Carl von Ossietzky Universität Oldenburg
Abteilung für Digitalisierte Energiesysteme

Vergleich von Supervised Learning und Reinforcement Learning in einem Deep Learning Modell auf das Optimal Power Flow Problem

Bachelorarbeit

vorgelegt von

Keno Ortmann

Geboren am 28.11.2000 in Papenburg

Erstprüferin:	Prof. Dr.-Ing. Astrid Nieße
Zweitprüfer:	M.Sc. Thomas Wolgast
Betreuer:	M.Sc. Thomas Wolgast

Oldenburg, den 24. April 2023

Kurzzusammenfassung

Der Optimal Power Flow (OPF) ist ein Optimierungsproblem der Energieübertragungssysteme, bei dem es unterschiedliche Lösungsmethoden gibt. Durch den Fortschritt des maschinellen Lernens konnten die Lösungsmethoden des OPF erfolgreich erweitert werden, sodass auch Deep Neural Networks (DNN) eingesetzt werden. Die Verfahren zum Trainieren der DNN, insbesondere Supervised Learning (SL) und Deep Reinforcement Learning (DRL), zeigen beide vielversprechende Ergebnisse bei der Berechnung des OPF. Dabei haben die Verfahren einen unterschiedlichen Trainingsablauf: SL lernt die Muster in einem vorher beschriftetem Datensatz zu erkennen, während DRL einen Agenten einsetzt, der mit einer Umgebung (in diesem Fall den OPF) interagieren kann und durch Belohnungen das Problem selbst lösen kann. Einen Vergleich auf gemeinsamer Grundlage der Methoden SL und DRL bezogen auf OPF fehlt in der Literatur, was in dieser Arbeit erreicht wurde. Die Tests in einem simulierten Energiesystem zeigen, dass beide Verfahren Vor- und Nachteile haben. SL hat verglichen mit DRL weniger Trainingszeit benötigt, einen geringeren Aufwand der Implementierung, einen geringeren Fehler zur optimalen Aktion und die Nebenbedingungen besser eingehalten. DRL hingegen hat die Zielfunktion des OPF leicht besser erfüllt und weniger oft ungültige Lösungen produziert. Damit hat DRL etwas besser abgeschnitten, da die Zielfunktion die wichtigste Metrik ist, jedoch gibt es in den anderen Kriterien die genannten Vor- und Nachteile.

Inhaltsverzeichnis

Abkürzungen

AC OPF	Alternating Current OPF
BP	Backpropagation
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DL	Deep Learning
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
DBN	Deep Belief Network
GPU	Graphic Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
KI	Künstliche Intelligenz
KNN	Künstliches Neuronales Netz
MAPE	Mean Absolute Percentage Error
MEP	Markow Entscheidungsproblem
ML	Maschinelles Lernen
MLP	Mehrlagiges Perzeptron
MSE	Mean Squared Error
OPF	Optimal Power Flow
PPO	Proximal Policy Optimization

ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RBM	Restricted Boltzmann Machine
SAC	Soft Actor Critic
SL	Supervised Learning
SGD	Stochastic Gradient Descent
TD3	Twin Delayed Deep Deterministic Policy Gradient
UL	Unsupervised Learning
DC OPF	Direct Current OPF

Formelzeichen

Lateinische Symbole

A	Zielwert
$\mathcal{A}$	Menge aller möglichen Aktionen
F	Vorhergesagter Wert
$\vec{P_G}$	Wirkleistungen der Generatoren
$\vec{P_L}$	Wirkleistungen aller Lasten
Q	Erwartete kumulative Belohnung von einem Zustand und Aktion
$\vec{Q_L}$	Blindleistungen aller Lasten
$\vec{Q_G}$	Blindleistungen der Generatoren
$\mathcal{S}$	Menge aller möglichen Zustände
V^{π}	Erwartete kumulative Belohnung von einer Strategie
a	Aktion
$c_{Blindleistung}$	minimale/maximale Blindleistung
c_{Last}	Netz-/Trafo-Last
$c_{Slack-Bus}$	Eingeschränkter Blindleistungsfluss über Slack-Bus
$c_{Spannung}$	Spannungsbänder
f(x)	Aktivierungsfunktion
f(u, v)	Zielfunktion
g(u, v)	Gleichheitsbedingungen
h(u, v)	Ungleichheitsbedingungen

$\mathbf{p}$	Bestrafung
$p_G^{\vec{Q}}$	Blindleistungspreise der Generatoren
$\mathbf{r}$	Belohnung
$\mathbf{s}$	Zustand
$\mathbf{u}$	Systemzustandsvariablen
$\mathbf{v}$	Kontrollvariablen
$\mathbf{w}$	Gewicht
$\mathbf{x}$	Eingehende Verbindung
$\mathbf{y}$	Ausgehende Verbindung

Griechische Symbole

γ	Diskontierungsrate
τ	Aktualisierungsrate der Zielnetzwerke
θ	Schwellwert
π	Strategiefunktion

Mathematische Symbole

$\mathbf{E}$	Erwarteter Wert
$\mathbb{R}$	Menge der reellen Zahlen

1 Einleitung

1.1 Motivation

Die Energiemärkte haben sich weltweit in den letzten zwei Jahrzehnten stark verändert. Die klassische Energieerzeugung durch Kohle und Atom soll weitestgehend durch erneuerbare Energien ersetzt werden. Da erneuerbare Energien stark fluktuieren können, z.B. bei Solarzellen durch das Wetter und die Tageszeit, wird auch das Stromnetz durch den Ausbau stärker schwanken. Aber nicht nur durch erneuerbare Energien werden größere Schwankungen beobachtet, denn die Nachfrage nach Energie ist immer größer geworden, beispielsweise durch die steigende Elektromobilität. Zudem gibt es einen Trend der Dezentralisierung von Strommärkten, wodurch Vorhersagen am Strommarkt stärker schwanken können. Durch diese Faktoren wird ein optimaler Leistungsfluss, der Optimal Power Flow (OPF), immer komplexer zu berechnen. Diese Berechnung ist notwendig, um z.B. die Verluste eines Stromnetzes zu minimieren und somit die Effizienz des Stromnetzes zu steigern. Die Formulierung des OPF beinhaltet viele Variablen und Nebenbedingungen und ist allgemein ein nichtlineares und nichtkonvexes Optimierungsproblem. Um den OPF zu lösen, gibt es Algorithmen, die auf den jeweiligen OPF spezialisiert sind. [1]
Angetrieben von der Minimierung des Rechenaufwands werden vermehrt alternative Lösungsmethoden wie maschinelles Lernen erforscht. Künstliche neuronale Netze sind dabei eine beliebte Methode, um die Lösungen für ein Problem (in diesem Fall den OPF) vorherzusagen. Dafür müssen die neuronalen Netze zuerst mit Daten trainiert werden, wofür es drei Methoden gibt: Supervised Learning (SL), Unsupervised Learning (UL) und Reinforcement Learning (RL). SL und UL benötigen zum Trainieren des Modells einen Datensatz, der dann im Training gelernt wird. Bei RL hingegen wird ein Agent eingesetzt, der mit einer Umgebung interagieren kann und Belohnungen erhält, wodurch er selbstständig lernt, das Problem zu lösen. Mit Deep Learning (DL) werden neuronale Netze um mehrere Schichten erweitert, damit sie noch leistungsfähiger sind. Wenn man RL mit DL verbindet, wird es auch Deep Reinforcement Learning (DRL) genannt. Dabei sind SL und DRL die Trainingsverfahren, die bereits bei der Lösung des OPF vielversprechende Ergebnisse geliefert haben [2][3]. SL und DRL sind sehr unterschiedlich und wurden nach aktuellem Kenntnisstand

bei der Lösung des OPF noch nicht in Bezug auf Lösungsgüte oder Komplexität verglichen.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist einen Vergleich von SL und DRL durchzuführen. Dafür wird zunächst das OPF Problem definiert, welches als Problemstellung dient. In dieser Arbeit werden zwei OPF Umgebungen verwendet. Es wird dann eine Grundlage geschaffen, was die beiden Trainingsverfahren gemeinsam haben, damit der Vergleich so fair wie möglich ist. So soll das gleiche neuronale Netz mit gleichen Parametern bei SL und DRL eingesetzt werden, sodass sich nur das Trainingsverfahren unterscheidet. Nachdem die neuronalen Netze trainiert wurden, werden dann die Modelle getestet und in folgenden Kriterien verglichen beziehungsweise analysiert:

- Wie hoch ist der Aufwand der Implementierung?
- Wie viel Zeit brauchen die Modelle zum Trainieren?
- Wie viel Zeit brauchen die Modelle, um eine Vorhersage zu machen?
- Was sind Vor- und Nachteile der Trainingsverfahren?
- Wie gut ist die Lösungsqualität der Modelle?

Dabei ist der letzte Punkt der wichtigste Faktor, da dieser aussagt, wie gut die Modelle die Lösung des OPF vorhergesagt haben.

1.3 Struktur

In Kapitel 2 werden zunächst die benötigten Grundlagen für diese Arbeit geschaffen. Um zu verstehen, wie der aktuelle Forschungsstand ist, werden in Kapitel 3 verwandte Arbeiten im Bereich SL und DRL bezogen auf den OPF kurz zusammengefasst. Danach kann die Implementierung in Kapitel 4 vorgenommen werden, wo zunächst die Umgebungen und alle Vergleichskriterien sowie Metriken definiert werden und dann das Programm erklärt wird. Nun können die Modelle trainiert werden. Die Ergebnisse des Trainings sowie die Ergebnisse des Vergleichs werden in Kapitel 5 vorgestellt. Anschließend werden die Ergebnisse in Kapitel 6 diskutiert und in Kapitel 7 das Fazit der Arbeit formuliert.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit vorgestellt. Der OPF wird in Abschnitt 2.1 erklärt. Danach wird alles zum Themengebiet maschinelles Lernen in Abschnitt 2.2 vorgestellt, dazu gehören die Begriffe Künstliches Neuronales Netz (KNN), DL, SL, RL und DRL.

2.1 Optimal Power Flow

Das OPF Problem berechnet optimalen Leistungsfluss in elektrischen Energieübertragungssystemen. Grundsätzlich ist OPF ein nichtlineares, nichtkonvexes und bedingt skalierbares Problem mit kontinuierlichen und diskreten Variablen. Ziele wie die Gesamtkosten des Systems, Planung, Sicherheit oder elektrische Verluste sollen optimiert werden; gleichzeitig müssen Einschränkungen des Systems oder der Infrastruktur eingehalten werden. Der OPF berechnet auf dieser Grundlage und den Systemzustandsvariablen u, darunter Strom, Spannung und Leistung, die entscheidenden Kontrollvariablen v. Zu den Kontrollvariablen gehören variable Betriebsgrößen von Blindleistungen, Transformatoreinstellungen, Busspannungsgrößen usw. Im Allgemeinem minimiert der OPF eine Zielfunktion $f(u, v)$. Die Funktionen $g(u, v)$ beziehungsweise $h(u, v)$ repräsentieren Gleichheits- und Ungleichheitsbedingungen des Systems. [1]

$$\begin{aligned} &\min f(u, v) \\ \text{u.d.N. } &g(u, v) = 0 \\ &h(u, v) \leq 0 \end{aligned} \tag{2.1}$$

Die Formulierung mit allen Nebenbedingungen des OPF wird auch Alternating Current OPF (AC OPF) genannt. Die Komplexität der Berechnung des AC OPF steigt erheblich mit präziserer Modellierung des Systems. Direct Current OPF (DC OPF) ist eine lineare Approximation des AC OPF und lässt sich leichter berechnen, allerdings sind die Lösungen von DC OPF keine zulässigen Lösungen des AC OPF. [4]

Es gibt mehrere Formulierungen des OPF, die auf ihre Einsatzgebiete optimiert sind und verschiedene Notationen, Variablen, Funktionen und Lösungsmethoden verwenden. Eine

allgemeine Lösungsmethode ist (noch) nicht bekannt. Fast jeder mathematische Ansatz wurde auf das Problem angewandt, z.B. Gradientenverfahren, Newtons Methode, Simplex Verfahren, usw. [1]

2.2 Maschinelles Lernen

Künstliche Intelligenz (KI) ist ein Forschungsgebiet der Informatik und befasst sich mit der Simulation von Intelligenz im Computer. Mit KI können z.B. Kaufentscheidungen vorhergesagt oder E-Mail-Postfächer automatisch kategorisiert werden. Ein durchdachtes Regelwerk, das vom Menschen programmiert wird, ermöglicht das intelligente Verhalten des Programms. Ein Teilgebiet von KI ist Maschinelles Lernen (ML). Dabei wird kein festes Regelwerk befolgt, denn der Computer lernt selbst, sich auf das Problem anzupassen. Dazu wird ein lernender Algorithmus eingesetzt, um aus Erfahrung Modelle zu erstellen, die Vorhersagen treffen können. Die Erfahrung wird in dem Training des Modells gewonnen und kann mit drei verschiedenen Methoden erfolgen: überwachtes Lernen (Supervised Learning), unüberwachtes Lernen (Unsupervised Learning) und bestärkendes Lernen (Reinforcement Learning). [5]
KNN sind wichtige lernende Algorithmen, die im nächsten Unterabschnitt 2.2.1 vorgestellt werden. Auf dieser Grundlage werden viele Anwendungen im Bereich vom ML realisiert. Mit DL werden die KNN erweitert, um sie noch leistungsstärker zu machen, was im Unterabschnitt 2.2.2 eingeführt wird. Die KNN müssen zuerst trainiert werden, damit sie ein Problem lösen können. Dafür können unterschiedliche Methoden angewendet werden. Zu diesen Methoden gehören SL und RL, welche in Unterabschnitt 2.2.3 und Unterabschnitt 2.2.4 vorgestellt werden. Diese Methoden werden auch verwendet, um DL Modelle zu trainieren. Der Name der Methoden wird mit *Deep* erweitert. Deep SL funktioniert analog zu SL. Deep RL hingegen unterscheidet sich von RL und wird im Unterabschnitt 2.2.5 präsentiert.

2.2.1 Künstliche Neuronale Netze

Ein KNN ist ein lernender Algorithmus, der vom Aufbau des Gehirns inspiriert wurde. Die zentralen Bestandteile dieser Netze sind Neuronen. In biologischen neuronalen Netzen sind Neuronen mit anderen Neuronen verbunden und leiten bei einer Aktivierung ein Signal weiter. Beim Übersteigen einer elektrischen Schwelle wird ein Neuron aktiviert. Ein Neuron kann mehrere eingehende Verbindungen haben, aber nur eine ausgehende Verbindung. [6]

McCulloch und Pitts haben in [7] anhand dieser Definition das M-P Neuron Modell aufgestellt, welches heute noch benutzt wird. In Abbildung 2.1 ist der Aufbau dargestellt.

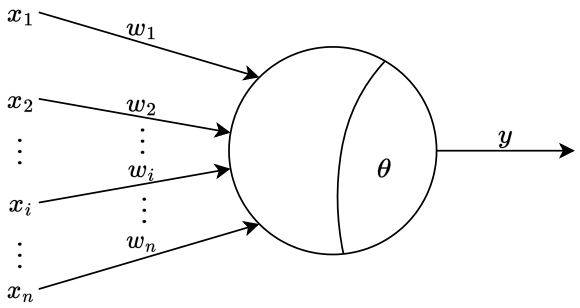

Abbildung 2.1: M-P Neuron Modell, basierend auf [6]

In dem Modell hat ein Neuron n Eingangsverbindungen von Neuronen mit den Gewichten $w_i (i = 1, 2, ..., n)$. Das Neuron produziert beim Überschreiten des Schwellwerts θ ein Signal an das nächste Neuron. Der Ausgang y kann mit der folgenden Gleichung berechnet werden:

$$y = f\left(\sum_{i=1}^{n} w_i x_i - \theta\right). \tag{2.2}$$

Der Schwellwert wird von der gewichteten Summe der Eingangsverbindungen subtrahiert. Das Ergebnis wird in eine Aktivierungsfunktion f eingesetzt.

Es gibt mehrere Aktivierungsfunktionen, die ein unterschiedliches Verhalten der Neuronen erzeugen. Typische Aktivierungsfunktionen sind z.B. Sigmoid, Rectified Linear Unit (ReLU) und Tangens hyperbolicus. Sigmoid wird definiert als:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

ReLU wird definiert als:

$$f(x) = \begin{cases} x & \text{wenn } x \geq 0 \\ 0 & \text{sonst.} \end{cases} \tag{2.4}$$

Tangens hyperbolicus wird definiert als:

$$f(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.5}$$

[8]

Ein Mehrlagiges Perzeptron (MLP) ist ein spezielles KNN und wird durch Ebenen von Neuronen konstruiert. Eine Ebene besteht aus Neuronen, die in einer Reihe platziert sind.

Die Neuronen haben Verbindungen zur nächsten Ebene und sind je nach Architektur anders gerichtet. Das MLP besteht aus mindestens zwei Ebenen, einer Eingangsebene und einer Ausgangsebene. Die Eingangsebene empfängt die Daten und die Ausgangsebene ist die vom Modell generierte Antwort auf die Eingangsdaten. Die Gewichte w_i und die Schwellwerte θ des KNN können durch Trainingsdaten gelernt werden. Ein typisches MLP ist in Abbildung 2.2 dargestellt. [6]

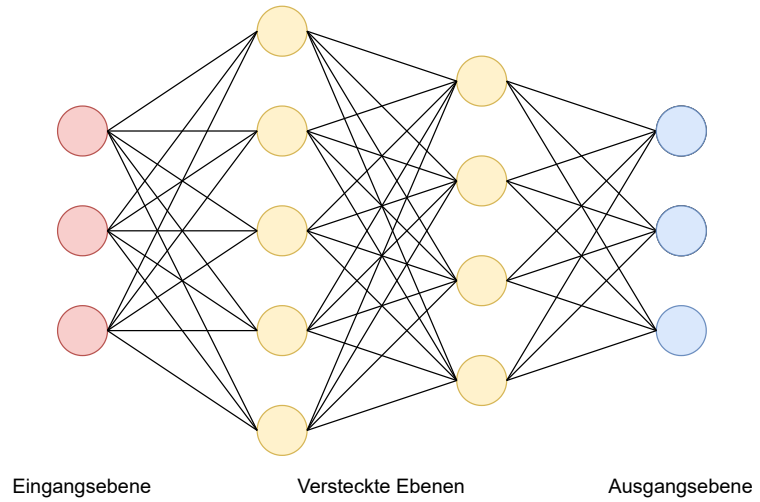

Abbildung 2.2: Beispiel für den Aufbau eines MLP, basierend auf [6]

In diesem Fall hat das MLP mehr als zwei Ebenen. Die Ebenen zwischen Eingangs- und Ausgangsebene sind *versteckt*. In Unterabschnitt 2.2.2 wird dieses MLP näher beschrieben.

