbb{R}^d$ leicht verrauscht. Über die verschiedenen Werte von J kann der Gradient in einer Umgebung von θ geschätzt werden. Dass diese, relativ simplen Methoden auch in der Praxis angewandt werden können, zeigen Nate Kohl und Peter Stone in [KS04]. Hier versuchen sie eine möglichst schnelle Gangart für einen vierbeinigen Roboter zu finden. Dafür haben sie Zugriff auf 12 verschiedene Parameter. Als Belohnung wird die Ganggeschwindigkeit eingesetzt. Sie nutzen dafür drei verschiedene Algorithmen: das Downhill-Simplex-Verfahren, einen genetischen Algorithmus, Gradienten Approximation über finite Differenzen und einen einfachen Bergsteigeralgorithmus. Der Bergsteigeralgorithmus ba-

[9] Der unterschiedliche Einfluss der Strategie und der Umgebung auf $J(\theta)$ wird im Beweis zu Theorem 1.3.2 offensichtlich.

siert darauf, dass iterativ die Parameter von einer Basiskonfiguration verrauscht werden. Die jeweils beste Konfiguration wird dann als neue Basiskonfiguration gewählt. Bei ihren Tests finden Kohl und Stone mit dem Bergsteigeralgorithmus und der Gradienten Approximation die besten Ergebnisse. Mit der Gradienten Approximation finden sie eine der schnellsten, zu dieser Zeit bekannten, Gangarten. Der Vorteil dieser Verfahren ist, dass sie keinen direkten Zugriff auf die Strategie benötigen. Diese muss noch nicht mal differenzierbar sein. Allerdings sind die Verfahren impraktikabel, sobald der Parameterraum hochdimensional wird. Wird zum Beispiel die Strategie durch ein künstliches neuronales Netz definiert, existieren selbst bei kleinen Netzen leicht mehrere 10.000 Parameter. Auch sind Verfahren, welche den Gradienten der Zielfunktion ausnutzen, oftmals effizienter. Wie kann also der Gradient des Startwerts $\nabla_\theta J(\theta)$ bezüglich des Parameters θ geschätzt werden, wenn der Einfluss von θ auf die Zustandsverteilung unbekannt ist? Hier hält das *Strategie Gradienten Theorem* eine Antwort bereit:

Theorem 1.3.2 (Strategie Gradienten Theorem)**.** *Für jede differenzierbare Strategie π_θ, jeden endlichen Horizont $H \in \mathbb{N}$ und für die Zielfunktion $J(\theta)$ wie in 1.3.1 definiert gilt:*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \cdot R_t\right].$$

Beweis. An dieser Stelle soll der Beweis lediglich skizziert werden. Dabei soll nur auf die wichtigsten Schritte eingegangen werden. Der vollständige Beweis wird im Anhang B.1 geführt und ist in den meisten Teilen nach [Abb17] aufgebaut.
Durch Umformungen lässt sich $\nabla_\theta J(\theta)$ als Erwartungswert des Produkts der log-Wahrscheinlichkeit der Trajektorien und der Belohnung schreiben; vergleiche (B.1.1):

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) R_\tau\right].$$

Die Notation ist für eine bessere Übersichtlichkeit bei der Beweisführung leicht verändert: R_τ steht für die kumulierte Belohnung einer gesamten Trajektorie τ und $p_{\pi_\theta}(\tau)$ steht für die Wahrscheinlichkeit der Trajektorie τ unter der Bedingung, dass der Agent der Strategie π_θ folgt. Hier bleibt zunächst das Problem, dass die Wahrscheinlichkeit einer Trajektorie auch von der Übergangswahrscheinlichkeit abhängt. Diese ist eine Funktion der Umgebung und in der Regel nicht einsehbar. Dadurch, dass die log-Wahrscheinlichkeit betrachtet wird, lässt sich diese Wahrscheinlichkeit aufteilen in eine Summe der Übergangswahrscheinlichkeiten und eine Summe der

Entscheidungswahrscheinlichkeiten der Strategie:

$$\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) = \nabla_\theta \left[\sum_{t=0}^{H} \log p(S_{t+1} \mid S_t, A_t) + \sum_{t=0}^{H} \log \pi_\theta(A_t \mid S_t)\right];$$

vergleiche im Beweis (B.1.2). Hier bezeichnet $p(S_{t+1} \mid S_t, A_t)$ die Übergangswahrscheinlichkeit. Da diese nicht von θ abhängt, fällt die erste Summe unter der Ableitung weg. Dadurch ergibt sich direkt die Behauptung. □

Das Theorem gibt also eine Darstellung des Gradienten der Zielfunktion

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta\left(A_t \mid S_t\right) \cdot R_t\right],$$

die nur von der Strategie und der Belohnung R_t abhängt. So kann der Gradient geschätzt werden, indem eine Stichprobe von m Trajektorien gezogen wird. Es ergibt sich als *erwartungstreue*[10] Schätzfunktion der *empirische Mittelwert*:

$$\nabla_\theta J\left(\theta\right) \approx \hat{g} := \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) \cdot r_t^{(i)}.$$

Die Zugehörigkeit zur i-ten Trajektorie wird durch das hochgestellte (i) symbolisiert. Man benötigt also nur die erfahrenen Belohnungen und den Gradienten der Strategie in einer Stichprobe aus m Trajektorien, um den Gradienten der Zielfunktion J zu schätzen. Mit Hilfe dieses Schätzers $\hat{g}$ lässt sich ein einfacher Algorithmus aufstellen: Algorithmus 1. Dieser Algorithmus gehört zu

Algorithmus 1 *Einfacher Strategie Gradienten Algorithmus*

Gegeben:
Eine differenzierbare Strategie π_θ
Die Schrittweite α

1: Initialisiere θ beliebig ▷ z.B. Zufällig
2: **while** True **do** ▷ Oder bis eine Abbruchbedingung erfüllt ist
3: Erzeuge m Trajektorien mit der Strategie π_θ
4: $\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) \cdot r_t^{(i)}$ ▷ Berechne Schätzer für den Gradienten
5: $\theta \leftarrow \theta + \alpha \hat{g}$ ▷ Update der Parameter
6: **return** θ

den grundlegenden Algorithmen im Reinforcement Learning. Er wurde unter anderem in [Wil92] beschrieben. Der oben beschriebene Schätzer ist zwar erwartungstreu, aber sehr verrauscht. Um

[10] Eine Schätzfunktion heißt *erwartungstreu*, falls der Erwartungswert des Schätzers gleich dem gesuchten Wert ist; vergleiche zum Beispiel [FLP15, S. 251].

die Varianz in den Stichproben zu mindern, führen wir eine Baseline ein; vergleiche [Abb17]. Diese Baseline kann die Varianz reduzieren, wobei die erwartungstreue beibehalten wird. Es gilt der folgende Satz:

Satz 1.3.3. *Es gelten die gleichen Voraussetzungen des Strategie Gradienten Theorem 1.3.2. Sei die Baseline $b \in \mathbb{R}$. So gilt:*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \cdot (R_t - b)\right].$$

Beweis. In der verkürzten Schreibweise wie in dem Beweis zu Theorem 1.3.2 ergibt sich analog zu (B.1.1):

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right)(R_\tau - b)\right].$$

Mit Theorem 1.3.2 bleibt zu zeigen, dass

$$\mathbb{E}\left[\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) b\right] = 0$$

gilt. Durch Ausschreiben des Erwartungswertes folgt:

$$\begin{aligned}\mathbb{E}\left[\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) b\right] &= \sum_\tau p_{\pi_\theta}(\tau)\, \nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) b \\ &= \sum_\tau \nabla_\theta\ p_{\pi_\theta}(\tau)\, b.\end{aligned}$$

Bei der zweiten Umformung wird ausgenutzt, dass $\nabla_x(\log f(x)) = \nabla_x(f(x))/f(x)$ gilt. Da die Anzahl der möglichen Trajektorien endlich ist, kann die Summe und der Gradient vertauscht werden und es folgt die Gleichung:

$$\sum_\tau \nabla_\theta\ p_{\pi_\theta}(\tau)\, b = b\ \nabla_\theta \left(\sum_\tau p_{\pi_\theta}(\tau)\right). \tag{1.3.2}$$

Die Summe über alle Wahrscheinlichkeiten ist konstant eins und verschwindet daher unter der Ableitung. Daraus folgt direkt die Behauptung.

□

Es lässt sich leicht nachrechnen, dass (1.3.2) auch gilt, wenn b eine Funktion der Zustände s_t ist. In der vorliegenden Arbeit wird ausschließlich eine konstante Baseline genutzt, da in dem vorliegenden Problem nur eine Belohnung am Ende der Trajektorie generiert wird. In [GBB04] werden verschiedene Alternativen für die Wahl von b getestet. Eine nahezu optimale Möglichkeit

b zu wählen, um die Varianz zu minimieren, ist der Erwartungswert der Belohnungen. Diesen kann man mit dem empirischen Erwartungswert annähern. Es ergibt sich, als immer noch erwartungstreuer Schätzer

$$\nabla_\theta J(\theta) \approx \hat{g} := \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) \cdot \left(r_t^{(i)} - \bar{R} \right). \tag{1.3.3}$$

Hier ist $\bar{R} = \frac{1}{m} \sum_1^m R_{\tau^{(i)}}$. Mit diesem neuen Schätzer muss der Algorithmus 1 leicht angepasst werden. Man erhält:

Algorithmus 2 *Strategie Gradienten Algorithmus mit Baseline*

Gegeben:
Eine differenzierbare Strategie π_θ
Die Schrittweite α

1: Initialisiere θ beliebig ▷ z.B. Zufällig
2: **while** True **do** ▷ Oder bis eine Abbruchbedingung erfüllt ist
3: Erzeuge m Trajektorien mit der Strategie π_θ
4: $\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta \left(a_t^{(i)} \mid s_t^{(i)} \right) \cdot \left(r_t^{(i)} - \bar{R} \right)$ ▷ Berechne Schätzer
5: $\theta \leftarrow \theta + \alpha \hat{g}$ ▷ Update der Parameter
6: **return** θ

2. Künstliche Neuronale Netze

Der Psychologe Frank Rosenblatt erstellte im Jahr 1958 ein einfaches Mathematisches Modell eines Neurons: das *Perzeptron*; vergleiche [Ros58]. Dieses ist, in leicht abgewandelter Form, noch immer ein Grundbaustein moderner künstlicher neuronaler Netze. Die Entdeckung des Perzeptrons geriet eine Zeitlang in Vergessenheit, nachdem Minsky und Papert 1969 in ihrem Buch *Perceptrons* [MP69] zeigten, dass die von Rosenblatt vorgeschlagenen Perzeptrone nicht die logische *XOR-Verknüpfung*[11] lernen können. Diese Einschränkung gilt allerdings nur für Netze, die lediglich aus einer Schicht bestehen. Cybenko konnte 1989 nachweisen, dass die Funktionen, welche durch ein Netz mit nur einer weiteren, sogenannten verdeckten, Schicht dargestellt werden können, dicht in den stetigen Funktionen liegen. Das heißt, eine ausreichend große, aber endliche Anzahl an Neuronen kann jede stetige Funktion beliebig genau approximieren; vergleiche [Cyb89]. Drei Jahre zuvor fanden Wissenschaftler heraus, dass ein relativ leichter Algorithmus die verborgenen Schichten eines Netzwerkes effektiv trainiert, der *Backpropagation Algorithmus* [RHW86]. Von diesem Zeitpunkt an dauerte es noch einige Zeit bis zu

[11] Die *XOR-Verknüpfung* oder die »*Kontravalenz* ist genau dann wahr, wenn beide Teilaussagen verschiedene Wahrheitswerte haben.« Wörtliches Zitat aus [Zog16, S. 38].

dem endgültigen Durchbruch dieser Technik, vor allem da Neuronale Netze sehr viele Daten und eine große Rechenleistung benötigen. Seit Mitte der 2000er Jahre gewinnt das Forschungsgebiet stark an Bedeutung und ist heutzutage sehr aktiv; siehe Abbildung 2.0.1. Gründe dafür sind zum einen die gestiegen Rechenleistung [Int11] und der Einsatz von Grafikkarten [Sch15] oder anderer Architekturen, die eigens dafür entwickelt wurden, wie die von Google konstruierten *Tensor Processing Units* [Jou+17]. Zum anderen existieren vielfältige, große und öffentlich zugängliche Datensätze beispielsweise [KH09], [Kag] und [HU15]. Des Weiteren könnte das Wachstum dadurch bedingt sein, dass es für nahezu jede Interessierte möglich ist selber aktiv zu partizipieren. Vor allem da die Software, welche zum Erstellen von modernen künstlichen neuronalen Netzen genutzt wird, wie zum Beispiel Googles *tensorflow* [Aba+15], frei verfügbar und in die Open Source Programmiersprache *Python*[12] eingebettet ist. Auch existiert durch Cloud-Computing Anbieter wie *Amazon Web Services*[13] und *Google Cloud*[14] die Möglichkeit selbst große Projekte auf darauf spezialisierten Rechenarchitekturen laufen zu lassen. Der Hauptgrund ist allerdings, dass viele Bildungsmaterialien frei verfügbar sind. Es gibt Vorlesungen von amerikanischen Eliteuniversitäten, exemplarisch sei auf den Kurs *Introduction to Deep Learning* am Massachusetts Institute of Technology [SA17] verwiesen, die mit Video und zugehörigen Folien verfügbar sind. Außerdem sind Bücher wie *Deep Learning* von Ian Goodfellow et al. [GBC16] oder *Neural Networks and Deep Learning* von Michael Nielsen [Nie15] und die Plattform *arXiv*[15] der Cornell Universität frei zugänglich. Auf arXiv werden fast alle Paper über neuronale Netze in irgendeiner Form veröffentlicht [Bea17].

In der vorliegenden Arbeit wird ein Feedforward Netz als parametrisierte Strategie für das Strategie Gradienten Verfahren genutzt. Ein Feedforward Netz ist eine grundlegende Architektur künstlicher neuronaler Netze. Für die Optimierung solcher Netze wird das *stochastische Gradientenverfahren* als elementarer Algorithmus vorgestellt, um dann den komplexeren *Adam* Algorithmus herzuleiten, welcher in dieser Arbeit angewandt wurde. Zum Abschluss dieses Kapitels wird der Backpropagation Algorithmus vorgestellt, welcher das Trainieren von künstlichen neuronalen Netzen erst möglich macht, indem er effizient den Gradienten eines Netzes bezüglich seiner Parameter berechnet.

[12] »All Python releases are Open Source.« Siehe [Pyt].
[13] Der Cloud-Computing Service von *Amazon* `https://aws.amazon.com/de/?nc1=h_ls`. Aufgerufen am 28.11.2018.
[14] Der Cloud-Computing Service von *Google* unter anderem mit den speziell für Neuronale Netze konstruierten *Tensor Processing Units* `https://cloud.google.com/tpu/?hl=de`. Aufgerufen am 28.11.2018.
[15] `https://arxiv.org/`.
[16] Erhältlich auf dem *Elsevier Developers* Portal unter `https://dev.elsevier.com/`. Aufgerufen am 26.06.2018.

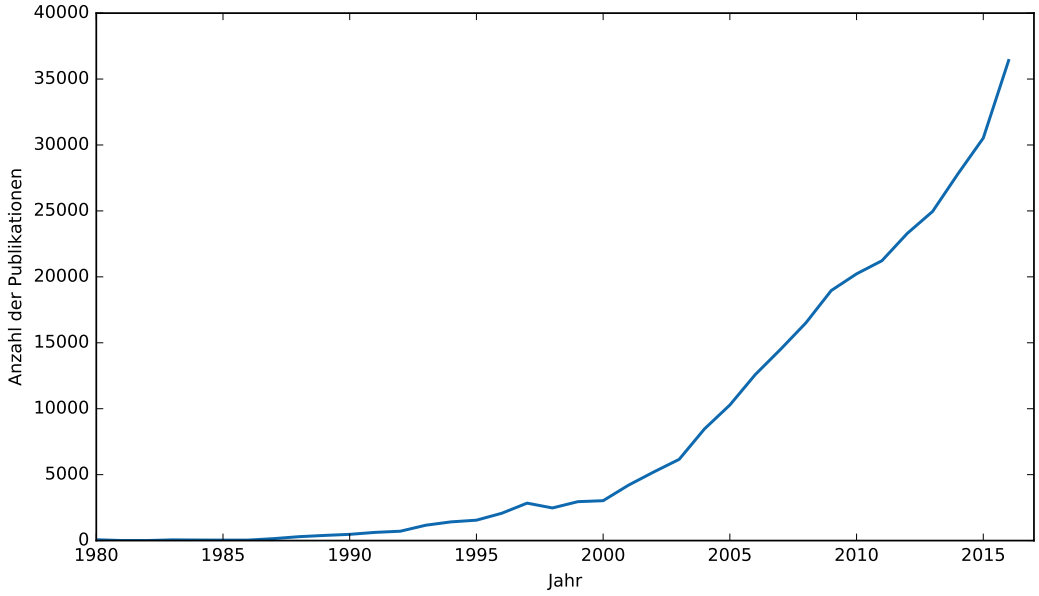


Abbildung 2.0.1: Anzahl der Einträge pro Jahr auf der Science Direct Plattform zu der Suchanfrage *Neural Networks* in Verbindung mit *Artificial Intelligence*. Daten erhoben mit Hilfe der ScienceDirect API.[16] Code für das automatisierte Herunterladen der Daten abgeändert von [Mau16].

2.1. Feedforward Netze

Künstliche Neuronale Netze sollen typischerweise einen Eingabevektor $\boldsymbol{x} \in \mathbb{R}^n$ auf einen Zielvektor $\boldsymbol{y} \in \mathbb{R}^m$ abbilden. Soll zum Beispiel unterschieden werden, ob auf einem graustufen Bild ein Hund oder eine Katze zu sehen ist, kann der Eingangsvektor die Grauwerte der Pixel des Bildes sein und der zugehörige Zielvektor ist der binäre Wert, ob auf dem Bild ein Hund ($\boldsymbol{y} = 1$) oder eine Katze ($\boldsymbol{y} = 0$) zu sehen ist [Val17]. Im folgenden Kapitel sollen Feedforward Netze beschrieben werden. Diese beschreiben eine grundlegende Struktur neuronaler Netze und werden in dieser Arbeit angewandt. Die kleinste Einheit in einem solchen Netz sind die *Sigmoid Neuronen*. Diese Neuronen bauen auf den oben genannten Perzeptronen Rosenblatts auf. Ein Perzeptron verarbeitet mehrere binäre Eingaben zu einer binären Ausgabe; siehe hierzu Abbildung 2.1.1.
Die Perzeptronen folgen einer simplen Rechenregel, um die Ausgabe y zu berechnen. Wenn die mit Gewichten w_i gewichtete Summe der Eingaben x_i einen bestimmten Schwellenwert b übersteigen, gibt das Perzeptron eine 1 und sonst eine 0 aus,

$$\begin{cases} y(\boldsymbol{x}) = 0 & \text{für } \langle \boldsymbol{w}, \boldsymbol{x} \rangle < b \\ y(\boldsymbol{x}) = 1 & \text{für } \langle \boldsymbol{w}, \boldsymbol{x} \rangle \geq b. \end{cases} \tag{2.1.2}$$

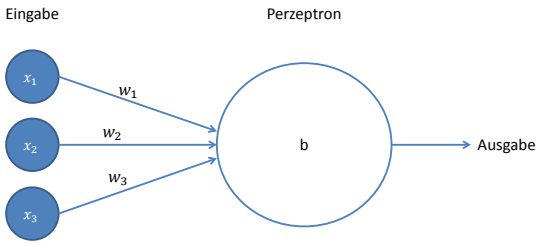


Abbildung 2.1.1: Eine Schematische Darstellung eines Perzeptrons. Der Eingabevektor $\boldsymbol{x} = (x_1, x_2, x_3)^T$ wird mit den Gewichten $\boldsymbol{w} = (w_1, w_2, w_3)^T$ und dem Schwellenwert b durch das Perzeptron zur Ausgabe verrechnet. Ähnlich zu der 3. Abbildung in [Nie15].

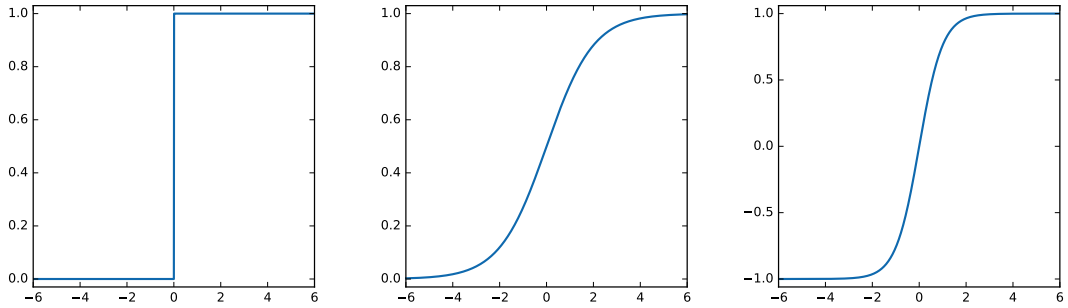
Die fett gedruckten Symbole stehen für den Gewichts- beziehungsweise Eingabevektor, $\langle \cdot, \cdot \rangle$ für das Skalarprodukt. Der Unterschied von Neuronen moderner neuronaler Netze zu dem ursprünglichen Perzeptron ist, dass diese anstelle der Stufenfunktion eine andere *Aktivierungsfunktion* besitzen. Häufige werden zum Beispiel die Logistische Funktion $\ell(x) = \frac{1}{1+e^{-x}}$ und der Tangens hyperbolicus tanh gewählt. Man kann diese Funktionen als geglättete Version der Stufenfunktion betrachten; siehe Abbildung 2.1.4. So verändert sich die Formel, (2.1.2) nach der die Aktivierung eines Perzeptrons berechnet wird, in unserem Fall zu

$$y(\boldsymbol{x}) = \sigma\left(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b\right), \tag{2.1.3}$$

wobei σ eine beliebige Aktivierungsfunktion ist.

Mit Hilfe eines einzelnen Neurons lassen sich keine komplexen Funktionen darstellen. Daher werden mehrere Neuronen in einem Netz miteinander verbunden. Die Ausgabe eines oder mehrerer Neuronen wird zu der Eingabe eines anderen Neurons. In einem Feedforward Netz sind die Neuronen in Schichten organisiert. Diese sind hintereinander gereiht, sodass die Neuronen der Schicht ℓ ausschließlich gerichtete Verbindungen zu allen Neuronen der darauffolgenden Schicht $\ell + 1$ haben. Das heißt, die Ausgaben der Schicht ℓ sind die Eingaben der Schicht $\ell + 1$; siehe Abbildung 2.1.5. Die Ausgabe eines Neurons nennt man auch *Aktivierung*. Dabei nennt man die letzte Schicht *Ausgabeschicht* und die vorhergehenden *verborgene Schichten*. Die Idee hinter dieser Verschachtelung ist, dass mit jeder Schicht eine komplexere Repräsentation des Eingabevektors gefunden wird.

Es soll an dieser Stelle beschrieben werden wie die Ausgabe eines Netzes berechnet wird. Diese



(a) Stufenfunktion (b) Logistische Funktion (c) Tangens Hyperbolicus

Abbildung 2.1.4: Verschiedene Aktivierungsfunktionen. Man beachte die abweichende y-Skala in Graph (c).

Berechnung wird in Kapitel 2.3 nochmal benötigt. Außerdem ist es für das Verständnis hilfreich, wenn man betrachtet mit welchen Operationen die Berechnung ausgeführt wird. Dafür ist zunächst einmal eine Einführung in die Notation notwendig. Diese ist an [Nie15] angelehnt. So ist

- w^ℓ_{jk} das *Gewicht*, welches das k-te Neuron der $(\ell-1)$-Schicht mit dem j-ten Neuron der ℓ-ten Schicht verbindet.
- b^ℓ_j steht für den *Schwellenwert* des j-ten Neuron in der ℓ-ten Schicht.
- a^ℓ_j steht für die *Aktivierung* des j-ten Neuron in der ℓ-ten Schicht.

Siehe hierfür auch Abbildung 2.1.5.
Da jedes Neuron der ℓ-ten Schicht mit allen Neuronen der vorhergehenden $(\ell-1)$-ten Schicht verbunden ist, setzt sich die Aktivierung des j-ten Neurons der ℓ-ten Schicht wie folgt zusammen,

$$a^\ell_j = \sigma\left(\sum_k w^\ell_{jk} a^{\ell-1}_k + b^\ell_j\right), \tag{2.1.6}$$

wobei die Summe über alle Neuronen der $(\ell-1)$-ten Schicht gebildet wird. Sowohl die Formel als auch die Indizierung lassen erkennen, dass sich dieser Term auch als eine Matrix-Vektor-Multiplikation schreiben lässt. So definieren wir eine *Gewichtsmatrix* W^ℓ mit den Gewichten, welche Schicht $(\ell-1)$ mit Schicht ℓ verbinden $\left(W^\ell\right)_{jk} = w^\ell_{jk}$. Analog erstellen wir einen Schwellenwert Vektor $\left(\boldsymbol{b}^\ell\right)_j = b^\ell_j$ und einen Aktivierungsvektor $\left(\boldsymbol{a}^\ell\right)_j = a^\ell_j$. Für die Aktivierungsfunktion σ nutzen wir die verkürzte Schreibweise $\sigma(\boldsymbol{v})$ für die elementweise Anwendung der Aktivierungsfunktion auf die Einträge des Vektors $\boldsymbol{v}$. Mit dieser Notation lässt sich die Gleichung

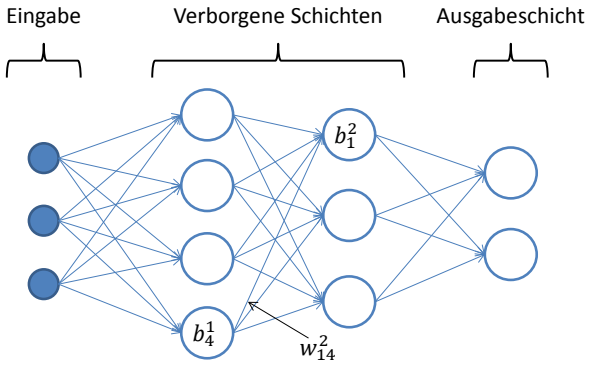


Abbildung 2.1.5: Eine schematische Darstellung eines Feedforward Netzes mit zwei verborgenen Schichten; vergleiche [Nie15].

(2.1.6) kompakt schreiben als

$$\boldsymbol{a}^\ell = \sigma\left(W^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell\right). \tag{2.1.7}$$

An dieser Schreibweise wird deutlich, wie die Aktivierung von einer Schicht zur nächsten weitergegeben wird. Die Hauptoperationen zur Berechnung der Aktivierungen sind Matrix-Vektor-Multiplikationen und Vektor Additionen. Diese Operationen lassen sich gut parallelisieren und mit Grafikkarten (GPUs) berechnen, da diese eine massiv parallele Struktur besitzen. Daher ist das Training auf GPUs auch bis zu 50-mal schneller ist als auf CPUs [Sch15]. Betrachtet man die einzelnen Rechenoperationen zeigt sich auch wie viele Parameter ein Neuronales Netz definieren. So hat das, in Abbildung 2.1.5 dargestellte Netz 30 Gewichte w und weitere neun Schwellenwerte b. Bei realen Netzen geht die Anzahl der Parameter schnell in den Bereich von mehreren Hunderttausend.