Aus mathematischer Sicht ist ein KNN eine Funktion mit vielen Parametern, die im Wesentlichen eine Menge X zu einer Menge Y zuordnet [9].

$$f : X \to Y \tag{2.6}$$

Um die Gleichung 2.6 zu lernen, werden Daten benötigt, die je nach Lernmethode anders definiert sind. Nach der mathematischen Definition wird für jedes Element in X ein Element in Y zugeordnet. Durch Messfehler, statistische Effekte usw. können in KNN jedoch mehrere Zuordnungen für Y vorliegen. Algorithmen müssen diese Effekte möglichst effektiv auflösen, um ein richtiges Ergebnis zu erhalten. [9]

2.2.2 Deep Learning

DL ist ein Teilgebiet vom ML. Ein DL Modell oder auch Deep Neural Network (DNN) ist ein KNN mit (mehreren) versteckten Ebenen. Die Ebenen heißen *versteckt*, da sie zwischen der

Eingangs- und Ausgangsebene platziert sind und somit von außen nicht „sichtbar“ sind. Dies wird in Abbildung 2.2 dargestellt. Zwischen der Eingangs- und Ausgangsebene befinden sich zwei versteckte Ebenen. Die versteckten Ebenen bewirken eine höhere Lernfähigkeit, weshalb die Netze *tief* genannt werden.

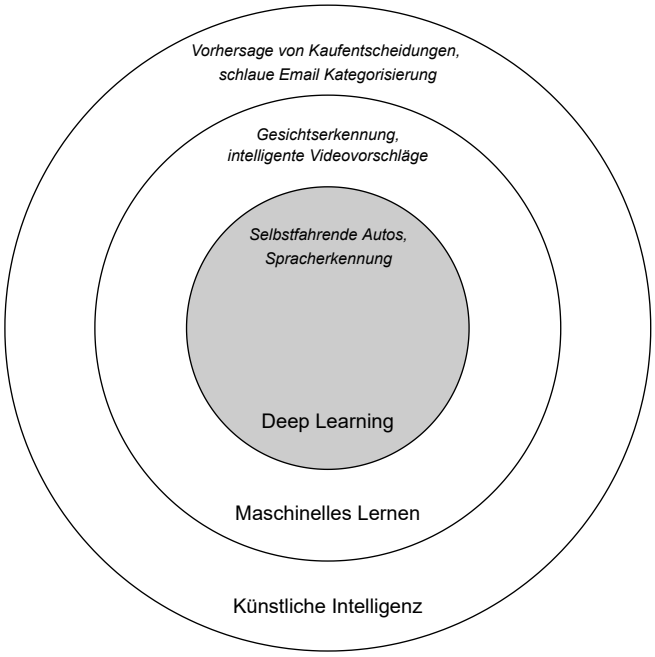

Abbildung 2.3: Beziehung zwischen KI, ML und DL, basierend auf [8]

In Abbildung 2.3 wird die Beziehung zwischen KI, ML und DL verdeutlicht. KI ist der Oberbegriff für dieses Forschungsgebiet und beinhaltet einfache intelligente Programme. ML Anwendungen gehören auch zum Themengebiet der KI, unterscheiden sich aber in der Implementierung. Die Programme befolgen keine Regelwerke mehr, sondern lernen mit Daten das Problem eigenständig. Dies ermöglicht komplexere Anwendungen wie Gesichtserkennung und intelligente Videovorschläge durch das Benutzerverhalten. DL erbt wiederrum die Techniken vom ML und erweitert sie, indem die Modelle *tiefer* werden und dadurch auch höheren Rechenaufwand erfordern. Die DL Modelle erzielen teilweise menschliche Leistungen in Einsatzgebieten wie selbstfahrende Autos und Spracherkennung. [8]

Um die Kapazität eines KNN zu steigern, d.h. eine bessere Fähigkeit komplexe Probleme zu lernen, kann man entweder die Anzahl der versteckten Ebenen oder die Anzahl der Neuronen erhöhen. Es zeigt sich, dass mehr versteckte Ebenen effektiver sind als mehr Neuronen, da mit jeder Ebene mehr Aktivierungsfunktionen verschachtelt werden. Dies führt zu erheblich höherem Rechenaufwand und braucht größere Datenmengen zum Trainieren, da

die Modelle sonst überangepasst sein können. Ein überangepasstes Modell hat die Muster in den Trainingsdaten nicht genügend erkannt und klassifiziert neue, unbekannte Daten fehlerhaft im Vergleich zu den Trainingsdaten. Mit dem Zeitalter des Cloud Computing und Big Data können diese Probleme immer besser gelöst werden und DL Modelle mit vielen versteckten Ebenen erfolgreich trainiert werden. Die Anzahl der versteckten Ebenen und die Anzahl an Neuronen ist nicht festgelegt und wird oft auf das Problem angepasst und ausprobiert. [5]

Wie gut ein ML Modell ist, hängt stark von der Repräsentation der Trainingsdaten ab. Die Daten müssen gut vorbereitet, gefiltert und analysiert werden, um das Modell erfolgreich zu trainieren. Dieser Prozess ist zeitaufwändig und schwierig. DNN haben die Fähigkeit aus ungefilterten und hochdimensionalen Daten Muster zu erkennen. Die rohen Daten werden in die Eingangsebene des DL Modells übergegeben und das Modell kann im folgenden Training die Abstraktionen aus den Eingangsdaten lernen. DNN sind ein wichtiger Schritt zur generellen künstlichen Intelligenz, die eigenständig neue Probleme lösen kann und der menschlichen Intelligenz immer näherkommt. [10]

Es gibt verschiedene DL Architekturen, die in einem DL Modell aneinandergereiht werden können. Die Architekturen werden im Folgenden kurz erklärt.

Eine Restricted Boltzmann Machine (RBM) ist ein neuronales Netz mit zwei Ebenen, einer Eingangsebene und einer versteckten Ebene. Innerhalb der beiden Ebenen existieren keine Verbindungen zwischen den Neuronen. Die Eingangsebene ist vollständig mit der versteckten Ebene verbunden, d.h. jedes Neuron der Eingangsebene hat eine Verbindung zu jedem Neuron der versteckten Ebene. RBM sind die zentralen Bestandteile moderner DL Architekturen. [8]

Ein Deep Belief Network (DBN) ist eine Erweiterung der RBM mit mehreren versteckten Ebenen. Es bildet die Grundlage für DNN. [8]

Autoencoder bestehen aus zwei symmetrischen DBN, wobei das erste DBN die Eingangsdaten enkodiert und das zweite DBN die Daten wieder dekodiert. Die Eingangs- und Ausgangsebene haben die gleiche Anzahl an Neuronen. In der Mitte der DBN befinden sich weniger Neuronen, da das enkodierende DBN die Daten komprimiert hat. Das dekodierende DBN versucht die komprimierten Daten wieder zu rekonstruieren. Dadurch entsteht ein Informationsverlust. Ein Autoencoder extrahiert damit wesentliche Merkmale der Eingangsdaten und wird beispielsweise für Datenkomprimierung oder Bildersuche eingesetzt. [8]

Ein Convolutional Neural Network (CNN) besteht aus drei Schichten: Convolutional Layer, Pooling Layer und Fully Connected Layer. Die Convolutional Layer berechnet mit einer Faltungsmatrix eine Konvolution mit den Eingangsdaten, bei CNN üblicherweise Pixel von Bilddaten oder Signale von Audiodaten. Das Ergebnis wird an die Pooling

Layer weitergegeben, die die Größe des Netzwerks verringert, während die wesentlichen Merkmale erhalten bleiben. Die Fully Connected Layer verknüpft die Pooling Layer mit der Ausgangsebene des Netzwerks, z.B. für Klassifizierungen. Die einzelnen Schichten können mehrfach aneinandergereiht werden. CNN sind nützlich für Anwendungen wie Bilderkennung und Spracherkennung. [10]

2.2.3 Supervised Learning

SL oder auch überwachtes Lernen ist eine klassische Methode, um ML Modelle zu trainieren. In Abbildung 2.4 ist das Verfahren abgebildet.

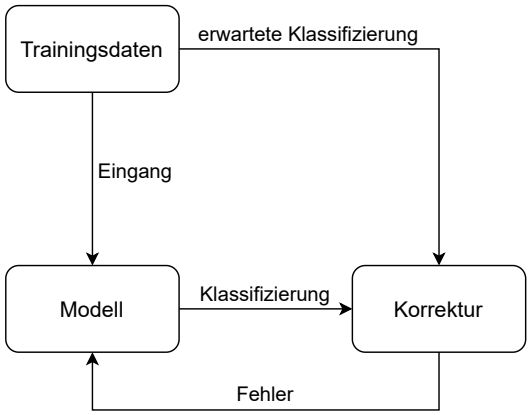

Abbildung 2.4: Supervised Learning, basierend auf [6]

Bei SL wird ein beschrifteter Datensatz benötigt, der Eingangsdaten und Zieldaten beinhaltet. Die Eingangsdaten sind für den Eingang des Modells vorgesehen und das Modell macht daraufhin eine Vorhersage bzw. Klassifizierung. Da für alle Eingangsdaten auch die Zieldaten im Datensatz vorhanden sind, kann der Ausgang des Modells mit der erwarteten Klassifizierung verglichen werden. Wenn es eine Abweichung (Fehler) von der erwarteten Klassifizierung gibt, werden die Parameter des Modells entsprechend korrigiert, um die Abweichung zu minimieren. In dem Training des Modells wird dieser Prozess so lange wiederholt, bis die Abweichung minimiert ist oder die Abweichung nicht weiter abnimmt. Dabei muss das Modell jedoch nicht den Datensatz auswendig lernen, sondern die Muster in dem Datensatz erkennen, um auf neue Daten, die nicht im Datensatz vorkommen, akkurat zu reagieren. Wenn das Modell auf den Trainingsdaten gute Ergebnisse liefert, aber auf ungesehene Daten schlechte Ergebnisse erzielt, ist das Modell überangepasst. [5]

SL wird für zwei Anwendungen benutzt: Klassifikationen und Regressionen. Bei der Klassifikation soll das Modell Daten in diskrete, vordefinierte Kategorien zuordnen. Eine typische Anwendung dafür ist Objekterkennung auf Bildern. Mit Regressionen werden

kontinuierliche Daten eingesetzt, die sich für Vorhersagen eignen, wie z.B. Kursvorhersagen am Aktienmarkt. [5]

Die wichtigsten Algorithmen für SL sind KNN mit Backpropagation (BP), k-nächster-Nachbar, Naiver Bayes-Klassifikator und Entscheidungsbäume. BP ist ein weit verbreiteter Algorithmus, um ein KNN überwacht zu trainieren. Dafür wird die Differenz von der Ausgabe des Modells und der erwarteten Ausgabe berechnet. Diese Differenz wird auch Fehler genannt. Der Algorithmus breitet sich rückwärts über das Netz aus und passt die Gewichte abhängig vom Einfluss auf den Fehler an. Dies wird mehrmals wiederholt, bis der Fehler möglichst minimiert ist. [11]

2.2.4 Reinforcement Learning

RL oder auch bestärkendes Lernen ist eine Methode, um ML Modelle zu trainieren und wird im Folgenden auf Grundlage von [11] erklärt. Bei RL wird ein lernender Entscheider, auch *Agent* genannt, eingesetzt, der in einer Umgebung Aktionen ausführt und dafür Belohnungen erhält. Durch das Ausprobieren von Aktionen kann der Agent über Zeit lernen, die bestmögliche Sequenz von Aktionen zu finden, um das Problem zu lösen. Dadurch unterscheidet sich RL grundlegend zu SL, da kein beschrifteter Trainingsdatensatz benötigt wird [12]. Zu den Bestandteilen von RL gehören Agent, Umgebung, Zustände, Aktionen und Belohnungen.

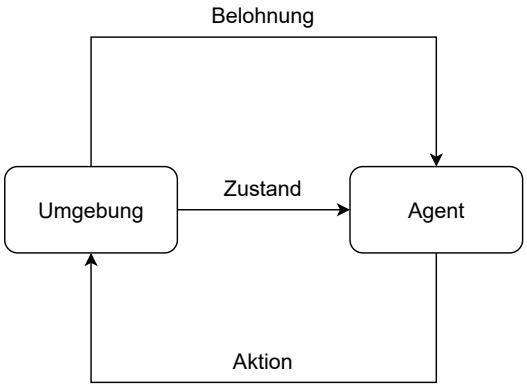

Abbildung 2.5: Reinforcement Learning, basierend auf [6]

In der Abbildung 2.5 wird der Zusammenhang zwischen den Bestandteilen verdeutlicht. Der Agent nimmt die Umgebung als einen Zustand $s_t \in \mathcal{S}$ zu einem Zeitpunkt t wahr, wobei $\mathcal{S}$ die Menge aller möglichen Zustände ist. Die Zeit t mit $t = 0, 1, 2, \ldots, T$ verläuft sequenziell. Durch eine Aktion a_t aus der Menge $\mathcal{A}(s_t)$ aller möglichen Aktionen zum Zustand s_t kann der Agent die Umgebung verändern. Der Zustand iteriert mit jeder

gewählten Aktion einen Zustand s_{t+1} weiter. Durch die Umgebung erhält der Agent eine Belohnung $r_t \in \mathbb{R}$. Das Problem wird als Markow Entscheidungsproblem (MEP) modelliert, bei dem Zustand und Belohnung der nächsten Iteration nur von dem aktuellen Zustand und Aktion abhängig ist. Wenn die Zustände und Aktionen endlich sind, dann wird das Problem als endliches MEP bezeichnet.

Das Ziel des Agenten ist die Belohnung zu maximieren, indem er eine Strategie π entwickelt, die über Zeit optimiert wird. Mit der Strategie $\pi : S \to A$ wählt der Agent zu einem Zustand s_t eine Aktion a_t. Die zu erwartende kumulative Belohnung bei einer Strategie zu einem Zustand s_t wird als $V^\pi(s_t)$ bezeichnet. $Q(s_t, a_t)$ gibt die zu erwartende kumulative Belohnung in einem Zustand s_t mit der Aktion a_t an. Der Q-Learning Algorithmus speichert und aktualisiert mit jedem Schritt die Werte von Q in einer Tabelle, um die bestmöglichen Aktion a_t zum Zustand s_t zu finden.

Der Agent muss eine Balance zwischen Erkundung der Umgebung und Ausbeuten einer guten Strategie halten. Um eine hohe Belohnung zu erhalten, muss der Agent die Umgebung erkunden und zufällige Aktionen ausprobieren, aber er muss gleichzeitig historisch gute Aktionen nutzen, um die Belohnung zu maximieren. [13]

Das Programm terminiert, wenn ein Endzustand erreicht wird, der vorher definiert wird (finiter Horizont). Modelle ohne Limitierungen des Endzustands werden infinite Horizonte genannt. Eine Episode ist eine Folge von Aktionen vom Startzustand bis zum Endzustand.

Die erwartete kumulative Belohnung $V^\pi(s_t)$ zur Zeit t im Modell des finiten Horizonts wird wie folgt definiert:

$$V^\pi(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]. \tag{2.7}$$

Im Modell des infiniten Horizonts sind die Schritte nicht begrenzt. Damit die Belohnung nicht gegen unendlich strebt, wird eine Diskontierungsrate $0 \leq \gamma < 1$ eingeführt:

$$V^\pi(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]. \tag{2.8}$$

2.2.5 Deep Reinforcement Learning

DRL ist ein Teilgebiet von RL und kombiniert RL Techniken mit DL. Wenn bei RL hochdimensionale und rohe Daten der Umgebung vorliegen, müssen diese zunächst in eine niedrigdimensionale Zustandsvariable s_t konvertiert werden [11]. Mit dem Einsatz von DNN müssen die Daten nicht konvertiert werden, sondern können direkt in das neuronale Netz übergeben werden [11]. Das DNN approximiert die Strategie Funktion $\pi(a|s; w)$

und/oder die Funktionen V und Q, wobei w die Gewichte des DNN repräsentieren [14]. Durch den Einsatz von DNN können komplexere Strategien entwickelt werden, da die Umgebung direkt und ungefiltert als Eingabe verwendet werden kann, insbesondere wenn die Umgebung große Zustands- und Aktionsmengen hat [10].

DRL wurde erfolgreich auf mehrere Einsatzgebiete angewendet, darunter auch Videospiele. Die Forscher von DeepMind haben mit einem Deep Q Network (DQN) Spiele der Atari 2600 Reihe gelernt. Ein DQN ist ein DNN, das mit dem Q-Learning Algorithmus trainiert wird. Die berechneten Pixel der Spiele wurden als hochdimensionale Eingabe (210 $\times$ 160 RGB-Video bei 60 Hz) für das DNN verwendet. Das Modell hatte vergleichbare oder gar bessere Fähigkeiten wie ein professioneller menschlicher Spieler. [10]

Der Deep Deterministic Policy Gradient (DDPG) Algorithmus erweitert die Idee von DQN, den Q-Wert mit neuronalen Netzen zu approximieren. Dabei wird ein DNN für den Aktor und für den Kritiker erstellt. Mit dem Aktor wird versucht, die optimale Aktion für einen Zustand vorherzusagen und der Kritiker bewertet dann die Aktion des Aktors. Um die Stabilität des Lernens zu verbessern, werden zwei weitere DNN benutzt: ein Ziel-Aktor-Netz und ein Ziel-Kritiker-Netz, welche periodisch mit der Aktualisierungsrate τ aktualisiert werden. Weiterhin wird ein Erfahrungspuffer (Replay Buffer) eingesetzt, um Erfahrungen des Agenten zu speichern und im Training zufällig wiederzuverwenden. [15]

3 Verwandte Arbeiten

Mit diesem Kapitel soll ein Überblick über den aktuellen Forschungsstand geschaffen werden. In Abschnitt 3.1 werden die SL Ansätze auf OPF näher beschrieben. Analog werden in Abschnitt 3.2 die DRL Ansätze vorgestellt. Weiterhin kann DRL mit SL kombiniert werden, was in Abschnitt 3.3 vorgestellt wird. Anschließend werden in Abschnitt 3.4 die Ansätze zusammengefasst und verglichen.

3.1 Supervised Learning Ansätze auf OPF

Pan et al. wenden in [2] SL auf das DC OPF Problem an. Die Autoren beschreiben, dass DNN mit ihrer Fähigkeit hochdimensionale Muster in Daten zu erkennen gut auf OPF anwendbar sind. Der Ansatz wird DeepOPF genannt. Das DNN hat drei versteckte Ebenen und wird mit optimalen Lösungen von klassischen Optimierungslösern trainiert. Das Parametertuning des DNN erfolgt mit Stochastic Gradient Descent (SGD). Der Ausgang des DNN kann nicht direkt verwendet werden, da das DNN im schlechtesten Fall zu 18.3 % unzulässige Lösungen liefert. Es wird ein Nachbearbeiten benötigt, bei dem eine zulässige Lösung des OPF berechnet wird. Dies verringert die Größe des DNN, da nur ein Teil der Variablen vorhergesagt werden muss. Mit den Institute of Electrical and Electronics Engineers (IEEE) 30, 57, 118 und 300-Bus Systemen wurde das Modell getestet. DeepOPF konnte das Problem um den Faktor 100 schneller berechnen als der Optimierungslöser. Die Lösungen hatten eine geringe Abweichung von 0.2 % zur optimalen Lösung und verletzten die Nebenbedingungen nicht.

Basierend auf dieser Arbeit erweitern Zhao et al. in [16] die Methode zu DeepOPF+ und wenden es auf das DC OPF Problem an. Während der DeepOPF Ansatz schnelle Ergebnisse liefert, wird das Nachbearbeiten aufwendig, wenn die Vorhersagen des DNN in Energiesystemen mit hoher Auslastung unpräziser werden. Dieses Problem soll DeepOPF+ lösen, indem die Trainingsdaten auf die Grenzen der Generatoren und Übertragungsleitungen kalibriert werden. Damit ist der Ausgang des DNN im erlaubten Lösungsbereich des DC OPF. Die Tests erfolgten auf den gleichen Testsystemem wie in [2] und der Ansatz

erzielte eine schnellere Berechnung als DeepOPF.

Zamzam und Baker wenden in [17] SL auf das AC OPF Problem an. Sie wurden durch die schnelle Berechnung mit erlaubten Ergebnissen und niedrigen Optimalitätsverlust in [2] motiviert, den Ansatz auf AC OPF zu erweitern. Das DNN hat drei versteckte Ebenen und wird mit generierten Daten trainiert, die erlaubte Lösungen des OPF sind. Analog wie Pan et al. in [2] wird ein Nachbearbeiten der Daten benötigt, bei dem der Ausgang des DNN als Variablen für eine Reihe von Leistungsflussgleichungen dient, um damit eine erlaubte Lösung zu berechnen. Es wurden die Testsysteme IEEE 39, 57 und 188-Bus verwendet. Die Ergebnisse zeigten, dass das DNN mit Nachbearbeiten signifikant schnellere Lösungen im Vergleich zu klassischen Optimierungslösern liefert. Die Lösungen waren nahezu optimal.

Kim und Kim wenden in [18] einen SL Ansatz auf das DC OPF Problem an. Das überwachte Lernen ist nach den Autoren ein vielversprechender Ansatz für OPF, hat aber das Problem, dass die Ergebnisse des DNN die Nebenbedingungen nicht immer einhalten. Sie verwenden eine Projektionsschicht, die den Ausgang des DNN auf eine lösungssichere Ebene projiziert. Das DNN hat drei versteckte Ebenen. Die Tests erfolgten auf den IEEE 39, 118, 162-Bus Systemen und zeigten nahezu optimale Lösungen ohne Verletzung der Nebenbedingungen.

Huang et al. wenden in [19] einen SL Ansatz auf das AC OPF Problem an. Die Autoren beschreiben, dass DNN die Berechnung des Problems wesentlich verkürzen können, was in den Energiesystemen immer dringender benötigt wird. Der Ausgang des DNN wird nicht direkt verwendet, sondern die Variablen des OPF werden durch die Teillösungen des DNN mit Leistungsflussgleichungen berechnet. Die Trainingsdaten werden durch Optimierungslöser generiert. Die Tests erfolgten auf den IEEE 118 und 300-Bus System sowie einem 2000-Bus System. Die Ergebnisse wurden bis zu 10000-mal schneller berechnet mit einem Optimalitätsverlust von weniger als 0.2 %, ohne die Nebenbedingungen zu verletzen.

Velloso und Hentenryck wenden in [20] SL auf das OPF Problem an. Nach den Autoren ist ML ein vielversprechender Ansatz, da viele historische Daten vorliegen und die Berechnung des OPF repetetiv ist. Das DNN hat drei versteckte Ebenen. Für das Parametertuning des DNN wird eine Lagrange-basierte Methode angewendet. Um die Verletzung der Nebenbedingungen zu reduzieren, führen die Autoren einen Algorithmus ein, der das näheste lösungssichere Ergebnis, basierend auf dem Ausgang des DNN, findet. Die Tests erfolgten auf dem IEEE 118-Bus System sowie zwei weiteren Testsystemen mit 1354 und 1888 Bussen. Der Ansatz konnte das Problem wesentlich schneller lösen und hat einen

Optimalitätsverlust von weniger als 0.1 %, ohne die Nebenbedingungen zu verletzen.

Huayi und Zhao wenden in [21] SL auf das DC OPF Problem an. Der Berechnungsaufwand des OPF Problems wird nach den Autoren durch ML wesentlich erleichtert. Das DNN wird aus drei Schichten von CNN und drei Schichten von DBN konstruiert. Um das Modell zu trainieren, werden optimale Lösungen generiert. Die Tests erfolgten auf den IEEE 14, 118 und 300-Bus Systemen. Der Ansatz zeigte eine bis zu 100-mal schnellere Berechnung als ein klassischer Optimierungslöser und erzielte die erforderte Lösungssicherheit.

3.2 Deep Reinforcement Learning Ansätze auf OPF

Cao et al. wenden in [3] den Proximal Policy Optimization (PPO) Algorithmus auf das OPF Problem an. Nach den Autoren ist DRL der aussichtsreichste Ansatz im Bereich von ML für Energiesysteme, da eine optimale Strategie gelernt werden kann, ohne das globale Optimum zu kennen. Das DNN hat drei versteckte Ebenen. Die Tests erfolgten auf einem modifizierten IEEE 33-Bus System mit simulierten Windkraftanlagen und Speicherkapazitäten, um näher die realen Bedingungen abzubilden. Der Ansatz zeigte die beste Kostenminimierung von Leistungsverlust gegenüber DQN und stochastischer Programmierung, lieferte Lösungen in Echtzeit und ist besser anpassbar auf neue, unbekannte Situationen.

Yan und Xu verwenden in [22] für das OPF Problem einen Lagrange-basierten DRL Ansatz. SL kann die vollen Nebenbedingungen nicht immer einhalten, weshalb die Autoren den DRL Ansatz wählen. Die DRL Methode besteht aus Parametertuning des DNN mit DDPG und einer Straffunktion, wenn Nebenbedingungen verletzt werden. Der Ansatz wurde auf dem IEEE 118-Bus getestet. Die Ergebnisse zeigen, dass der OPF in Echtzeit nahezu optimal gelöst werden konnte und dabei alle Nebenbedingungen eingehalten wurden. Der Vergleich zu klassischen Optimierungslösern und einem SL Ansatz zeigt die Vorteile in Geschwindigkeit und Optimalitätsverlust.

Zeng et al. wenden in [23] DRL auf das OPF Problem an. Nach den Autoren ist DRL gut geeignet für den OPF, da DRL die rohen und hochdimensionalen Eingangsdaten direkt verarbeiten kann. Sie implementieren DRL mit zwei verschiedenen Ansätzen: einen DQN Ansatz und einen DDPG Ansatz. Beide Ansätze zeigen effiziente Ergebnisse auf den IEEE 14, 36, 118-Bus Testsystemen. Außerdem führen die Autoren einen False Data Injection Attack Algorithmus ein, der während des Trainings das DNN mit Angriffen auf das Stromnetz stört. Dadurch soll das Modell robuster gegen Sicherheitsangriffe werden, ein Risiko, das

mit der Digitalisierung des Stromnetzes weiter ansteigt. Der vorgeschlagene Ansatz wird mit anderen DRL Ansätzen verglichen und zeigt eine robustere und effizientere Berechnung.

In [24] wenden Sayed et al. DRL auf das AC OPF Problem an. Eine Variante des Soft Actor Critic (SAC) mit konvexen Bedingungen wurde als DRL Algorithmus gewählt. Die Autoren beschreiben, dass der SL Ansatz im Vergleich zu DRL im Nachteil ist, da SL auf einen Datensatz vertraut, der aus optimalen Lösungen bestehen muss, was aber nicht garantiert werden kann. DRL hingegen kann selbstständig die optimale Lösung finden. Um die Lösungssicherheit zu garantieren, wird der Ausgang des DNN in eine konvexe Sicherheitsebene übergeben. Der Ansatz lieferte optimale Ergebnisse in einer simulierten Umgebung bis zu 34-mal schneller als klassische Optimierungslöser, ohne die Nebenbedingungen des OPF zu verletzen und konnte in Echtzeit angewendet werden.

Nie et al. wenden in [25] DRL auf ein OPF Problem an. Die Autoren beschreiben, dass DRL ein vielversprechender Ansatz für dynamische und komplexen Umgebungen wie OPF ist. Für die DRL Implementierung wird der Twin Delayed Deep Deterministic Policy Gradient (TD3) Algorithmus gewählt. Die Tests erfolgten auf einem modifizierten IEEE 30-Bus System mit simulierten elektrischen Autos und Ladeanschlüssen. Der vorgeschlagene Ansatz wird mit einem DQN und DDPG Ansatz verglichen und weist eine bessere Leistung auf. Nach den Autoren ist der Ansatz skalierbar auf komplexere und größere Systeme.

3.3 Deep Reinforcement Learning mit Supervised Learning Ansätze auf OPF

Zhou et al. wenden in [26] DRL auf das AC OPF Problem an. Die Autoren beschreiben, dass der SL Ansatz nicht die Lösungssicherheit mit allen Nebenbedingungen garantieren kann. Durch dieses Problem und inspiriert durch andere Arbeiten in dem Bereich, entwickelten sie einen DRL Ansatz. Als DRL Algorithmus wird PPO eingesetzt. Das DNN mit drei versteckten Ebenen nähert die Strategiefunktion an. Der Aktions- und Zustandsraum des Agenten ist sehr groß, weshalb das Training komplex werden kann. Imitation Learning soll bei dem Training helfen, indem ein unabhängiges DNN mit SL auf generierten Daten trainiert wird und als initiale Strategie für das DNN benutzt wird. Die Tests erfolgten auf den IEEE 14-Bus und Illinois 200-Bus Energiesystemen und zeigen, dass der Ansatz mindestens siebenmal schneller ist als klassische Optimierungslöser mit niedrigem Optimalitätsverlust. Weiterhin ist die Methode auch unter wechselnden Netzwerkstopologien effizent.

Guo et al. wenden in [27] DRL auf das AC OPF Problem an. Sie verwenden Imitation Learning, um die Parameter des DNN zu initialisieren und das Training zu verkürzen. Die Autoren beschreiben, dass DRL besser geeignet ist als SL, da der benötigte Datensatz bei SL möglicherweise nicht alle Zustände enthält, wohingegen ein gut trainierter Agent diese Zustände besser und eigenständig lernen kann. Als DRL Algorithmus wird PPO eingesetzt. Auf dem IEEE 30-Bus Testsystem wurde der Ansatz getestet und die Ergebnisse zeigen, dass das Problem in Echtzeit gelöst wird, ohne die Nebenbedingungen zu verletzen.

3.4 Zusammenfassung

Allgemein wurden SL und DRL auf verschiedene OPF Probleme wie DC OPF und AC OPF angewendet. SL ist der klassische Ansatz für ML und erzielt in den Arbeiten eine wesentliche Steigerung in der Berechnungszeit im Vergleich zu herkömmlichen Optimierungslösern, während die Ergebnisse nahezu optimal sind [2] [16] [17] [18] [19] [20] [21]. Ein Nachteil ist die Lösungssicherheit der Ergebnisse, denn die DNN liefern nicht immer eine erlaubte Lösung des OPF. Dafür nutzen viele Arbeiten eine Vor- oder Nachbearbeitung der Daten, damit die Lösungssicherheit garantiert werden kann [2] [16] [17] [18] [19] [20]. Auch Ansätze mit DRL liefern effiziente Berechnungen und nahezu optimale Lösungen [26] [3] [22] [23] [24] [27] [25]. Einige Autoren der DRL Arbeiten argumentieren, dass der Ansatz besser für OPF geeignet ist, da der Erfolg von SL von den Trainingsdaten abhängig ist, die möglicherweise nicht alle Zustände und optimale Lösungen enthalten [24] [27]. Ein möglicher Nachteil von DRL ist der Trainingsaufwand, da der Zustands- und Aktionsraum des Agenten in der OPF Umgebung sehr groß ist, weshalb einige Autoren DRL mit SL kombinieren [26] [27]. Um die Lösungssicherheit zu garantieren, wurde in [24] ebenfalls ein Nachbearbeiten der Daten angewendet. Bei beiden Ansätzen wurden oft DNN mit drei versteckten Ebenen gewählt [2] [17] [20] [21] [26] [3]. Die Tests erfolgten auf verschiedenen IEEE Systemen [2] [16] [17] [18] [19] [20] [21] [26] [3] [22] [23] [27] [25] oder anderen Systemen mit mehr Bussen [19] [20]. Die SL, DRL und kombinierten Ansätze erzielen eine wesentliche Steigerung in der Berechnungszeit und können die Nebenbedingungen einhalten, während der Optimalitätsverlust niedrig ist. Aufgrund dieser Arbeiten kann die Frage gestellt werden, ob SL oder DRL besser für das OPF Problem geeignet ist. Ein Vergleich wurde bereits in [22] hergestellt und zeigt, dass DRL bessere Ergebnisse erzielt als SL. Allerdings wollen die Autoren ihren DRL Ansatz präsentieren, weshalb der Vergleich möglicherweise nicht unbeeinflusst ist. Es fehlt ein neutraler Vergleich, bei dem das gleiche DNN eingesetzt wird und sich nur das Training unterscheidet. Das soll in dieser Arbeit erreicht werden.

4 Methodik

In diesem Kapitel wird die Methodik der Arbeit vorgestellt. Dafür wird zunächst die Problemstellung in Abschnitt 4.1 definiert. Da SL und DRL einige Unterschiede haben, werden diese in Abschnitt 4.2 näher beleuchtet und danach beschrieben, auf welchen Grundlagen der Vergleich stattfinden soll. Die verwendeten Metriken zum Training und Vergleich der Modelle werden in Abschnitt 4.3 vorgestellt. Schließlich wird dann die Implementierung in Abschnitt 4.4 und Abschnitt 4.5 vorgenommen, wobei für DRL der DDPG Algorithmus verwendet wird. Zum Schluss werden in Abschnitt 4.6 die Kriterien für den Vergleich aufgestellt. Nach dem Training der Modelle werden Hyperparameter optimiert, die in Abschnitt 4.7 erklärt werden, um dann im nächsten Kapitel die Ergebnisse zu betrachten.
Die Implementierung von SL, DRL und der Umgebung wurden mit der Programmiersprache Python[1] vorgenommen. Python wird typischerweise für ML eingesetzt und hat eine große Auswahl an Bibliotheken.

4.1 Problemstellung

In dieser Arbeit wird ein Blindleistungsmarkt als Problemstellung verwendet, der mit SL und DRL optimiert werden soll. In einem Blindleistungsmarkt wird Blindleistung genutzt, um die Verluste bei der Übertragung von Wirkleistung zu minimieren. Dabei muss die Blindleistung der Stromerzeuger bzw. Generatoren reguliert werden. Das Ziel ist die Kosten für die Blindleistung und für den Verlust zu minimieren. Weiterhin müssen die Nebenbedingungen

- Spannungsbänder $c_{Spannung}$
- Netz-/Trafo-Last c_{Last}
- minimale/maximale Blindleistung $c_{Blindleistung}$
- Eingeschränkter Blindleistungsfluss über Slack-Bus $c_{Slack-Bus}$

eingehalten werden. [28]

[1] `https://www.python.org/`, Python Version: 3.7.0, aufgerufen am 11.04.2023

Energiesystem

Der Blindleistungsmarkt verwendet ein bestimmtes elektrisches Energiesystem aus der Simbench[2] Bibliothek, welcher ein umfangreicher Benchmark-Datensatz für die Simulation von Stromnetzen aus Deutschland ist. Dieser enthält Modelle von verschiedenen Arten von Stromnetzen, einschließlich Übertragungs- und Verteilungsnetzen, sowie unterschiedliche Arten von erneuerbaren Energiequellen wie Windkraftanlagen und Photovoltaikanlagen. Außerdem enthält der Datensatz Zeitreihendaten von Erzeugungseinheiten und Lasten über ein Jahr, die nützlich für die Generierung von Zuständen sind und auch in dieser Arbeit verwendet werden. Für die Blindleistungsmarkt-Umgebung wird das Energiesystem mit dem Simbench Code „1-LV-urban6–0-sw“ eingesetzt. Das Energiesystem ist ein Niederspannungsnetz mit urbanem Charakter und hat fünf Photovoltaikanlagen als Generatoren. Weiterhin wurden die Bibliotheken Pandapower[3] und OpenAI Gym[4] verwendet. Mit Pandapower können elektrische Systeme modelliert werden und Berechnungen wie OPF ausgeführt werden. OpenAI Gym definiert die grundlegende Struktur für die Umgebung und den Agenten und bietet nützliche Methoden.

Im Folgenden werden die Zustände, Aktionen und Belohnungen definiert, welche wichtig für den DRL Agenten sind.