Neben den hier beschriebenen Feedforward Netzen, gibt es eine Vielzahl anderer Netzarchitekturen. Besonders hervorzuheben sind die rekurrente Netze und die Konvolutionsnetze. Bei rekurrenten Netzen besitzen Neuronen einer Schicht Verbindungen zu Neuronen derselben oder einer vorangegangenen Schicht. Das heißt, wenn man die Netzwerkarchitektur als gerichteten Graph auffasst, sind diese Architekturen nicht mehr Kreisfrei. Die populärste Version dieser Netze sind LSTM-Netze[17] [HS97]. Diese werden zum Beispiel in der Erkennung und Verarbeitung von (natürlicher) Sprache verwendet [GMH13]. Konvolutionsnetze sind hingegen spezialisiert für die Bildverarbeitung. Die Neuronen der Konvolutionsnetze operieren mit

[17] Long short-term memory.

Faltungen auf den Pixeln von Bilder, ähnlich der Faltungsfilter, welche aus der Bildverarbeitung bekannt sind. Ein weiterer Unterschied der Konvolutionsnetze zu Feedforward Netzen ist, dass die Neuronen einer Schicht geteilte Gewichte haben. Dies hilft die Anzahl der Parameter zu verringern, da für Bilder sonst eine extrem hohe Anzahl an Parametern nötig ist.[18] Für genauere Erklärungen zu dieser Art von Netzen sei auf [GBC16] Kapitel 9 und 10 verwiesen.

2.2. Training

Nachdem das vorherige Kapitel dem Aufbau eines Feedforward Netzes gewidmet war, soll nun beschrieben werden, wie das *Training* der Netze funktioniert: Wie können die Gewichte und Schwellenwerte angepasst werden, sodass die Ausgabe eines Netzes sich der gewünschten Ausgabe annähert? Zunächst soll für das Verständnis des Trainings ein grundlegender Algorithmus eingeführt werden. Dabei halten wir uns vor allem an [Nie15] und [Roj96]. In dem darauffolgenden Abschnitt über den *Adam* Algorithmus wird hauptsächlich das Paper [KB14] genutzt.
Für das Training sind Daten nötig, die zum Lernen genutzt werden: das *Trainingsset* $\mathcal{T}$. Diese besteht aus mehreren Tupel $(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}))$, wobei $\boldsymbol{y}(\boldsymbol{x})$ die gewünschte Ausgabe zu dem Eingabevektor $\boldsymbol{x}$ ist. Als Gütekriterium führen wir eine *Verlustfunktion*[19] L ein, die minimiert werden soll. Wir nutzen beispielhaft den mittleren quadratischen Fehler. In diesem Fall ergibt sich:

$$L(\theta) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \|\boldsymbol{y}(\boldsymbol{x}) - f(\boldsymbol{x}, \theta)\|^2. \tag{2.2.1}$$

Hier ist θ die Gesamtheit der Parameter, also die Gewichte $\boldsymbol{w}$ und die Schwellenwerte $\boldsymbol{b}$. Die Ausgabe des Netzes zu dem Eingabevektor $\boldsymbol{x}$ und Parameter θ wird mit $f(\boldsymbol{x}, \theta)$ bezeichnet. Diese Wahl der Verlustfunktion bietet sich an, da offensichtlich $L(\theta) \geq 0$ gilt. Des Weiteren gilt $L(\theta) = 0$ genau dann, wenn das Netz für alle Eingaben $\boldsymbol{x}$ im Trainingsset $\mathcal{T}$ die gewünschte Ausgabe $\boldsymbol{y}(\boldsymbol{x})$ liefert. Wir suchen also das globale Minimum von L über den Parametern θ. Dafür kann das *Gradientenverfahren* genutzt werden, da die Ausgabe des Netzes als Verkettung differenzierbarer Aktivierungsfunktionen differenzierbar ist. Damit ist auch die Funktion des gesamten Netzes als Verkettung der Aktivierungsfunktionen differenzierbar. Somit kann der

[18] Angenommen der Eingangsvektor ist ein Farbbild mit 224×224 Pixeln und 3 Farbkanälen. Dann ergeben sich in einem Feedforward Netz mit nur einer Ausgabeschicht von m Neuronen $224 \cdot 224 \cdot 3 \cdot m$ Gewichte und m Schwellenwerte. Das heißt, dieses Netz besitzt mehr als $150.000m + m$ Parameter.
[19] engl. *Loss*.

Gradient der Verlustfunktion

$$\nabla L(\theta) = \left(\frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, \dots \frac{\partial L(\theta)}{\partial \theta_k} \right)$$

bestimmt werden. Folgt man dem negativen Gradienten iterativ mit einer Schrittweite η so erhält man folgendes Update der Parameter:

$$\theta \longleftarrow \theta - \eta \nabla L.$$

Wenn die Schrittweite in jedem Schritt intelligent angepasst wird, kann garantiert werden, dass man auf diese Art zu einem lokalen Minimum gelangt [BB88]. Diese Anpassung der Schrittweite ist allerdings nicht trivial. Wir werden im weiteren Verlauf darauf eingehen, wie die Schrittweite in dem hier genutzten Algorithmus angepasst wird.
Die Verlustfunktion L lässt sich auch als Summe von Verlusten jedes einzelnen Trainingselements $\boldsymbol{x} \in \mathcal{T}$ schreiben:

$$L(\theta) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} L_{\boldsymbol{x}}(\theta),$$

mit $L_{\boldsymbol{x}}(\theta) = \|\boldsymbol{y}(\boldsymbol{x}) - f(\boldsymbol{x}, \theta)\|^2$. So kann auch der Gradient über die gemittelten Gradienten der verschiedenen Eingabevektoren berechnet werden:

$$\nabla L(\theta) = \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{x} \in \mathcal{T}} \nabla L_{\boldsymbol{x}}(\theta).$$

Es ist offensichtlich, dass die Berechnung des Gradienten bei einer großen Anzahl von Trainingsdaten sehr aufwändig ist und das Lernen nur langsam erfolgt. Das *stochastische Gradientenverfahren* kann genutzt werden, um das Training zu beschleunigen. Dafür wählt man eine zufällige Teilmenge $\mathcal{B} \subset \mathcal{T}$ der Trainingsdaten. Diese Teilmenge $\mathcal{B}$ wird *Batch* genannt. Nun approximiert man den Gradienten mit

$$\nabla L(\theta) \approx \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} \nabla L_{\boldsymbol{x}}(\theta). \tag{2.2.2}$$

Es ergibt sich das einfache Stochastische Gradientenverfahren mit fester Schrittweite; siehe Algorithmus 3.
Dieses Verfahren ist die Grundlage der meisten Optimierungsverfahren für neuronale Netze. Es hat allerdings noch einige Schwächen; vergleiche [Rud16] Abschnitt 3:

- Die Wahl der Lernrate η ist schwierig. Während eine zu kleine Lernrate zu einer langsamen Konvergenz führt, kann eine zu große Lernrate dazu führen, dass die Verlustfunktion um

Algorithmus 3 *Einfaches Stochastisches Gradientenverfahren*

Gegeben:
Eine differenzierbare Verlustfunktion L
Ein Trainingsdatensatz $\mathcal{T}$
Die Schrittweite η
Eine Batchgröße $m \in \mathbb{N}$

1: Initialisiere θ beliebig ▷ z.B. Zufällig
2: **while** True **do** ▷ Oder bis eine Abbruchbedingung erfüllt ist
3: Unterteile den $\mathcal{T}$ in Batches $\mathcal{B}_i$ der Größe m
4: **for** jeden Batch $\mathcal{B}_i$ **do**
5: $\boldsymbol{g} \leftarrow \frac{1}{|\mathcal{B}_i|} \sum_{\boldsymbol{x} \in \mathcal{B}_i} \nabla L_{\boldsymbol{x}}(\theta)$
6: $\theta \leftarrow \theta - \eta \boldsymbol{g}$ ▷ Aktualisiere θ

ein Minimum fluktuiert oder sogar divergiert.

- Wird die Lernrate nach einem Zeitplan reduziert, muss dieser Zeitplan vor dem Starten des Algorithmus aufgestellt werden und kann sich so nicht an die spezifischen Charakteristiken eines Datensatzes anpassen.

- Die gleiche Lernrate gilt für alle Parameter θ_i. Wenn die Eingabevektoren $\boldsymbol{x}$ dünn besetzt sind und die Einträge unterschiedlich oft vorkommen, ist es wünschenswert, dass für Einträge, die selten vorkommen, größere Updates ausgeführt werden.

- Ein weiteres Problem ist das Minimieren der nicht-konvexen Verlustfunktion. Das stochastische Gradientenverfahren verfängt sich häufig in lokalen Minima. In [Dau+14] wird argumentiert, dass auch Sattelpunkte ein großes Problem für die Optimierung sind, da sich um diese herum typischerweise ein Plateau des gleichen oder ähnlichen Fehlers befindet. Das führt dazu, dass der Gradient an dieser Stelle im einfachen stochastischen Gradientenverfahren verschwindet.

Es gibt mehrere Verfahren, welche diese Probleme auf unterschiedliche Weise angehen. In der vorliegenden Arbeit behandeln wir den *Adam*-Algorithmus [KB14]. Da es für Optimierungsprobleme einer Verlustfunktion keinen optimalen Algorithmus gibt, vergleiche das *No free lunch theorem in optimization* [WM97],[20] wählt man am besten einen Algorithmus, welcher sich in der Praxis bewährt hat. Der Adam Algorithmus wird von unterschiedlicher Seite als bevorzugte Wahl eines Optimierungsalgorithmus genannt. So beendet Sebastian Ruder sein *Overview of gradient descent optimization algorithms* mit der Empfehlung »Adam might be the best over-

[20] Das *No free lunch theorem in optimization* besagt, dass über die Menge aller Optimierungsprobleme der durchschnittliche Aufwand für alle Lösungsmethoden gleich ist.

all choice« siehe [Rud16, S.10]. Auch Andrej Karpathy schreibt in der Zusammenfassung des Stanford Kurses *CS231n: Convolutional Neural Networks for Visual Recognition*: »The two recommended updates to use are either SGD+Nesterov Momentum or Adam.« Siehe [Kar18], Abschnitt »Summary.«

Der Name *Adam* ist abgeleitet von *adaptive moment estimation*. Der Algorithmus wird eingeführt als ein Minimierungsalgorithmus für stochastische Funktionen. In unserem Fall ist die zu optimierende Funktion stochastisch, da, wie im *einfachen stochastisches Gradientenverfahren*, der Gradient durch den Gradienten auf einer Teilmenge der Trainingsdaten angenähert wird. Der Algorithmus berechnet adaptive Schätzungen des ersten Momentes – der Mittelwert – und des zweiten Momentes des Gradienten ∇L.[21] Das Vorgehen ist in Algorithmus 4 als

Algorithmus 4 *Adam* Optimierungsalgorithmus nach [KB14]. Es wird auf die fett gedruckten Buchstaben um Vektoren zu kennzeichnen verzichtet. Bei $\theta_i, m_i, v_i, \hat{m}_i, \hat{v}_i, g_i$ handelt es sich um Vektoren. g^2 beschreibt die elementweise Quadratur $g \odot g$. Alle anderen Operationen mit Vektoren sind auch elementweise durchzuführen. In [KB14] vorgeschlagene Standardwerte: $\eta = 0{,}001$; $\beta_1 = 0{,}9$; $\beta_2 = 0{,}999$ und $\epsilon = 10^{-8}$.

Gegeben:
Eine differenzierbare Verlustfunktion L
$\beta_1, \beta_2 \in [0, 1)$ ▷ Exponentielle Abklingrate der Momenten Schätzern
Ein Trainingsdatensatz $\mathcal{T}$
Die Schrittweite η
Eine Batchgröße $m \in \mathbb{N}$

Initialisiere θ beliebig ▷ z.B. Zufällig
$m_0 \leftarrow 0$ ▷ Initialisiere 1. Momenten Vektor
$v_0 \leftarrow 0$ ▷ Initialisiere 2. Momenten Vektor
$t \leftarrow 0$ ▷ Initialisiere Zeitschritt
while True **do** ▷ Oder bis eine Abbruchbedingung erfüllt ist
    Unterteile den $\mathcal{T}$ in Batches $\mathcal{B}_i$ der Größe m
    for jeden Batch $\mathcal{B}_i$ **do**
        $t \leftarrow t + 1$
        $g_t \leftarrow \frac{1}{|\mathcal{B}_i|} \sum_{\boldsymbol{x} \in \mathcal{B}_i} \nabla L_{\boldsymbol{x}}(\theta_{t-1})$ ▷ Berechne Gradienten
        $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Aktualisiere verzerrten Schätzer des 1. Moments
        $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ Aktualisiere verzerrten Schätzer des 2. Moments
        $\hat{m}_t \leftarrow m_t / (1 - {\beta_1}^t)$ ▷ Berechne korrigierten Schätzer des 1. Moments
        $\hat{v}_t \leftarrow v_t / (1 - {\beta_2}^t)$ ▷ Berechne korrigierten Schätzer des 2. Moments
        $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Aktualisiere Parameter θ

Pseudocode beschrieben. Der angenäherte Gradient zum Zeitpunkt t wird g_t genannt und wie im einfachen stochastischen Gradientenverfahren berechnet. Daraufhin werden die exponentiell gleitenden Durchschnitte des Gradienten (m_t) und des quadrierten Gradienten v_t aktualisiert;

[21] Als *k-tes Moment* einer Zufallsvariable X wird der Erwartungswert $\mathbb{E}(X^k)$ der k-ten Potenz von X bezeichnet [OLD98].

(a) Gradientenverfahren ohne Momentum (b) Gradientenverfahren mit Momentum

Abbildung 2.2.3: Schaubild zu Gradientenverfahren mit und ohne Momentum in steilen Einkerbungen; siehe [Rud16] Figure 2.

vergleiche Alorithmus 4, Z.10f. Diese gleitenden Durchschnitte sind Schätzer für das erste Moment und das zweite Moment des Gradienten. Da diese Momente mit 0 initialisiert wurden, sind die Schätzer in Richtung Null verzerrt. Diese Schätzer werden im Folgenden – Algorithmus 4, Z.12f – so korrigiert, dass $\hat{m}_t$ beziehungsweise $\hat{v}_t$ die erwartungstreuen Schätzer der Momente sind. Die Erwartungstreue von $\hat{m}_t$ und $\hat{v}_t$ lässt sich nachrechnen; siehe [KB14] Abschnitt 3. Der Parametervektor θ_t wird daraufhin aktualisiert mit $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$. Hierbei ist ϵ ein typischerweise kleiner Wert, der das Dividieren durch Null verhindern soll. Der in [KB14] vorgeschlagener Standardwert ist $\epsilon = 10^{-8}$.

Es gibt mehrere Vorteile das erste Moment zu berechnen anstatt nur den Gradienten. Es führt eine gewisse Trägheit in die Minimierung ein. So kann ein solches Verfahren Plateaus mit verschwindendem Gradienten überwinden, da auch vergangene Gradienten noch mitbetrachtet werden. Die Berechnung des ersten Momentes hilft auch, wenn die Fehlerfunktion eine steile Einkerbung besitzt; so heben sich die fluktuierenden Anteile des Gradienten gegenseitig auf und verschwinden im Mittelwert; siehe Abbildung 2.2.3. Eine weitere Eigenschaft des Adam Algorithmus ist die vorsichtige Aktualisierung der Parameter. Unter der Annahme das $\epsilon = 0$ ist, ergibt sich als effektive Schrittweite im Parameterraum

$$\Delta_t = \eta \, \frac{\hat{m}_t}{\sqrt{\hat{v}_t}}.$$

Für diese gilt die folgenden oberen Schranken:

$$\|\Delta_t\|_\infty \leq \begin{cases} \eta \cdot (1-\beta_1)/\sqrt{1-\beta_2} & \text{falls } (1-\beta_1) > \sqrt{1-\beta_2} \\ \eta & \text{sonst.} \end{cases}$$

Die obere Schranke kann man als *Trust Region* um den aktuellen Parameter verstehen, außerhalb welcher der aktuelle Schätzer des Gradienten keine ausreichende Information liefert. Die obere

Schranke wird nur erreicht, wenn der Gradient an der Position zum ersten Mal einen Wert ungleich Null besitzt. Das heißt bei sehr seltenen Einträgen ist der Gradient größer als bei solchen die häufiger vorkommen. Kingma und Ba nennen in [KB14] den Quotient $\hat{m}_t/\sqrt{\hat{v}_t}$ *Signal-zu-Rauschen Verhältnis* (SRV). Wenn eine größere Unsicherheit über die Richtung des Gradienten herrscht, ist das SRV klein und somit wird auch die Schrittweite Δ_t kleiner. Da das SRV in der Nähe eines Optimums typischerweise kleiner wird, kann man diese Eigenschaft auch als eine Art automatisches Reduzieren der Schrittweite beschreiben; vergleiche [KB14]. Eine weitere Eigenschaft des Algorithmus ist, dass dieser invariant gegenüber Skalierung des Gradienten und somit auch der Verlustfunktion ist. Skaliert man den Gradienten ∇L mit λ, wird $\hat{m}_t$ mit λ und $\hat{v}_t$ mit λ^2 skaliert. In der effektiven Schrittweite heben sich diese Faktoren auf.
Abschließend lassen sich die Vorteile des Adam Algorithmus wie folgt zusammenfassen:

- Die Nutzung des Moments hilft Plateaus zu überwinden und verhindert die Fluktuation in steilen Einkerbungen.
- Es existiert eine obere Schranke für die Schrittweite.
- Bei größerer Unsicherheit, typischer Weise auch in der Nähe von Minima, wird die Schrittweite verringert.
- Anteile des Gradienten, die seltener vorkommen, besitzen eine größere Schrittweite.
- Der Algorithmus ist invariant gegenüber Skalierungen der Verlustfunktion.
- Funktioniert gut auf einer Vielzahl von Optimierungsproblemen künstlicher Neuronaler Netze [KB14].

2.3. Backpropagation Algorithmus

Im vorherigen Kapitel wurde gezeigt, wie künstliche neuronale Netze trainiert werden können. Das Training wurde im wesentlichen darauf reduziert, den Gradienten der Verlustfunktion ∇L zu bestimmen. Allerdings haben wir in Kapitel 2.1 gesehen, dass Feedforward Netze sehr viele Parameter besitzen für die jeweils eine partielle Ableitung bestimmt werden muss. Dies geschieht effizient mit dem *Backpropagation Algorithmus* [RHW86], welcher in diesem Kapitel beschrieben wird. Dafür halten wir uns hauptsächlich an [Nie15, Kap. 2].
Der Backpropagation Algorithmus bestimmt effizient partielle Ableitungen. Er wurde zuerst in den späten 1960er Jahren erfunden. Allerdings wurde er in verschiedenen Forschungsgebieten

unter unterschiedlichen Namen mehrmals wieder erfunden. So wird er sowohl für Wettervorhersage Simulationen als auch für die Analyse von numerischer Stabilität eingesetzt. Der generelle anwendungsunabhängige Name ist *Rückwärtsmodus Differenzieren*;[22] vergleiche [Gri10]. Im Jahr 1986 zeigte RUMELHART et al. in [RHW86], dass dieser Algorithmus auf neuronale Netze angewandt werden kann und dass diese so effektiv trainiert werden können.
Um den Backpropagation Algorithmus zu beschreiben, erinnern wir uns an die Notationen aus Kapitel 2.1. Die Berechnung der Aktivierung einer Schicht ℓ lässt sich in Vektorschreibweise schreiben als

$$\boldsymbol{a}^\ell = \sigma\left(W^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell\right);$$

siehe Gleichung (2.1.7). Wir definieren die *gewichtete Eingabe* $\boldsymbol{z}^\ell$ mit $\boldsymbol{z}^\ell := W^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell$. Mit dieser Definition vereinfacht sich die Gleichung (2.1.7) für die Berechnung der Aktivierung einer Schicht zu

$$\boldsymbol{a}^\ell = \sigma\left(\boldsymbol{z}^\ell\right).$$

Es soll der Gradient der Verlustfunktion bezüglich aller Parameter, also aller Gewichte w und aller Schwellenwerte b berechnet werden. Für bessere Übersichtlichkeit, wird der zu berechnende Gradient aus (2.2.2) wiederholt:

$$\nabla L(\theta) \approx \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x}\in\mathcal{B}} \nabla L_{\boldsymbol{x}}(\theta), \tag{2.3.1}$$

wobei $L_{\boldsymbol{x}}$ sich mit der Definition der Aktivierungen als

$$L_{\boldsymbol{x}}(\theta) = \|\boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{a}^N(\boldsymbol{x},\theta)\|^2 \tag{2.3.2}$$

schreiben lässt. Hier ist N die Anzahl an Schichten und folglich $\boldsymbol{a}^N$ die Aktivierung der letzten Schicht für die Eingabe $\boldsymbol{x}$ und die Parameter θ. Wir bleiben im Folgenden in den Beispielen bei der Verlustfunktion (2.3.2). Der Backpropagation Algorithmus gilt aber auch für weitere differenzierbare Verlustfunktionen. Diese müssen nur zwei Annahmen genügen: Erstens muss die Verlustfunktion als eine Mittelwert $L(\theta) = \frac{1}{|\mathcal{B}|}\sum_{\boldsymbol{x}\in\mathcal{B}} L_{\boldsymbol{x}}(\theta)$ der Verlustfunktion einzelner Trainingselemente $\boldsymbol{x}$ geschrieben werden können. Zweitens muss die Verlustfunktion als Funktion $L_{\boldsymbol{x}} = L(\boldsymbol{a}^N)$ der Ausgabe des Netzwerks geschrieben werden können wie in (2.3.2) geschehen.
Um nun die partiellen Ableitungen $\frac{\partial L}{\partial w^\ell_{jk}}$ beziehungsweise $\frac{\partial L}{\partial b^\ell_j}$ zu berechnen, führen wir den Fehler

[22] Engl. *reverse-mode differentiation.*

δ_j^ℓ des j-ten Neuron in Schicht ℓ ein, mit

$$\delta_j^\ell := \frac{\partial L}{\partial z_j^\ell}. \tag{2.3.3}$$

Wie bei den anderen Größen schreiben wir $\boldsymbol{\delta}^\ell$ für den Fehlervektor der Schicht ℓ. Mit Hilfe des Backpropagation Algorithmus werden die Fehler der Neuronen $\boldsymbol{\delta}^\ell$ Schichtweise berechnet, um dann mit den gesuchten Größen $\frac{\partial L}{\partial w_{jk}^\ell}$ beziehungsweise $\frac{\partial L}{\partial b_j^\ell}$ in Relation gebracht zu werden. Dazu sind vier Gleichungen notwendig, die im Folgenden beschrieben werden sollen. Für die Beweise sei auf [Nie15, Kap. 2] verwiesen.
Die erste der vier Gleichungen gibt uns einen Ausdruck für den Fehler $\boldsymbol{\delta}^N$ der Ausgabeschicht. Es gilt:

$$\delta_j^N = \frac{\partial L}{\partial a_j^N} \sigma'\left(z_j^N\right). \tag{2.3.4}$$

Das lässt sich mit der Kettenregel nachrechnen. Es sei angemerkt, dass die einzelnen Bestandteile von (2.3.4) einfach zu berechnen sind: Die gewichtete Eingabe z_j^N wird während der normalen Berechnung der Ausgabe des Netzwerkes berechnet. Die Berechnung von $\sigma'(z_j^N)$ hängt natürlich von der Wahl der Aktivierungsfunktion ab, stellt allerdings, bei den klassischer Weise genutzten Aktivierungsfunktionen, keinen großen rechnerischen Aufwand dar. Die Berechnung von $\frac{\partial L}{\partial a_j^N}$ hängt von der Wahl der Verlustfunktion L ab. Wird die quadratische Verlustfunktion wie in (2.3.3) gewählt, lässt sich leicht nachrechnen, dass in dem Fall $\frac{\partial L}{\partial a_j^N} = 2(a_j^N - y_j)$ gilt. Nun lässt sich die Formel (2.3.4) auch in Vektorschreibweise formulieren:

$$\boldsymbol{\delta}^N = \nabla_{\boldsymbol{a}} L \odot \sigma'\left(\boldsymbol{z}^N\right). \tag{2.3.5}$$

Hier, wie auch im Folgenden, bezeichnet $\odot$ das *Hadamard-Produkt*, welches Vektoren elementweise multipliziert. Mit der quadratischen Verlustfunktion ergibt sich der folgende Ausdruck

$$\boldsymbol{\delta}^N = 2\left(\boldsymbol{a}^N - \boldsymbol{y}\right) \odot \sigma'\left(\boldsymbol{z}^N\right).$$

Die nächste Gleichung beschreibt den Fehler $\boldsymbol{\delta}^\ell$ einer Schicht ℓ in Abhängigkeit des Fehlers in der folgenden Schicht $\boldsymbol{\delta}^{\ell+1}$. Es gilt

$$\boldsymbol{\delta}^\ell = \left(\left(W^{\ell+1}\right)^T \boldsymbol{\delta}^{\ell+1}\right) \odot \sigma'\left(\boldsymbol{z}^\ell\right), \tag{2.3.6}$$

hier ist $\left(W^{\ell+1}\right)^T$ die transponierte Gewichtsmatrix. Wie schon bei der vorherigen Gleichung (2.3.4) wird die gewichtete Eingabe $\boldsymbol{z}^\ell$ während der Berechnung der Netzwerkausgabe berechnet. Das heißt, der Fehler $\boldsymbol{\delta}^\ell$ einer Schicht kann mit einer Matrixmultiplikation und einer elementweisen Vektormultiplikation berechnen werden, wenn der Fehler der folgenden Schicht $\boldsymbol{\delta}^{\ell+1}$ bekannt ist.

In Gleichung (2.3.5) ist ein Ausdruck für die Berechnung des Fehlers der letzten Schicht $\boldsymbol{\delta}^N$ beschrieben. Mit Formel (2.3.6) kann sukzessive der Fehler der vorhergehenden Schichten berechnet werden. Die Nächsten zwei Formeln bringen diese Fehler $\boldsymbol{\delta}$ nun mit den eigentlich gesuchten Größen $\frac{\partial L}{\partial w^\ell_{jk}}$ und $\frac{\partial L}{\partial b^\ell_j}$ in Verbindung. So gilt für die partielle Ableitung bezüglich der Schwellenwerte b mit

$$\frac{\partial L}{\partial b^\ell_j} = \delta^\ell_j, \tag{2.3.7}$$

dass diese schon mit den Fehlern der Schichten berechnet wurden. Für die partielle Ableitung bezüglich der Gewichte w gilt:

$$\frac{\partial L}{\partial w^\ell_{jk}} = a^{\ell-1}_k \delta^\ell_j. \tag{2.3.8}$$

Auch hier wurden die Aktivierungen $\boldsymbol{a}^\ell$ schon, bei der Berechnung der Ausgabe des Netzes, berechnet. Mit den Gleichungen (2.3.4) – (2.3.8) wird deutlich, wie der Algorithmus die Ableitungen bestimmt. Zunächst wird die Ausgabe zu einem Trainingselement $\boldsymbol{x}$ wie in Kapitel 2.1 berechnet. Dabei werden die Aktivierungen $\boldsymbol{a}^\ell$ und die gewichteten Eingaben $\boldsymbol{z}^\ell$ gespeichert. Mit Hilfe dieser gespeicherten Werte wird der Fehler der letzten Schicht $\boldsymbol{\delta}^N$ berechnet und bis zur ersten Schicht zurück propagiert. Aus diesen Fehlern $\boldsymbol{\delta}^\ell$ werden dann die partiellen Ableitungen berechnet. Dies ist nochmal als Pseudocode in Algorithmus 5 festgehalten. Dieser Algorithmus schafft es also die partiellen Ableitungen aller Parameter zugleich zu berechnen. Der Rechenaufwand ist dabei ungefähr so hoch wie zwei Auswertungen des Netzwerkes. Dies hat das Trainieren moderner Netze erst möglich gemacht. Wählt man zum Beispiel den naiven Ansatz und perturbiert einen Eintrag des Parametervektors θ und nähert den Gradient durch

$$\frac{\partial L}{\partial \theta_i} \approx \frac{L(\theta + \varepsilon e_i) - L(\theta)}{\varepsilon}$$

an. So benötigte man für jeden Parameter den Rechenaufwand einer Auswertung des Netzes.

Algorithmus 5 Der *Backpropagation Algorithmus* berechnet den Gradienten ∇L zu einem Trainingsbeispiel $(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}))$.

Gegeben:
Eine differenzierbare Verlustfunktion L
Ein Trainingsbeispiel $(\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x}))$

$\boldsymbol{a}_0 \leftarrow x$ ▷ Setze die Aktivierung $\boldsymbol{a}_0$ auf den Eingabevektor $\boldsymbol{x}$
Feedforward:
for jede Schicht $\ell = 1, 2, \ldots, N$ **do**
    berechne: ▷ Wie in der standardmäßigen Berechnung der Ausgaben
    $\boldsymbol{z}^\ell = W^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell$
    $\boldsymbol{a}^\ell = \sigma\left(\boldsymbol{z}^\ell\right)$
Backpropagate:
$\boldsymbol{\delta}^N = \nabla_{\boldsymbol{a}} L \odot \sigma'\left(\boldsymbol{z}^N\right)$ ▷ Berechne Fehler der Ausgabeschicht nach (2.3.5)
for jedes $\ell = N-1, N-2, \ldots, 1$ **do**
    $\boldsymbol{\delta}^\ell = \left(\left(W^{\ell+1}\right)^T \boldsymbol{\delta}^{\ell+1}\right) \odot \sigma'\left(\boldsymbol{z}^\ell\right)$ ▷ Berechne die Fehler nach (2.3.6)
return
    $\frac{\partial L}{\partial b_j^\ell} = \delta_j^\ell$ ▷ Nach (2.3.7)
    $\frac{\partial L}{\partial w_{jk}^\ell} = a_k^{\ell-1} \delta_j^\ell$ ▷ Nach (2.3.8)

3. Künstliche Neuronale Netze in Reinforcement Learning

In diesem Kapitel werden die Ergebnisse aus Kapitel 1 und Kapitel 2 zusammengeführt. Es soll ein künstliches neuronales Netz als parametrisierte Strategie in einem Reinforcement Learning Algorithmus genutzt werden. In einem Reinforcement Learning Problem soll eine Zielfunktion J maximiert werden. Dafür werden m Trajektorien mit einer Strategie ausgeführt, um dann in (1.3.3) den erwartungstreuen Schätzer des Gradienten

$$\nabla_\theta J(\theta) \approx \hat{g} := \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta\left(a_t^{(i)} \mid s_t^{(i)}\right) \cdot \left(r_t^{(i)} - \bar{R}\right) \tag{3.0.1}$$

zu erhalten. Hier ist $\bar{R} = \frac{1}{m} \sum_1^m R_{\tau^{(i)}}$. Der Einfachheit halber betrachten wir die verschiedenen Zustände s, Aktionen a und Belohnungen r unabhängig davon zu welcher Trajektorie sie gehören. So ergibt sich

$$\hat{g} = \frac{1}{m} \sum_{j=1}^{H \cdot m} \nabla_\theta \log \pi_\theta(a_j \mid s_j) \cdot \left(r_j - \bar{R}\right).$$

Die Strategie π_θ soll nun durch ein Neuronales Netz beschrieben werden, wobei θ die Parameter, nämlich die Schwellenwerte b und Gewichte w, darstellt. Dafür muss die Ausgabe des Netzes eine Wahrscheinlichkeitsverteilung der möglichen Aktionen sein. Das wird realisiert, in-

dem die Ausgabeschicht so viele Neuronen enthält, wie mögliche Aktionen existieren. Damit die Ausgabe des Netzes Wahrscheinlichkeiten darstellt, müssen die Ausgaben so normiert sein, dass die Summe der Ausgaben eins ergibt. Dies wird standardmäßig mit Hilfe einer *Softmax*-Aktivierungsfunktion erreicht. Diese Beschreibt die Aktivierung α_j^N des j-ten Neuron in der letzten Schicht N folgendermaßen:[23]

$$\alpha_j^N = \frac{e^{z_j^N}}{\sum_i e^{z_i^N}}. \tag{3.0.2}$$

Der Index i im Nenner geht über alle Neuronen der Ausgabeschicht. Es ist offensichtlich, dass die Summe über alle Aktivierungen der Ausgabeschicht eins ergibt. So ist $\pi_\theta(a_j \mid \boldsymbol{s}_j) = \alpha_j^N(\boldsymbol{s}_j)$ die Aktivierung des j-ten Neuron der Ausgabeschicht mit dem Netzwerkeingabevektor $\boldsymbol{s}_j$. Wir definieren uns nun eine Ersatz-Verlustfunktion durch

$$\tilde{L}(\theta) = -\frac{1}{m} \sum_{j=1}^{H \cdot m} \log \pi_\theta(a_j \mid \boldsymbol{s}_j) \cdot \left(r_j - \bar{R}\right). \tag{3.0.3}$$

Es lässt sich leicht sehen, dass diese Verlustfunktion die Bedingungen einer Verlustfunktion für den Backpropagation Algorithmus erfüllt. Die erste Bedingung ist, dass sich die Funktion als Mittelwert von Verlustfunktionen für einzelne Trainingselemente schreiben lässt. Dies ist in (3.0.3) offensichtlich der Fall mit den Trainingselementen $(\boldsymbol{s}_j, (a_j, r_j))$. Die zweite Bedingung ist, dass die Verlustfunktion eine Funktion der Ausgabe des Netzes ist. Da die Ausgabe des Netz in unserem Fall genau $\pi_\theta(a_j \mid \boldsymbol{s}_j)$ ist, ist auch diese Bedingung erfüllt. Also können wir auf diese Verlustfunktion den in Algorithmus 4 beschriebenen Adam Algorithmus anwenden. Dieser versucht über den Gradienten die Verlustfunktion zu minimieren, da der Gradient der Verlustfunktion $\tilde{L}$ dem Schätzer $\hat{g}$ bis auf das Vorzeichen gleicht, wird so die Zielfunktion $J(\theta)$ maximiert. Es ergibt sich der Algorithmus 6. Dieser unterscheidet sich nur geringfügig von dem Strategie Gradienten Verfahren mit Baseline in Algorithmus 2. Der Hauptunterschied ist, dass das Update der Parameter nicht mehr mit einer festen Schrittweite geschieht, sondern der Adam-Algorithmus genutzt wird, da die Strategie nun explizit in einem neuronalen Netz gespeichert ist. Der Gradient der Strategie wird effizient mit Hilfe des Backpropagation Algorithmus berechnet.

[23] α wird hier als Symbol für die Aktivierung genutzt, um Verwechslungen mit den Aktionen zu vermeiden.

Algorithmus 6 *Strategie Gradienten Algorithmus mit Baseline angewandt auf ein künstliches Neuronales Netz mit Adam Optimierung*

Gegeben:
Eine differenzierbare Strategie π_θ

1: Initialisiere θ beliebig ▷ z.B. Zufällig
2: **while** True **do** ▷ Oder bis eine Abbruchbedingung erfüllt ist
3: Erzeuge m Trajektorien mit der Strategie π_θ
4: Nutze den *Adam* Algorithmus zum Aktualisieren der mit der Verlustfunktion (3.0.3)
5: Der Gradient wird mit Hilfe des Backpropagation Algorithmus berechnet
6: Ein Batch $\mathcal{B}$ besteht dabei aus den erzeugten $m \cdot H$ Aktionen, Zuständen und Belohnungen der Trajektorien
7: **return** θ

Anwendung

4. Problembeschreibung

In diesem Kapitel soll die *Umwelt* des Reinforcement Learning Problems beschrieben werden. Der Agent kann mit der Umwelt interagieren, kann aber nicht ihre Regeln beeinflussen; vergleiche hierzu [SB18, Kap. 2]. In dem vorliegenden Anwendungsfall ist die Umwelt ein konzentrierendes solarthermisches Kraftwerk, in dem die Putzaktivitäten mit Hilfe eines Reinforcement Learning Algorithmus gesteuert werden sollen. Die Umwelt beinhaltet zum einen, den Aufbau des Kraftwerks und die Klimabedingungen mit den spezifischen Verschmutzungsraten, zum anderen welche Aktionen möglich sind und wie sich diese auf die Umwelt auswirken.

4.1. Greenius

Die Modellierung des Solarkraftwerks erfolgt mit Greenius; siehe [Qua+01] und [Der15]. Greenius ist eine vom deutschen Luft- und Raumfahrtzentrum entwickelte Simulationssoftware für erneuerbare-Energien-Kraftwerke. Auf Basis der Kraftwerkspezifikationen und Wetterdaten bestimmt Greenius verschiedene finanzielle Parameter über die gesamte Laufzeit des Kraftwerkes. Dafür benötigt es ein Jahr stündlich aufgelöster Wetterdaten bestehend aus Außentemperatur, Luftdruck, Luftfeuchtigkeit, DHI,[24] DNI,[25] Windrichtung, Windgeschwindigkeit und dem Stand der Sonne. Es ist nicht möglich eine zeitlich variable Sauberkeit des Solarfeldes oder die Putzkosten direkt in Greenius einzugeben. Dies wird extern in einem, im Rahmen dieser Arbeit geschriebenen, Python-Programm simuliert. Die Verschmutzung wird, mit der Einstrahlung verrechnet, während die jährlichen Kosten auf die greeniusinternen Betrieb und Wartungskosten aufaddiert werden.

4.2. Kraftwerkspezifikation

Die in der vorliegenden Arbeit genutzten Kraftwerkspezifikationen sind an das Kraftwerk Andasol I in der spanischen Provinz Granada angelehnt. Es handelt sich um ein Parabolrinnenkraftwerk. Hier sind in mehreren Reihen, sogenannte *Loops*, parabolförmige Spiegel aufgestellt. In dem Brennpunkt der Spiegel verläuft ein Absorberrohr, in dem eine Flüssigkeit fließt, die so auf mehrere hundert Grad erhitzt wird; siehe Abbildung 0.0.1 und 4.2.1. Mit Hilfe der Wärme

[24] Diffuse Horizontal Irradiance, die Bestrahlungstärke auf einer horizontalen Empfangsfläche.
[25] Direct Normal Irradiance, die Bestrahlungstärke auf einer senkrecht zur Sonne ausgerichteten Empfangsfläche.

Abbildung 4.2.1: Bild des Kraftwerks Andasol III. Dieses Kraftwerk weißt viele Ähnlichkeiten zu Andasol I auf. Im Vordergrund auf der linken Seite ist der thermische Salzspeicher zu sehen. Im Hintergrund sieht man die Parabolspiegel. Es lässt sich erkennen, wie die Spiegel in Loops aufgestellt sind. Das Bild ist aus der Pressemitteilung zur Eröffnung des Kraftwerkes entnommen [RWE11].

der Flüssigkeit kann Dampf erzeugt werden, der wiederum eine Turbine antreibt. Ein Teil der Wärme kann auch in thermische Speicher geleitet werden. Diese Speicher sorgen dafür, dass auch dann Strom produziert werden kann, wenn keine Sonne scheint.
Das hier betrachtete Kraftwerk verfügt über eine Turbine mit einer Nennleistung von knapp $50\,\mathrm{MW_{el}}$ und einem thermischen Speicher, der ausreicht, um die Turbine für 7,5 Stunden auf Vollast zu betrieben. Das Solarfeld ist, mit einer Aperturfläche von $510\,000\,\mathrm{m}^2$, entsprechend der Kapazitäten dimensioniert. Insgesamt besteht das Solarfeld aus 156 Loops. Sowohl die technischen, als auch die finanziellen Parameter sind übereinstimmend mit den Parametern in [Wol+18] gewählt, um die Vergleichbarkeit zu erhalten. Ein Auszug der Spezifikationen ist in Tabelle 4.2.2 gelistet. Die vollständigen Daten mit Quellen sind in [Wol+18] beschrieben.

4.3. Verschmutzung und Reinigung

Die Sauberkeit eines Loops i wird mit $\xi_i \in [0, 1]$ bezeichnet. Sie gibt das Verhältnis der aktuellen Reflektivität $\rho(t)$ gegenüber der Reflektivität eines optimal sauberen Spiegel ρ_{cl} an. So ist die Sauberkeit des Loops i zum Zeitpunkt t definiert als

$$\xi_i(t) = \frac{\rho(t)}{\rho_{cl}}.$$

Tabelle 4.2.2: Auflistung der wichtigsten, in der Simulation mit Greenius genutzten technischen Daten. Eine vollständige Liste findet sich in [Wol+18].

Parameter	**Wert**
Nennleistung der Turbine	49,9 MW_{el}
Anzahl der Loops	156
Aperturfläche des Solarfeldes	510 000 m^2
Kühlung	Wasser
geplante Laufzeit	25 Jahre
Grundstückkosten	2 € $/m^2$
Zinssatz	3,2%
Eigenkapitalquote	30%

Die Verschmutzungsrate SR ist die Veränderung der Sauberkeit durch Umwelteinflüsse über die Zeit. Manuelles Putzen wird nicht in der Verschmutzungsrate festgehalten. Wird ein Loop nicht gereinigt, so lässt sich die Verschmutzungsrate als

$$\mathrm{SR} = \frac{\xi_i(t + \Delta t) - \xi_i(t)}{\Delta t}$$

definieren, wobei Δt der Zeitraum zwischen zwei Messungen ist. Hier wird die Verschmutzungsrate SR ausschließlich pro Tag, das heißt mit der Einheit $1/d$ angegeben. Es wird angenommen, dass die Verschmutzungsrate gleichmäßig auf dem gesamten Solarfeld wirkt. Die Verschmutzungsrate ist in der Regel negativ, außer bei natürlichen Reinigungsereignissen – typischerweise bei starkem Regen.

Das manuelle Putzen wirkt der Verschmutzung entgegen. Dies geschieht loopweise mit Hilfe von Putzfahrzeugen. Wird ein Loop gereinigt, wird die Sauberkeit nicht auf 1, sondern auf $\hat{\xi} = 0{,}986$ gesetzt, da das Reinigen nicht perfekt ist. Der Wert ist an das hier verwendete, und im nächsten Kapitel beschriebene, Putzfahrzeug *Albatros* abgestimmt und aus [Kai11] entnommen. Jedes Putzfahrzeug kann sowohl in einer Nachtschicht, als auch in einer Tagesschicht eingesetzt werden. Dabei wird immer der Loop als nächstes gereinigt bei dem die letzte Reinigung am längsten vergangen ist. Vor der Nacht hat ein Loop i die Sauberkeit $\xi_i(t)$ und der Agent kann auf Basis des sichtbaren Zustandes die Aktion für die Nacht und den nächsten Tag wählen. Daraus ergeben sich drei mögliche Tagessauberkeiten für den folgenden Tag $\xi_i^d(t + 1)$. Wird Loop i nicht gereinigt bleibt die Tagessauberkeit bei $\xi_t(t)$. Wenn in der Nacht gereinigt wurde, wird die Tagessauberkeit auf $\hat{\xi}$ gesetzt. Wird tagsüber geputzt, wird die Sauberkeit auf den Mittelwert zwischen vorhergehender Sauberkeit und der Sauberkeit nach dem Putzen gesetzt $\frac{\xi_t(t)+\hat{\xi}}{2}$. Dies soll einbeziehen, dass während des Tages der Loop für eine gewisse Zeit noch in der alten und für die restliche Zeit der geputzten Sauberkeit vorliegt. Zusammengefasst ergibt sich für die

Tagessauberkeit also folgende Fallunterscheidung:

$$\xi_i^d(t+1) = \begin{cases} \hat{\xi} & \text{, falls nachts geputzt} \\ \frac{\hat{\xi}+\xi_i(t)}{2} & \text{, falls tagsüber geputzt} \\ \xi_i(t) & \text{sonst.} \end{cases} \tag{4.3.1}$$

Für die Sauberkeit am Abend des folgenden Tages $\xi_i(t+1)$ wird dann die Verschmutzungsrate einberechnet. Dabei wird beachtet, dass der Loop der Verschmutzung kürzer ausgesetzt war, falls er tagsüber geputzt wurde. Es ergeben sich wieder 3 Fälle:

$$\xi_i(t+1) = \begin{cases} \hat{\xi} + \mathrm{SR}(t+1) \cdot 1d & \text{, falls nachts geputzten} \\ \hat{\xi} + \mathrm{SR}(t+1) \cdot 0{,}5d & \text{, falls tagsüber geputzt} \\ \xi_i(t) + \mathrm{SR}(t+1) \cdot 1d & \text{sonst.} \end{cases} \tag{4.3.2}$$

Abschließend wird die durchschnittliche Tagessauberkeit über dem ganzen Solarfeld bestimmt

$$\xi_{\mathrm{Feld}}^d(t+1) = \frac{1}{N} \sum_{i=1}^{N} \xi_i^d(t+1). \tag{4.3.3}$$

Hier ist N die Anzahl der Loops in dem Solarfeld. Die durchschnittliche Feldsauberkeit wird im Folgenden nur Feldsauberkeit genannt. Es ist allerdings, wenn nicht ausdrücklich anders beschrieben, die abendliche durchschnittliche Feldsauberkeit

$$\xi_{\mathrm{Feld}}(t+1) = \frac{1}{N} \sum_{i=1}^{N} \xi_i(t+1) \tag{4.3.4}$$

gemeint, da auf dieser Basis die Entscheidungen getroffen werden.
Ein weiterer Einfluss, den die Putzaktionen auf den Ertrag des Solarfeldes haben, ist die Verfügbarkeit. Wenn tagsüber geputzt wird, müssen die entsprechenden Loops defokussiert werden. Dabei wird ein Loop so lange defokussiert, wie dieser geputzt wird. Die Verfügbarkeit α_i ist der Anteil des Tages an welchem Loop i nicht defokussiert ist. Es ergibt sich

$$\alpha_i(t) = \begin{cases} 1 - \frac{T_{cl}}{T_d} & \text{falls tagsüber an Tag } t \text{ geputzt} \\ 1 & \text{sonst.} \end{cases} \tag{4.3.5}$$

Hierbei ist T_d die Länge des Solartages – also von Sonnenaufgang bis Sonnenuntergang. T_{cl} ist die Zeit, die das Putzfahrzeug benötigt einen Loop zu reinigen. Wir gehen davon aus, dass bis zu zwei Putzfahrzeuge einen Loop gleichzeitig putzen können und sich in diesem Fall T_{cl} halbiert. Analog zu der Feldsauberkeit wird die Feldverfügbarkeit als Durchschnitt der Verfügbarkeit über alle Loops gewählt,

$$\alpha_{\text{Feld}}(t) = \frac{1}{N} \sum_{i=1}^{N} \alpha_i(t). \tag{4.3.6}$$

Weder die Verfügbarkeit noch die täglich aufgelöste Feldsauberkeit kann Greenius direkt übergeben werden. Daher behelfen wir uns, wie in [Wol+18] damit, die beiden Parameter auf die Einstrahlung umzurechnen. Wir verringern die Einstrahlung um den Anteil, welcher durch Verschmutzung oder Defokussierung verloren geht.

$$\text{DNI}_{\text{mod}}(s) = DNI(s) \cdot \alpha_{\text{Feld}}(t) \cdot \xi_{\text{Feld}}(t) \quad \forall s \in \text{Tag } t. \tag{4.3.7}$$

Man beachte, dass die DNI Werte im Gegensatz zu der Verfügbarkeit und der Feldsauberkeit, stündlich aufgelöst sind. Diese so modifizierten DNI Zeitreihen werden Greenius übergeben.

4.4. Putzkosten

In dieser Arbeit wurde, wie in [Wol+18], das Putzfahrzeug *Albatros* [Alb13] gewählt. Die jährlichen Kosten, welche durch das Putzen entstehen, setzen sich aus festen und aus aktionsabhängigen Kosten zusammen. Die festen Kosten sind die Abschreibungskosten der Putzfahrzeuge, bestehend aus dem Produkt der Anzahl der Putzfahrzeuge N_{Pfz} und der Abschreibung pro Putzfahrzeug K_A.[26] Die aktionsspezifischen Kosten hängen von der Anzahl der geputzten Loops ab. Es lassen sich aus den Kennwerten in Tabelle 4.4.1 und 4.4.2 die Wasserkosten P_W, Lohnkosten P_L und Treibstoffkosten P_T pro gereinigtem Loop berechnen. Multipliziert man diese Kosten mit der Anzahl der gereinigten Loops, erhält man die variablen Putzkosten. Diese lassen sich durch die Strategie beeinflussen. Die gesamten Putzkosten $\mathcal{K}$ lassen sich schreiben als,

$$\mathcal{K} = N_{\text{Pfz}} \cdot K_A + \sum_{d=1}^{365} (P_W + P_L + P_T) \cdot L(d),$$

wobei $L(d)$ die Anzahl der geputzten Loops an Tag d sind. Diese jährlichen Putzkosten werden zu den jährlichen Betriebs- und Wartungskosten in Greenius addiert.

[26] Die relevanten Kosten sind in Tabelle 4.4.2 aufgeführt, die Spezifikationen des Putzfahrzeug Albatros in Tabelle 4.4.1.

Tabelle 4.4.1: Spezifikationen des Putzfahrzeug Albatros. Die Daten wurden wie in [Wol+18] gewählt und stammen hauptsächlich aus [Kai11]. Alle Angaben sind für ein Putzfahrzeug.

Parameter	**Wert**
Putzgeschwindigkeit	9 Loops pro 8 Stunden Schicht
Betriebspersonal	1
Treibstoffverbrauch	7 l/Loop
Sauberkeit nach Reinigung $\hat{\xi}$	0.986
Wasserverbrauch (demineralisiert)	1 m^3/Loop
geschätzte Lebensdauer	15 Jahre

Tabelle 4.4.2: Für die Reinigung relevante Kosten. Die Daten wurden wie in [Wol+18] gewählt. Dort sind auch die jeweiligen Quellen angegeben.

Parameter	**Kosten**
Jahresgehalt für eine Person	48000 $\frac{€}{\text{Pers·a}}$
Wasserkosten (demineralisiert)	0,39 $\frac{€}{\text{m}^3}$
Treibstoffkosten	1,5 $\frac{€}{l}$
Abschreibung Putzfahrzeug (K_A)	51400 $\frac{€}{\text{Fahrzeug·a}}$

Für die Lohnkosten P_L wird angenommen, dass ein Arbeitsjahr aus 250 acht Stunden Schichten besteht. Nur die Stunden, die für das Reinigen benötigt werden, fließen in die Putzkosten mit ein. Das bedeutet, dass der Arbeiter andere Aufgaben in Betrieb und Wartung übernehmen muss, wenn er nicht putzt. Es ergibt sich

$$P_L = \frac{48000 \frac{€}{\text{Jahr}}}{250 \frac{\text{Schichten}}{\text{Jahr}} \cdot 9 \frac{\text{Loops}}{\text{Schicht}}} = 21{,}\overline{3} \frac{€}{\text{Loop}}.$$

Für die Wasserkosten pro Loop P_W ergibt sich ein Wert von 0,39 €/Loop und für die Treibstoffkosten pro Loop P_T ergibt sich 10,5 €/Loop. Das bedeutet, dass die Wasserkosten den deutlich geringsten Anteil an den variablen Kosten haben, die Personalkosten hingegen nehmen den größten Anteil ein. Sie sind mehr als ein 50-faches höher als die Wasserkosten und stark standortabhängig. Zum Beispiel werden in [Wol+18] die Lohnkosten in Marokko um einen Faktor sieben kleiner geschätzt als die hier für Spanien geschätzten Lohnkosten.

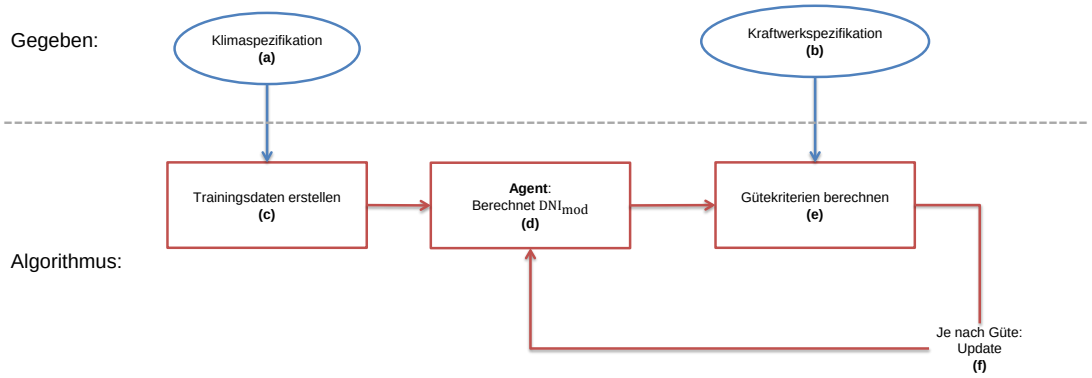


Abbildung 5.0.1: Schematische Darstellung des Algorithmus. Blau werden die benötigten Voraussetzungen dargestellt, rot die Bausteine des Algorithmus.

5. Aufstellen des Algorithmus

Ziel des hier beschriebenen Algorithmus ist es einen Agenten zu erstellen, welcher die Putzaktivität in einem konzentrierenden solarthermischen Kraftwerk steuert. Ein Schaubild des Algorithmus ist in Abbildung 5.0.1 zu sehen. Anhand dieses Schaubildes soll der Aufbau des Algorithmus skizziert werden. Damit die Umgebung des Agenten ausreichend definiert ist, benötigt der Algorithmus Informationen über das Kraftwerk (b) und dessen Klima (a). Die Kraftwerkspezifikation, welche in der vorliegenden Arbeit genutzt werden, sind in Kapitel 4.2 aufgeführt. Es ist festzuhalten, dass der vorliegende Algorithmus analog mit allen Kraftwerktypen funktioniert, so lange diese in Greenius simuliert werden können. Die benötigten Klimadaten wurden über mehrere Jahre an der *Plataforma Solar de Almería* (PSA) gemessen.

Reinforcement Learning Algorithmen benötigen viele Daten, damit nicht einzelne, möglicherweise atypische Jahre auswendig gelernt werden. Stattdessen soll der Algorithmus eine möglichst generell gültige Strategie lernen. Daher müssen wir die Daten durch statistische Methoden künstlich erweitern (c). In Kapitel 5.1 wird beschrieben auf welche Art die Daten in der vorliegenden Arbeit erzeugt wurden.

Der Agent wird nach der in Kapitel 1 beschriebenen Reinforcement Learning Methode auf den generierten Daten trainiert. Er wählt Aktionen durch welche die Feldsauberkeit und die Putzkos-

ten bestimmt werden (d). Die Putzkosten und die modifizierte Einstrahlung, vergleiche (4.3.7), werden Greenius übergeben. Greenius berechnet daraus die finanziellen Parameter, welche das Kraftwerk über die gesamte Laufzeit erwirtschaftet (e). Aus diesen wird das Gütekriterium berechnet, welches als Belohnung im Reinforcement Learning Algorithmus fungiert (f). Welches Gütekriterium gewählt und wie dieses berechnet wird, ist in Kapitel 5.2 aufgeführt.