Zustand

Der Zustand s enthält kontinuierliche Werte über den aktuellen Systemzustand als Tupel. Dies wird mit der Formel

$$s = (\vec{P_L}, \vec{Q_L}, \vec{P_G}, \vec{p_G^Q}) \tag{4.1}$$

beschrieben, wobei $\vec{P_L}$ für den Vektor der Wirkleistungen aller Lasten, $\vec{Q_L}$ für den Vektor der Blindleistungen aller Lasten, $\vec{P_G}$ für den Vektor der Wirkleistungen der Generatoren und $\vec{p_G^Q}$ für den Vektor der Blindleistungspreise der Generatoren steht. [28]

[2] `https://pypi.org/project/simbench/`, Simbench Version: 1.3.0, aufgerufen am 11.04.2023
[3] `https://pypi.org/project/pandapower/`, Pandapower Version: 2.2.0, aufgerufen am 11.04.2023
[4] `https://pypi.org/project/gym/`, Gym Version: 0.25.2, aufgerufen am 11.04.2023

Aktion

Mit einer Aktion a wird die Blindleistung der Generatoren angepasst. Die kontinuierlichen Werte werden ebenfalls in einem Tupel abgespeichert.

$$a = (\vec{P_G}) \tag{4.2}$$

Dabei ist $\vec{P_G}$ der Vektor für die Blindleistung der Generatoren. [28]

Belohnung und Bestrafung

Die Belohnung r setzt sich aus der Minimierung der Kosten (die Zielfunktion) sowie einer Bestrafung bei Verletzung der Nebenbedingungen zusammen. Die Kosten werden mit der polynomiellen Kostenfunktion von Pandapower berechnet und beinhalten die Kosten für Wirkleistung und Blindleistung. Bei Verletzung der Nebenbedingungen wird eine Bestrafung berechnet.

$$r = -costs + p(c_{Spannung}, c_{Last}, c_{Blindleistung}, c_{Slack-Bus}) \tag{4.3}$$

In der Formel sind *costs* die Kosten (Zielfunktion) und die Funktion p die Verletzung der Nebenbedingungen, wobei jede Nebenbedingung einen eigenen Bestrafungsfaktor besitzt, der jeweils unterschiedlich gesetzt sein kann. Es gilt: $r < 0$ und je näher r an 0 ist, desto besser ist die Belohnung. [28]

4.2 Unterschiede und Gemeinsamkeiten der Trainingsverfahren

Die Trainingsverfahren SL und DRL sind unterschiedlich und dadurch schwer zu vergleichen. Im Folgenden werden die Unterschiede in Unterabschnitt 4.2.1 näher beleuchtet, um in Unterabschnitt 4.2.2 die Grundlage für den Vergleich zu schaffen.

4.2.1 Unterschiede in den Trainingsverfahren

Datensatz

SL benötigt einen beschrifteten Datensatz, der die Zielwerte enthält, die das Modell lernen soll. Dieser muss vorher erzeugt werden. DRL interagiert mit der Umgebung, um Daten zu sammeln und braucht daher keinen beschrifteten Datensatz.

Interaktion mit der Umgebung

Die Interaktion mit der Umgebung ist für DRL essenziell. Der DRL Agent kann in der Umgebung Aktionen ausführen und sie dadurch verändern. Das ist bei SL nicht möglich, da keine Interaktion mit der Umgebung stattfindet.

Lernverfahren

Mit SL wird versucht, das Muster in dem Datensatz zu lernen. DRL hingegen lernt nach Versuch und Irrtum durch das Interagieren mit der Umgebung. Außerdem muss der Agent Belohnungen für seine Aktionen bekommen, was bei SL nicht berücksichtigt wird. DRL findet die optimalen Aktionen also durch Belohnungen und Interaktion mit der Umgebung, während SL die optimalen Aktionen bereits durch den Datensatz gegeben hat.

Ziel des Trainings

Weiterhin ist das Ziel der Trainingsverfahren unterschiedlich: SL versucht, den Fehler zwischen Vorhersage und Zielwert zu minimieren, während der DRL Agent versucht, die Belohnung zu maximieren. Außerdem muss der DRL Agent eine Balance zwischen Exploration und Ausbeutung einer guten Strategie halten.

Epochen und Schritte

Die Architektur der Trainingsverfahren ist ebenfalls unterschiedlich. In einer *Epoche* bei SL lernt das Modell einmal den ganzen Datensatz. Dabei wird der Datensatz in Batches aufgeteilt. Bei DRL hingegen werden in einem *Schritt* die Umgebungsdaten als Batch verarbeitet. Epochen und Schritte sind demnach nicht vergleichbar, da in einer Epoche viel mehr Informationen verbarbeitet werden (der ganze Datensatz gegenüber einem Batch von Umgebungsdaten).

4.2.2 Grundlage für den Vergleich

Um den Vergleich der Trainingsverfahren möglichst fair zu gestalten, werden die folgenden Kriterien gleich implementiert. Für die Implementierung von DNN eignet sich die PyTorch Bibliothek[5], die in dieser Arbeit verwendet wird.

[5] `https://pypi.org/project/torch/`, Torch Version: 1.13.1, aufgerufen am 11.04.2023

Neuronales Netz

Die beiden Trainingsverfahren benutzen das gleiche DNN. Dafür wurde die Klasse `DDPGActorNet` aus [29] verwendet. SL verwendet das DNN, um das Muster in dem Datensatz zu lernen, während DRL das DNN für den Aktor und den Ziel-Aktor verwendet. Die Implementierung von DRL mit DDPG erfordert noch ein zwei weitere DNN für den Kritiker und Ziel-Kritiker, wofür sich die Klasse `DDPGCriticNet` eignet [29]. Der Aufbau von `DDPGCriticNet` ist ähnlich mit `DDPGActorNet`. Die DNN haben drei versteckte Ebenen mit jeweils 256 Neuronen pro versteckter Ebene und nutzt die Aktivierungsfunktion Tangens hyperbolicus $\tanh(x)$.

Optimierer

Ein Optimierer ist ein Algorithmus, der verwendet wird, um die Parameter eines Modells zu optimieren, sodass die Leistung des Modells auf den Trainingsdaten verbessert wird. In dieser Arbeit wird der Adam-Optimierer für beide Trainingsverfahren eingesetzt. Der Adam-Optimierer ist ein häufig verwendeter Gradientenabstiegs-Algorithmus und eine Verbesserung des klassischen SGD. Adam ist besonders nützlich für komplexe Optimierungsprobleme mit großen Datenmengen und vielen Parametern. Es ist robust gegenüber schwankenden Lernraten und Ausreißern, was zur schnelleren Konvergenz und besseren Modellleistungen führt. Die Lernrate beeinflusst, wie groß die Schritte des Optimierers sind beziehungsweise wie schnell die Gradienten aktualisiert werden. Da die Lernrate die Konvergenz beeinflusst, kann dieser Parameter auf die Modelle angepasst werden und muss nicht gleich sein.

Trainingsalgorithmus

Als Trainingsalgorithmus wird für beide Trainingsverfahren der Backpropagation (BP) Algorithmus eingesetzt, um die Gewichte der Neuronen entsprechend anzupassen und um die Leistung des DNN zu verbessern.

Verlustfunktion

Als Verlustfunktion wird der Mean Squared Error (MSE) eingesetzt. Dieser wird nach der Formel

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(A_i - F_i)^2 \tag{4.4}$$

berechnet, bei der n die Anzahl an Beobachtungen ist, A_i der Zielwert und F_i der vorhergesagte Wert. Der MSE berechnet den erwarteten quadratischen Abstand des vorhergesagten

Werts F_i zum Zielwert A_i.
SL berechnet mit der Funktion den Fehler zwischen Zielwert vom Datensatz A_i und der vorhergesagte Wert vom Modell F_i. DRL verwendet im DDPG Algorithmus die Verlustfunktion zur Berechnung des Fehlers im Kritiker Netz, indem der Fehler von den vorhergesagten Q-Werten F_i und den Ziel Q-Werten A_i berechnet wird.

Hardware

Für das Training bei SL und DRL sowie der Datensatzgenerierung wurde die gleiche Hardware verwendet, damit die Zeit gemessen werden kann und fair verglichen wird. Das verwendete System ist das High Perfomance Cluster der Universität Oldenburg. Dabei wurde der Rechenknoten MPC-Big[6] des Clusters CARL benutzt, der die folgenden Spezifikationen hat:

- 2x Intel Xeon CPU E5-2667 v4 8C mit 3.2GHz
- 16x 32GB TruDDR4 modules @2400MHz (8GB RAM wurde verwendet)
- 2x NVIDIA GTX 1080 (eine Grafikkarte wurde verwendet)

Bei der Ausführung des Vergleichs von SL und DRL wurde eine andere Hardware mit folgenden Eigenschaften verwendet:

- Intel i7-4790k@4GHz
- 16GB DDR4 RAM
- NVIDIA GTX 1060 6GB

Trainingszeit

Die Trainingszeit kann entweder auf ein Maximum festgelegt werden oder die Modelle werden so lange trainiert, bis sie konvergieren. In dieser Arbeit werden die Modelle bis zur Konvergenz trainiert. Damit wird der Fokus mehr auf die Lösungsqualität gelegt, denn die Trainingsverfahren haben so viel Zeit wie nötig, um ein gutes Trainingsergebnis zu erzielen. Die Modelle sollen nicht von einer Maximalzeit gekappt werden.

[6] `http://wiki.hpcuser.uni-oldenburg.de/index.php`, aufgerufen am 11.04.2023

Berechnung von Zuständen und Aktionen

Bei der Implementierung von DRL müssen Variablen wie Zustände, Aktionen, Belohnungen usw. berechnet werden. Um den SL Datensatz zu generieren, müssen Zustände und optimale Aktionen berechnet werden (siehe Unterabschnitt 4.4.1). Dafür können die Methoden von DRL genutzt werden. Damit haben die Trainingsverfahren die gleiche Schnittstelle zur Berechnung von Zuständen und (optimalen) Aktionen.
Ein zufälliger Zustand wird mit dem folgenden Programm erzeugt:

```
env = QMarketEnv()

state = env.reset(test=False)
```

Listing 4.1: Zustand

Dafür wird die `reset` Methode der Umgebung `env` aufgerufen. Die Umgebung wird mit `QMarketEnv` für den Blindleistungsmarkt initialisiert. Mit `reset` wird der Zustand zufällig und mit etwas Rauschen zurückgesetzt, d.h. der Zustand ist zufällig aus dem Simbench Datensatz ausgewählt und die Werte werden zufällig noch etwas verändert. Der Simbench Datensatz hat 35136 Datenpunkte, was einem Datenpunkt alle 15 Minuten in einem Schaltjahr entspricht (4 * 24 * 366 = 35136). Dabei kann `test=False` oder `test=True` gesetzt werden, je nachdem, ob Testdaten verwendet werden sollen oder nicht. Die Testdaten können zum Testen der Modelle verwendet werden, da diese bei `test=False` nicht verwendet werden und somit die Modelle die Testdaten im Training nicht gesehen haben.
Um eine optimale Aktion zu berechnen, muss zunächst ein optimaler Zustand der Umgebung hergestellt werden. Dieser wird mit dem Pandapower OPF Löser berechnet. Der Aufruf sieht wie folgt aus:

```
pp.runopp(net)
```

Listing 4.2: Pandapower OPF Aufruf

Die `runopp` Methode von der Bibliothek Pandapower `pp` startet eine OPF Berechnung auf dem Simbench Netz `net`. Der OPF könnte unter Umständen nicht konvergieren, sodass die Methode eine `OPFNotConverged` Exception wirft. Wenn es aber keine Exception gab, dann kann mit dem folgenden Programm die optimalen Aktionen geholt werden:

```
action = env.get_current_actions()
```

Listing 4.3: Optimale Aktionen

Die Aktion `action` ist nach dem Pandapower OPF Löser die optimale Aktion für den aktuellen Zustand der Umgebung. [28]

Skalierer

Um die Zustandsdaten zu skalieren und somit eine bessere Lernfähigkeit für das Modell zu erreichen, wird ein Skalierer eingesetzt. Dieser Skalierer ist bei beiden Trainingsverfahren gleich und heißt `ScalerObs` [29]. Damit werden die Werte der Umgebung auf $[-1, 1]$ skaliert. Die Verwendung sieht wie folgt aus:

```
scaler = ScalerObs(env.observation_space.low, env.observation_space.high)

state = env.reset()
state = scaler(state)
```

Listing 4.4: Zustand

Der Skalierer `scaler` wird mit den niedrigsten und höchsten Werten der Zustände von `env` initialisiert. Wenn ein Zustand `state` hergestellt wird kann mit dem Aufruf von `scaler` der Zustand skaliert werden.

4.3 Metriken

Zum Vergleich der Modelle werden die Metriken Mean Absolute Percentage Error (MAPE) und Root Mean Squared Error (RMSE) verwendet. Diese Metriken beschreiben, wie die Abweichung des wahren Wertes beziehungsweise Zielwerts A_i von dem vorhergesagten Wert F_i ist. n repräsentiert die Anzahl an Experimenten. Der MAPE wird nach der Formel

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{A_i - F_i}{A_i} \right| \times 100 \tag{4.5}$$

berechnet. Das Ergebnis wird mit 100 multipliziert, um es zu einer Prozentzahl zu konvertieren. Der RMSE wird nach der Formel

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(A_i - F_i)^2}{n}} \tag{4.6}$$

berechnet. Je niedriger MAPE und RMSE sind, desto geringer ist die Abweichung der Vorhersage F_i vom Zielwert A_i.

Schwächen der Metriken

Wenn der Zielwert A nahe null liegt, wird der MAPE sehr groß. Die Werte von A dürfen niemals null sein, denn sonst wird durch null geteilt, was nicht erlaubt ist.
Der RMSE ist empfindlicher gegen Ausreißer als der MAPE, da die Differenz von A und F quadriert wird. [30]

4.4 Implementierung Supervised Learning

In diesem Abschnitt wird die Implementierung von SL vorgenommen. Dafür wird zunächst ein Datensatz benötigt, der in Unterabschnitt 4.4.1 generiert wird. Mit dem Datensatz wird das Modell in Unterabschnitt 4.4.2 trainiert.

4.4.1 Datensatzgenerierung

Ein Datensatz zum Trainieren von neuronalen Netzen muss Eingangsdaten und Zieldaten enthalten. Da Datensätze für OPF nicht weit verbreitet sind und für diesen Anwendungsfall nicht vorliegen, muss der Datensatz in dieser Arbeit generiert werden.
Der DRL Agent entscheidet sich in einem Zustand für eine Aktion. Im SL Modell soll dies ähnlich funktionieren. Das Modell soll einen Zustand als Eingang erhalten und daraufhin eine Vorhersage für eine Aktion machen. Daher muss der Datensatz Zustände und Aktionen enthalten. Die Zustände müssen zufällig und gestreut generiert werden, um viele mögliche Zustände im Datensatz abzudecken. Für jeden Zustand wird eine optimale Aktion berechnet. Mit den Methoden aus Unterabschnitt 4.2.2 können die Zustände und Aktionen generiert werden. Weiterhin muss die Datensatzgröße mit dem Hyperparameter `DATASET_SIZE` festgelegt werden. Allgemein gilt, dass je mehr Daten vorhanden sind, desto mehr kann das Modell lernen. Allerdings ist der Simbench Datensatz auf 35136 Datenpunkte begrenzt. Hyperparameter können variiert werden, sodass das Modell eine bessere Leistung erzielt.
Um den Datensatz zu generieren, wurde das folgende Programm geschrieben:

```
import csv
from time import time
from thesis_envs import QMarketEnv
from agent import ScalerObs

env = QMarketEnv()
scaler = ScalerObs(env.observation_space.low, env.observation_space.high)
DATASET_SIZE = ...
start = time()
index = 0
```

```
for index in range(DATASET_SIZE):
    input = env.reset(test=False)
    input = scaler(input)

    success = env.optimal_power_flow()
    if not success: # opf not converged
        index -= 1
        continue

    target = env.get_current_actions()

    with open("inputs.csv", "a") as f:
        writer = csv.writer(f)
        writer.writerow(input)
    with open("targets.csv", "a") as f:
        writer = csv.writer(f)
        writer.writerow(target)

    index +=1

end = time()
execution_time = end - start
[...]
```

Listing 4.5: Datensatzgenerierung

Zunächst wird die Umgebung `env` und der Skalierer `scaler` initialisiert. Danach wird die Datensatzgröße mit `DATASET_SIZE` festgelegt und die Zeit gestartet. Die `for` Schleife iteriert über die Länge des Datensatzes. Mit der `reset` Methode wird ein neuer Zustand generiert, der dann mit dem `scaler` skaliert wird und in der Variable `input` gespeichert wird. Dabei muss `test=False` gesetzt werden, damit keine Testdaten verwendet werden. Um die optimalen Aktionen zu dem Zustand zu berechnen, wird erst mit der `optimal_power_flow` Methode der OPF ausgeführt. Die `optimal_power_flow` Methode ruft die Pandapower OPF Methode auf und gibt wahr oder falsch zurück, je nachdem, ob der OPF konvergiert ist oder nicht. Falls der OPF nicht konvergiert, steht `success` auf falsch und die Schleife wird erneut ausgeführt. Wenn das nicht passiert, werden mit `get_current_actions` die optimalen Aktionen in der Variable `target` abgespeichert. Mit den `writer` von der Python CSV Bibliothek werden die Variablen in einer CSV-Datei gespeichert. Am Ende wird die Zeit gestoppt und die Ausführungszeit kann berechnet werden.