5.1. Daten Generieren

Für den hier angewandten Algorithmus werden viele Wetterdaten benötigt. Dafür werden auf Basis von real gemessenen Daten künstliche Wetterdaten erzeugt.[27] Es ist wichtig darauf zu achten, dass bei den erzeugten Daten die zeitliche Abhängigkeit ähnlich zu den realen zeitlichen Abhängigkeiten ist, da mit den hier gewählten Parametern 18 Putzschichten benötigt werden, damit das Feld komplett gereinigt wurde und daher beeinflussen auch die Aktionen vorhergehender Tage die aktuelle Feldsauberkeit.
Die Datenbasis besteht aus Messreihen von 16 Jahren für die in Greenius benötigten Wetterparameter. Für die Anwendung der Putzstrategien werden allerdings auch die Verschmutzungsraten benötigt. Diese liegen lediglich für die letzten sechs Jahre vor, daher werden die Wetterparameter und die Verschmutzungsrate separat modelliert.
Für die Wetterparameter werden die Messtage in drei verschiedene Klassen eingeteilt. Diese Einteilung geschieht auf Grund der Schwankung der Einstrahlung. Die Messtage werden, wie in [SH+18] vorgeschlagen, minutenweise in acht verschiedene *variability*-Klassen (var-Klasse) aufgeteilt. In der ersten var-Klasse sind die Tage, die kaum von der theoretischen Einstrahlung abweichen. Mit steigender var-Klasse steigt auch die Abweichung zu der theoretischen Einstrahlung bis die achte var-Klasse nahezu vollkommene Bewölkung darstellt. Eine höhere var-Klasse bedeutet hierbei nicht ausschließlich eine niedrigere Einstrahlung, sondern auch eine größere Schwankung in der Einstrahlung.
Die Anzahl der Klassen wird mit Hilfe der folgenden Kriterien auf drei reduziert: In der ersten Klasse hier, sind Tage die zu mehr als 76% als var-Klasse 1 oder 2 klassifiziert sind. Wenn mehr als die Hälfte des Tages in var-Klasse 7 oder 8 eingeordnet sind, werden sie hier in die dritte Klasse eingeteilt. Die restlichen Tage werden in die zweite Klasse eingeordnet. Betrachtet man die Verteilung der Klassen über die Tage des Jahres, wird sichtbar, dass klarer Himmel – also Klasse 1 – häufiger im Sommer auftritt. Während des restlichen Jahres besteht ungefähr ein Drittel der Tage aus Klasse 1; vergleiche Abbildung 5.1.1.

[27] Eine Beschreibung der Wetterdaten und Messmethoden finden sich [Wol16, Kap.5].

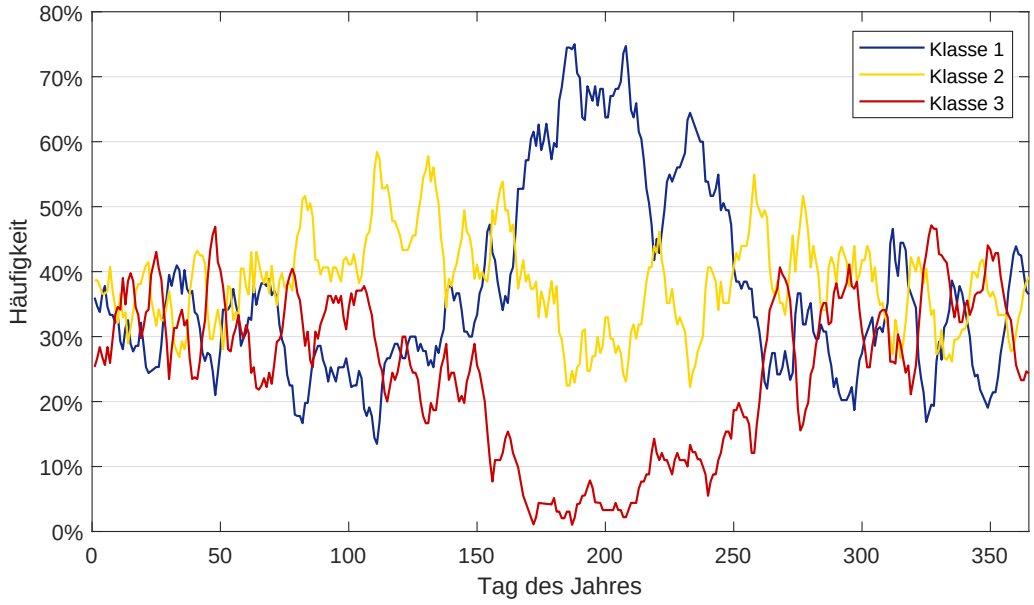


Abbildung 5.1.1: Häufigkeiten der verschiedenen Klassen über das Jahr. Die gezeigten Häufigkeiten sind der gleitende Durchschnitt von 15 Tagen.

Es werden nun für jeden Tag des Jahres[28] i die Übergänge zwischen den einzelnen Klassen in einem Zeitfenster von $i \pm 7 \pmod{365}$ gezählt und daraus die Übergangswahrscheinlichkeiten bestimmt. In Tabelle 5.1.1 ist die globale Übergangsmatrix dargestellt. Es wird sichtbar, dass die Wahrscheinlichkeit in einer Klasse zu bleiben am höchsten ist. Am unwahrscheinlichsten ist der direkte Wechsel von Klasse 1 zu Klasse 3 und umgekehrt. Durch die Übergangswahrscheinlichkeiten wird sichergestellt, dass die zeitlichen Abhängigkeiten zwischen den verschiedenen Tagen ähnlich sind, wie die Abhängigkeiten der realen Messtage. Um die synthetischen Daten zu erzeugen, wird zunächst die Klasse für den ersten Tag gezogen. Die Wahrscheinlichkeiten für den ersten Tag werden durch die Häufigkeit der Klassen in den Messtagen zwischen dem 358. und dem 8. Tag des Jahres bestimmt. Für die darauffolgenden Tage wird jeweils aus den Übergangswahrscheinlichkeiten der aktuellen Klasse und des aktuellen Tages gezogen. Wenn für alle Tage die Klasse bestimmt ist, wird für jeden Tag des Jahres ein realer Messtag gezogen, welcher die gleiche Klasse besitzt und zur gleichen Zeit des Jahres – wieder in einem Zeitfenster von 15 Tagen – aufgenommen wurde. Es werden reale Messtage genommen, da so die Abhängigkeiten innerhalb der verschiedenen Parameter wie Einstrahlung, Temperatur und Luftdruck erhalten bleiben. Auch die zeitlichen Abhängigkeiten innerhalb eines Tages bleiben so erhalten.

Die Verschmutzungsrate muss gesondert simuliert werden, da nur Messwerte der letzten sechs

[28] Tag des Jahres i steht für den i-ten Kalendertag des Jahres.

Tabelle 5.1.1: Globale Übergangswahrscheinlichkeiten zwischen den Klassen in den Messdaten. In den Zeilen ist die Ausgangsklasse und die Spalte definiert die Zielklasse.

	Klasse 1	Klasse 2	Klasse 3
Klasse 1	58 %	32 %	10 %
Klasse 2	31 %	45 %	24 %
Klasse 3	17 %	38 %	45 %

Jahre vorliegen. Für die Modellierung der Verschmutzungsrate werden die Messwerte des Regens genutzt, da Regen einen starken Einfluss auf die Verschmutzungsrate hat [Wol16]. Die Niederschlagsmessungen liegen für die letzten acht Jahre vor. Sowohl der Regen als auch die Verschmutzungsrate werden nach der Intensität in drei Klassen eingeteilt. Die Verschmutzungsrate ist unterteilt in hohe Verschmutzung für eine Verschmutzungsrate von mehr als 2% ($SR < -0{,}02$), in normale Verschmutzung ($SR \in [-0{,}02; 0)$) und natürliche Reinigung also positive Verschmutzungsrate ($SR > 0$). Der Regen wird anhand des täglichen kumulierten Niederschlags geteilt. Ein Tag wird bei einem Niederschlag von unter 0,05 mm als trocken gewertet, bei 0,05 bis 5 mm pro Tag als schwacher Niederschlag und über 5 mm als starker Niederschlag gewertet.

Mit Hilfe dieser Einteilungen wird die Verschmutzungsrate modelliert. Dafür wird in einem ersten Schritt festgelegt welche Verschmutzungsintensität vorliegt und in einem zweiten Schritt wird aus einer dieser Intensität entsprechenden Wahrscheinlichkeitsverteilung der genaue Wert gezogen. Es werden dabei einige Annahmen aufgestellt. Zunächst nehmen wir an, dass der Mittelwert der hohen Verschmutzung Jahreszeitunabhängig ist. Das heißt, bei einem starken Verschmutzungsevent ist die Verteilung aus der gezogen wird Jahreszeitunabhängig. Diese Annahme muss getroffen werden, da in den Messwerten zu wenig starke Verschmutzungsevents aufgetreten sind, um eine Jahreszeitabhängige Verteilung zu finden. Des Weiteren nehmen wir an, dass hohe Verschmutzungsraten nur während schwachem Regen auftreten. Diese Annahme wurde aufgrund der Tatsache getroffen, dass die meisten starken Verschmutzungsereignisse auf nasse Depositionsprozesse wie *roter Regen*[29] oder Ablagerung von Saharastaub zurückgehen. Abschließend nehmen wir an, dass natürliche Reinigung genau dann auftritt, wenn starker Regen auftritt. Das heißt, mit der Schwelle für starken Regen haben wir eine Niederschlagsmenge definiert ab welcher Regen reinigt. Dieser Schwellenwert basiert auf mehreren Studien zur Reinigung von Regen auf Photovoltaikmodulen. So finden die Autoren in [MK13] einen Schwellenwert von 2,5 mm während Kimber et al. in [Kim+06] auf einen

[29] *Roter Regen* bezeichnet leichten Niederschlag, der eine große Menge an Aerosolen aus der Atmosphäre wäscht, welche sich auf Oberflächen ablagern; siehe [Wol16].

Schwellenwert von 5-10 mm täglichen Niederschlag kommen. In der vorliegenden Arbeit wird eine Schwelle von 5 mm gewählt.
Mit Hilfe dieser Annahmen modellieren wir die Verschmutzungsrate wie folgt: Für jeden Tag des Jahres und jede Klasse berechnen wir die Wahrscheinlichkeiten für die verschiedenen Regenklassen. Dies ist nötig, da die Messreihen des Regens, wie die der Verschmutzungsrate nicht die gesamte Messdauer abdecken. Gemäß diesen Wahrscheinlichkeiten ziehen wir für jeden Tag und Klasse die Art des Niederschlag an diesem Tag – kein, schwacher oder starker Niederschlag. Je nachdem welche Regenklasse gezogen wird unterscheiden wir drei Fälle:

1. Wird *starker Regen* gezogen, bedeutet das, dass die Verschmutzungsrate positiv ist. Zu diesem Zweck ziehen wir aus einer stetigen Gleichverteilung zwischen 0,7 und 1. Das Ergebnis dieser Verteilung ist der Anteil der aktuellen Verschmutzung auf den Kollektoren, welche gereinigt wird. Ist zum Beispiel die Sauberkeit eines bestimmten Kollektors vor der natürlichen Säuberung $\xi_i(0) = 0{,}95$ und der gezogene Anteil ist $f = 0{,}8$, so ist die Sauberkeit des Kollektors nach der Reinigung $\xi_i(1) = \xi_i(0) + (1 - \xi_i(0)) \cdot f = 0{,}99$. Die Verschmutzungsrate für den Kollektor i ist also $\mathrm{SR}_i = (1 - \xi_i(0)) \cdot f = 0{,}04$. Mit dieser Methode wird sichergestellt, dass es unterschiedlich starke Säuberung durch Regen gibt und auch die vorherige Sauberkeit des Kollektors Einfluss auf die Säuberung hat.

2. Bei einem *trockenen Tag* wird die Verschmutzung durch das Ziehen aus einer Exponentialverteilung ermittelt. Die Dichtefunktion einer Exponentialverteilung ist durch den Mittelwert definiert.[30] Der Mittelwert dieser Verteilung wird aufgrund der gemessenen Verschmutzung zu dieser Zeit des Jahres angepasst. Die Exponentialverteilung wurde gewählt, da sie gut mit den gemessenen Daten übereinstimmt; siehe hierzu Abbildung 5.1.2.

3. An einem Tag mit *schwachem Regen* ist die Verschmutzungsrate entweder normal oder hoch. Ob das Verschmutzungsereignis normal oder hoch ist, wird zufällig festgelegt. Die Wahrscheinlichkeiten hierfür werden so gewählt, dass der Erwartungswert der starken Verschmutzungsereignisse über das Jahr mit der gemessenen Häufigkeit übereinstimmt. Wird ein starkes Verschmutzungsereignis gezogen, wird die Verschmutzung aus einer eigenen Exponentialverteilung gezogen, welche um -0,02 versetzt ist; siehe auch hierzu Anhang D.1. Diese Exponentialverteilung ist unabhängig von dem Tag des Jahres. Wird eine normale Verschmutzungsrate für den Tag festgelegt, dann wird wie bei einem trockenen Tag die

[30] Für die Dichtefunktion der Exponentialverteilung siehe Anhang D.1.

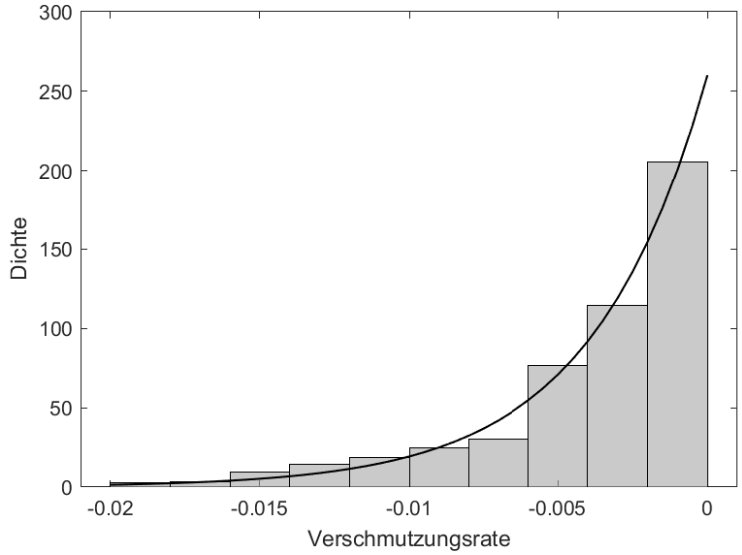


Abbildung 5.1.2: Dichte Histogramm der gemessenen Verschmutzungsrate. Hier wird nur die Verschmutzung zwischen -0,02 und 0 betrachtet. Die schwarze Linie ist die Dichtefunktion der Exponentialverteilung mit dem gleichen Mittelwert wie die angezeigten Daten.

Verschmutzung bestimmt.

5.2. Gütekriterien berechnen

Die Gütekriterien zum Vergleich der verschiedenen Strategien sind an [Wol+18] und [Wol16] angelehnt. Da Strategien verglichen werden sollen, wird eine Referenzstrategie genutzt. Diese besteht aus einem Putzfahrzeug, welches konstant zwei Schichten pro Tag putzt. Diese Strategie wurde gewählt, da sie in zwei verschiedenen Kraftwerken angewandt wird, die von Standort und Auslegung vergleichbar mit dem hier verwendeten Kraftwerk sind; vergleiche [Wol16] beziehungsweise [Fre13].

Greenius berechnet mehrere finanzielle Parameter über die Laufzeit eines Kraftwerkes. Für den Vergleich verschiedener Strategien werden der Greenius Ausgabewert *Gesamtdividenden* D_{tot} genutzt. Diese beschreiben den Gewinn des Kraftwerks über eine 25-jährige Laufzeit. Die Gesamtdividenden setzen sich zum einen aus den Erlösen durch den Stromverkauf und zum anderen aus verschiedenen Kosten, wie zum Beispiel Betriebs-, Versicherungs-, Finanzierungs- oder die Reinigungskosten zusammen. Die jährlichen Dividenden variieren stark. Sie sind negativ während der Bauphase und steigen an bis zur Abbezahlung der Kredite. Unterschiede aufgrund von verschiedenen Wetterbedingungen zwischen den Jahren werden nicht beachtet, da Greenius den gleichen Meteorologischen Datensatz für alle simulierten Jahre nutzt. Um nun die Güte

einer Strategie zu bewerten, wird der relative Mehrgewinn eines Projekts (RMG) gegenüber der Referenzstrategie berechnet,

$$\mathrm{RMG} = \left(D_{\mathrm{tot}}/D_{\mathrm{tot}}^{\mathrm{ref}} - 1\right) \cdot 100\%. \tag{5.2.1}$$

Hier sind $D_{\mathrm{tot}}^{\mathrm{ref}}$ die Gesamtdividenden der oben beschriebenen Referenzstrategie.

5.3. Reinforcement Learning Anwendung

Die in der vorliegenden Arbeit verwendete Reinforcement Learning Methode wurde in den Grundlagen eingehend beschrieben. Hier soll auf die Anwendung dieser Methode eingegangen werden, da erst in dieser einige Details spezifiziert werden. Auch lässt sich der genutzte Algorithmus besser verstehen, wenn dieser an dem Anwendungsbeispiel detailierter ausgeführt wird. Es wird das Strategie Gradienten Verfahren genutzt, wobei die Strategie in einem Feedforward Netz kodiert ist (Algorithmus 6)

Für die Anwendung der Reinforcement Learning Methoden werden zwei Datensätze mit künstlichen Daten erzeugt. Ein Datensatz wird als Trainingsdatensatz, der andere als Validierungsdatensatz genutzt. Der Trainingsdatensatz besteht aus 2000 und der Validierungsdatensatz aus 20 synthetischen Jahren. Der Agent startet mit einer zufälligen Strategie. Die Initialisierungen der Gewichte und Schwellenwerte werden aus einer Gleichverteilung zwischen -0,05 und 0,05 gezogen. Während eines Trainingsschrittes wird die Strategie zehn mal auf das selbe Jahr angewandt. Für jeden Tag des Jahres erhält der Agent den aktuellen sichtbaren Zustand und berechnet die Wahrscheinlichkeiten für jede mögliche Aktion. Aus diesen Wahrscheinlichkeiten wird die ausgeführte Aktion gezogen. Daher unterscheiden sich die Ergebnisse leicht für jedes der zehn Jahre. Damit die Strategie im Verlauf des Trainings weiter exploriert, wird darauf geachtet, dass die Wahrscheinlichkeit für eine Aktion 95% nie überschreitet. Gibt der Agent eine Wahrscheinlichkeit von über 95% für eine Aktion aus, wird diese im Trainingsfall auf 95% gesetzt und die übrigen 5% werden gleichmäßig auf die anderen Aktionen verteilt. Nachdem die Aktionen für ein Jahr gewählt wurden, wird die modifizierte Einstrahlung an Greenius übergeben. Greenius berechnet daraufhin die Gesamtdividenden. Die Gesamtdividenden der verschiedenen Jahre werden normiert, indem der Mittelwert abgezogen und durch die Standardabweichung dividiert wird. Die so normierten Gesamtdividenden werden als Belohnung des Algorithmus 6 genutzt. Durch die Normierung wird sichergestellt, dass die Jahre die schlechter als der Mittelwert sind eine negative Belohnung und die anderen eine positive Belohnung erfahren. Die Division durch die Standard-

abweichung stellt sicher, dass die Belohnungen während verschiedener Trainingsschritte ähnliche Größenordnungen besitzen. Die Strategie wird nach den in Algorithmus 6 festgelegten Regeln aktualisiert. Nach diesem Update der Gewichte und Schwellenwerte wird die aktualisierte Strategie auf das nächste Jahr im Trainingsdatensatz angewandt. Alle 15 Trainingsschritte wird der Fortschritt der Strategie auf dem unabhängigen Validierungsdatensatz getestet. Dafür wird die Strategie auf allen 20 Jahren des Datensatzes ausgeführt und die durchschnittlichen Gesamtdividenden werden gespeichert. Damit eine Vergleichbarkeit der Validierungsschritte gewährleistet werden kann, wird für jeden Tag die Aktion gewählt, für die die höchste Wahrscheinlichkeit bestimmt wurde. Auf diese Art wird aus der stochastischen Strategie im Validierungsfall eine deterministische Strategie. Der Validierungsschritt ist nötig, um den Trainingsfortschritt auf unabhängigen Daten zu verfolgen. Wir nennen einen Agenten *austrainiert*, wenn er an 20 aufeinanderfolgenden Validierungsschritten das beste Ergebnis nicht übertroffen hat; frühestens aber nach 25 Validierungsschritten. Ist ein Agent austrainiert, wird das Training abgebrochen.

6. Implementierung

Das Programm, welches im Zuge dieser Arbeit erstellt wird, lässt sich in drei Teile gliedern. Im ersten Teil werden die gemessenen Wetterdaten aufbereitet und künstliche Wetterdaten, wie in Kapitel 5.1 beschrieben, generiert. In dem darauffolgenden wichtigsten Abschnitt ist das Reinforcement Learning Problem definiert. Abschließend wird die Nachbearbeitung und Auswertung durchgeführt.

An dieser Stelle soll ein Überblick über die Implementierung gegeben werden, dabei wird nicht der genaue Ablauf beschrieben, sondern lediglich die Vorgehensweise und die genutzte Software. Im Fokus steht die Beschreibung der Reinforcement Learning Implementierung.

Der erste Teil, der sich mit dem Erstellen künstlicher Wetterdaten beschäftigt, ist in Matlab geschrieben, da der Zugriff auf die Datenbanken so bestmöglich erfolgen kann. Zunächst werden die Daten aus der Datenbank gelesen. Daraufhin werden die acht var-Klassen minutenweise berechnet, um daraus eine Klasse für den gesamten Tag abzuleiten; vergleiche Kapitel 5.1. Im Anschluss werden die gemessenen Wetterdaten den Tagen des Jahres zugeordnet und die Wetterstatistik über ein Zeitfenster erstellt. Daraus werden dann im letzten Schritt die synthetischen Wetterdaten generiert, wie in Kapitel 5.1 beschrieben. Diese werden in einer *.mat[31] Datei gespeichert. Es sei bemerkt, dass dieser Schritt des Algorithmus nur einmal ausgeführt werden

[31] *.mat ist ein Matlab eigenes Speicherformat.

muss. Auch die leichte Modifizierung der Daten, zum Beispiel die Einführung mehrerer starker Verschmutzungsereignisse, kann auf den einmal gespeicherten Daten an späterer Stelle durchgeführt werden.

Der Hauptteil ist im Gegensatz zu dem vorangegangenen Teil in Python implementiert, da es dort viele Pakete für neuronale Netze gibt. In Abbildung 6.0.2 wird der Programmaufbau schematisch dargestellt. Zur genaueren Erklärung sind in Anhang D.3 Kurzbeschreibungen zu allen Klassenattributen und -methoden aufgelistet.

Dieser Programmteil ist in zwei Blöcke unterteilt: Einer definiert die Umgebung, während der andere den Agenten definiert. Außerdem existiert noch ein Modul, welches für die Kommunikation mit Greenius zuständig ist. Die Umgebung setzt sich zusammen aus einer Klasse für die Wetterdaten, einer Klasse für das Solarfeld und einer für die Spezifikation des Putzfahrzeugs. In der übergeordneten Klasse *Environment*, werden zum einen die Spezifikationen für den Standort gesammelt, wie zum Beispiel die Wasserkosten oder die Wetterdaten. Zum anderen ist die Klasse dafür zuständig, während der Simulation eines Jahres, die Sauberkeit und die Verfügbarkeit der Spiegel zu verfolgen.

Der Agent ist in einer eigenen Klasse implementiert. Diese Klasse beinhaltet einerseits Methoden für die Ausführung des Agenten, zum Beispiel eine Funktion zum Trainieren des Agenten oder eine für die Wahl einer Aktion zu einem gegebenen Zustand. Andererseits ist in dieser Klasse auch der Aufbau des Agenten gespeichert. Das beinhaltet auch das neuronale Netz, welches die Strategie kodiert. Das neuronale Netz ist mit Hilfe der Python-Bibliothek Keras [Cho+15] implementiert. Keras ist eine high-level Bibliothek für die Implementierung neuronaler Netze und wird mit Tensorflow [Aba+15] ausgeführt. Dies stellt sicher, dass die neuronalen Netze effizient implementiert sind. Außerdem sind die meisten Optimierer, so wie der *Adam* Algorithmus, schon in der Bibliothek vorimplementiert. Durch den modulartigen Aufbau von Keras lassen sich leicht verschiedene Konfigurationen und Optimierer testen, um schnell eine gute Architektur für das Problem zu finden.

Die Auswertung erfolgte für jedes getestete Problem separat und wird daher nicht genauer beschrieben. Für die Erstellung der Plots wurde meistens auf die Python basierte Darstellungs-Bibliothek Matplotlib [Hun07] zurückgegriffen.

Bei der Implementierung ist es wichtig die Laufzeit des Trainings zu beobachten. Die Laufzeiten für die anderen Teile des Programms sind weniger kritisch, da diese nur einmalig ausgeführt werden. Die Laufzeit des Trainings setzt sich aus verschiedenen Programmteilen zusammen. Für einen Trainingsschritt muss ein *Batch* von zehn Jahren mit Hilfe von Greenius ausgewertet

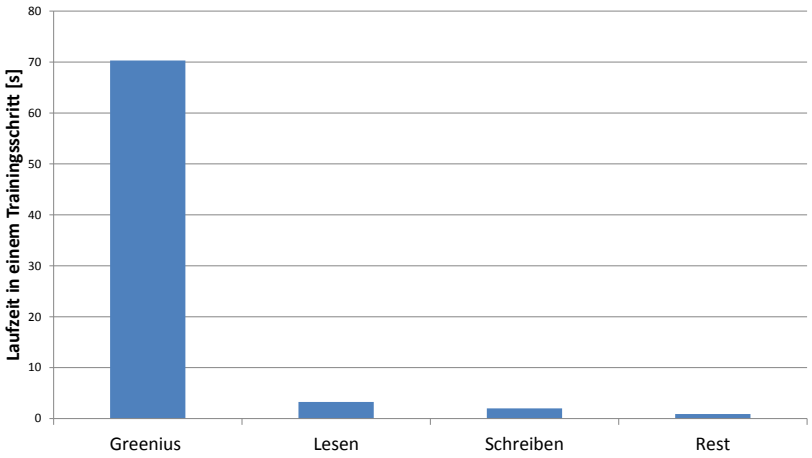


Abbildung 6.0.1: Laufzeiten der verschiedenen Programmteile in Sekunden in einem Trainingsdurchlauf mit einer Batchgröße von zehn Jahren. Ausgeführt auf dem Laptop, der für die Auswertung in dieser Arbeit genutzt wurde: Intel Core i7-3667U CPU @ 2.00 GHz 2.50 GHz, 8GB RAM.

werden. Auf Basis der Ergebnisse dieser Auswertungen wird die Strategie aktualisiert. Ein solcher Trainingsschritt dauert im Schnitt 76,49s.[32] Dabei entfallen 70,31s auf die Ausführung von Greenius. Weitere 2,02s fallen für das Schreiben der Dateien an, welche für die Auswertung von Greenius benötigt werden, in denen die spezifischen Wetterdaten und Putzkosten für das Jahr an Greenius übergeben werden. Weitere 3,26s werden für das Laden der relevanten Kennwerte aus den Ausgabedateien von Greenius benötigt, sodass die, von Greenius unabhängigen Operationen lediglich 0,9s pro Trainingsschritt benötigen; siehe Abbildung 6.0.1. Betrachtet man diese Zeiten wird deutlich, dass der Performance-Engpass für dieses Programm die Ausführung von Greenius ist, die so ohne Weiteres nicht verbessert werden kann. Für einen Validierungsschritt, bestehend aus 15 Trainingsschritten und einer Auswertung des Validierungsdatensatz von 20 Jahren, werden ungefähr 21,5 Minuten benötigt. So erhält man für einen kompletten Trainingsdurchlauf mit 50-70 Validierungsschritten eine Laufzeit von ungefähr 18-25 Stunden.

[32]Die angegebenen Werte sind die Mittelwerte über zehn Trainingsschritte. Ausgeführt auf dem Laptop, welcher für die Auswertung genutzt wurde: Intel Core i7-3667U CPU @ 2.00 GHz 2.50 GHz, 8GB RAM.

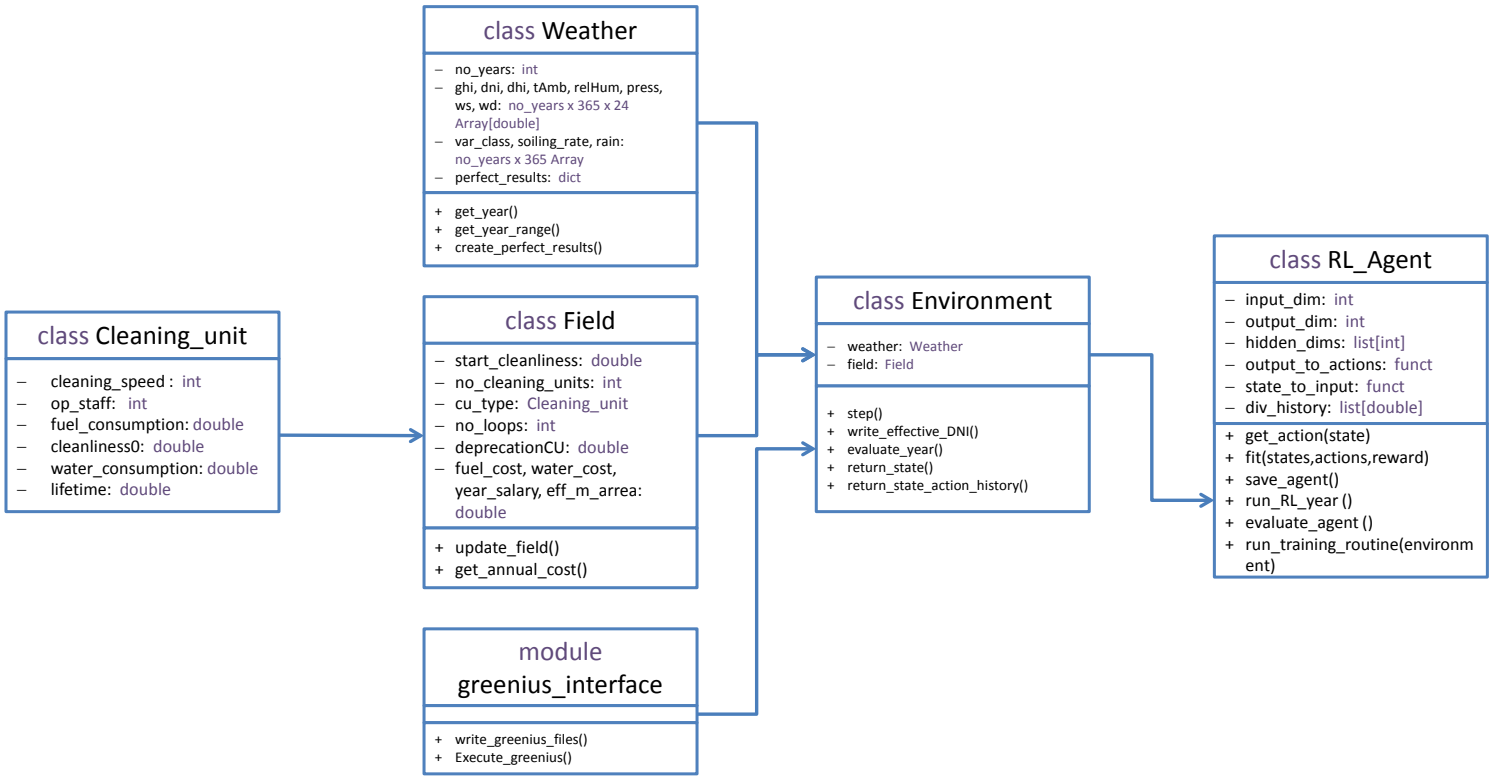


Abbildung 6.0.2: Schematische Darstellung der Implementierung des Reinforcement Learning Algorithmus. – steht für Attribute und + für Methoden. Dabei sind nicht alle Methoden und Attribute aufgelistet, sondern nur die, welche für das Verständnis wichtig sind. Weder die Erstellung der Trainingsdaten, noch die Auswertung, wird in diesem Schaubild betrachtet. Im Anhang D.3 findet sich eine Kurzbeschreibung der Attribute und Methoden.

7. Ergebnisse

7.1. Untersuchung des Algorithmus

In diesem Kapitel soll der Algorithmus in Verbindung mit dem vorliegenden Problem evaluiert werden. Dabei wird zunächst an einem Beispiel untersucht, wie gut der Algorithmus an die optimale Lösung heranreicht. Daraufhin wird die Robustheit des Algorithmus betrachtet, da der Algorithmus an mehreren Stellen zufällige Elemente beinhaltet. Zum Abschluss wird ein Agent aus dem Robustheitstesst genauer analysiert. Insbesondere werden die Entscheidungen zu verschiedenen Trainingszeitpunkten beobachtet. Dies soll einerseits einen Einblick geben, auf welche Art der Agent lernt, andererseits können so die Entscheidungen des austrainierten Agenten einer Plausibilitätsprüfung unterzogen werden.

7.1.1. Vergleich mit optimalem Algorithmus

Zuerst wollen wir überprüfen, ob der Algorithmus bei einem Problem, bei dem die optimale Strategie bekannt ist, eine vergleichbare Lösung findet. In [Wol+18] wurde eine Schwellenwert basierte Strategie untersucht. Dafür wurde ausschließlich die durchschnittliche Feldsauberkeit genutzt. Ab einem bestimmten Schwellenwert der Feldsauberkeit wurde mit allen verfügbaren Einheiten nachts geputzt. Dieses Vorgehen wurde für verschiedene Schwellenwerte und Anzahlen von verfügbaren Putzfahrzeugen auf einem Jahr tatsächlich gemessener Werte durchgeführt. Für jede Ausführung wurde der RMG berechnet. Das beste Ergebnis erzielt eine Strategie mit drei Putzfahrzeugen und einem Schwellenwert für die Feldsauberkeit von ungefähr 0,975; siehe Abbildung 7.1.1.
Da in der vorliegenden Arbeit die Auswertung nicht auf einem gemessenen meteorologische Jahr, sondern auf 20 simulierten Jahren beruht, wurde das Vorgehen wiederholt. Dafür wurden zwei verschiedene Strategien getestet: Die erste besteht wie in dem Paper [Wol+18] daraus, dass ab einem bestimmten Schwellenwert der Feldsauberkeit mit allen verfügbaren Putzfahrzeugen ausschließlich nachts gereinigt wird. Diese Strategie wird mit n bezeichnet. Bei der zweiten Strategie wird mit allen verfügbaren Putzfahrzeugen ab einem bestimmten Schwellenwert Tag und Nacht geputzt bis die Sauberkeit wieder über dem Schwellenwert liegt. Diese Strategie wird mit tn bezeichnet. In [Wol+18] wurde die Strategie tn nicht weiter betrachtet, da im Fall einer konstanten Strategie die Strategie n deutlich besser war.
In Abbildung 7.1.2 werden die Ergebnisse auf den 20 Validierungsdaten für beide Strategien, tn und n dargestellt. Bei dem Vergleich der Strategie n aus [Wol+18] und der vorliegenden Arbeit

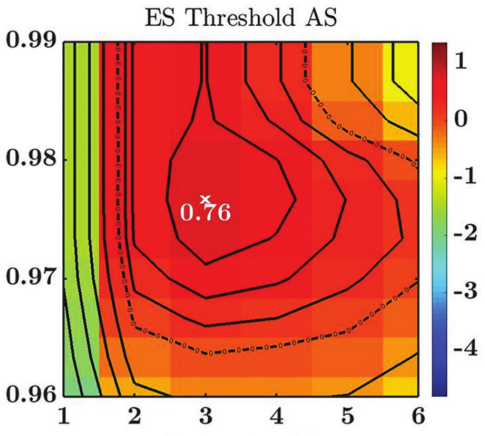


Abbildung 7.1.1: Der relative Mehrgewinn für einen Schwellenwert Algorithmus. Auf der X-Achse ist die Anzahl der Putzfahrzeuge und auf der Y-Achse ist der Schwellenwert der Feldsauberkeit, ab dem mit allen verfügbaren Fahrzeugen nachts geputzt wird, abgebildet. Die Abbildung ist, mit freundlicher Genehmigung von Fabian Wolfertstetter, aus [Wol+18] übernommen. Die Parameter und Kosten des Solarfeldes und der Putzprozedur sind identisch zu den Einstellungen in dieser Masterarbeit. Die zugrunde liegenden Wetterdaten bestehen in diesem Fall aus einem realen Messjahr.

wird deutlich, dass die Ergebnisse qualitativ ähnlich sind. So ist in beiden Fällen eine Strategie mit drei Putzfahrzeugen und einem Schwellenwert von ungefähr 0,975 Feldsauberkeit die beste gefundene Strategie. Der RMG liegt mit 0,85% zwar etwas über dem RMG von 0,76% in [Wol+18], allerdings lässt sich die Abweichung mit den unterschiedlichen Wetterdaten erklären. Bei dem Vergleich der Strategie *tn* und *n* wird deutlich, dass die Strategie *tn* einen höheren RMG als die Strategie *n* erreicht. Bei einem Schwellenwert von 0,97 und zwei Putzfahrzeugen erreicht die Strategie *tn* einen RMG von 1,07%. Daher wird im Folgenden eine Strategie mit zwei Putzfahrzeugen gewählt

Um den besten Schwellenwert für die Strategie *tn* mit zwei Putzfahrzeugen herauszufinden, werden die Schwellenwerte kleinschrittiger überprüft als in Abbildung 7.1.2. Dafür werden auf dem Validierungsdatensatz die Schwellenwerte 0,95 bis 0,985 in 0,001er Schritten ausgewertet. Der beste Algorithmus besitzt einen Schwellenwert von 0,973 für die Feldsauberkeit und erreicht einen RMG von 1,0875%; vergleiche Abbildung 7.1.3. Ist nur die Feldsauberkeit bekannt und sind die Aktionsmöglichkeiten derart eingeschränkt, dass nur mit zwei Tag- und Nachtschichten oder gar nicht geputzt werden kann, liegt es nahe, dass der optimale Algorithmus ein einfacher Schwellenwertalgorithmus ist.

Um nun zu überprüfen, wie nah der Reinforcement Learning Algorithmus in diesem Fall an das Optimum heranreicht, wird ein Agent unter diesen Bedingungen trainiert. Er

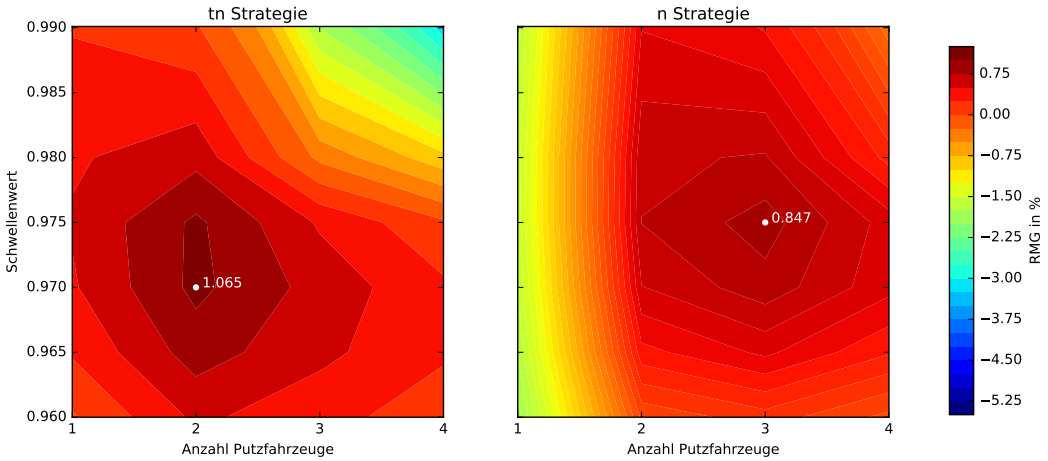


Abbildung 7.1.2: Der relative Mehrgewinn für die Strategien n und tn auf den 20 Validierungsjahren der künstlich erzeugten Wetterdaten für eine unterschiedliche Anzahl an Putzfahrzeugen.

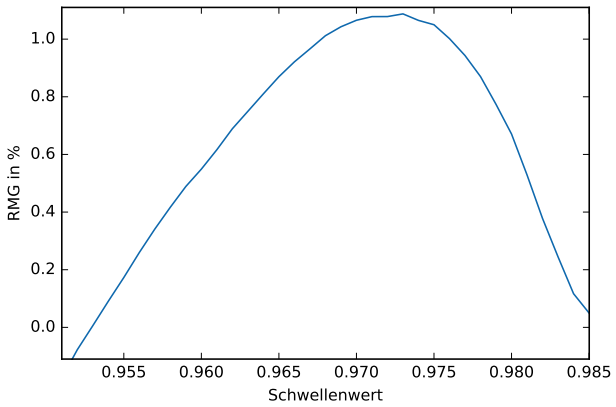


Abbildung 7.1.3: Der relative Mehrgewinn für die Strategien tn mit zwei Putzfahrzeugen auf den 20 Validierungsjahren der künstlich erzeugten Wetterdaten.

erhält die Feldsauberkeit als einzige Eingabe und lediglich die binäre Entscheidung mit allen Einheiten zwei Schichten putzen oder nicht putzen.[33] Der Agent wird wie gewohnt auf dem Trainingsdatensatz trainiert und alle 15 Epochen auf dem Validierungsdatensatz validiert. Wenn der Agent *austrainiert* nach der Definition in Kapitel 5.3 ist, wird der Agent mit dem besten Validierungsergebnis gewählt. Es ist anzumerken, dass der Agent nur durch diese letzte Auswahl direkt auf den Validierungsdatensatz angepasst wird. Der Schwellenwert Algorithmus wird direkt auf dem Validierungsdatensatz optimiert. Trotzdem findet der Reinforcement Learning Algorithmus eine Strategie mit der ein RMG von 1,0867% erreicht wird. Das heißt, die Lösung ist nur um etwa $8 \cdot 10^{-4}$% schlechter als die optimale

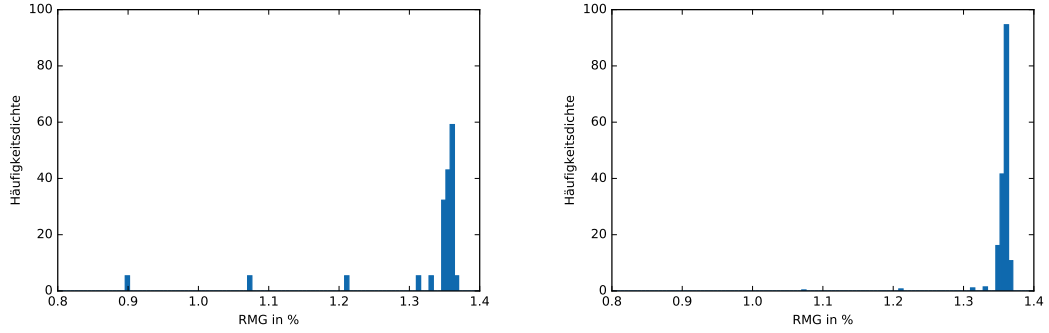
[33]Die genaue Beschreibung aller genutzten Netzarchitekturen findet sich in Anhang D.5.

gefundene Lösung. Daraus lässt sich schließen, dass der Algorithmus zumindest für solche reduzierte Probleme eine nahezu optimale Lösung findet. Die Hoffnung ist, dass dies auch für komplexere Probleme zutrifft. Allerdings lässt sich hier die optimale Lösung nicht bestimmen.

7.1.2. Robustheit des Algorithmus

Nachdem gezeigt wurde, dass der Algorithmus zumindest auf leichten Problemen eine nahezu optimale Lösung findet, soll im folgenden Abschnitt die Robustheit des Algorithmus überprüft werden. An mehreren Stellen des Algorithmus spielt der Zufall eine Rolle. Es werden die Gewichte zufällig initialisiert und die Auswahl der Aktionen während des Trainings werden nach bestimmten Wahrscheinlichkeiten zufällig gewählt. Dadurch ändern sich die Trajektorien und somit auch der Teil des Problems, der exploriert wird. Daher müsste für jede Einstellung, die getestet werden soll, eine ausreichend große Stichprobe gezogen werden, um ein repräsentatives Ergebnis zu erhalten. Allerdings ist dies aufgrund der langen Laufzeiten nicht möglich. Exemplarisch überprüfen wir die Robustheit des Algorithmus an einem spezifischen Problem: Dafür betrachten wir den Fall, dass der Agent als Eingabe die durchschnittliche Feldsauberkeit und eine Vorhersage von zwei Tagen erhält. Das heißt, für den kommenden und den darauffolgenden Tag erhält der Agent die Information, ob die Verschmutzungsrate hoch $\mathrm{SR} < -0{,}02$, normal $\mathrm{SR} \in [-0{,}02; 0)$ oder positiv $\mathrm{SR} > 0$ ist und zu welcher Klasse der Tag gehört.[34] Mit diesen Einstellungen wurde ein Agent 31 mal austrainiert. Die Ergebnisse sind in Abbildung 7.1.4 (a) aufgeführt. 26 von 31 Ergebnisse liegen in einem schmalen Bereich von $1{,}357 \pm 0{,}011$ % RMG. Es gibt drei deutliche und zwei weniger deutliche Ausreißer nach unten. Wahrscheinlich sind diese Verfahren zu einem lokalen Minimum konvergiert. Der Mittelwert der Ergebnisse ist 1,326% mit einer empirischen Standardabweichung von 0,096% RMG. Das spricht dafür, dass das Verfahren recht stabil ist. Allerdings sind ungefähr 16% der Fälle deutliche Ausreißer nach unten.
Um zu verhindern, dass in der weiteren Analyse diese Ausreißer gewertet werden, wird im Folgenden jede Ausführung doppelt ausgeführt. Als abschließendes Ergebnis wird dann die bessere der beiden Strategien gewählt. Dies hat gegenüber dem Mittelwert den Vorteil, dass eine Strategie in einem lokalen Minimum nur gewertet wird, wenn beide ausgeführten Strategien in einem lokalen Minimum enden. Zieht man aus den 31 Durchläufen alle möglichen zweier Kombinationen und wertet je den besseren, dann erhält man einen Mittelwert von 1,357% und eine empirische Standardabweichung von 0,017%; siehe Abbildung 7.1.4 (b). Das heißt, durch dieses Vorgehen

[34] Für die genaue Beschreibung der Architektur siehe wieder Anhang D.5.



(a) Einzelergebnisse

(b) Bester Durchlauf von zwei

Abbildung 7.1.4: Die Häufigkeitsdichte für die verschiedenen relativen Mehrgewinne. (a) zeigt die Einzelergebnisse. (b) zeigt die Ergebnisse mit dem Verfahren zwei Durchläufe zu ziehen und den besseren der beiden zu wählen.

konnte auf der vorliegenden Stichprobe die empirische Standardabweichung um einen Faktor 0,18 verringert werden. Daher wird im Folgenden dieses Vorgehen gewählt: Jede Strategie die getestet wird, wird zweimal unter identischen Bedingungen, aber mit unterschiedlichen zufälligen Initialisierungen durchgeführt. Als Ergebnis wird nur das Resultat der besseren Ausführung gewertet.

7.1.3. Verhalten des Algorithmus

Es soll exemplarisch ein Trainingsdurchlauf aus den oben aufgeführten 31 Durchläufen betrachtet werden, um einen Einblick zu erhalten, wie die gelernten Strategie funktioniert. So hat auch der hier betrachtete Agent die Möglichkeit aus null bis zwei Schichten pro Tag und Nacht zu wählen. Als Eingabe erhält der Agent sowohl die Verschmutzungsrate und Klasse des aktuellen und folgenden Tages, als auch die aktuelle durchschnittliche Feldsauberkeit. Der gewählte Trainingsverlauf ist in Abbildung 7.1.5 dargestellt. Es sind deutlich zwei Plateaus in der Lernkurve erkennbar, in denen sich der Algorithmus wenig oder gar nicht verbessert. Das erste Plateau befindet sich ungefähr bei Validierungsschritt zwei bis zwölf und das zweite bei 26 bis 42. Zwei so deutliche Plateaus in der Lernkurve sind eher untypisch.[35] Dieser spezielle Durchlauf wurde gewählt, da sich der Algorithmus anhand dieser Plateaus gut untersuchen lässt. Dafür betrachten wir den Agenten zu drei verschiedenen Trainingszeitpunkten: in dem ersten Plateau (Validierungsschritt 7), zweiten Plateau (Validierungsschritt 33) und den besten Durchlauf (Validierungsschritt 74); siehe dazu die Markierungen in Abbildung 7.1.5.

[35] Vergleiche mit zehn zufällig ausgewählten Lernkurven im Anhang D.4.

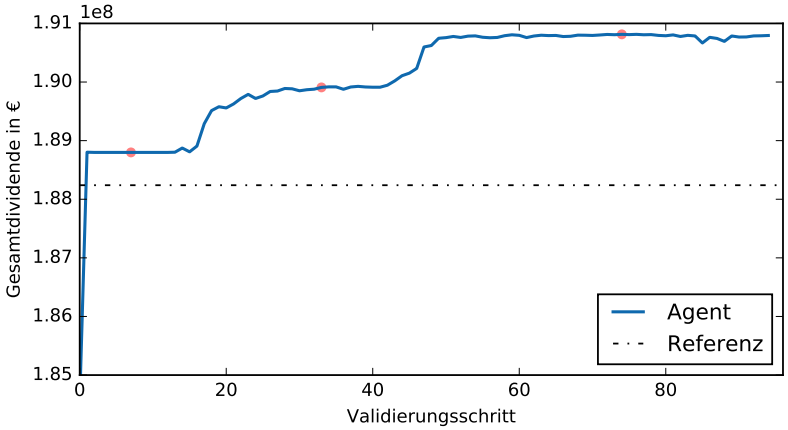


Abbildung 7.1.5: Validierungsergebnisse während des Trainings. Die Vaidierungsschritte 7, 33 und 74, welche genauer betrachtet werden, sind hier rot markiert.

Die Entscheidungskriterien der trainierten Agenten lassen sich schwer einsehen, da diese in einem künstlichen neuronalen Netz gespeichert sind. Am sinnvollsten ist es, die Entscheidung des Agenten für verschiedenen Zuständen zu beobachten. Um die Entscheidungen auf realistischen Zuständen zu untersuchen, betrachten wir die Entscheidungen auf dem Validierungsdatensatz. Dieser besteht aus 20 Jahren mit je 365 Tagen, das heißt aus 7300 Tagen und aus ebenso vielen Entscheidungen für die Putzaktivität. Es sei angemerkt, dass die Zustände, die in dem Validierungsdatensatz erreicht werden, auch von der gewählten Strategie abhängen. So würde eine Strategie, die jeden Tag mit vier Schichten putzt mit einer hohen Feldsauberkeit konfrontiert werden.

In dem ersten Plateau, Validierungsschritt 7, erzielt der Algorithmus einen relativen

Tabelle 7.1.6: Vergleich der Agenten zu verschiedenen Validierungsschritten. Dafür werden mehrere Kennwerte auf dem Validierungsdatensatz gelistet. SpT steht für Schichten pro Tag und ∅-Cl.[%] bezeichnet die durchschnittliche Feldsauberkeit über alle Tage des Datensatzes.

Val-Schritt	Gesamtdividende [€]	RMG [%]	∅-Cl.[%]	SpT
7	$1{,}888 \cdot 10^8$	0,30	96,42	2
33	$1{,}899 \cdot 10^8$	0,89	96,97	2,32
74	$1{,}908 \cdot 10^8$	1,37	96,99	1,82

Mehrgewinn von 0,30% gegenüber der Referenzstrategie; siehe Tabelle 7.1.6. Die Referenzstrategie besteht wie in Kapitel 5.2 beschrieben daraus, dass konstant Tag und Nacht mit einer Putzeinheit gereinigt wird. Im Gegensatz dazu fährt der Agent an allen Tagen des Validierungsdatensatzes zwei Schichten nachts und keine tagsüber. Die Verbesserung gegenüber der Referenzstrategie ergibt sich daraus, dass der Algorithmus gelernt hat nicht tagsüber zu

putzen. So müssen die Parabolspiegel während des Putzens nicht defokussiert werden. Dies wiegt die niedrigeren Anschaffungskosten für nur eine Reinigungseinheit der Referenzstrategie auf. Allerdings passt sich der Algorithmus nicht auf die Wetterbedingungen an, sondern fährt eine konstante Strategie. Ein solcher Agent erreicht eine durchschnittliche Feldsauberkeit von 96,42% auf den 20 Validierungsjahren.

Auf dem zweiten Plateau, Validierungsschritt 33, hat sich die Strategie deutlich verbessert und erzielt nun einen relativen Mehrgewinn von 0,89%. Der Agent steigert die Feldsauberkeit um circa 0,5% auf 96,97%. Dies wird erreicht indem der Agent an manchen Tagen vier Schichten fährt. Das bedeutet er nutzt die volle Kapazität der zwei Putzfahrzeuge. Er fährt an etwa 6100 Tagen zwei und an den restlichen circa 1200 Tagen vier Schichten. Die Strategie kommt so auf eine durchschnittliche Anzahl von 2,3 Schichten pro Tag. Der Agent putzt vor allem in vier Schichten, wenn die Durchschnittliche Feldsauberkeit niedrig ist. So schafft es der Agent nach hohen Verschmutzungsraten schnell wieder in akzeptable Bereiche der Feldsauberkeit zu gelangen. So ist der größte Anteil der Tage des Validierungsdatensatz im Bereich zwischen 0,96 und 0,98; vergleiche Abbildung 7.1.7.

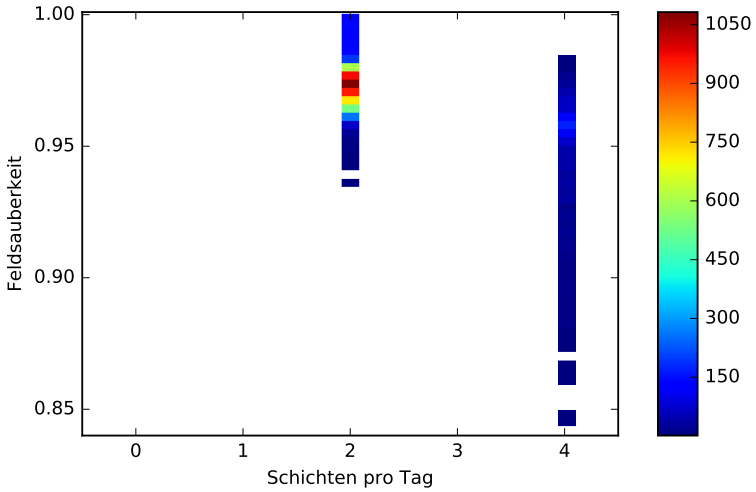


Abbildung 7.1.7: Schichten auf dem Validierungsdatensatz für den Agent des Validierungsschritt 33, aufgetragen gegen die Feldsauberkeit. Die Farbskala repräsentiert die Häufigkeit im Validierungsdatensatz.