Datensatzvorbereitung

Nachdem der Datensatz generiert wurde, muss der Datensatz geladen und vorbereitet werden. Bei SL bietet es sich an, den Datensatz in einem Trainings-Datensatz (Train) und einem Validierungs-Datensatz (Valid) aufzuteilen. Train wird zum Trainieren des Modells benutzt. Mit Valid wird das Modell, nachdem es den Train-Datensatz einmal durchlaufen

hat, validiert. Die Validierungsdaten sind die Daten, die das Modell noch nicht gelernt hat und ist daher eine Metrik für die Leistung des Modells auf ungesehene Daten. Der finale Test des Modells erfolgt auf Testdaten, die auch für das DRL Modell genutzt werden (siehe Abschnitt 4.6). Mit dem folgenden Programm wurde der Datensatz vorbereitet:

```
import numpy as np
import torch.utils.data.dataset as dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split

BATCH_SIZE = ...

inputs = np.genfromtxt("inputs.csv", delimiter=',')
targets = np.genfromtxt("targets.csv", delimiter=',')

data = Dataset(inputs, targets)

# 80 % train, 20% valid
train_size = int(0.80 * len(data))
valid_size = len(data) - train_size

train, valid = random_split(data, [train_size, valid_size])

train_dl = DataLoader(train, batch_size=BATCH_SIZE)
valid_dl = DataLoader(valid, batch_size=BATCH_SIZE)
```

Listing 4.6: Datensatzvorbereitung

Zunächst werden die Datensätze mit der `genfromtxt` Funktion von NumPy geladen. Danach werden die Datensätze in ein PyTorch `Dataset` Objekt konvertiert, damit auf den Datensatz in einem Objekt zugegriffen werden kann. Als nächstes wird der Datensatz in einen Trainings- und einen Validierungsdatensatz aufgeteilt. Valid enthält 20% der Daten, während der Rest für Train verwendet wird. Dafür wird die `random_split` Funktion von PyTorch verwendet, welche den Datensatz zufällig aufteilt. Schließlich werden die Datensätze in `DataLoader` Objekte konvertiert. Diese Objekte ermöglichen es, die Daten in Batches aufzuteilen und für das Training eines neuronalen Netzes zu verwenden. Ein Batch ist dabei ein kleiner Teil des Datensatzes. Eine hohe `BATCH_SIZE` bedeutet, dass viele Eingangsdaten in einem Schritt verarbeitet werden. Die Batchgröße gehört zu den Hyperparametern, welche die Leistung des Modells beeinflussen. Im nächsten Schritt werden die Dataloader für das Training verwendet.

4.4.2 Training

Nachdem der Datensatz generiert, geladen und vorbereitet wurde, kann das Training des Modells beginnen. Dafür wurde das folgende Programm geschrieben:

```
import torch
from time import time
from thesis_envs import QMarketEnv, EcoDispatchEnv
from agent import ScalerObs
from utils import *

EPOCHS = ...
LEARNING_RATE = ...
env = QMarketEnv()
model = initialize_model(env)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
scaler = ScalerObs(env.action_space.low, env.action_space.high)
start = time()

for epoch in range(EPOCHS):

    [...]
    train_epoch_loss = []
    train_epoch_mape = []

    # Train
    model.train()
    for _, (inputs, targets) in enumerate(train_dl):
        inputs = inputs.to(DEVICE)
        targets = targets.to(DEVICE)

        optimizer.zero_grad()

        output = model(inputs)

        loss = criterion(output, targets)
        train_epoch_loss.append(loss.item())

        train_epoch_mape.append(mape(outputs, targets))

        loss.backward()

        optimizer.step()

    [...]
    end = time()
    execution_time = end - start
    [...]
```

Listing 4.7: Training

Im ersten Schritt werden die Hyperparameter `EPOCHS` und `LEARNING_RATE` festgelegt, womit die Anzahl an Epochen beziehungsweise die Lernrate bestimmt wird. Außerdem wird die Umgebung `env` mit `QMarketEnv` ausgewählt sowie das Modell `model` initalisiert. Die Methode `initialize_model` erstellt das DNN mit den Parametern der Umgebung (Grö-

ße des Zustands- und Aktionsraum). Danach wird noch die Verlustfunktion sowie der Optimierer definiert, also der MSE-Verlust und Adam-Optimierer. Der Optimierer erhält außerdem die Lernrate. Weiterhin wird der Skalierer `scaler` initialisiert und die Zeit abgespeichert, um die Trainingszeit zu messen. Nun kann das Training beginnnen, indem eine `for` Schleife über die festgelegte Epochenzahl mit `EPOCHS` iteriert. Im ersten Schritt des Trainings werden einige Variablen wie `train_epoch_loss` und `train_epoch_mape` initialisiert, um den Verlust und MAPE in jedem Trainingsdurchlauf zu berechnen. Das Modell wird mit der `train` Methode in den Trainingsmodus gesetzt. Damit wird sichergestellt, dass die Gradienten im Training aktualisiert werden. Mit dem Dataloader `train` werden die Eingangsdaten `inputs` und Zieldaten `targets` entpackt. Damit die Berechnung auf der Grafikkarte ausgeführt wird, werden die Daten mit der `to` Methode auf `DEVICE` gesetzt, der die Graphic Processing Unit (GPU) als Zeichenkette repräsentiert.
Als nächstes werden die Gradienten auf null gesetzt, um sicherzustellen, dass beim Berechnen der Gradienten nur die aktuellen Batches berücksichtigt werden. Dann wird eine Vorwärtsdurchlauf durch das Modell durchgeführt, wobei die `inputs` als Eingabe verwendet werden, um die Ausgabe des Modells `outputs` zu erhalten. Mit den `outputs` des Modells kann der Verlust `loss` berechnet werden, weshalb ein Aufruf auf `criterion` mit `outputs` und den Zieldaten `targets` gestartet wird. Der Verlust wird in `train_epoch_loss` gespeichert. Zur Berechnung des MAPE wird eine `mape` Methode aufgerufen und das Ergebnis in `train_epoch_mape` gespeichert.
Das Ziel des Trainings besteht darin, den Verlust zu minimieren, weshalb BP auf dem Verlust `loss` ausgeführt wird, um die Gradienten des Fehlers bezüglich der Gewichte des Modells zu berechnen. Im letzten Schritt müssen die Gewichte mit der `step` Funktion des Optimierers aktualisiert werden. Die Schleife iteriert über den gesamten Trainingsdatensatz mit den aufgeteilten Batches.

Validierung

Nachdem der Train-Datensatz einmal durchlaufen wurde, kann das Modell validiert werden. Dabei ist das Vorgehen wie beim Training und wird mit dem folgenden Programm erklärt:

```
import torch
from time import time
from thesis_envs import QMarketEnv, EcoDispatchEnv
from agent import ScalerObs

best_valid_mape = float('inf')
[...]
for epoch in range(EPOCHS):
```

```
# Train
[...]

valid_epoch_loss = []
valid_epoch_mape = []

# Validate
model.eval()
for i, (inputs, targets) in enumerate(valid):
    inputs = inputs.to(DEVICE)
    targets = targets.to(DEVICE)

    outputs = model(inputs)

    loss = criterion(outputs, targets)
    valid_epoch_loss.append(loss.item())

    valid_epoch_mape.append(mape(outputs, targets))

[...]

if best_valid_mape > valid_epoch_mape:
    best_valid_mape = valid_epoch_mape
    torch.save(model.state_dict(), "model.pt")

[...]
```

Listing 4.8: Validierung

Der relevante Programmausschnitt befindet sich nach dem Training des Modells während einer Epoche. Da das Modell noch im Trainingsmodus ist, wird es in den Evalierungsmodus gesetzt, um sicherzustellen, dass keine Gradienten aktualisiert werden. Es wird einmal über den gesamten Validierungsdatensatz iteriert. Wie beim Training müssen die Daten auf die Berechnung der GPU eingestellt werden. Auch der Vorwärtsdurchlauf sowie die Berechnung des Verlusts und MAPE sind identisch.
Nach dem Validieren wird überprüft, ob der MAPE sich verbessert (verringert) hat. Dafür wird vor dem Training `best_valid_mape` auf unendlich gesetzt und immer der beste MAPE abgespeichert. Falls der MAPE sich verringert, wird das Modell abgespeichert. Somit ist nach dem Training immer das beste Modell bezüglich des Validierungs-MAPE abgespeichert. Bei DRL wurde es identisch implementiert.

4.5 Implementierung Deep Reinforcement Learning

Der DRL Teil dieser Arbeit basiert auf [29], welcher den DDPG Algorithmus wie in [31] implementiert. Es wurden kleine Änderungen vorgenommen, um das Modell während des Trainings abzuspeichern (identisch wie bei SL).

4.6 Vergleichskriterien

Zum Vergleich der Trainingsverfahren werden vorher folgende Vergleichskriterien aufgestellt:

Aufwand der Implementierung

Der Aufwand der Implementierung ist kein wichtiges Kriterium, soll aber beachtet werden. Wenn ein Trainingsverfahren komplexer ist, jedoch bessere Ergebnisse liefert, sollte es den Aufwand wert sein. Hierbei muss beachtet werden, dass die DRL Implementierung von einer anderen Arbeit genutzt wurde (siehe Abschnitt 4.5) und der SL Teil selbst implementiert wurde.

Trainingszeit

Die Trainingszeit soll verglichen werden, denn beim Training eines Modells werden viel Zeit und Ressourcen verbraucht, die durch ein schnelleres und effizienteres Trainingsverfahren möglicherweise eingespart werden können. Dabei ist es wichtig, dass bei SL die Zeit der Datensatzgenerierung dazugezählt wird, denn ohne die Datensatzgenerierung kann das Modell nicht trainiert werden. Weiterhin ist durch den Datensatz schon definiert, was das SL Modell trainieren soll, während der DRL Agent das im Training selbst herausfinden muss. Die Zeit wird vor den entsprechenden Programmabschnitten gestartet und danach gestoppt. Da die gleiche Hardware für beide Trainingsverfahren genutzt wird, ist die Zeit vergleichbar. Bei DRL werden während des Trainings regelmäßig Tests durchgeführt, um die Leistung des Modells zu messen und um das beste Modell zu speichern. Die Testzeit wird daher auch gemessen.

Berechnungszeit

Es soll weiterhin die Berechnungszeit der Modelle für eine Aktion verglichen werden. Dies ist eine wichtige Metrik für den Einsatz in ein Echtzeitsystem. Da die Modelle die gleiche Klasse für das DNN verwenden, kann erwartet werden, dass die Zeiten ähnlich sind. Die Berechnugnszeit des Pandapower OPF Lösers soll auch verglichen werden, um die Zeitersparnis von neuronalen Netzen zu verdeutlichen. Denn SL und DRL sollten wesentlich schneller in der Berechnung sein, wie es einige Arbeiten schon gezeigt haben [19](siehe Kapitel 3. Um diesen Vergleich durchzuführen, wird erst ein zufälliger Zustand erzeugt. Danach wird die Zeit gestartet und das jeweilige Modell macht eine Vorhersage anhand des erzeugten Zustands. Dann wird die Zeit gestoppt. Da die gleiche Hardware genutzt wird, ist diese Zeit auch vergleichbar.

Lösungsqualität Aktionen, Belohnungen und Zielfunktion

Die Lösungsqualität der Modelle ist die wichtigste Metrik. Dabei wird zwischen Lösungsqualität der Aktionen, Belohnungen und Zielfunktionswerte unterschieden. Bei der Lösungsqualität der Aktion wird der MAPE und RMSE von der optimalen Aktion A und der vorhergesagten Aktion des Modells F berechnet. Dafür muss vorher ein zufälliger Zustand mit den Testdaten (`test=True`) generiert werden, anhand dessen der Pandapower OPF Löser eine optimale Aktion berechnet und je SL und DRL eine Aktion berechnen. Da die Aktion noch nicht aussagt, wie gut das OPF Problem gelöst wurde, ist diese Metrik weniger wichtig. Um eine Belohnung zu erhalten, werden die drei Aktionen mit der `step` Funktion jeweils auf den Zustand angewendet und die Belohnung gespeichert, wobei die Belohnung der optimalen Aktion als optimale Belohnung gilt. Allerdings gibt es in manchen Fällen eine Bestrafung auf die Belohnung, die eventuell nicht genug mit den Bestrafungsfaktoren bestraft wird. Denn der DRL Agent könnte über Zeit lernen, eine Bestrafung zu dulden, weil sie nicht so stark bestraft wird. Damit fällt die Belohnung am Ende besser aus als ohne Bestrafung. Dieses Problem tritt bei SL nicht auf, da die Informationen von Belohnung und Bestrafung dem Modell nicht bekannt sind. Um den Vergleich fair zu behalten, werden die Belohnungen aussortiert, die eine Bestrafung $p < 0$ enthalten. Da sich die Belohnung aus Zielfunktion und Bestrafung zusammensetzt, sind die übrigen Belohnungen die Zielfunktionswerte. Bei den Zielfunktionswerten werden ebenfalls die Metriken MAPE und RMSE vom optimalen Zielfunktionswert A und Zielfunktionswert der Modelle F berechnet. Die Lösungsqualität der Zielfunktionswerte ist die wichtigste Metrik, danach folgen die Belohnungen und Aktionen.

Einhaltung der Nebenbedingungen

Weiterhin wird untersucht, ob die Nebenbedingungen eingehalten wurden. Mit jeder Belohnung wird auch eine Bestrafung mitberechnet, die aussagt, wie stark die Nebenbedingung verletzt wurde. Bei der Berechnung der Lösungsqualität wird auch die Bestrafung abgespeichert. Für den Vergleich werden dann die Bestrafungen nach der Formel

$$penalties = \frac{1}{n} \sum_{i=1}^{n} p(c_{Spannung}, c_{Last}, c_{Blindleistung}, c_{Slack-Bus}) \tag{4.7}$$

aufsummiert, wobei p die Bestrafungsfunktion, $penalties$ das Ergebnis und n die Anzahl an Experimenten repräsentieren.

Für die Kriterien Berechnungszeit, Lösungsqualität und Einhaltung der Nebenbedingungen müssen Zustände generiert werden. Die Anzahl der generierten Zustände n wird auf 10000 festgelegt. Es wird dann mit jedem Zustand jeweils bei dem Pandapower OPF Löser,

SL und DRL die Aktion, Berechnungszeit, Belohnung und Bestrafung abgespeichert.

4.7 Training

Während des Trainings wird bei DRL das Modell in Intervallen getestet und der MAPE zu den optimalen Belohnungen berechnet. Dies gilt jedoch nicht für SL, denn im Datensatz werden keine Belohnungen berücksichtigt. Es wird stattdessen der MAPE und Verlust zu den optimalen Aktionen berechnet. Das Ziel ist es, ein Modell zu finden, welches den niedrigsten MAPE zu den optimalen Belohnungen bzw. optimalen Aktionen erzielt. Hyperparameter wie die Lernrate oder Batchgröße beeinflussen die Trainingsergebnisse und müssen bestmöglich eingestellt werden. Im Folgenden werden die Hyperparameter kurz erklärt.

Lernrate

Die Lernrate bestimmt, wie stark das neuronale Netz bei jeder Iteration seine Gewichte aktualisiert. Eine zu hohe Lernrate kann zu instabilen Gewichtungen und zu schnellem Überlernen führen, während eine zu niedrige Lernrate dazu führen kann, dass das Netzwerk zu langsam konvergiert. Beim Überlernen ist das Modell zu sehr auf dem Datensatz angepasst, sodass Vorhersagen auf ungesehene Daten ungenau sind. [32]

Batchgröße

Die Batchgröße bestimmt, wie viele Datenpunkte auf einmal durch das Netzwerk propagiert werden. Dabei kann es vom Vorteil sein, eine hohe Batchgröße zu verwenden, da viele Daten auf einmal verwendet werden und das Training verkürzt wird. Eine kleine Batchgröße kann dagegen zu zufälligen Aktualisierungen der Gewichte führen, sodass die Leistung des Modells verbessert oder verschlechtert wird. [32]

Epochen-/Schrittanzahl

Eine hohe Epochen-/Schrittanzahl bewirkt ein längeres Training und somit generell ein besseres Ergebnis [32]. Bei SL kann jedoch eine hohe Epochenanzahl zu Überlernen führen. Das Ziel der Arbeit ist es, die besten Modelle zu vergleichen, somit sollte dieser Parameter möglichst hoch gewählt werden, damit die Modelle genug Zeit zum Trainieren haben.

Datensatzgröße

Dieser Hyperparameter ist nur für SL wichtig. Je mehr Daten zur Verfügung stehen, desto mehr kann ein Modell lernen. In dieser Arbeit wird ein Simbench Datensatz verwendet, der 35156 Datenpunkte hat. Daraus wird der Datensatz generiert und kann beliebig groß gewählt werden, da die Zustände zufällig und mit Rauschen erzeugt werden (siehe Unterabschnitt 4.4.1). Es muss herausgefunden werden, ob ein großer Datensatz einen Vorteil bringt oder nur Ressourcen verschwendet.

Weitere Hyperparameter

Es gibt noch weitere Hyperparameter, wie z.B. die Anzahl an Neuronen in den versteckten Ebenen, andere Verlustfunktionen, andere Optimierer usw. Diese werden in der Arbeit nicht beachtet und sind für SL und DRL gleich. Allerdings hat DRL weitere Hyperparameter, die SL nicht besitzt. Diese werden im Folgenden aufgelistet:

- Erfahrungspuffergröße = 100000
- $\gamma = 0.99$
- $\tau = 0.001$
- Kritiker Netz Lernrate = 0.0005
- Testintervall = 20000
- Testschritte = 2000

Das Testintervall gibt an, nach wie vielen Schritten ein Test des Agenten durchgeführt wird. Dabei werden dann die gesetzte Anzahl an Testschritten verwendet.

Da bei jedem Training andere Ergebnisse erzielt werden (durch zufällig gewählte Startgewichte, Exploration, usw.) werden die Modelle mit den besten Ergebnissen dreimal trainiert. Damit ergeben sich jeweils für SL und DRL drei Modelle. Danach werden die Modelle getestet und verglichen. Die Modelle, die den besten und schlechtesten MAPE zu den optimalen Zielfunktionswerten haben, werden nicht mehr beachtet. Somit wird am Ende das Modell „in der Mitte“ für den Vergleich gewählt.

5 Ergebnisse

Im vorherigen Kapitel wurden SL und DRL implementiert und alle Metriken sowie Vergleichskriterien aufgestellt. Nun können die Modelle trainiert werden. Die Ergebnisse des Trainings werden zunächst in Abschnitt 5.1 und Abschnitt 5.2 präsentiert. Dadurch soll gezeigt werden, dass die Modelle fertig trainiert sind und ein gutes Ergebnis erzielt haben. Mit den trainierten Modellen kann dann der Vergleich durchgeführt werden. Die Ergebnisse werden in Abschnitt 5.3 vorgestellt. Im nächsten Kapitel werden die Ergebnisse diskutiert.

5.1 Supervised Learning Training

Beim Training des besten gefundenen Modells wurden die folgenden Hyperparameter verwendet:

- Lernrate = 0.0005
- Batchgröße = 8
- Epochenanzahl = 1000
- Datensatzgröße = 70272

Eine kleine Batchgröße eignete sich am besten, denn bei einer hohen Batchgröße gab es eine starke Tendenz zum Überlernen. Die Lernrate hatte einen kleinen Effekt auf die Geschwindigkeit der Konvergenz. Es könnte auch eine kleinere Epochenanzahl gewählt werden, aber so konnte genug Zeit für die Konvergenz gewährt werden. Die Datensatzgröße war entscheidend, denn mit einem kleinen Datensatz von 35136 Datenpunkten war die Leistung schlechter. Daraufhin wurde die Datensatzgröße verdoppelt und die Leistung verbesserte sich. Mit einem noch größeren Datensatz wurde keine nennenswerte Leistungssteigerung erzielt.