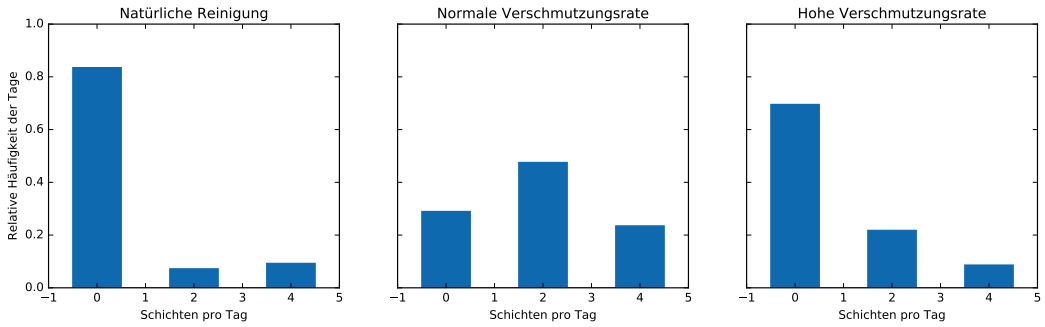


Abbildung 7.1.8: Die relative Häufigkeit der Anzahl der Schichten, aufgeschlüsselt für die verschiedenen Verschmutzungsraten des folgenden Tages. Auf der linken Seite ist die relative Häufigkeit der Schichten für die Tage dargestellt, an denen für den folgenden Tag eine natürliche Reinigung vorhergesagt wurde.

Der beste Agent (Validierungsschritt 74) erreicht eine nochmalige Verbesserung des relativen Mehrgewinn um circa 0,5% auf 1,37%. Dies erreicht er indem die Effizienz des Reinigens deutlich erhöht wird. So verbleibt die Feldsauberkeit gegenüber Validierungsschritt 33 auf einem ähnlichen Niveau, der Agent benötigt allerdings nur 1,82 Schichten pro Tag. Das heißt die durchschnittliche Anzahl der Schichten ist um eine halbe Schicht pro Tag niedriger als im Validierungsschritt 33. Dies wird zum einen dadurch erreicht, dass bei hoher Feldsauberkeit weniger geputzt wird; siehe Abbildung 7.1.9. Zum anderen wird die Vorhersage der kommenden Verschmutzungsraten genutzt, wie in Abbildung 7.1.8 zu sehen. So spart der Agent Säuberungskosten, wenn für den folgenden Tag eine natürliche Reinigung oder eine hohe Verschmutzungsrate vorhergesagt wird. Das zeigt, dass der Agent gelernt hat, sowohl die aktuelle, als auch die vorhergesagte Verschmutzungsrate sinnvoll zu nutzen und darauf zu reagieren.

Vergleicht man die Strategien indem man die Feldsauberkeit und die Aktionen an einem Beispieljahr betrachtet, wird deutlich, dass die Feldsauberkeit des ersten Agenten sehr ähnlich ist zu der Referenzstrategie. Da die Strategie auch jeden Tag zwei Schichten fährt, liegt der einzige Unterschied darin, dass bei der gelernten Strategie nur nachts geputzt wird. Die Verschmutzung der geputzten Spiegel weicht nur minimal ab, vergleiche Gleichung (4.3.2). Die Strategien des Validierungsschritt 33 beziehungsweise 74 erreichen im Gegensatz zu der Referenz nach starken Verschmutzungsereignissen schnell wieder ein akzeptables Niveau der Feldsauberkeit um die 97%; siehe die detaillierte Darstellung eines Beispieljahres für die Strategien des Validierungsschritt 33 und 74 in Abbildung 7.1.10. Allerdings spart nur der beste Agent Putzkosten in Erwartung von hoher Verschmutzung oder natürlicher Säuberung. Gut zu sehen ist dies in Abbildung 7.1.10 für natürliche

Reinigung zum Beispiel die Tage des Jahres 74, 95 und 114, beziehungsweise für starke Verschmutzung die Tage 106 oder 125. In der gleichen Abbildung wird auch deutlich, dass der austrainierte Agent bei guter durchschnittlicher Feldsauberkeit eine etwas niedrigere Sauberkeit in Kauf nimmt, um dafür Putzkosten zu sparen, vergleiche Tag des Jahres 34-60.

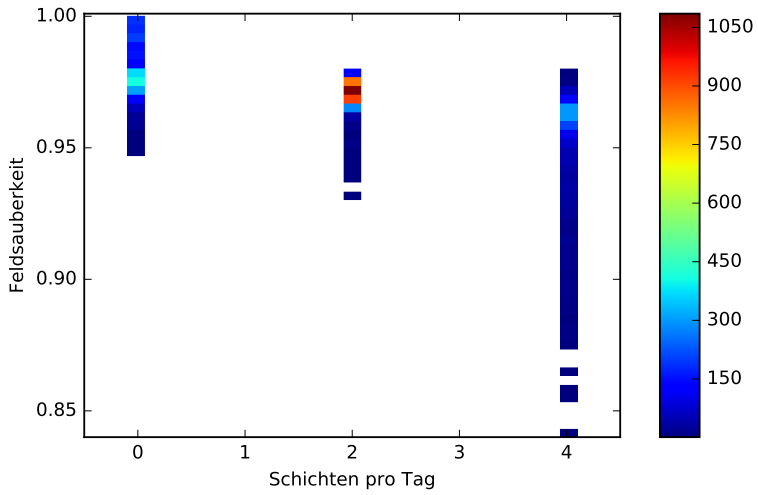


Abbildung 7.1.9: Schichten auf dem Validierungsdatensatz für den Agent des Validierungsschritt 74, aufgetragen gegen die Feldsauberkeit. Die Farbskala repräsentiert die Häufigkeit im Validierungsdatensatz.

Der Vergleich der Feldsauberkeiten macht einen Fehler in den Putzannahmen sichtbar. So setzt das Putzen die geputzten Spiegel, wie in Kapitel 4.3 besprochen, auf 0,986. Allerdings auch dann, wenn der Spiegel sauberer als 0,986 ist. So wird die Sauberkeit bei sehr hoher Sauberkeit durch das Putzen verringert. Daher fällt bei sehr geringer Verschmutzung die Sauberkeit schneller wenn geputzt wird. Dies wird in Abbildung 7.1.10 an Tag des Jahres 0 deutlich, da alle Spiegel mit der Ausgangssauberkeit von 100% starten oder nach den zwei aufeinanderfolgenden natürlichen Reinigungen der Tage 253-254. An diesen Tagen fällt die Sauberkeit bei der Referenzstrategie und der Strategie zu Validierungsschritt 33 schneller als bei der austrainierten Strategie, die bei so hoher Sauberkeiten nicht putzt. Bei den folgenden Berechnungen ist weiter mit diesem Fehler gerechnet worden. Man kann es als eine Möglichkeit sehen Putzen bei hoher Sauberkeit künstlich zu penaltisieren. Außerdem ist der Einfluss dieses Fehlers nicht sehr groß: Berichtigt man den Referenzagenten, sodass bei einer Feldsauberkeit von über 0,986 das Putzen keinen Einfluss hat, führt dies zu einer Verbesserung der Gesamtdividenden um lediglich 0,05%.

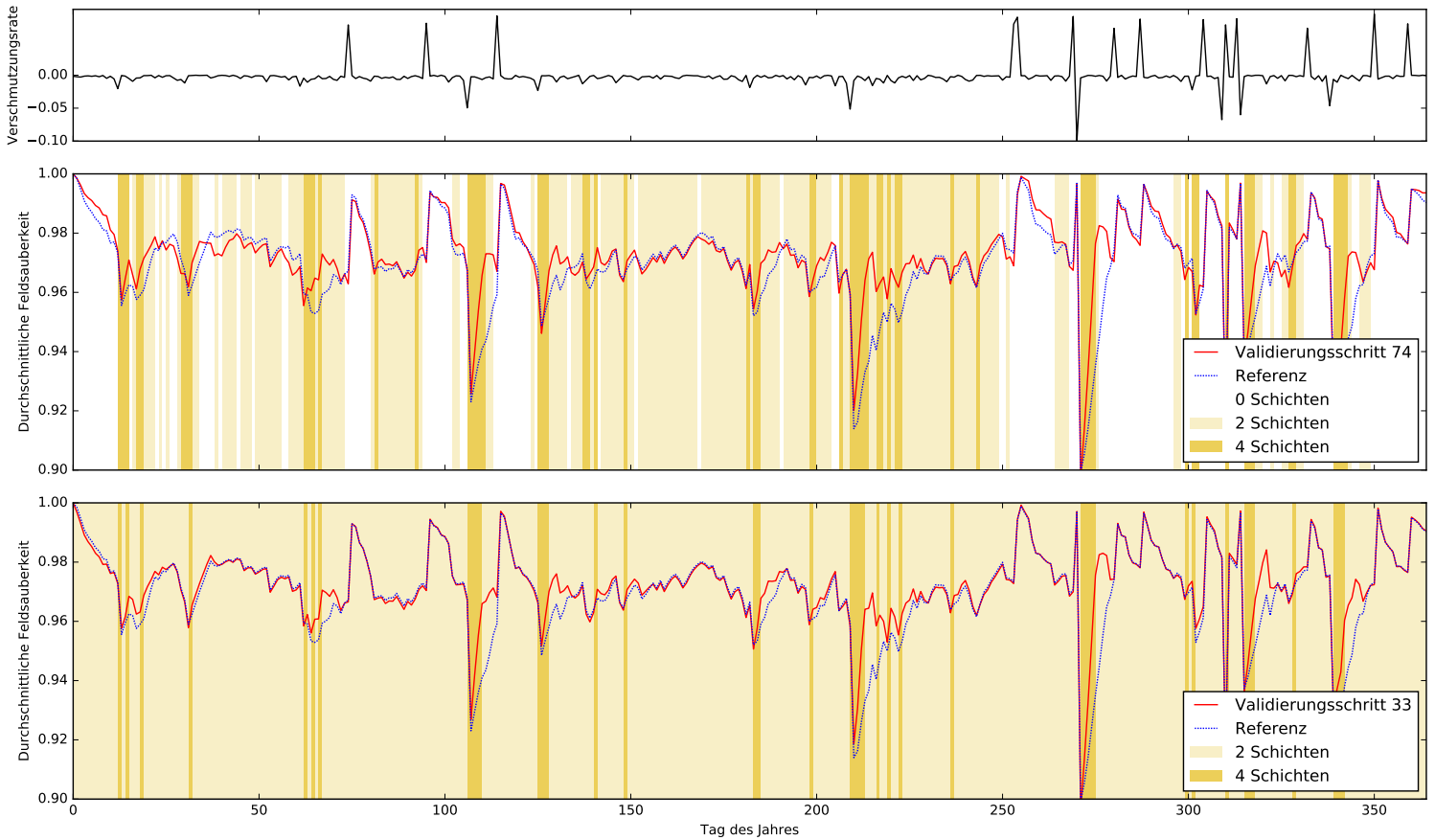


Abbildung 7.1.10: Vergleich der Strategien auf einem Beispieljahr des Validierungsdatensatzes. Oben ist die Verschmutzungsrate des spezifischen Jahres dargestellt. Die Verschmutzungsrate eines natürlichen Reinigungsevents hängt, wie in Kapitel 5.1 beschrieben, von der aktuellen Sauberkeit des Loops ab. Daher ist hier beispielhaft die positive Verschmutzungsrate zu einer Sauberkeit von 90% dargestellt. Darunter ist das Verhalten der Agenten zu Validierungsschritt 74 (Mitte) beziehungsweise Validierungsschritt 33 (unten) gezeigt. Die Feldsauberkeit wird in beiden Fällen mit der konstanten Referenzstrategie verglichen. Die Hintergrundfarben signalisieren die Aktion an dem dazugehörigen Tag.

7.2. Auswertung

Nachdem im vorhergehenden Kapitel der Algorithmus und das gelernte Verhalten untersucht wurden, soll nun mit Hilfe des Algorithmus verschiedene Fragen beantwortet werden. Zunächst werden verschiedene Vorhersagehorizonte für die Verschmutzungsrate untersucht. Dabei soll evaluiert werden, wie viele Vorhersagetage für eine optimale Putzsteuerung nötig sind. Im Folgenden soll untersucht werden, wie der Agent auf eine veränderte Umgebung reagiert. Es wird einerseits der Wasserpreis und somit die Putzkosten erhöht und andererseits werden mehrere starke Verschmutzungsereignisse in die Daten eingefügt.

7.2.1. Vergleich mehrerer Vorhersagehorizonte

Mit Hilfe des in dieser Arbeit erstellten Algorithmus kann untersucht werden welchen Einfluss die Verfügbarkeit verschiedener Eingabeparameter auf die Qualität der Strategie hat. So wurde im Rahmen dieser Arbeit analysiert, inwiefern die Vorhersage der Verschmutzungsrate einen Vorteil für die Putzstrategie bringt. Diese Untersuchungen sind wichtig, da momentan noch keine genauen Vorhersagen für die Verschmutzungsraten möglich sind. Bevor Anstrengungen unternommen werden um Vorhersagen zu erstellen, sollte Abgeschätzt werden welche Verbesserung diese für die Anpassung der Putzstrategien bringt. Die Ergebnisse werden in [Wol+19] veröffentlicht.

Es wurden Agenten mit fünf verschiedenen Vorhersagehorizonten trainiert. Für jeden Vorhersagetag sind die Klasse und die Verschmutzungsrate für den Agenten verfügbar. Die Verschmutzungsrate wurde allerdings nicht genau vorhergesagt, sondern in drei Klassen eingeteilt – starke ($SR < -0{,}02$) und normale ($SR \in [-0{,}02; 0)$) Verschmutzung und natürliche Reinigung ($SR > 0$). Das ist sinnvoll, da eine exakte Vorhersage der Verschmutzungsrate unrealistisch ist. Außerdem hat jeder Agent Zugriff auf die aktuelle Feldsauberkeit.

Werden die Ergebnisse der verschiedenen Vorhersagehorizonte verglichen, kann abgeschätzt werden, inwiefern die Verschmutzungsvorhersage eine Verbesserung der Putzstrategie möglich macht. Hat der Agent lediglich Zugriff auf die aktuelle Feldsauberkeit, dass heißt es existiert keine Verschmutzungsvorhersage, wird ein RMG von 1,28% erreicht. Wenn der Agent Zugriff auf die Vorhersage des nächsten Tages hat, steigt der RMG auf 1,33% und bei einer Vorhersage von zwei Tagen auf 1,36%. Bei einer weiteren Vorhersage ist keine signifikante Verbesserung des RMG mehr möglich; vergleiche Tabelle 7.2.1.

Bei der genauen Betrachtung des Agenten mit einem Vorhersagehorizont von zwei Tagen wurde in [Wol+19] sowie in Kapitel 7.1.3 deutlich, dass eine Strategie zur Verbesserung der Gesamtdivi-

denden, das Sparen von Putzkosten in Erwartung einer natürlichen Reinigung ist. Eine natürliche Reinigung bedeutet in unserem Modell eine tägliche Niederschlagmenge von über 5mm. Da die Regenvorhersage losgelöst von der Verschmutzungsrate ist und es gängige Praxis ist den Regen Vorherzusagen, soll im Folgenden noch untersucht werden, wie gut der Putzalgorithmus werden kann, wenn nur Regen vorhergesagt wird. Dafür wird ein Agent trainiert, der als Eingabe die aktuelle Feldsauberkeit und die jeweils binäre Vorhersage für die nächsten zwei Tage erhält, ob ein natürliches Reinigungsereignis eintritt. Der Vorhersagehorizont von zwei Tagen wurde gewählt, da dass der Punkt war, ab dem sich der Mehrgewinn stabilisiert hat und eine weitere Vorhersage keine signifikante Verbesserung mehr erzielen konnte.
Ein solcher Agent erzielt einen RMG von 1,31%. Dies ist besser als der relative Mehrgewinn von 1,28%, der nur durch die Feldsauberkeit erzielt werden kann. Es lohnt sich also die Vorhersage des Regens in die Strategie einzubinden. Allerdings erzielt ein Agent mit dem gleichen Vorhersagehorizont, welcher auch noch Zugriff auf die Verschmutzungsrate hat, einen relativen Mehrgewinn von 1,36%. Das heißt, eine Vorhersage der Verschmutzungsraten kann die Strategie noch einmal signifikant verbessern. Für die vollständigen Daten siehe Tabelle C.1.1.

Tabelle 7.2.1: Die relative Mehrgewinn gegenüber der Bezugsstrategie (RMG) für verschiedene Vorhersagehorizonte in %. Die Werte sind auf die zweite Nachkommastelle gerundet. Jedes Setting wurde zwei mal trainiert. Der höhere Wert wurde als Ergebnis betrachtet. Der Vollständigkeit halber finden sich beide Ergebnisse in Tabelle C.1.1.

Horizont	RMG [%]
0	1,28
1	1,33
2	1,36
3	1,37
6	1,36
2 (Nur natürliche Reinigung)	1,31

7.2.2. Erhöhte Verschmutzungsraten

In der Region des Nahen Osten und Nord Afrika (MENA-Region), welche für CSP aufgrund der hohen Einstrahlung interessant ist, kommt es deutlich häufiger zu Sandstürmen als in Südspanien [Mah+07]. Daher ist es sinnvoll zu prüfen, wie sich die Strategien verhalten, wenn mehrere Ereignisse mit hoher Verschmutzungsrate über die Jahre verteilt werden. Dies wurde auch für die Schwellenwert basierten Strategien in [Wol+18] getestet. Dafür werden auf jedes Jahr sowohl im Trainings- als auch im Validierungsdatensatz zufällig zehn *Sandstürme* mit einer Verschmutzungsrate von -0,1 verteilt. Daraufhin wird ein Agent, mit Zugriff auf die durchschnittliche

Feldsauberkeit und einer Verschmutzungsratenvorhersage von zwei Tagen, wie im vorherigen Kapitel 7.2.1, auf den veränderten Datensatz trainiert. Dieser Agent wird Agent_s genannt. Als Vergleich wird ein auf dem normalen Datensatz trainierte Agent (Agent_n) genutzt, der Zugriff auf die gleichen Parameter hat.[36]

Zunächst wird deutlich, dass bei höherer Verschmutzung der relative Mehrgewinn einer adaptiven Strategie deutlich höher liegt. So erreicht der auf dem modifizierten Datensatz trainierte Agent einen RMG von 2,51%. Auf dem normalen Datensatz erreicht ein hier trainierter Agent einen RMG von 1,36%; vergleiche Tabelle 7.2.2. Dieses Ergebnis deckt sich mit den Ergebnissen in [Wol+18]. Auch dort wurde herausgefunden, dass bei häufigeren starken Verschmutzungsereignissen ein deutlich höherer RMG erzielt werden kann.

Tabelle 7.2.2: Vergleich der Agenten Agent_s und Agent_n. Ausgeführt auf dem Validierungsdatensatz einmal mit und einmal ohne das künstliche Hinzufügen von Verschmutzungsereignissen. D_tot sind die Gesamtdividenden über die Laufzeit des Kraftwerkes. SpT steht für Schichten pro Tag und ∅-Cl.[%] bezeichnet die durchschnittliche Feldsauberkeit über alle Tage des Validierungsdatensatzes.

Normaler Datensatz

Strategie	D_tot [€]	RMG[%]	∅-Cl.[%]	SpT
Referenz	$1{,}882 \cdot 10^8$	0	96,86	2
Agent_n	$1{,}908 \cdot 10^8$	1,36	97,38	1,87
Agent_s	$1{,}907 \cdot 10^8$	1,34	97,35	1,83

Datensatz mit Sandstürmen

Strategie	D_tot [€]	RMG[%]	∅-Cl.[%]	SpT
Referenz	$1{,}841 \cdot 10^8$	0	95,88	2
Agent_n	$1{,}888 \cdot 10^8$	2,52	96,86	2,08
Agent_s	$1{,}888 \cdot 10^8$	2,51	96,95	2,04

Ebenso interessant ist, wie gut die Agenten auf den jeweils anderen Datensätzen eingesetzt werden können. Es wird deutlich, dass der auf dem normalen Datensatz trainiert Agent_n auf beiden Datensätzen etwas besser abschneidet. Allerdings ist der Unterschied auf dem Datensatz mit den künstlichen Sandstürmen nur minimal. So ist der RMG des Agent_n um unter 0,005% RMG höher als der von Agent_s.[37] Auf dem Standarddatensatz ist der Unterschied mit 0,02% RMG deutlicher, aber immer noch klein. Das spricht dafür, dass sich die Strategien nicht so stark unterscheiden. Es ist sinnvoll, dass das Prinzip der Steuerung ähnlich bleibt. So ergibt es auf beiden Datensätzen Sinn Wasser zu sparen, wenn möglich; zum Beispiel vor Reinigungsereignissen oder hohen Verschmutzungsraten. Auch bei niedriger Feldsauberkeit schnell wieder auf ein

[36] Für die genaue Beschreibung der Parameter siehe auch hier Anhang D.5.

[37] Die ungerundeten RMG Werte sind 2,519128…% für Agent_n beziehungsweise 2,514260…% für Agent_s.

akzeptables Niveau zu gelangen ist sinnvoll, unabhängig davon ob dies häufiger oder seltener pro Jahr der Fall ist.

7.2.3. Erhöhter Wasserpreis

Mit Hilfe des Algorithmus kann beobachtet werden, wie die Putzstrategie auf verschiedene Voraussetzungen reagiert. So sollen in diesem Kapitel die Wasserkosten von den für Südspanien angenommenen 0,39 €/m^3 auf 3 €/m^3 erhöht werden. Zum einen gibt es Regionen bei denen der Preis für demineralisiertes Wasser deutlich über den 0,39 €/m^3 liegt. Zum anderen kann das Erhöhen des Wasserpreises auch genutzt werden, um den Wasserverbrauch künstlich zu penaltisieren, damit eine Strategie gefunden wird, welche weniger Wasser benötigt. Dies ist wünschenswert, um die Wasserknappheit nicht weiter zu verschlimmern, welche in vielen Regionen vorherrscht, die für CSP Kraftwerke geeignet sind.
Das Erhöhen der Wasserkosten um 2,61 €/m^3 ist äquivalent dazu, die Putzkosten pro Loop um 2,61 € zu erhöhen, da kein Unterschied gemacht wird wo die Kosten anfallen. Es werden also die Putzkosten pro Loop von 32,22 €/Loop auf 34,83 €/Loop, um ungefähr 8%, erhöht; vergleiche hierzu Kapitel 4.4. Der Agent, der hier betrachtet werden soll, verfügt über einen Vorhersagehorizont von zwei Tagen. Das bedeutet, er erhält einerseits die aktuelle Feldsauberkeit und andererseits sowohl die Klasse als auch die Verschmutzungsraten-Vorhersage der folgenden zwei Tage. Er wird mit den erhöhten Wasserpreisen trainiert. Als Vergleich wird, wie im vorherigen Kapitel, der Agent mit einem Vorhersagehorizont von zwei Tagen aus Kapitel 7.2.1 gewählt. Das heißt, die Agenten haben exakt die gleichen Eingabeparameter. Der einzige Unterschied ist, dass der Vergleichsagent mit den normalen Wasserkosten trainiert wurde. Wir nennen den Agenten, der auf hohen Wasserkosten trainiert wurde Agent_w und den Agent, der auf normalen Wasserkosten trainiert wurde Agent_n.
Beide Agenten werden zweimal auf dem Validierungsdatensatz getestet: Einmal mit normalen und einmal mit erhöhten Wasserkosten. Der relative Mehrgewinn ist immer gegenüber der konstanten Referenz unter den jeweiligen Wasserkosten angegeben. Agent_w findet eine Strategie, welche im Schnitt auf dem gesamten Validierungsdatensatz nur 1,75 Schichten pro Tag fährt. Agent_n fährt 1,87 Schichten pro Tag. Dabei ist die durchschnittliche Feldsauberkeit um lediglich 0,1% höher. Da die Schichten pro Tag direkt proportional zu dem Putzwasserverbrauch sind, bedeutet dies, dass Agent_n 7% mehr Wasser verbraucht. Der durchschnittliche Wasserverbrauch der Referenzstrategie ist sogar um mehr als 14% höher, wobei die durchschnittliche Feldsauberkeit niedriger ist; siehe Tabelle 7.2.3.

Tabelle 7.2.3: Vergleich der Agenten Agent_w und Agent_n. Ausgeführt auf dem Validierungsdatensatz einmal mit hohen und einmal mit normalen Wasserkosten. D_tot sind die Gesamtdividenden über die Laufzeit des Kraftwerkes. SpT steht für Schichten pro Tag und ∅-Cl.[%] bezeichnet die durchschnittliche Feldsauberkeit über alle Tage des Datensatzes. Es sei angemerkt, dass die Schichten pro Tag direkt proportional zu dem Wasserverbrauch während des Putzvorgangs sind.

Bei hohen Wasserkosten

Strategie	D_tot [€]	RMG[%]	∅-Cl.[%]	SpT
Referenz	$1{,}879 \cdot 10^8$	0	96,86	2
Agent_w	$1{,}905 \cdot 10^8$	1,37	97,29	1,75
Agent_n	$1{,}905 \cdot 10^8$	1,37	97,38	1,87

Bei normalen Wasserkosten

Strategie	D_tot [€]	RMG[%]	∅-Cl.[%]	SpT
Referenz	$1{,}882 \cdot 10^8$	0	96,86	2
Agent_w	$1{,}908 \cdot 10^8$	1,35	97,29	1,75
Agent_n	$1{,}908 \cdot 10^8$	1,36	97,38	1,87

Die Gesamtdividenden sind bei hohen Wasserkosten offensichtlich niedriger als bei normalen Wasserkosten. Im Gegensatz dazu steigt RMG mit höheren Wasserkosten. Das liegt daran, dass die trainierten Agenten weniger Schichten fahren als die Referenzstrategie und daher auch die Putzkosten weniger ins Gewicht fallen. Es ist davon auszugehen, dass bei einer weiteren Erhöhung der Wasserkosten oder einem anderen Teil der Putzkosten der RMG weiter stiege. Dann würden sich auch die wenigeren Schichten pro Tag bei Agent_w mehr auszahlen. Bei Wasserkosten von $3€/\text{m}^3$ sind die beiden trainierten Agenten bei hohen Wasserkosten gleichauf mit einem RMG von 1,37%. Bei normalen Wasserkosten schneidet Agent_w minimal schlechter ab als der auch auf normalen Wasserkosten trainierte Agent. Diese sehr geringe Verschlechterung zeigt, dass die auf den hohen Wasserkosten gelernte Strategie auch gut unter *normalen* Bedingungen funktioniert. Es kann also ein sinnvolles Vorgehen sein, die Putzkosten künstlich zu penaltisieren, um eine wassersparende Strategie zu finden.

7.3. Diskussion

In diesem Kapitel sollen die Schwierigkeiten, die im Verlauf der Arbeit aufgetreten sind, diskutiert werden. Zunächst wurde, wie in Kapitel 7.2.1 beschrieben, ein Fehler in der Definition der Putzstrategien gemacht. So wird die Sauberkeit der Spiegel auf 0,986 gesetzt, auch wenn diese vor dem Putzen höher war. Dies geschieht allerdings sehr selten, da die Spiegel zyklisch geputzt werden, sodass immer die aktuell am wenigsten sauberen Spiegel

geputzt werden. Dass der Einfluss dieses Fehlers klein ist, sieht man auch daran, dass sich die Gesamtdividenden der Referenzstrategie nach der Behebung des Fehlers nur um 0,05% steigern. Außerdem ist dieser Fehler für alle Strategien gleich und die austrainierten Agenten haben ohnehin gelernt bei so hoher Sauberkeit nicht zu putzen. Daher macht dieser Fehler für den Vergleich unter den austrainierten Agenten keinen Unterschied. Er sorgt lediglich für eine leichte Verbesserung der Agenten gegenüber der Referenz.

Außerdem wurde in dieser Arbeit darauf verzichtet einen weiteren dritten unabhängigen Testdatensatz einzusetzen. Dieser wird normalerweise genutzt, um eine Überanpassung an den Validierungsdatensatz zu verhindern. Diese Anpassung geschieht hauptsächlich bei der Hyperparameter-Optimierung[38]. Dabei wird ein Agent mit verschiedenen Einstellungen mehrmals ausgeführt und aufgrund der Ergebnisse auf dem Validierungsdatensatz entschieden, welche Einstellung genutzt werden soll. So kann es sein, dass die Hyperparameter genau auf den speziellen Validierungsdatensatz angepasst werden und nicht gute generelle Ergebnisse liefern. Hier wurde auf diesen dritten Datensatz verzichtet, da die einzige Anpassung auf den Validierungsdatensatz die Auswahl des besten Agenten darstellt. So konnte Rechenzeit gespart werden, indem kein weiterer Datensatz ausgewertet werden musste.

Eine weitere mögliche Fehlerquelle sind die erzeugten Daten. Es wird nicht ausreichend genau untersucht, wie gut diese die Wirklichkeit abbilden.[39] Sollte der Algorithmus zur Optimierung eines realen Kraftwerkes genutzt werden, ist es sinnvoll diesen Zusammenhang genauer zu untersuchen. Falls möglich könnte man als dritten Testdatensatz einen Datensatz mit realen Daten nehmen. In dieser Arbeit war das Hauptziel der Konzeptnachweis, dafür ist es nicht so wichtig, wie gut die Daten die Realität abbilden, da hauptsächlich gezeigt werden sollte, dass der Algorithmus lernt und wie sich bei verschiedenen Einstellungen die Agenten unterscheiden. Es kann davon ausgegangen werden, dass dies ebenso auf realen Daten funktionieren würde. Außerdem wurde bei der genaueren Untersuchung des gelernten Verhaltens in Kapitel 7.1.3 deutlich, dass die Strategie nachvollziehbare Verhalten hervorgebracht hat. Daraus lässt sich vermuten, dass diese Strategien auch bei der Anwendung in der Realität sinnvoll wären. Bei der Untersuchung der erhöhten Verschmutzungsraten in Kapitel 7.2.2 haben wir gesehen, dass die gelernten Verhalten auf veränderten Daten ähnlich gut funktioniert haben. So haben

[38] Als *Hyperparamer* versteht man die Parameter und Einstellungen, die vor dem Training festgesetzt werden. Im Gegensatz dazu werden Schwellenwerten und Gewichten der Netze Parameter genannt, da diese während des Trainings angepasst werden. Zu den Hyperparametern zählen zum Beispiel der Aufbau des Netzes – wie Anzahl der Schichten und Neuronen – oder auch die Einstellung der Optimierungsverfahren – wie zum Beispiel die Schrittweite.

[39] Im Anhang D.2 wird auf die Analyse der Daten eingegangen.

der auf dem normalen Datensatz trainierte Agent und der auf dem veränderten Datensatz trainierte Agent auf beiden Datensätzen ähnlich gut funktioniert. Das bedeutet, dass das gelernte Verhalten auch unter veränderten Bedingungen wünschenswert ist.

Fazit und Ausblick

Die Anwendung von Reinforcement Learning Algorithmen auf die Putzsteuerung kann den Profit eines Kraftwerkes erhöhen. So kann ein Agent nur mit Hilfe der aktuellen mittleren Feldsauberkeit einen relativen Mehrgewinn von 1,28% gegenüber einer konstanten Referenzstrategie erwirtschaften. Ein einfacher Schwellenwert basierter Algorithmus erreicht lediglich 1,09% RMG. Nimmt man noch eine Verschmutzungsraten Vorhersage von drei Tagen hinzu, wird sogar ein RMG von 1,37% erreicht; vergleiche Kapitel 7.1.1 und 7.2.1. Diese Ergebnisse und die genauere Untersuchung des Verhaltens von Agenten in Kapitel 7.1.3 lassen darauf schließen, dass Reinforcement Learning Algorithmen für diese Problemstellung sinnvolle Lösungen finden.
Außerdem lässt sich mit dieser Methode gut abschätzen, wie wichtig die Verfügbarkeit verschiedener Parameter für die Anpassung der Putzstrategie ist. So deuten die Untersuchungen in Kapitel 7.2.1 darauf hin, dass eine Verschmutzungsvorhersage für mehr als zwei Tage die Putzstrategie nicht mehr signifikant verbessern kann. Auch wird deutlich, dass die Vorhersage von natürlichen Reinigungsereignissen zwar eine Verbesserung bringt, allerdings signifikant weniger als die zusätzliche Vorhersage der Verschmutzungsraten.
Eine weitere Anwendung, die dieser Algorithmus ermöglicht, ist die Untersuchung wie sich die Putzstrategien auf unterschiedliche Bedingungen anpassen. So wurde gezeigt, dass ein Agent der auf hohen Wasserkosten trainiert wird, eine Strategie findet, die weniger Wasser verbraucht. Ein solches Vorgehen kann eine Möglichkeit sein unerwünschtes Verhalten künstlich zu bestrafen, um eine Strategie zu finden, welche dieses Verhalten weniger zeigt. Insgesamt wurde gezeigt, dass für die durch Reinforcement Learning Methoden gefundenen Agenten, sowohl an Standorten mit höheren Wasserpreisen, als auch bei mehreren starken Verschmutzungsereignissen, ein noch höherer relativer Mehrgewinn zu erwarten ist.

Für die Laufzeit des Programms stellt die Dauer, die eine Ausführung von Greenius benötigt, den größten Anteil dar; vergleiche Kapitel 6. Daher wäre ein Ansatz zur Verbesserung dieses Programms, die angewandte Reinforcement Learning Methode durch ein Verfahren zu ersetzen, welches Stichproben effizienter[40] ist. Das bedeutet, dass die Auswertungen von Greenius effizienter genutzt würden, um die Strategie zu verbessern.

[40] Engl.: *sample-efficient*.

Dafür gibt es verschiedene Ansätze; siehe zum Beispiel [Gu+16] oder [Yu18]. Es müsste für die Anwendung getestet werden, welcher Ansatz hier am besten funktioniert. Dies wäre vor allem sinnvoll, wenn weitere ausführliche Tests gemacht werden sollen. Für die Optimierung der Putzstrategien in einem festgelegten Setting, zum Beispiel einem bestehenden Kraftwerk, ist die Laufzeit ausreichend kurz, da die Optimierung in diesem Fall nur einmal ausgeführt werden muss. Bei der Ausführung der Strategie müsste lediglich einmal pro Tag zu den aktuellen Wetterdaten das neuronale Netz ausgewertet werden. Dies geschieht innerhalb von Sekundenbruchteilen.

Eine weitere Möglichkeit die vorliegende Arbeit zu erweitern, wäre dem Agent noch weitere Entscheidungsmöglichkeiten zu geben. So haben Wolfertstetter et al. in [Wol+18] gezeigt, dass es bei hohen Verschmutzungsraten sinnvoll sein kann, nach extremen Verschmutzungsereignissen kurzfristig zusätzliche Putzeinheiten einzustellen, die die Spiegel manuell reinigen.

Außerdem könnte mit dem hier vorgestellten Algorithmus der Einfluss von weiteren Parametern auf die Putzstrategie untersucht werden. So könnte, wie in [Ba+17], der zeitlich schwankende Strompreis dem Agenten zur Steuerung übergeben werden. Dafür müsste allerdings getestet werden, wie ein schwankender Strompreis in Greenius realisiert werden kann.

Anhang

A. Notationsverzeichnis

In der folgenden Übersicht werden die gebräuchlichsten Zeichen und Variablen, die in dieser Arbeit verwendet werden, kurz erläutert.

CSP	Konzentrierende Solarkraft vom Englischen *concentrating solar power*
$\mathbb{E}_\pi$	Erwartungswert einer Zufallsvariable unter der Bedingung, dass der Agent der Strategie π folgt
log	Der natürliche Logarithmus
PV	Photovoltaik
PSA	Plataforma Solar de Almería
RMG	Relativer Mehrgewinn gegenüber der Referenzstrategie – Putzen in zwei Schichten pro Tag mit einem Putzfahrzeug
var-Klasse	variability-Klasse nach [SH+18]

B. Beweise

B.1. Strategie Gradienten Theorem

Theorem B.1.1 (Strategie Gradienten Theorem)**.** *(siehe 1.3.2).*
Für jede differenzierbare Strategie π_θ, jeden endlichen Horizont $H \in \mathbb{N}$ und für die Zielfunktion $J(\theta)$ wie in 1.3.1 definiert gilt:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \cdot R_t\right],$$

Beweis. Vergleiche [Abb17]. Für eine bessere Übersichtlichkeit, führen wir zunächst eine verkürzte Notation ein. So sei die Zufallsvariable

$$R_\tau := \sum_{t=0}^{H} R_t$$

definiert als die Summe der Belohnungen über eine Trajektorie τ. Dann lässt sich $J(\theta)$ schreiben als

$$\begin{aligned} J(\theta) :=& \mathbb{E}\left[\sum_{t=0}^{H} R_t\right] \\ =& \mathbb{E}\left[R_\tau\right]. \end{aligned}$$

Schreibt man den Erwartungswert aus erhält man:

$$J(\theta) = \sum_{\tau} p_{\pi_\theta}(\tau) R_\tau.$$

Hier ist $p_\pi(\tau)$ die Wahrscheinlichkeit für die Trajektorie τ unter der Bedingung, dass der Agent Strategie π folgt. Es soll nun der Gradient $\nabla_\theta J(\theta)$ bezüglich des Parametervektors θ betrachtet werden. Hier nutzen wir aus, dass Aufgrund des endlichen Horizonts H und der endlichen Aktions- und Zustandsräume $\mathcal{A}$ bzw. $\mathcal{S}$ die Summe über alle Trajektorien endlich ist und somit der Gradient und die Summe vertauscht werden kann. Es ergibt sich:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} p_{\pi_\theta}(\tau) R_\tau \\ &= \sum_{\tau} \nabla_\theta p_{\pi_\theta}(\tau) R_\tau \\ &= \sum_{\tau} \frac{p_{\pi_\theta}(\tau)}{p_{\pi_\theta}(\tau)} \nabla_\theta p_{\pi_\theta}(\tau) R_\tau. \end{aligned}$$

Nun nutzen wir aus das $\nabla_x(\log f(x)) = \nabla_x(f(x))/f(x)$ gilt.[41] Wir erhalten also

$$\nabla_\theta J(\theta) = \sum_{\tau} p_{\pi_\theta}(\tau) \nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) R_\tau.$$

Es lässt sich leicht sehen, dass diese Formulierung einen Erwartungswert beschreibt. Nämlich

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) R_\tau\right]. \tag{B.1.1}$$

Das Problem bei dieser Beschreibung ist, dass die Wahrscheinlichkeit einer bestimmten Trajektorie auch von der Umgebung abhängt. Um dieses Problem zu lösen, teilen wir zunächst den Term auf, dass die Unterscheidung zwischen Funktionen der Umgebung und der Strategie deutlich werden. Wir betrachten dazu den Term $\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right)$. Dieser setzt sich auf Zeitschrittebene

[41] Hierbei steht log für den natürlichen Logarithmus.

zusammen als Produkt der Wahrscheinlichkeiten für die Aktionen unter bestimmten Zuständen und der Übergangswahrscheinlichkeiten zum nächsten Zustand unter der Aktion und dem aktuellen Zustand:

$$\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) = \nabla_\theta \log\left[\prod_{t=0}^{H} p(S_{t+1} \mid S_t, A_t) \cdot \pi_\theta(A_t \mid s_t)\right].$$

Hier hängt $p(S_{t+1} \mid S_t, A_t)$ von der Umgebung ab und $\pi_\theta(A_t \mid S_t)$ wird von der Strategie bestimmt. Der Logarithmus eines Produkts lässt sich schreiben als die Summe von Logarithmen. Daraus folgt, dass

$$\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) = \nabla_\theta \left[\sum_{t=0}^{H} \log p(S_{t+1} \mid S_t, A_t) + \sum_{t=0}^{H} \log \pi_\theta(A_t \mid S_t)\right] \tag{B.1.2}$$

gilt. Man sieht sofort, dass nur die letzte der beiden Summen von θ abhängt. Der erste Summand fällt also bei der Ableitung weg. Durch vertauschen der Summe und des Gradienten erhält man

$$\nabla_\theta \log\left(p_{\pi_\theta}(\tau)\right) = \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(A_t \mid S_t).$$

Setzt man diesen Ausdruck nun bei (B.1.1) ein erhält man die Behauptung des Theorems:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(A_t \mid S_t) \cdot R_t\right].$$

□

C. Daten

C.1. Auswertung

Daten zu Kapitel 7.2.

Tabelle C.1.1: Die Verbesserung der Gesamtdividenden gegenüber der Bezugsstrategie (RMG) für verschiedene Vorhersagehorizonte in %. Die Werte sind gerundet auf die zweite Nachkommastelle, da wie in Kapitel 7.1 beschrieben von einer Unsicherheit von etwas unter 0,02% ausgegangen werden kann. Jedes Setting wurde zweimal trainiert. Der höhere Wert wurde in die Spalte Maximum und der niedrigere Wert in die Spalte Minimum eingetragen.

Vorhersagehorizont	Maximum	Minimum
0	1,28	1,26
1	1,33	1,33
2	1,36	0,89
3	1,37	1,36
6	1,36	1,36
2 (Nur natürliche Reinigung)	1,31	1,27

D. Ergänzung

D.1. Exponentialverteilung

Die Dichtefunktion einer Exponentialverteilten Zufallsvariablen X, mit dem Parameter $\lambda > 0$ ist:

$$f_\lambda(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{D.1.1}$$

der Mittelwert dieser Verteilung ist $\mu = 1/\lambda$.

Da die Verschmutzungsrate negativ ist, muss nachdem aus der Verteilung gezogen wurde, das Vorzeichen des Ergebnisses getauscht werden. Dafür muss auch das Vorzeichen des Mittelwertes geändert werden. Dies ist äquivalent wie aus der Dichtefunktion

$$f_{\hat{\lambda}}(x) = \begin{cases} -\hat{\lambda} e^{-\hat{\lambda} x} & x \leq 0 \\ 0 & x > 0 \end{cases} \tag{D.1.2}$$

zu ziehen. Hier ist $\hat{\lambda} < 0$ der Kehrwert des Mittelwerts.

Die Exponentialverteilung wird um einen Wert z versetzt, indem der Schwellenwert nicht bei 0, sondern bei z liegt:

$$f_{\hat{\lambda}}(x) = \begin{cases} -\hat{\lambda} e^{-\hat{\lambda} x} & x \leq z \\ 0 & x > z. \end{cases} \tag{D.1.3}$$

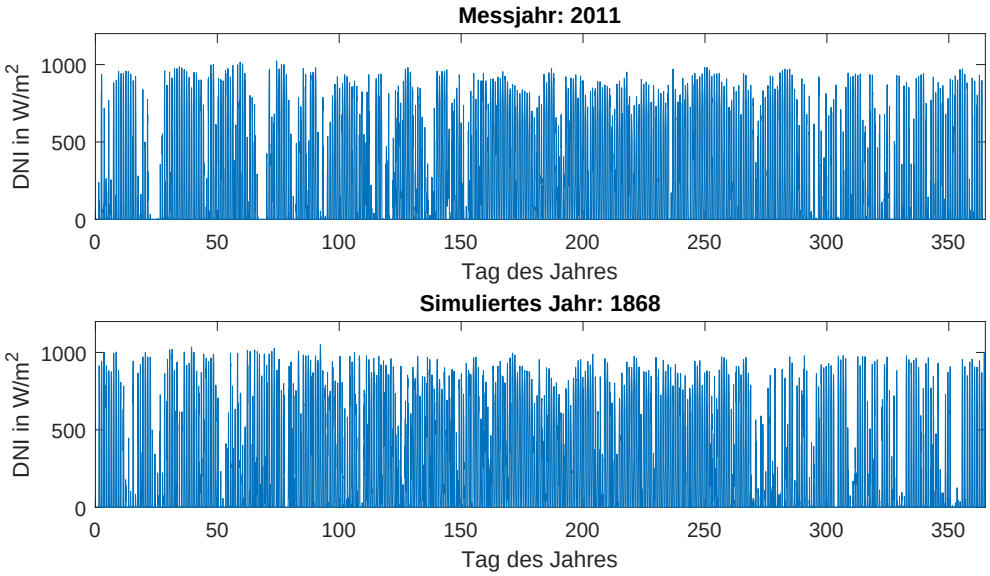


Abbildung D.2.1: DNI Kurven eines realen Messjahres und eines simulierten Jahres.

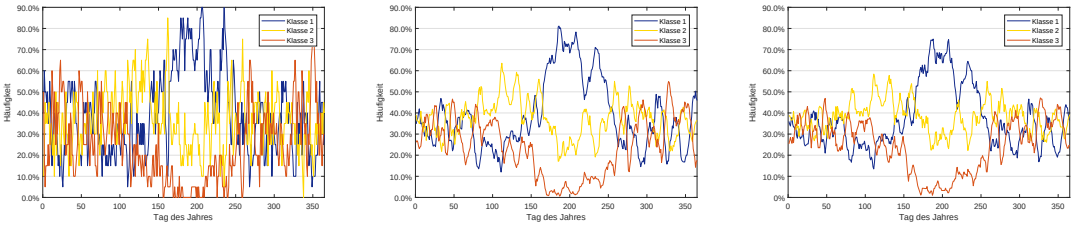
D.2. Generierte Daten

In diesem Kapitel soll darauf eingegangen werden, wie gut die generierten Daten die Messdaten abbilden. Die Untersuchung ist eher oberflächlich gehalten, da die Genauigkeit der Daten, für das Konzept von geringer Bedeutung sind.

Die innertägliche Abhängigkeiten zwischen den Wetterparametern wird gewährleistet, da für einen simulierten Tag immer nur ganze Messtage des gleichen Tag des Jahres (± 7 Tage) und der gleichen Klasse gezogen werden. Daher soll die jährliche Abhängigkeit der Daten qualitativ untersucht werden, wobei vor allem die DNI betrachtet wird, da diese den größten Einfluss auf den Ertrag eines CSP Kraftwerkes hat und da durch die Ziehung ganzer Messtage die Abhängigkeit der Wetterparameter untereinander gewährleistet sein sollte. Bei der Betrachtung der DNI eines zufällig ausgewählten Messjahres und eines simulierten Jahres, Abbildung D.2.1, werden verschiedene Ähnlichkeiten deutlich: Zum Beispiel treten Tage mit niedriger DNI sowohl in dem gemessenen Jahr, als auch in dem simulierten Jahr meistens gebündelt auf. Außerdem ist die Periode der meisten hohen DNI Tagen in beiden Fällen im Sommer ungefähr zwischen den Tagen des Jahres 150-250.

Beim Vergleich der jährlichen DNI Summe von neun gemessenen und neun simulierten Jahren, wird die Ähnlichkeit zwischen den Jahren deutlich; siehe Tabelle D.2.2.[42] In den neun Messjahren ist die durchschnittliche jährliche DNI Summe $2{,}24 \pm 0{,}11$ mWh/(m^2·a), wobei

[42] Es wurden lediglich neun gemessene Jahre gelistet, da die anderen Jahre größere Messlücken der DNI aufweisen.



(a) 20 Simulierte Jahre (b) 2000 Simulierte Jahre (c) Messdaten

Abbildung D.2.3: Die Verteilung der Klassen über das Jahr. Hierbei wird die Verteilung in 20 beziehungsweise 2000 simulierten Jahren mit der gemessenen Verteilung verglichen.

0,11 die empirische Standardabweichung ist. In der Stichprobe der neun gelisteten simulierten Jahre erhält man eine jährliche Einstrahlung von $2{,}22 \pm 0{,}1$ mWh/(m^2·a). Auf 2000 simulierten Daten wird eine Einstrahlung von $2{,}25 \pm 0{,}09$ mWh/(m^2·a) erreicht.

Tabelle D.2.2: Die Jahres DNI in mWh/(m^2·a) für neun zufällige simulierte Jahre und die Messjahre 2008 - 2016. Dabei sind die Jahre aufsteigend, nach der Jahreseinstrahlung, sortiert.

Simulation	Messjahre
2,10	2,09
2,12	2,15
2,17	2,17
2,19	2,23
2,20	2,23
2,25	2,24
2,27	2,31
2,31	2,36
2,41	2,42

Durch die Wahl der Übergangswahrscheinlichkeiten wird sichergestellt, dass sich die Verteilung der Klassen der gemessenen Verteilung immer weiter annähert; siehe Abbildung D.2.3.

D.3. Implementierung

Hier soll eine Kurzbeschreibung aller in Schaubild 6.0.2 aufgeführten Attribute und Methoden nach Klassen sortiert aufgeführt werden. Dabei werden nicht alle Funktionsweisen dargestellt. Für eine genauere Betrachtung müssen die *.py-Dateien eingesehen werden.

***class* Cleaning_unit** Klasse der Putzfahrzeuge. Hier werden die Eigenschaften der Putzfahrzeuge spezifiziert. In den Klammern ist der Wert für das, in der vorliegenden Arbeit genutzte,

Putzfahrzeug Albatros angegeben.

- cleaning_speed: int (9)

  Die Putzgeschwindigkeit des Putzfahrzeugs in Loops/Schicht

- op_staff: int (1)

  Anzahl der Mitarbeiter zur Bedienung eines Putzfahrzeugs

- fuel_consumption: double (7.)

  Kraftstoffverbrauch in Liter/Loop

- cleanliness0: double (0.986)

  Die Reflektivität des Spiegels nach der Reinigung

- water_consumption: double (1.)

  Der Wasserverbrauch in m^3/Loop

- lifetime: double (15.)

  Die erwartete Betriebsdauer eines Putzfahrzeugs

***class* Weather** In dieser Klasse werden die Wetterbedingungen für ein oder mehrere Jahre gespeichert.

Attribute

- no_years: int

  Anzahl der Jahre, die in diesem Objekt gespeichert sind

- ghi, dni, dhi, tAmb, relHum, press, ws, wd: no_years x 365 x 24 Array[double]

  Stündlich aufgelöste Wetterdaten GHI, DNI, DHI, Umgebungstemperatur, relative Luftfeuchtigkeit, Luftdruck, Windgeschwindigkeit und Windrichtung

- var_class, soiling_rate, rain: no_years x 365 Array

  Täglich aufgelöste Klasse, Verschmutzungsrate und Regenmenge

- perfect_results: dict

  Gespeicherte perfekte Ergebnisse, das heißt Gesamtdividenden wenn kein Putzkosten anfallen und die Reflektivität dauerhaft bei 1 ist.[43]

[43] Die Abweichung von diesen perfekten Ergebnissen war zuerst als ein weiteres Gütekriterium gedacht. Für die Vergleichbarkeit mit [Wol+18], wurde im Endeffekt der RMG als Gütekriterium genutzt.

Methoden

- get_year()

  Übergibt i-te Jahr als Objekt der Weather Klasse. Wenn kein i übergeben wird, wird ein zufälliges Jahr zurückgegeben

- get_year_range(range)

  Übergibt einen in *range* spezifizierten Bereich an Jahren als Objekt der Weather Klasse

- create_perfect_results()

  Erstellt die oben genannten perfekten Ergebnisse und speichert diese im Attribut *perfect_results* (muss nur einmal ausgeführt werden)

***class* Field** In den Objekten dieser Klasse werden die Kostenparameter spezifiziert. Während einer Simulation, werden in dieser Klasse sowohl die aktuelle Sauberkeit und Verfügbarkeit der einzelnen Loops, als auch die Kosten gespeichert.

Attribute

- start_cleanliness: double

  Die Reflektivität der Spiegel bei der Initialisierung

- no_cleaning_units: int

  Anzahl der Putzfahrzeuge

- cu_type: Cleaning_unit

  Spezifikation der Putzfahrzeuge in einem Objekt der Klasse Cleaning_unit

- no_loops: int

  Anzahl der Loops

- deprecationCU: double

  Abschreibungskosten der Putzfahrzeuge in € / (Jahr· Putzfahrzeug)

- fuel_cost: double

  Kraftstoffpreis in € / Liter

- water_cost: double

  Wasserkosten in € / m^3

- year_salary: double

  Gehalt eines Arbeiters in € / 250 Schichten

- eff_m_area: double

  Effektive Spiegelfläche oder auch Aperturfläche in m^2

- cleanliness_history, availability_history: 356 x no_loops Array[double]

  Die nach Loops und Tagen aufgelöste Sauberkeit und Verfügbarkeit. Diese wird während der Simulation eines Jahres gespeichert

Methoden

- update_field(soiling_rate, actions)

  Updatet die Attribute des Feldes nach einem Tag mit der Verschmutzungsrate *soiling_rate* und den Aktionen *actions*

- get_annual_costs()

  Gibt die jährlichen Kosten nach einem simulierten Jahr zurück

***module* greenius_interface** Gesammelte Funktionen für das Starten von Greenius über Python.

- write_greenius_files()

  Schreibt die von Greenius benötigten Daten, mit den Wetterdaten und Kosten eines simulierten Jahres

- execute_greenius()

  Führt Greenius für die Dateien im angegebenen Pfad aus und liest das Ergebnis aus der von Greenius erstellten .csv Datei. Gibt das Ergebnis als *Dictionary* zurück

***class* Environment** Die übergeordnete Klasse für die Umgebung. Benötigt alle bis hierhin aufgeführten Klassen.

Attribute:

- weather: Weather

  Ein Jahr an Wetterdaten als Objekt der Wetter-Klasse

- field: Field

  Ein Objekt der Klasse Field, mit den Feldspezifikationen

- day: int

  Der aktuelle Tag in der Simulation

- action_history: 4 x 365 Array[int]

  Speichert die Aktionen, dabei stehen die Einträge für die Putzschichten in der Reihenfolge tagsüber, nachts, tagsüber mit Extraeinheiten und nachts mit Extraeinheiten[44]

Methoden:

- step(actions)

  Updatet die Umgebung für einen Tag mit den übergebenen Aktionen. Erhöht *day* um eins

- write_effective_dni()

  Berechnet die effektive Einstrahlung, indem die Verschmutzung und die Verfügbarkeit mit der Einstrahlung verrechnet wird, wie in Gleichung (4.3.7) beschrieben

- evaluate_year()

  Wertet ein Jahr mit Greenius aus, nachdem die Aktionen und die Verschmutzung dieses Jahr simuliert wurden. Gibt die wichtigen Kennwerte für dieses Jahres zurück und speichert diese in einer .csv Datei

- return_state()

  Gibt den aktuellen Zustand der Umgebung in einem *dictionary* zurück

- return_state_action_history()

  Gibt die Zustands – Aktions Paare für jeden Tag des Jahres zurück. Kann nur aufgerufen werden, wenn die Simulation abgeschlossen ist, das heißt $day == 365$

***class* RL_Agent** Die Klasse des Agenten, mit den zugehörigen Methoden zum Trainieren und Ausführen der Strategie.

Attribute:

- input_dim: int

  Dimension der Eingabeschicht des neuronalen Netzes

- output_dim: int

  Dimension der Ausgabeschicht des neuronalen Netzes

[44] Die Extraeinheiten sind wie in [Wol+18] definiert, werden aber in der vorliegenden Arbeit nicht weiter betrachtet.