Trainingsergebnisse

Das Training des besten gefundenen Modells wird in den folgenden Abbildungen dargestellt.

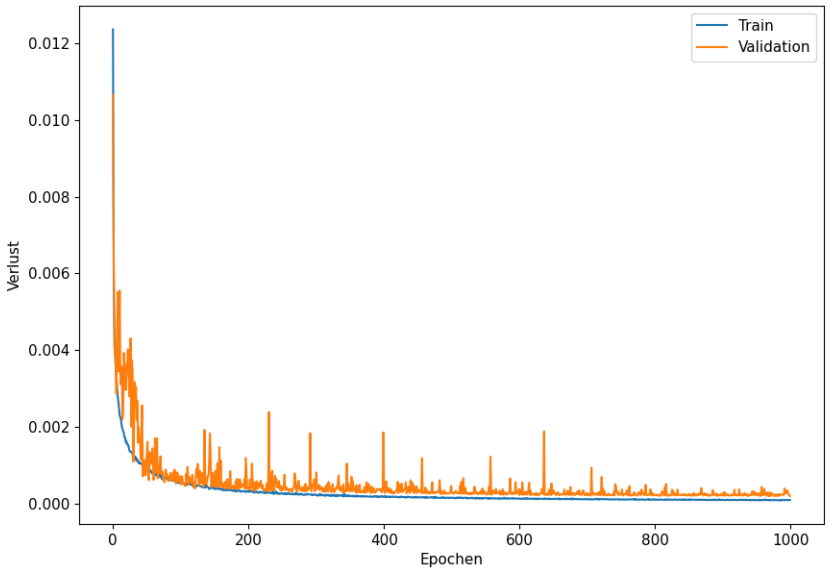

Abbildung 5.1: Verlust des Modells zu den optimalen Aktionen während des Trainings

In Abbildung 5.1 wird der Verlust des Modells zu den optimalen Aktionen während des Trainings dargestellt. Auf der y-Achse befindet sich der Verlust und auf der x-Achse die Epochen. Beim Training wurden 1000 Epochen durchlaufen. Dabei wird der Train-Datensatz (blau) und Validation-Datensatz (orange) angezeigt. Der Verlust von Train startet bei ca. 0.012 und konvergiert gegen ca. 0.0001, während Validation einen leicht höheren Wert anstrebt. Validation folgt allgemein dem Verlauf von Train mit leicht höheren Werten und mehr Ausreißern.

Auf dem Diagramm ist nicht eindeutig erkennbar, ob das Modell bereits fertig trainiert ist, weshalb in der nächsten Abbildung ein Ausschnitt von der Epoche 200 bis 1000 gezeigt wird.

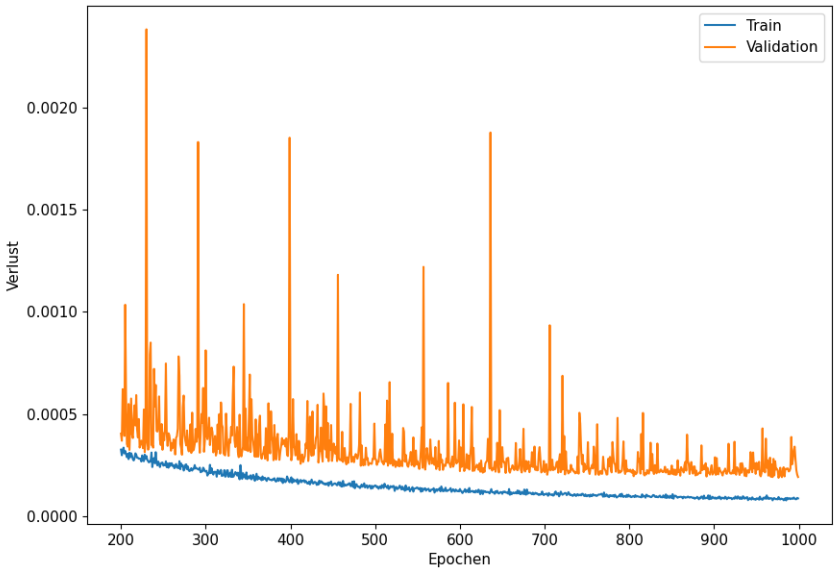

Abbildung 5.2: Verlust des Modells zu den optimalen Aktionen während des Trainings, Ausschnitt

In der Abbildung 5.2 wird der Verlust des Modells zu den optimalen Aktionen während des Trainings von der Epoche 200 bis 1000 gezeigt. Die Legenden- und Achsenbeschriftung sind dabei gleichgeblieben. Nun wird deutlich, dass die Werte von Train und Validation in den Epochen 200 bis 1000 weiter fallen und langsam konvergieren.

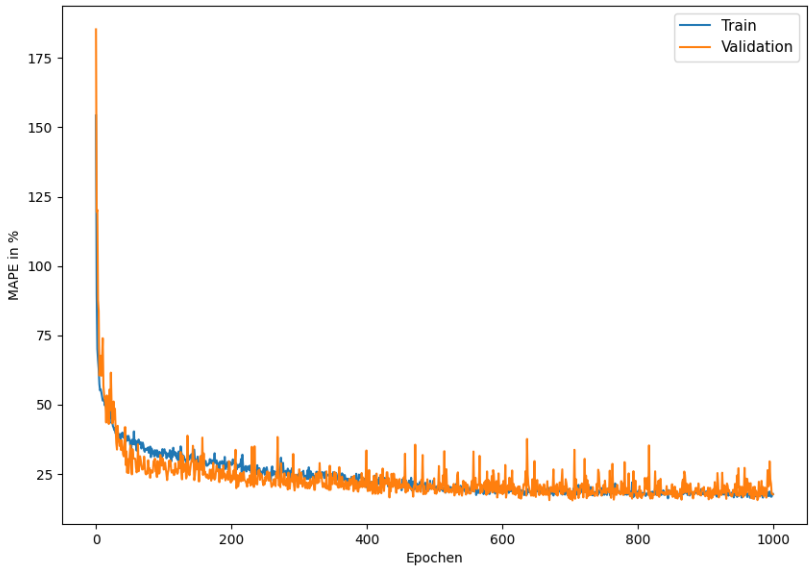

Abbildung 5.3: MAPE des Modells zu den optimalen Aktionen während des Trainings

Mit der Abbildung 5.3 wird der MAPE des Modells zu den optimalen Aktionen während des Trainings dargestellt. Auf der x-Achse wird der MAPE in % und auf der y-Achse die Epochen angezeigt. Dabei ist die Legende zur vorherigen Abbildung gleich geblieben. Der Verlauf ist ähnlich der Verlustkurve, wobei hier der Validations-MAPE am Anfang des Trainings von ca. 20 bis 400 Epochen niedriger ist als der Train-MAPE. Der Train-MAPE nähert sich langsam dem Validations-MAPE an und gegen Ende des Trainings sind sie ungefähr gleich. Bemerkenswert ist, dass der MAPE am Ende ca. 24 % beträgt, während der Verlust fast gegen 0 strebt. In der nächsten Abbildung 5.4 wird ebenfalls ein Ausschnitt von der Epoche 200 bis 1000 gezeigt, um die Werte besser betrachten zu können.

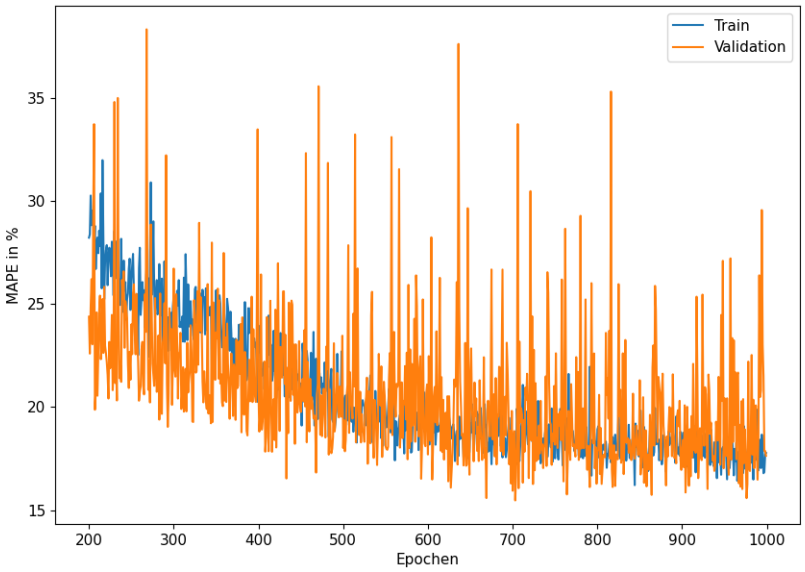

Abbildung 5.4: MAPE des Modells zu den optimalen Aktionen während des Trainings, Ausschnitt

Die Legenden- und Achsenbeschriftung sind dabei gleichgeblieben. Der MAPE von Train fällt noch um etwa 15 % von Epoche 200 bis 1000. Auch der Validations-MAPE fällt weiter, jedoch nicht so signifikant wie Train.

5.2 Deep Reinforcement Learning Training

Das beste Ergebnis konnte das Modell mit folgenden Hyperparametern erzielen:

- Lernrate = 0.0001
- Batchgröße = 1024
- Schrittanzahl = 1000000

Die Batchgröße muss im Gegensatz zu SL hoch eingestellt sein, um ein gutes Ergebnis zu erzielen. Die Schrittanzahl mit 1 Million Schritten ist hoch gewählt, um genug Zeit für die Konvergenz zu gewähren.

Trainingsergebnisse

Das Training des besten gefundenen Modells wird in den folgenden Abbildungen dargestellt.

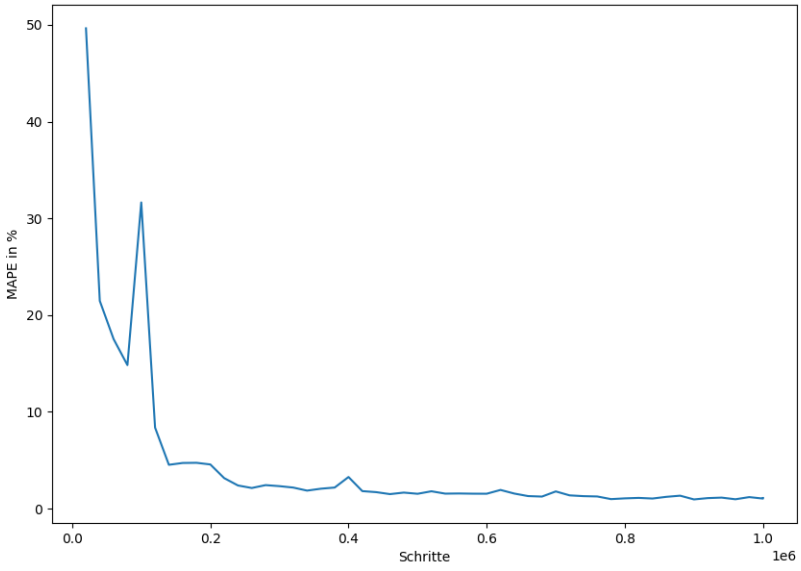

Abbildung 5.5: MAPE des Modells zu den optimalen Belohnungen während des Trainings

In Abbildung 5.5 wird der MAPE des Modells zu den optimalen Belohnungen während des Trainings abgebildet. Damit unterscheidet sich die Abbildung zu den SL Trainingsergebnissen, da bei SL der MAPE zu den optimalen Aktionen angezeigt wird. Auf der y-Achse befindet sich der MAPE in % und auf der x-Achse die Schritte. Bei dem Training

wurden 1 Million Schritte durchlaufen. Am Anfang des Trainings war der MAPE bei ca. 50% und fällt in den ersten 100000 Schritten auf ca. 15%. Danach steigt der MAPE wieder auf über 30% und fällt dann schnell auf 5%. Ab hier konvergiert der MAPE langsam gegen ca. 1%. Um die Konvergenz des Trainings besser sichtbar zu machen, wird in der nächsten Abbildung 5.6 einen Ausschnitt von den Schritten 200000 bis 1 Million abgebildet.

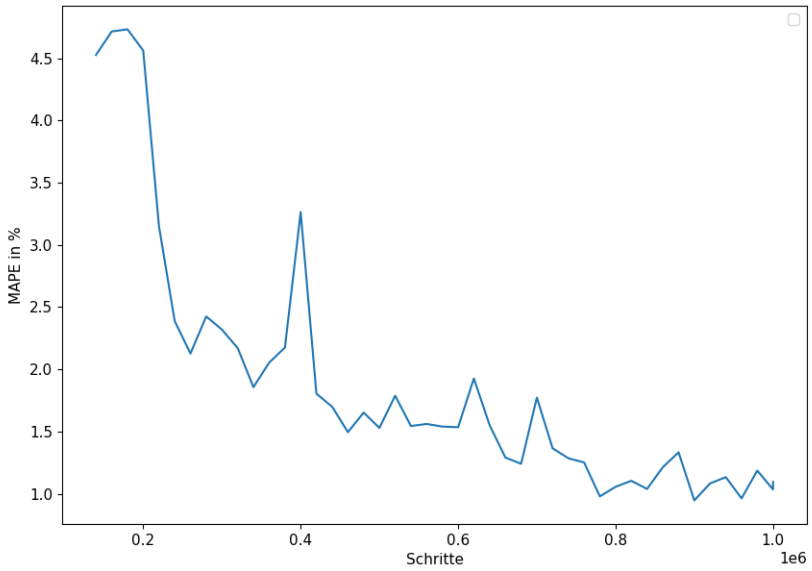

Abbildung 5.6: MAPE des Modells zu den optimalen Belohnungen während des Trainings, Ausschnitt

Dabei ist die Achsenbeschriftung gleichgeblieben. Der MAPE fällt im Verlauf des Trainings immer weiter, bis bei ca. 1 % MAPE sich der Wert nicht verbessert.

5.3 Vergleich der Ergebnisse

5.3.1 Trainingszeit

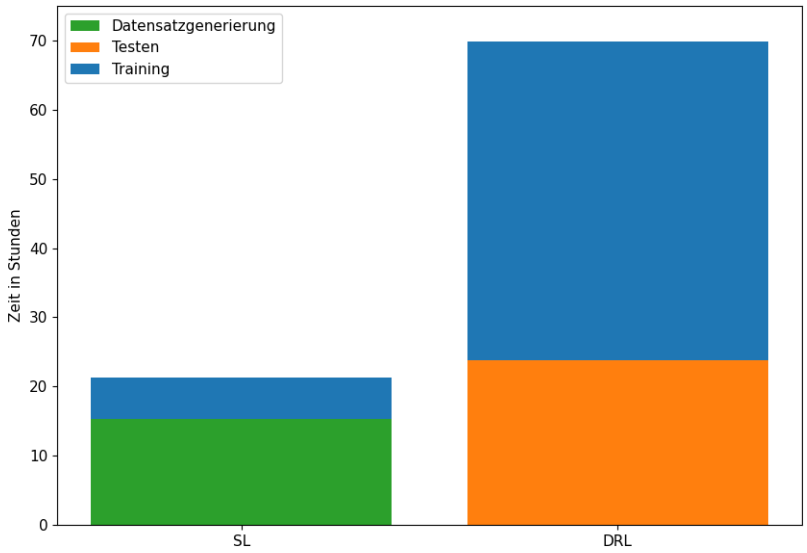

Abbildung 5.7: Trainingszeit der Modelle

In der Abbildung 5.7 wird die Trainingszeit von SL und DRL verglichen. Auf der y-Achse wird die Trainingszeit in Stunden angezeigt. Dabei wird zwischen Trainingszeit (blau), der Zeit zur Datensatzgenerierung (grün) und der Testzeit (orange) unterschieden. Dabei ist die Datensatzgenerierung exklusiv für SL und die Testzeit exklusiv für DRL. Die Testzeit wird von den Hyperparametern Testintervall und Testschritte beeinflusst, sodass diese Zeit auch kürzer sein kann. SL benötigt 6 Stunden zum Trainieren und 15.3 Stunden, um den Datensatz zu generieren, womit sich eine Gesamtzeit von 21.3 Stunden ergibt. Das entspricht ungefähr der Testzeit von DRL, welche 23.77 Stunden beträgt. Am längsten hat das Training von DRL mit 46.08 Stunden gedauert. Insgesamt ergibt sich für DRL eine Zeit von 69.85 Stunden. DRL hat mit Testzeit ungefähr 3.5-mal länger und ohne Testzeit doppelt so lange trainiert verglichen mit der gesamten Zeit von SL.

5.3.2 Berechnungszeit einer Aktion

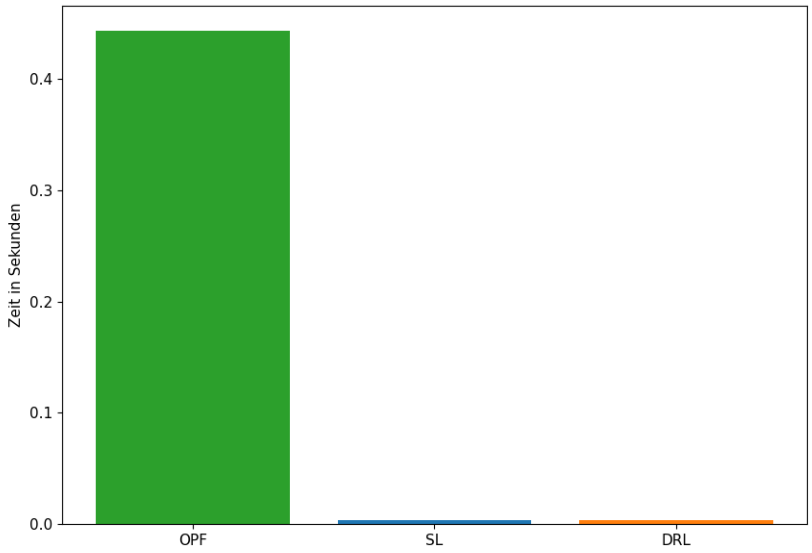

Abbildung 5.8: Durchschnittliche Berechnungszeit für eine Aktion

In Abbildung 5.8 wird die durchschnittliche Berechnungszeit einer Aktion verglichen. Auf der y-Achse wird die Zeit in Sekunden angegeben und auf der x-Achse SL, DRL und der Pandapower OPF Löser. Letzterer benötigte für die Berechnung ca. 0.44 Sekunden, während SL und DRL ca. 0.003 Sekunden gebraucht haben. Damit sind die Modelle etwa 146-mal schneller in der Berechnung einer Aktion.