- hidden_dims: list[int]

  Dimension der verdeckten Schichten des neuronalen Netzes

- output_to_actions: funct

  Funktion, welche die Ausgabe des neuronalen Netzes auf die Aktionen abbildet. Die Aktionen sind ein 4-dimensionales Array mit der Anzahl der Tagschichten, Nachtschichten und Tag- beziehungsweise Nachtschichten der Extraeinheiten

- state_to_input: funct

  Funktion, die den Zustand der Umgebung auf die Eingabe für das Netzwerk abbildet. An dieser Stelle wird festgelegt, welchen Teil des Zustandes der Agent sieht

- div_history: list[double]

  Liste der Gesamtdividenden bei den verschiedenen Validierungsschritten

Methoden:

- get_action(state)

  Gibt die Aktionen für einen gegebenen Zustand zurück. Dabei kann unterschieden werden, ob die Aktionen zufällig nach den Wahrscheinlichkeiten (Trainings-Fall) gezogen werden oder deterministisch (Test-Fall) die höchste Wahrscheinlichkeit gewählt wird

- fit(states, actions, rewards)

  Updatet die Gewichte der Strategie nach dem *Adam* Algorithmus zu gegebenen Zuständen, Aktionen und Belohnungen

- save_agent()

  Speichert sowohl die Attribute, als auch die aktuellen Gewichte des neuronalen Netzes in einer Datei

- run_RL_year(env)

  Führt den Agenten auf einem Jahr aus, welches in dem Objekt env, der Klasse Environment gespeichert ist. Dabei kann sowohl der stochastische, als auch der deterministische Fall betrachtet werden. Gibt verschiedene Kennwerte, unter anderem die Gesamtdividenden, zurück

- evaluate_agent()

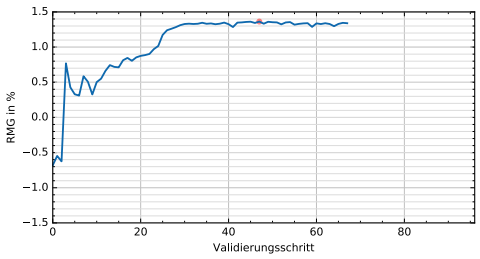
  Wertet den Agenten im deterministischen Test-Fall auf mehreren Jahren aus. Errechnet

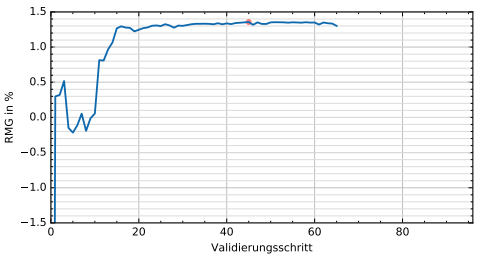
die durchschnittlichen Gesamtdividenden und andere Kennwerte auf diesem Datensatz. Speichert die Ergebnisse in einer *.csv Datei, wenn gewünscht

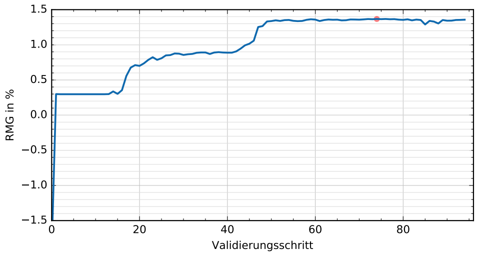
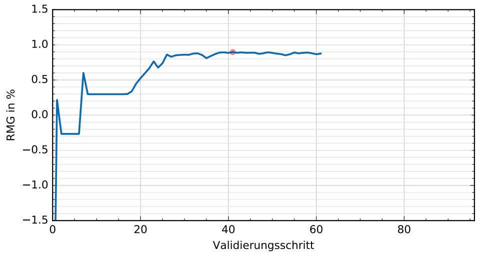
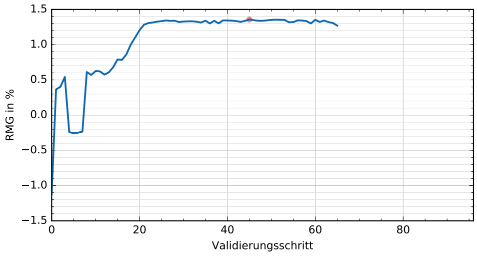
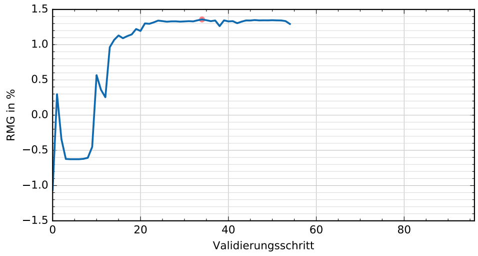
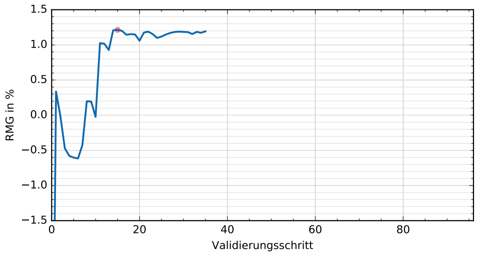
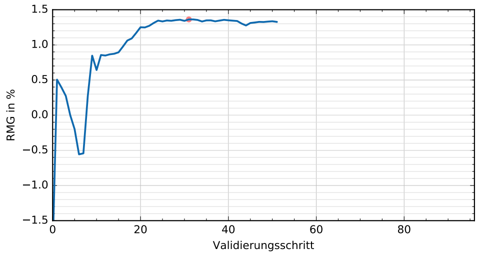
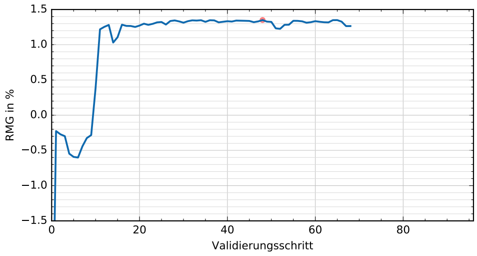
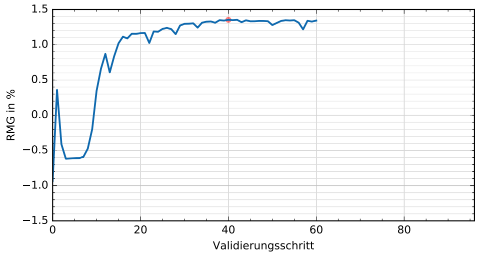
- run_training_routine()
  Führt das Training auf dem Trainingsdatensatz aus. Validiert in regelmäßigen (default=10 Trainingsschritte) Abständen. Bricht ab, wenn der Agent *austrainiert* ist

D.4. Lernkurven

Hier sollen 10 der 31 Lernkurven des Robustheitstests in Kapitel 7.1.2 aufgezeigt werden. Diese Lernkurven sind zufällig ausgewählt und sollen ein Einblick in die Lernverläufe geben. Der jeweils beste Validierungsschritt ist rot markiert. Es wird deutlich, dass die Lernverläufe einander ähneln und dass die Agenten meistens nach 50 bis 70 Validierungsschritten austrainiert sind. Des Weiteren sieht man, dass es ohne Vergleichsmöglichkeit schwer zu erkennen ist, ob ein Ergebnis nah am Optimum ist, da die Lernverläufe der schlechteren Durchläufe den guten Durchläufen qualitativ ähnlich sehen – siehe den Trainingsverlauf in der zweiten Reihe auf der rechten Seite.






RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

RMG in %
Validierungsschritt

D.5. Netzarchitekturen

Tabelle D.5.1: Tabelle der verschiedenen genutzten Netzwerkarchitekturen. Genauere Erklärung der Einträge folgen weiter unten.

Beschreibung	Input-Dim.	Ausgabe-Dim	verborgene Schichten	Aktivierungsfunktion	#Parameter	sichtbarer Zustand
Robustheit Test Kapitel 7.1.2	13	9	[30,25]	tanh + softmax	1429	SR[1,2], Klasse[1,2], ∅ − Sauberkeit
Schwellenwert Agent Kapitel 7.1.1	1	2	[20,15]	tanh + softmax	387	∅ − Sauberkeit
Horizont Vergleich Kapitel 7.2.1						
Horizont 0	1	9	[35,30]	tanh + softmax	1429	∅ − Sauberkeit
Horizont 1	7	9	[35,30]	tanh + softmax	1639	SR[1] Klasse[1], ∅− Sauberkeit
Horizont 2	13	9	[35,30]	tanh + softmax	1849	SR[1,2], Klasse[1,2], ∅ − Sauberkeit
Horizont 3	19	9	[35,30]	tanh + softmax	2059	SR[1:3], Klasse[1:3], ∅ − Sauberkeit
Horizont 6	37	9	[35,30]	tanh + softmax	2689	SR[1:6], Klasse[1:6], ∅ − Sauberkeit
Nat.Reinigung	3	9	[35,30]	tanh + softmax	1499	nat. Reinigung [1,2], ∅−Sauberkeit
Sandstürme Kapitel 7.2.2	13	9	[30,25]	tanh + softmax	1429	SR[1,2], Klasse[1,2], ∅ − Sauberkeit
Hoher Wasserpreis Kapitel 7.2.3	13	9	[30,25]	tanh + softmax	1429	SR[1,2], Klasse[1,2], ∅ − Sauberkeit

In Tabelle D.5.1 werden die Parameter der genutzten Netze aufgelistet. Dies dient der besseren Reproduzierbarkeit und der detaillierteren Beschreibung der Neuronalen Netze.
Die Input- beziehungsweise Ausgabe-Dimension beschreibt die Anzahl der Knoten in der Eingabe- und Ausgabeschicht. Die Eingabe-Dimension wird durch den sichtbaren Zustand bestimmt. Die Feldsauberkeit benötigt einen Eingabeknoten. Diesem Knoten wird die skalierte Feldsauberkeit $\xi_{scal} = (\xi - 0{,}95)/0.05$ übergeben. Wobei ξ die durchschnittliche Feldsauberkeit über allen Loops ist. Diese Skalierung wurde vorgenommen, da der Bereich der Feldsauberkeit zwischen 0,9 und 1 als am wichtigsten eingestuft wurde.
Die Klasse wird dem Netzwerk in *one-hot-encoding* übergeben. Das heißt, das Netzwerk erhält einen 3-dimensionalen Vektor mit zwei Nulleinträgen und einer eins. Die Stelle des nichtnull Eintrages zeigt an, welche Klasse gegeben ist. Die Verschmutzungsrate (SR) benötigt ebenso eine dreidimensionale Eingabe. Hier zeigt der nichtnull Eintrag an, ob eine natürliche Reinigung (SR>0), starke Verschmutzung (SR<-0,02) oder normale Verschmutzung (sonst) vorliegt. Es wurde one-hot-encoding genutzt, da dieses für kategorische Eingaben oftmals das beste Ergebnis liefert.
Wenn ausschließlich die natürliche Reinigung übergeben wurde, benötigt man nur einen Eingabeknoten. Dieser ist 1, falls ein natürliches Reinigungsereignis vorliegt und 0 sonst. Die Zahlen in den eckigen Klammern geben an für welche Tage der Wert vorliegt, dabei bezeichnet 1 den nächsten Tag. Die Notation 1:n steht für alle Zahlen von 1 bis inklusive n: $[1, 2, \ldots, n]$.
In der Spalte verborgene Schichten werden die Anzahl der Knoten in den verborgenen Schichten, von links nach rechts aufgleistet. Als Aktivierungsfunktion wurde für alle Netze bei den verborgenen Schichten tanh genutzt; siehe Abbildung 2.1.4 (c). Die Ausgabeschicht besitzt eine Softmax Aktivierungsfunktion; siehe (3.0.2).
In der Spalte #Parameter, ist die Anzahl aller optimierbaren Parameter, das heißt die Gewichte und Schwellenwerte des neuronalen Netzes, angegeben. Die Anzahl der Gewichte zwischen zwei Schichten ergibt sich aus dem Produkt der Knotenanzahl der beiden Schichten. Die Anzahl der Schwellenwerte ist gleich der Anzahl der Knoten. Ist nun N_i die Anzahl der Knoten der i-ten Schicht, wobei die 0-te Schicht die Eingabeschicht und die n-te Schicht die Ausgabeschicht ist, erhält man

$$\#\text{Parameter} = \sum_{i=0}^{n-1} N_i \cdot N_{i+1} + \sum_{i=1}^{n} N_i. \tag{D.5.2}$$

Literatur

[Aba+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu und Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[Abb17] Pieter Abbeel. *Policy Gradients and Actor Critic*. Lecture during Deep RL Bootcamp at University of California, Berkeley `https://sites.google.com/view/deep-rl-bootcamp/lectures`. Aufgerufen 13.7.2018. Aug. 2017.

[Ba+17] H. Truong Ba, M.E. Cholette, R. Wang, P. Borghesani, L. Ma und T.A. Steinberg. "Optimal condition-based cleaning of solar power collectors". In: *Solar Energy* 157 (2017), S. 762 –777. ISSN: 0038-092X. DOI: `https://doi.org/10.1016/j.solener.2017.08.076`. URL: `http://www.sciencedirect.com/science/article/pii/S0038092X17307582`.

[BB88] Jonathan Barzilai und Jonathan M. Borwein. "Two-Point Step Size Gradient Methods". In: *IMA Journal of Numerical Analysis* 8.1 (1988), S. 141–148. DOI: `10.1093/imanum/8.1.141`. eprint: `/oup/backfile/content_public/journal/imajna/8/1/10.1093/imanum/8.1.141/2/8-1-141.pdf`. URL: `http://dx.doi.org/10.1093/imanum/8.1.141`.

[Bea17] Andrew L. Beam. *Deep Learning 101 - Part 1: History and Background*. `https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html`. Aufgerufen am 01.07.2018. 2017.

[BT95] D. P. Bertsekas und J. N. Tsitsiklis. "Neuro-dynamic programming: an overview". In: *Proceedings of 1995 34th IEEE Conference on Decision and Control*. Bd. 1. 1995, 560–564 vol.1. DOI: `10.1109/CDC.1995.478953`.

[Cho+15] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[Cyb89] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (1989), S. 303–314. ISSN: 1435-568X. DOI: `10.1007/BF02551274`.

[Dau+14] Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli und Yoshua Bengio. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *CoRR* abs/1406.2572 (2014). arXiv: `1406.2572`. URL: `http://arxiv.org/abs/1406.2572`.

[Der15] J Dersch. *greenius User Manual, Version 4.1*. Techn. Ber. DLR, 2015.

[DGS86] Danny M. Deffenbaugh, Steve T. Green und Steve J. Svedeman. "The effect of dust accumulation on line-focus parabolic trough solar collector performance". In: *Solar Energy* 36.2 (1986), S. 139 –146. ISSN: 0038-092X. DOI: `https://doi.org/10.1016/0038-092X(86)90118-0`. URL: `http://www.sciencedirect.com/science/article/pii/0038092X86901180`.

[Doy96] Kenji Doya. "Temporal difference learning in continuous time and space". In: *Advances in neural information processing systems*. 1996, S. 1073–1079.

[FLP15] Gerd Fischer, Matthias Lehner und Angela Puchert. *Einführung in die Stochastik*. Springer, 2015.

[Fre13] Niels Jonas Freise. "A comparative cost analysis of soiling effects on CSP mirrors under variable cleaning scenarios: Tunisia, Morocco and Spain". Bachelor Thesis. 2013.

[GBB04] Evan Greensmith, Peter L Bartlett und Jonathan Baxter. "Variance reduction techniques for gradient estimates in reinforcement learning". In: *Journal of Machine Learning Research* 5.Nov (2004), S. 1471–1530.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[GMH13] Alex Graves, Abdel-rahman Mohamed und Geoffrey E. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *CoRR* abs/1303.5778 (2013). arXiv: `1303.5778`. URL: `http://arxiv.org/abs/1303.5778`.

[Gri10] Andreas Griewank. "Who Invented the Reverse Mode of Differentiation ?" In: 2010.

[Gu+16] Shixiang Gu, Timothy P. Lillicrap, Zoubin Ghahramani, Richard E. Turner und Sergey Levine. "Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic". In: *CoRR* abs/1611.02247 (2016). arXiv: `1611.02247`. URL: `http://arxiv.org/abs/1611.02247`.

[Har17] Fiona Harvey. *Syria signs Paris climate agreement and leaves US isolated*. `https://www.theguardian.com/environment/2017/nov/07/syria-signs-paris-climate-agreement-and-leaves-us-isolated`. Aufgerufen am 13.09.2018. 2017.

[Hee+17] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller und David Silver. "Emergence of Locomotion Behaviours in Rich Environments". In: *CoRR* abs/1707.02286 (2017). arXiv: `1707.02286`. URL: `http://arxiv.org/abs/1707.02286`.

[Hel13] Malt Helmert. *Grundlagen der Künstlichen Intelligenz: 19. Handeln unter Unsicherheit: Grundlagen*. Vorlesung an der Universität Basel `http://ai.cs.unibas.ch/_files/teaching/fs13/ki/slides/ki19.pdf`. Aufgerufen 17.7.2018. 2013.

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), S. 1735–1780.

[HU15] National Institute of Health U.S. *Tox21 Data Challenge*. `https://tripod.nih.gov/tox21/challenge/index.jsp`. Aufgerufen am 29.06.2018. 2015.

[Hun07] J. D. Hunter. "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3 (2007), S. 90–95. ISSN: 1521-9615. DOI: `10.1109/MCSE.2007.55`.

[Jou+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn,

Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox und Doe Hyun Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760. URL: http://arxiv.org/abs/1704.04760.

[Kai11] Andreas Kaiser. "Reflektivitätsmessungen und Waschfahrzeugauswahl in einem Parabolrinnen-Kraftwerk". unterliegt Geheimhaltungsvereinbarung. Bachelorarbeit. 2011.

[Kar18] Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition.* Lecture at University of California, Berkeley http://cs231n.github.io/neural-networks-3/. Aufgerufen 05.07.2018. 2018.

[KB14] Diederik P. Kingma und Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[KH09] Alex Krizhevsky und Geoffrey Hinton. "Learning multiple layers of features from tiny images". In: *Citeseer* (2009).

[Kim+06] A Kimber, L Mitchell, S Nogradi und H Wenger. "The effect of soiling on large grid-connected photovoltaic systems in California and the southwest region of the United States". In: *Photovoltaic Energy Conversion, Conference Record of the 2006 IEEE 4th World Conference on.* Bd. 2. IEEE. 2006, S. 2391–2395.

[KS04] Nate Kohl und Peter Stone. "Machine learning for fast quadrupedal locomotion". In: *AAAI.* Bd. 4. 2004, S. 611–616.

[KVH12] Kyle Kattke und Lorin Vant-Hull. *Optimum Target Reflectivity for Heliostat Washing.* SolarPACES Conference. 2012.

[Li+17] Yuanlong Li, Yonggang Wen, Kyle Guan und Dacheng Tao. "Transforming Cooling Optimization for Green Data Center via Deep Reinforcement Learning". In: *CoRR* abs/1709.05077 (2017). arXiv: 1709.05077. URL: http://arxiv.org/abs/1709.05077.

[Mah+07] N. M. Mahowald, J. A. Ballantine, J. Feddema und N. Ramankutty. "Global trends in visibility: implications for dust sources". In: *Atmospheric Chemistry and Physics* 7.12 (2007), S. 3309–3339. DOI: 10.5194/acp-7-3309-2007. URL: https://www.atmos-chem-phys.net/7/3309/2007/.

[Mau16] Gianluca Mauro. *Science_Direct_API_trends_analysis.* GitHub repository. Commit: 7a93606a8dfc0de66e5b2a99d98312af33baa1fb. 2016. URL: https://github.com/gianlucahmd/Science_Direct_API_trends_analysis.

[MK13] Felipe A Mejia und Jan Kleissl. "Soiling losses for solar photovoltaic systems in California". In: *Solar Energy* 95 (2013), S. 357–363.

[Mni+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), S. 529.

[MP69] Marvin Minsky und Seymour A Papert. *Perceptrons: an introduction to computational geometry.* Cambridge, MA: MIT Press, 1969.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html.

[NM65] John A Nelder und Roger Mead. “A simplex method for function minimization”. In: *The computer journal* 7.4 (1965), S. 308–313.

[OLD98] Maria Overbeck-Larisch und Wolfgang Dolejsky. *Stochastik mit Mathematica*. Vieweg+Teubner Verlag, Wiesbaden, 1998.

[Pie+14] Robert Carl Pietzcker, Daniel Stetter, Susanne Manger und Gunnar Luderer. “Using the sun to decarbonize the power sector: The economic potential of photovoltaics and concentrating solar power”. In: *Applied Energy* 135 (2014), S. 704 –720. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2014.08.011.

[Qua+01] Volker Quaschning, Winfried Ortmanns, Rainer Kistner und Michael Geyer. “Greenius: a new simulation environment for technical and economical analysis of renewable independent power projects”. In: Solar Forum. 2001, S. 413–418.

[RHW86] David E Rumelhart, Geoffrey E Hinton und Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), S. 533.

[Roc+17] Johan Rockström, Owen Gaffney, Joeri Rogelj, Malte Meinshausen, Nebojsa Nakicenovic und Hans Joachim Schellnhuber. “A roadmap for rapid decarbonization”. In: *Science* 355.6331 (2017), S. 1269–1271. ISSN: 0036-8075. DOI: 10.1126/science.aah3443. eprint: http://science.sciencemag.org/content/355/6331/1269.full.pdf. URL: http://science.sciencemag.org/content/355/6331/1269.

[Roj96] Raul Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag, 1996. URL: https://page.mi.fu-berlin.de/rojas/neural/.

[Ros+07] Alexandra G. Rosati, Jeffrey R. Stevens, Brian Hare und Marc D. Hauser. “The Evolutionary Origins of Human Patience: Temporal Preferences in Chimpanzees, Bonobos, and Human Adults”. In: *Current Biology* 17.19 (2007), S. 1663 –1668. ISSN: 0960-9822. DOI: https://doi.org/10.1016/j.cub.2007.08.033. URL: http://www.sciencedirect.com/science/article/pii/S0960982207018507.

[Ros58] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), S. 386–408.

[Rud16] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[SA17] Ava Soleimany und Alexander Amini. *Introduction to deep learning*. Lecture at Massachusetts Institute of Technology http://introtodeeplearning.com/. Aufgerufen am 01.07.2018. 2017.

[SB18] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second Edition. The MIT Press, 2018.

[Sch15] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural Networks* 61 (2015), S. 85 –117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0893608014002135.

[Sch17] John Schulman. *Reinforcement learning with policy gradient*. Lecture at University of California, Berkeley http://rail.eecs.berkeley.edu/deeprlcoursesp17/docs/lec2.pdf. Aufgerufen 13.7.2018. Feb. 2017.

[SH+18] Marion Schroedter-Homscheidt, Miriam Kosmale, Sandra Jung und Jan Kleissl. “Classifying ground-measured 1 minute temporal variability within hourly intervals for direct normal irradiances”. In: *Meteorologische Zeitschrift* (2018).

[Sil+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel und Demis Hassabis. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Okt. 2017), S. 354. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[Sil15] David Silver. *Reinforcment Learning*. Lecture at University College London http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html. Aufgerufen 3.7.2018. 2015.

[Sol18a] SolarPACES. *CSP Doesn't Compete With PV – it Competes with Gas*. https://www.solarpaces.org/csp-competes-with-natural-gas-not-pv/. Aufgerufen am 13.09.2018. 2018.

[Sol18b] SolarPACES. *How CSP Works: Tower, Trough, Fresnel or Dish*. https://www.solarpaces.org/how-csp-works/. Aufgerufen am 13.09.2018. 2018.

[Sze10] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Bd. 4. Morgan & Claypool Publishers, Jan. 2010.

[Val17] Venelin Valkov. "Building a Cat Detector using Convolutional Neural Networks - TensorFlow for Hackers (Part III)". In: *Medium* (2017).

[Wil92] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Reinforcement Learning*. Hrsg. von Richard S. Sutton. Boston, MA: Springer US, 1992, S. 5–32. ISBN: 978-1-4615-3618-5. DOI: 10.1007/978-1-4615-3618-5_2. URL: https://doi.org/10.1007/978-1-4615-3618-5_2.

[WM97] D. H. Wolpert und W. G. Macready. "No free lunch theorems for optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), S. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

[Wol+18] Fabian Wolfertstetter, Stefan Wilbert, Jürgen Dersch, Simon Dieckmann, Robert Pitz-Paal und Abdellatif Ghennioui. "Integration of Soiling-Rate Measurements and Cleaning Strategies in Yield Analysis of Parabolic Trough Plants". In: *Journal of Solar Energy Engineering* 140.4 (2018), S. 041008.

[Wol+19] Fabian Wolfertstetter, Felix Terhag, Stefan Wilbert, Tobias Hirsch und Oliver Schaudt. "Optimization of Cleaning Strategies Based on ANN Algorithms Assessing the Benefit of Soiling Rate Forecasts". In: *SolarPACES conference*. Forthcoming. 2019.

[Wol16] F. Wolfertstetter. "Auswirkungen von Verschmutzung auf konzentrierende solarthermische Kraftwerke". PhD. 2016.

[Yu18] Yang Yu. "Towards Sample Efficient Reinforcement Learning". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, Juli 2018, S. 5739–5743. DOI: 10.24963/ijcai.2018/820. URL: https://doi.org/10.24963/ijcai.2018/820.

[Zog16] Thomas Zoglauer. *Einführung in die formale Logik für Philosophen*. 5. Auflage. Vandenhoeck & Ruprecht, 2016. ISBN: 9783838545929. URL: https://www.utb-studi-e-book.de/einfuehrung-in-die-formale-logik-fuer-philosophen-2804.html.

[Alb13] Grupo Albatros. *Grupo Albatros CSP Cleaning Vehicles*. http://albatrosexport.com/albatros-the-market-leader-in-mirror-cleaning-vehicles-for-csp-plants/. Aufgerufen am 07.09.2018. 2013.

[Int11] Intel Corporation. *Over 6 Decades of Continued Transistor Shrinkage, Innovation*. Santa Clara, CA: Press Release, Mai 2011. https://www.intel.com/content/www/us/en/silicon-innovations/standards-22-nanometers-technology-backgrounder.html. 2011.

[Kag] Kaggle Inc. *Datasets*. https://www.kaggle.com/datasets. Aufgerufen am 29.06.2018.

[Pyt] Python Documentation. *History and License*. https://docs.python.org/3/license.html. Aufgerufen am 30.06.2018.

[RWE11] RWE Innogy GmbH. *Deutsches Konsortium weiht Solarthermiekraftwerk Andasol 3 feierlich ein*. Presseinformation 30. September 2011. https://www.rwe.com/web/cms/en/113648/rwe/press-news/press-release/?pmid=4006895. 2011.

[UNF15] UNFCCC. "Paris Agreement". In: United Nations Framework Convention on Climate Change. 2015.

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Köln, den 3. Dezember 2018

Felix Terhag