5.3.3 Lösungsqualität Aktionen

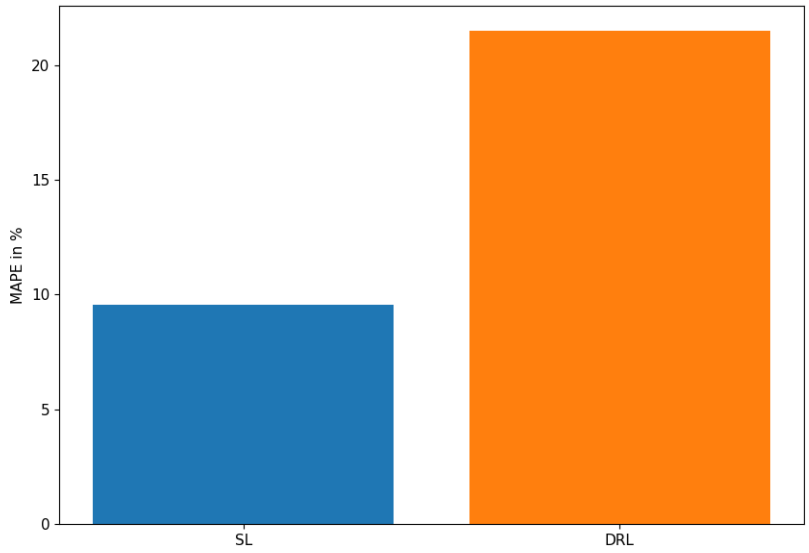

Abbildung 5.9: MAPE zu den optimalen Aktionen

In der Abbildung 5.9 wird der MAPE zu den optimalen Aktionen dargestellt. Auf der y-Achse befindet sich der MAPE in % und auf der x-Achse SL und DRL. Die berechneten Aktionen von SL haben einen MAPE von 9.54 % zu den optimalen Aktionen, während die Aktionen von DRL bei 21.51 % liegen.

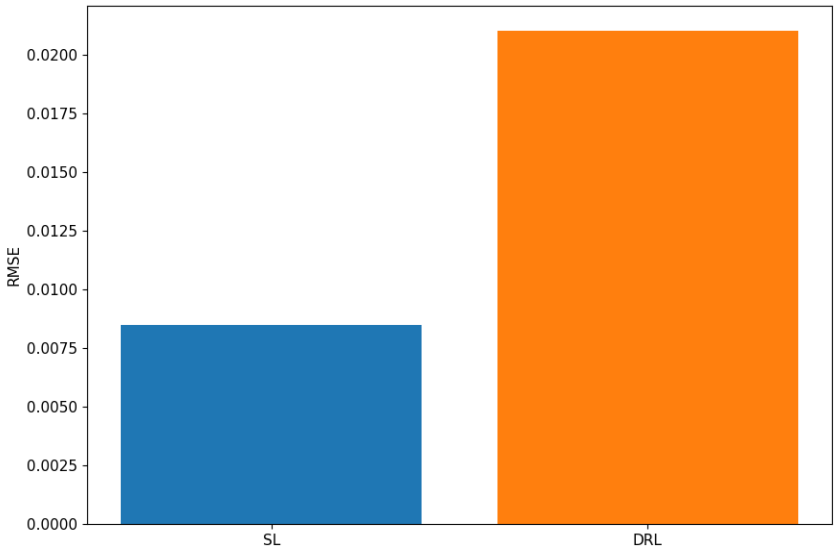

Abbildung 5.10: RMSE zu den optimalen Aktionen

In der Abbildung 5.10 wird der RMSE zu den optimalen Aktionen dargestellt. Auf der y-Achse befindet sich der RMSE und auf der x-Achse SL und DRL. Der RMSE von SL und DRL ist 0.008 beziehungsweise 0.021. Damit verhalten sich MAPE und RMSE bezogen auf die optimalen Aktionen fast gleich: SL erzielt in beiden Metriken verglichen mit DRL mehr als zweimal bessere Ergebnisse.

5.3.4 Lösungsqualität Belohnungen

Nun wird die Lösungsqualität der Belohnungen ohne Abzug der Bestrafungen näher beleuchtet.

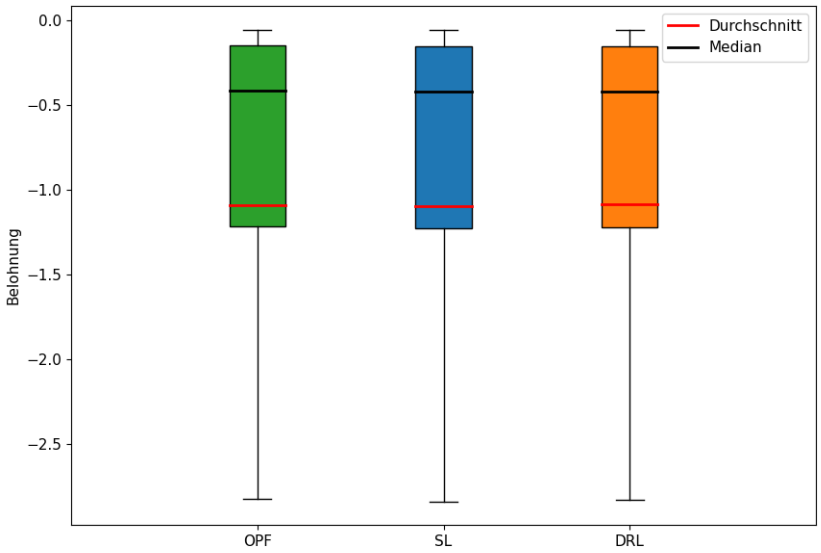

Abbildung 5.11: Boxplots der Belohnungen

In der Abbildung 5.11 werden die Belohnungen in Boxplots abgebildet. Auf der y-Achse befinden sich die Belohnungen und auf der x-Achse jeweils der Pandapower OPF Löser, SL und DRL. In den Boxplots wird der Durchschnittswert (rot) und der Median (schwarz) angezeigt. Auffällig ist, dass alle Werte fast identisch sind, also der Durchschnittswert, Median, die Antennen (unten und oben) und die Quartile. Der Durchschnittswert vom Pandapower OPF Löser liegt bei -0.844, während SL bei -0.847 und DRL bei -0.846 liegen. Die unteren Antennen haben einen Wert von ca. -2.6 und die oberen Antennen einen Wert von ca. -0.08. In der nächsten Abbildung 5.12 wird dasselbe Diagramm mit Ausreißern dargestellt, da in dieser Abbildung die Ausreißer herausgefiltert wurden.

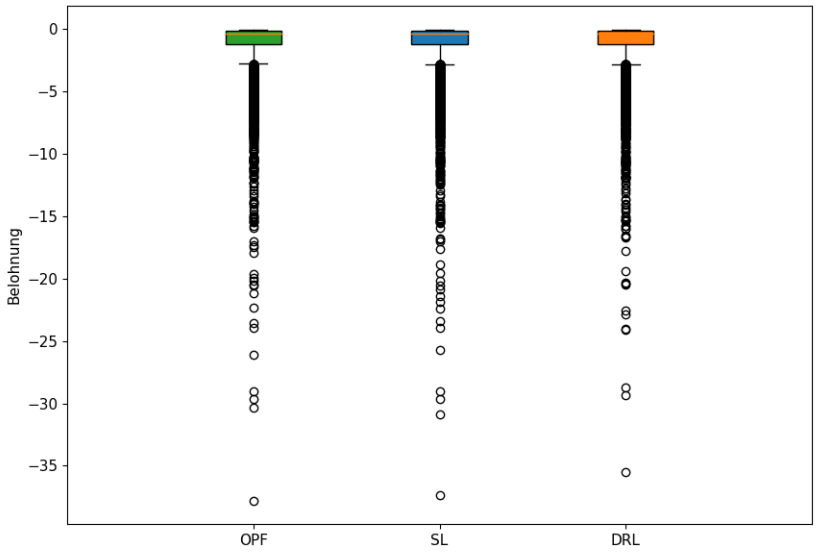

Abbildung 5.12: Boxplots der Belohnungen inklusive Ausreißer

Der OPF Löser hat den größten Ausreißer bei einer Belohnung von ca. -39, während der größte Ausreißer bei SL bei ca. -38 und bei DRL bei ca. -35 liegt. Am meisten befinden sich Ausreißer bei allen Verfahren in dem Bereich von ca. -3 bis -16. Bemerkenswert ist, dass viele Ausreißer bei den Verfahren ähnlich sind. So sind z.B. drei Ausreißer bei OPF in dem Bereich von -30, die bei SL auch zu finden sind, während DRL dort zwei Ausreißer hat. Tendenziell hat DRL jedoch weniger und nicht so intensive Ausreißer verglichen mit SL und OPF.

5.3.5 Lösungsqualität Zielfunktion

Bei der Lösungsqualität für die Zielfunktion werden nur Belohnungen gewertet, die keine Bestrafungen enthalten.

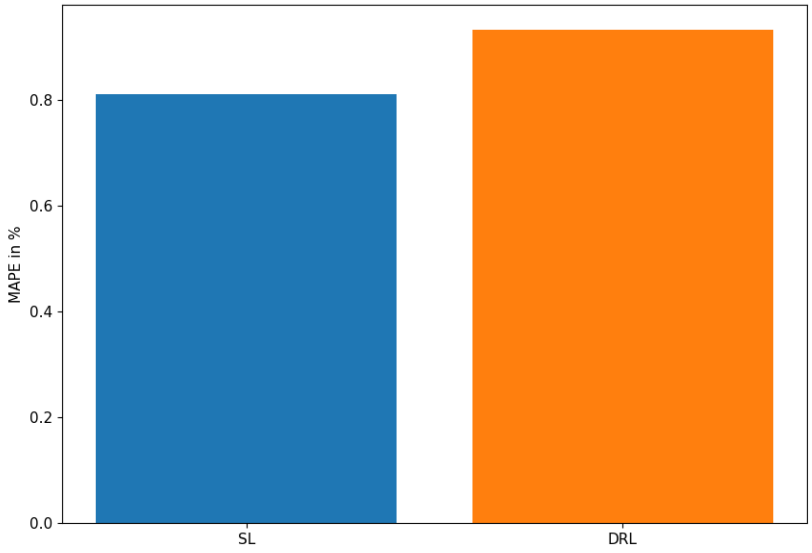

Abbildung 5.13: MAPE zu den optimalen Zielfunktionswerten

In der Abbildung 5.13 wird der MAPE zu den optimalen Zielfunktionswerten dargestellt. Auf der y-Achse befindet sich der MAPE in % und auf der x-Achse SL und DRL. SL erzielt einen MAPE von 0.81 % und DRL 0.93 %.

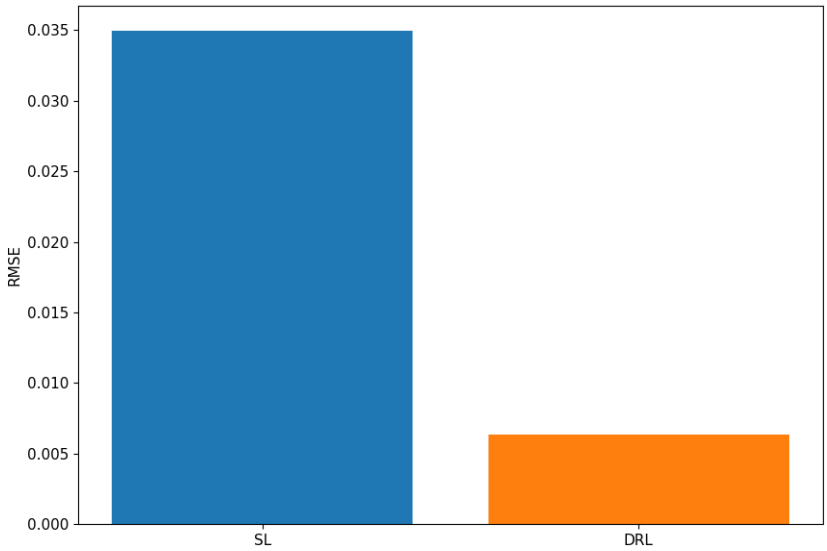

Abbildung 5.14: RMSE zu den optimalen Zielfunktionswerten

In der Abbildung 5.14 wird der RMSE zu den optimalen Zielfunktionswerten dargestellt. Auf der y-Achse befindet sich der RMSE und auf der x-Achse SL und DRL. SL erzielt einen RMSE von 0.0345 und DRL 0.006. Damit ist DRL bei dem RMSE mehr als fünfmal besser als SL, allerdings ist SL bei dem MAPE um etwa 0.12 % besser.

5.3.6 Einhaltung der Nebenbedingungen

Nun werden die Bestrafungen betrachtet.

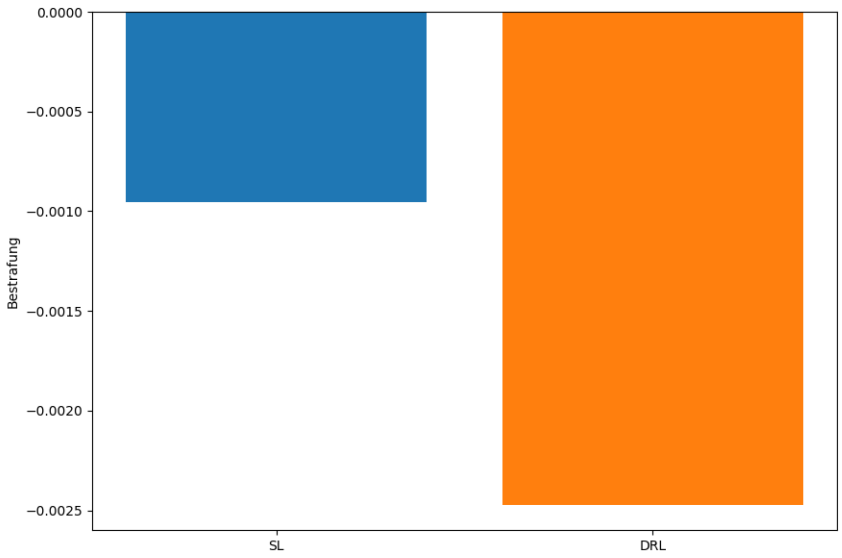

Abbildung 5.15: Durchschnittliche Bestrafungen

In der Abbildung 5.15 werden die durchschnittlichen Bestrafungen von den gesamten Lösungen ($n = 10000$) angezeigt. Auf der y-Achse befindet sich die Bestrafung und auf der x-Achse SL und DRL. Der OPF Löser hat gar keine Bestrafungen erhalten, weshalb dieser nicht in die Abbildung aufgenommen wurde. SL hat eine durchschnittliche Bestrafung von ca. -0.001. DRL liegt bei ca. -0.0025. Damit ist die Höhe der Bestrafung bei SL durchschnittlich etwa 2.5-mal geringer als bei DRL.

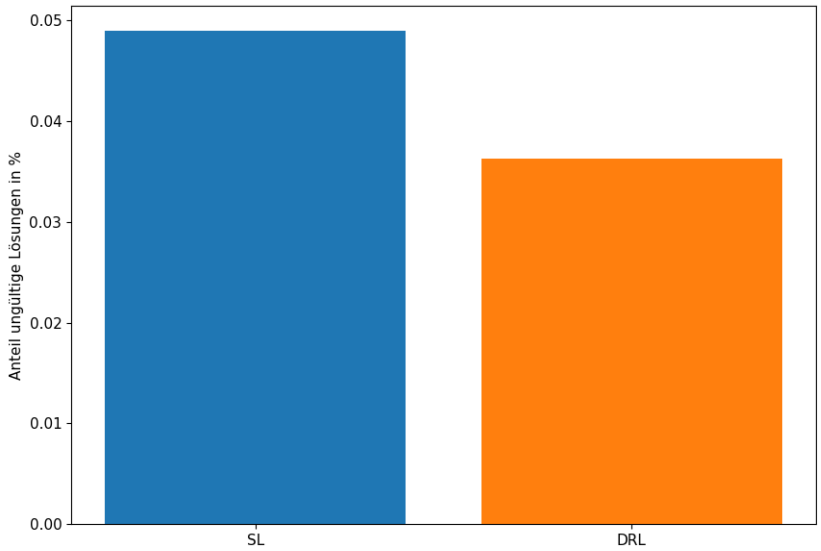

Abbildung 5.16: Anteil an ungültigen Lösungen von den gesamten Lösungen

In der Abbildung 5.16 wird der Anteil an ungültigen Lösungen von den gesamten Lösungen ($n = 10000$) angezeigt. Auf der y-Achse wird der Anteil an ungültigen Lösungen in % angezeigt und auf der x-Achse SL und DRL. SL und DRL haben einen Anteil an ungültigen Lösungen von ca. 0.049 % bzw. 0.036 %.

6 Diskussion

Im Folgenden werden die Ergebnisse aus Kapitel 5 diskutiert. In Abschnitt 6.4 werden Ergebnisse aus einer anderen Umgebung betrachtet, die nicht in die Arbeit aufgenommen wurden. Der Aufwand der Implementierung wird in Abschnitt 6.5 eingeschätzt. Zum Schluss werden in Abschnitt 6.6 die Schwierigkeiten beim Vergleich von SL und DRL beschrieben.

6.1 Supervised Learning Training

In der Abbildung 5.1 werden die Verluste während des Trainings angezeigt. Dabei fällt auf, dass der Verlust fast gegen 0 geht, während der MAPE in Abbildung 5.3 am Ende noch ca. 24 % beträgt. Da die Aktionen im Wertebereich $[0, 1]$ liegen, kann der MAPE hoch ausfallen, da durch den wahren Wert A geteilt wird.
In der Abbildung 5.2 und Abbildung 5.4 ist erkennbar, dass die Leistung des Modells im Verlauf des Trainings weiter verbessert wird. Bei einem längeren Training würde vermutlich die Leistung gering besser werden. Das Training ist insgesamt erfolgreich und ein gutes Modell wurde gefunden.

6.2 Deep Reinforcement Learning Training

Bei der Abbildung 5.6 wird der MAPE während des Trainings angezeigt, wobei die ersten 200000 Schritte nicht in der Abbildung vorkommen. Damit wird deutlich, dass der MAPE im Verlauf des Trainings weiter fällt, jedoch wird nicht ersichtlich, ob der MAPE seinen niedrigsten Punkt erreicht hat oder noch weiter fallen würde. Vermutlich würde der MAPE mit einem längeren Training noch weiter fallen, die Veränderungen auf die Ergebnisse wären aber gering. Damit hat das Modell ein erfolgreiches Training gehabt und eine gute Leistung erzielt.

6.3 Vergleich der Ergebnisse

Im Folgenden werden die Ergebnisse des Vergleichs diskutiert.

Trainingszeit

DRL hat mit Testzeit ungefähr 3.5-mal länger und ohne Testzeit doppelt so lange trainiert wie SL. Dabei ist zu beachten, dass die Trainingszeit bewusst lange gewählt wurde, jedoch ist der Trend erkennbar: DRL braucht länger zum Trainieren. Der Agent muss selbst das Optimum finden, wobei vier neuronale Netze verwendet werden (im Gegensatz zu einem neuronalen Netz bei SL). Dabei hätten die SL und DRL Zeit länger gewählt werden können, jedoch ist das Training bei beiden Verfahren konvergiert, sodass die Leistung sich vermutlich eher gering verbessern würde.

Berechnungszeit einer Aktion

In der Abbildung 5.8 wird deutlich, dass der OPF Löser eine deutlich längere Berechnungszeit für eine Aktion benötigt als SL und DRL, welche ungefähr die gleiche Zeit haben. Dadurch wird der Vorteil von neuronalen Netzen deutlich: Sie benötigen wenig Zeit, um eine Vorhersage zu machen, aber viel Zeit im Voraus, um trainiert zu werden. Da SL und DRL das gleiche neuronale Netz verwenden, ist die Zeit etwa identisch.

Lösungsqualität Aktionen

Bei der Lösungsqualität der Aktionen hat SL in den Metriken MAPE und RMSE mehr als zweimal bessere Ergebnisse erzielt. Da SL die optimalen Aktionen aus dem Datensatz gelernt hat, ist es nachvollziehbar, dass SL hier besser abschneidet.

Lösungsqualität Belohnungen

Der OPF Löser, SL und DRL haben durchschnittlich etwa die gleichen Belohnungen erhalten. In der Abbildung 5.12 wird deutlich, dass SL in etwa die gleichen Ausreißer wie OPF hat, was daran liegt, dass SL die Aktionen vom OPF Löser gelernt hat. Es ist außerdem auffällig, dass die Ausreißer von DRL nicht so stark sind wie bei OPF und SL. Der Agent hat vermutlich gelernt, eine allgemein bessere Belohnung zu finden oder eine Bestrafung zu dulden, um dennoch eine bessere Belohnung zu erhalten.

Lösungsqualität Zielfunktion

Es ist nicht eindeutig, ob SL oder DRL besser in dieser Kategorie abschneidet, da der MAPE bei SL um 0.12 % besser ist, aber der RMSE bei DRL ist mehr als fünfmal besser. Da DRL weniger starker Ausreißer hat, ist der RMSE besser, da der RMSE empfindlicher bei Ausreißern ist als der MAPE (siehe Abschnitt 4.3). Beide Modelle haben eine gute Leistung erzielt, wobei DRL etwas besser abschneidet, da der RMSE deutlich niedriger ist.

Einhaltung der Nebenbedingungen

Die Höhe der durchschnittlichen Bestrafung ist bei SL etwa 2.5-mal geringer als bei DRL, jedoch ist der Anteil an ungültigen Lösungen, also die Lösungen, die Bestrafungen erhalten, bei DRL um 0.013 % niedriger. Der DRL Agent hat also weniger oft Bestrafungen erhalten. Wenn er jedoch Bestrafungen erhält, fallen sie eher hoch aus. Da die Ausreißer der Belohnungen aus Abbildung 5.12 bei DRL weniger stark waren, lässt sich vermuten, dass der Agent in manchen Fällen die Bestrafung duldet, um eine bessere Belohnung zu erhalten, selbst wenn die Bestrafung hoch ist. Denn die Belohnung ist dann am Ende immer noch höher als eine Belohnung ohne Bestrafung.

Zusammenfassung

Bei der Trainingszeit hat SL einen klaren Vorteil. Die vier neuronalen Netze von DRL brauchen mehr Zeit zum Trainieren. Auch ohne Testzeit hat das Training von DRL mehr als zweimal länger gedauert als die Datensatzgenerierung und Training von SL.
Mit dem Vergleich der durchschnittlichen Berechnungszeit einer Aktion wird deutlich, dass ein großer Vorteil von neuronalen Netzen die Zeitersparnis ist. So sind SL und DRL etwa 146-mal schneller als der Pandapower OPF Löser.
SL hat verglichen mit DRL einen mehr als zweimal geringeren Fehler zu den optimalen Aktionen erzielt. Bei den Belohnungen hat DRL etwas geringere Ausreißer, insgesamt sind die durchschnittlichen Belohnungen vom Pandapower OPF Löser, SL und DRL aber ungefähr gleich. Der MAPE zu den optimalen Zielfunktionswerten ist bei SL 0.12 % besser, aber beim RMSE ist DRL mehr als fünfmal besser, da die Ausreißer geringer sind. Somit hat DRL etwas bessere Ergebnisse bei den Zielfunktionswerten, was die wichtigste Metrik ist.
Weiterhin haben DRL 0.036 % und SL 0.049 % ungültige Lösungen. Die Höhe der durchschnittlichen Bestrafung ist bei SL etwa 2.5-mal geringer als bei DRL. Es lässt sich vermuten, dass der DRL Agent gelernt hat, Bestrafungen zu dulden, um eine höhere Belohnung zu bekommen.

6.4 Wirkleistungsmarkt-Umgebung

Es wurde noch eine Wirkleistungsmarkt-Umgebung getestet, die aber nicht in die Arbeit aufgenommen wurde, da viel am DRL Algorithmus optimiert wurde, um die Ergebnisse zu verbessern. So wurden die Bestrafungsfaktoren angepasst und eine andere Metrik für das Training verwendet. Bei SL hat jedoch die Zeit gefehlt, um Optimierungen vorzunehmen, denn die Ergebnisse waren noch nicht optimal. Die Verlustkurven von Train und Validation

blieben im Verlauf des Trainings weit auseinander (siehe Abbildung A.1). Das bedeutet, dass das Modell Probleme hat, auf ungesehene Daten akkurat zu reagieren.
Die Umgebung ist etwas komplexer als die Blindleistungsmarkt-Umgebung, da z.B. das Energiesystem größer ist. In Abschnitt A.1 sind die Ergebnisse abgebildet. Dabei sind die Ergebnisse der Trainingszeit in Abbildung A.3 ähnlich zum Blindleistungsmarkt. Ebenfalls ist die Lösungsqualität der Aktionen bei SL wieder besser (siehe Abbildung A.4 und Abbildung A.5). Bei der Lösungsqualität der Zielfunktion gibt es jedoch ein eindeutiges Ergebnis: der MAPE und RMSE in Abbildung A.6 und Abbildung A.7 ist bei DRL etwa sechsmal niedriger verglichen mit SL. Die Höhe der durchschnittlichen Bestrafungen in Abbildung A.10 sind bei SL verglichen mit DRL wieder deutlich niedriger. Auch bei dem Anteil an ungültigen Lösungen ist SL wieder etwas besser, wobei beide Modelle hier mehr ungültige Lösungen haben als beim Blindleistungsmarkt (siehe Abbildung A.11).

6.5 Aufwand der Implementierung

Da der DRL Teil dieser Arbeit nicht selbst implementiert wurde, kann der Aufwand nur grob eingeschätzt werden. Die folgenden Punkte sind bei DRL aufwendig zu implementieren:

- Umgebung modellieren: Die Umgebung zu modellieren ist sehr aufwendig und erfordert ein tiefes Verständnis des Problems, damit der Agent in der Umgebung reibungslos Aktionen ausführen kann und angemessene Belohnungen erhält. Durch das OpenAI Gym Modul werden schon einige Methoden bereitgestellt, jedoch ist die Implementierung der Umgebung immer noch ein aufwendiger und komplexer Prozess von DRL.
- Training: Es gibt mehrere Trainingsalgorithmen für DRL, die unterschiedlich sind, aber bereits in der Literatur implementiert sind, d.h. der Programmcode kann von der Literatur übernommen werden bzw. der Pseudocode ist schon vorgegeben.
- Hyperparameter optimieren: Nach dem Training werden die Hyperparameter anhand verschiedener Metriken optimiert, um die Leistung des Modells zu steigern.

Nun werden die Punkte aufgezählt, die bei SL aufwendig sind:

- Datensatzgenerierung: Die Datensatzgenerierung oder die Beschaffung von geeigneten Daten inklusive Vorbereitung der Daten ist ist eine aufwendige Teilaufgabe von SL, da der Datensatz maßgeblich für das Ergebnis verantwortlich ist und daher von guter Qualität sein muss.

- Training: Das Training des neuronalen Netzes in einer Trainingsschleife hat bei SL oft den gleichen Ablauf, der auf das Problem angepasst wird.
- Hyperparameter optimieren: Nachdem das Modell trainiert ist, werden die Metriken wie MAPE und Verlust begutachtet und die Hyperparameter optimiert.

Zusammengefasst ist die Implementierung von DRL vermutlich aufwendiger als die SL Implementierung. Besonders die Modellierung der Umgebung erfordert viel Aufwand, was nicht im Vergleich mit einer Datensatzgenerierung bei SL steht. Bei der Implementierung des Trainingsalgorithmus ist DRL auch komplexer, da die Algorithmen viele Schritte umfassen. So benötigt der DDPG Algorithmus beispielsweise vier neuronale Netze, während SL ein neuronales Netz benutzt. Auch das Parametertuning am Ende ist bei DRL komplexer, da es mehr Hyperparameter besitzt, wie z.B. die Diskontierungsrate γ oder die Aktualisierungsrate τ, die SL nicht besitzt.

6.6 Schwierigkeiten beim Vergleich

Da SL und DRL sehr unterschiedlich sind, gab es Schwierigkeiten, die Trainingsverfahren zu vergleichen. Zunächst ist das größte Problem das Ziel des Trainings: SL lernt eine optimale Aktion zu einem Zustand zu finden, während DRL lernt eine maximale Belohnung zu einem Zustand zu finden. Da die Belohnung bzw. der Zielfunktionswert am Ende am wichtigsten ist, könnte SL einen Nachteil haben, da es nicht das Ziel des Trainings ist und gar keine Information über Belohnung (oder Bestrafung) vorliegt.
Weiterhin gibt es Probleme bei den Metriken. Bei der Lösungsqualität der Zielfunktion ist SL beim MAPE besser und DRL beim RMSE besser. Auch beim Training der Modelle wurde die Metrik MAPE eingesetzt, wobei bei SL damit der Fehler zur optimalen Aktion und bei DRL der Fehler zur optimalen Belohnung berechnet wurde.
Die Trainingszeit war ebenfalls schwierig zu vergleichen, da SL in Epochen trainiert und DRL in Schritten, was sehr unterschiedliche Werte sind. DRL muss vier neuronale Netze trainieren, während SL ein neuronales Netz trainiert, was deutlich weniger Rechenaufwand bedeutet.
Außerdem ist es nicht einfach, geeignete Hyperparameter zu finden. So können bei DRL Hyperparameter angepasst werden, die bei SL nicht existieren. Auch gemeinsame Hyperparameter wie der Optimierer oder die Verlustfunktion können unterschiedliche Effekte bei SL und DRL bewirken, sodass ein Verfahren eventuell Vorteile hat, während das andere Verfahren Nachteile hat. Diese wurden für beide Verfahren vorher festgelegt, jedoch ist in dieser Arbeit nicht getestet worden, ob andere Hyperparameter die Ergebnisse verändern.

7 Fazit

In dieser Arbeit wurden SL und DRL eingesetzt, um die Lösungen für ein OPF Problem vorherzusagen. Dafür wurde die Blindleistungsmarkt-Umgebung eingesetzt. Eine weitere Umgebung, der Wirkleistungsmarkt, wurde auch getestet, jedoch nicht in die Arbeit aufgenommen. Um den Vergleich durchzuführen, wurde zunächst eine gemeinsame Grundlage geschaffen. So ist die Architektur des DNN, der Optimierer, die Verlustfunktion, die Hardware etc. bei SL und DRL gleich. Dennoch haben sich einige Schwierigkeiten beim Vergleich ergeben, beispielsweise die unterschiedlichen Hyperparameter und andere Metriken beim Training. Im Training der Modelle haben SL und DRL gute Leistungen erzielt. Danach wurden die Modelle getestet und anhand mehrerer Kriterien verglichen.
Jeweils SL und DRL haben gezeigt, dass neuronale Netze eine geeignete Lösungsalternative zur Berechnung des OPF sind, denn der MAPE zur optimalen Lösung beträgt etwa 1 % mit einer 146-mal schnelleren Berechnung. Damit decken sich die Ergebnisse mit den verwandten Arbeiten aus Abschnitt 3.4 außer bei der Einhaltung der Nebenbedingungen. Beide Trainingsverfahren berechneten zu 0.05 % der gesamten Lösungen ungültige Ergebnisse. Somit konnte die Lösungssicherheit nicht garantiert werden.
In den Vergleichskriterien haben die Verfahren unterschiedlich abgeschnitten. Bei der Trainingszeit hat allein das Training von DRL mehr als zweimal länger gedauert als die Datensatzgenerierung und Training von SL. Weiterhin hat SL einen mehr als zweimal geringeren Fehler zur optimalen Aktion und durchschnittlich 2.5-mal geringere Bestrafungen erhalten. Auch der Aufwand der Implementierung ist bei SL eher geringer als bei DRL. Einen Vorteil hat DRL bei der Lösungsqualität der Zielfunktionswerte, da der RMSE fünfmal niedriger ist verglichen mit SL. Der MAPE ist jedoch bei SL um etwa 0.12 % besser, was aber nicht im Vergleich mit dem deutlichen Vorteil beim RMSE ist. Bei den Belohnungen zeigen SL und DRL etwa den gleichen Durchschnittswert, wobei DRL etwas geringere Ausreißer hat. DRL produziert verglichen mit SL 0.013 % weniger ungültige Lösungen. Die durchschnittliche Zeit zur Berechnung einer Aktion ist mit etwa 0.003 Sekunden bei beiden Verfahren gleich. Insgesamt sind die Ergebnisse je nach Vergleichskriterium unterschiedlich, jedoch hat DRL bei den Zielfunktionswerten leicht bessere Ergebnisse erhalten, sodass DRL etwas vielversprechender ist.

Ausblick

In dieser Arbeit wurde eine Umgebung verwendet, jedoch sollten für den Vergleich mehr als eine Umgebung verwendet werden, um sicherzustellen, dass die Ergebnisse von der Umgebung unabhängig sind. In Abschnitt A.1 werden die Ergebnisse von der Wirkleistungsmarkt-Umgebung dargestellt, die nicht in die Arbeit aufgenommen werden sollten, da viele Optimierungen bei DRL vorgenommen wurden und bei SL nicht. Dennoch sind die Ergebnisse von DRL in der Lösungsqualität der Zielfunktionswerte vielversprechend und deuten darauf hin, dass DRL tendenziell bessere Lösungen liefert.
Außerdem sollten die Hyperparameter weiter experimentiert werden, da eventuell noch bessere Modelle gefunden werden können. Dazu zählen auch die gemeinsamen Hyperparameter wie Optimierer oder Verlustfunktion, die in dieser Arbeit nicht mehr geändert wurden. Auch die Anzahl an trainierten Modellen sollte erhöht werden, um sicherzustellen, dass ein durchschnittliches Modell für den Vergleich gewählt wird.
Weiterhin wurden Techniken wie Random Dropout nicht benutzt, bei der zufällig Neuronen deaktiviert werden, um die Leistung des neuronalen Netzes zu steigern.
Bei der Berechnungszeit von Aktionen wurde die Zeit gemessen, die das neuronale Netz für einen Zustand benötigt. Neuronale Netze können aber größere Daten in Batches empfangen, sodass die Zeitersparnis größer wird. Der Zeitunterschied zum OPF Löser würde dadurch noch weiter ansteigen.

A Anhang

A.1 Wirkleistungsmarkt-Umgebung Ergebnisse

A.1.1 Supervised Learning Training

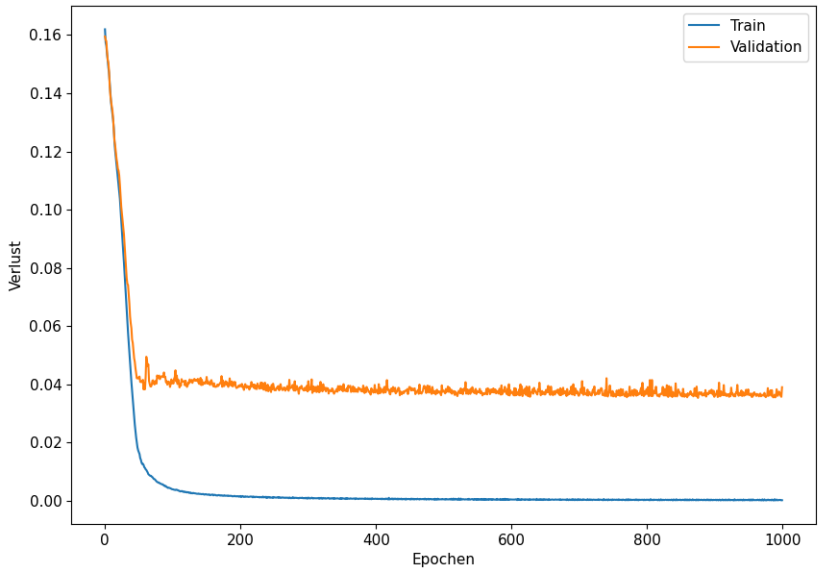

Abbildung A.1: Verlust des Modells zu den optimalen Aktionen während des Trainings (Wirkleistungsmarkt)

A.1.2 Deep Reinforcement Learning Training

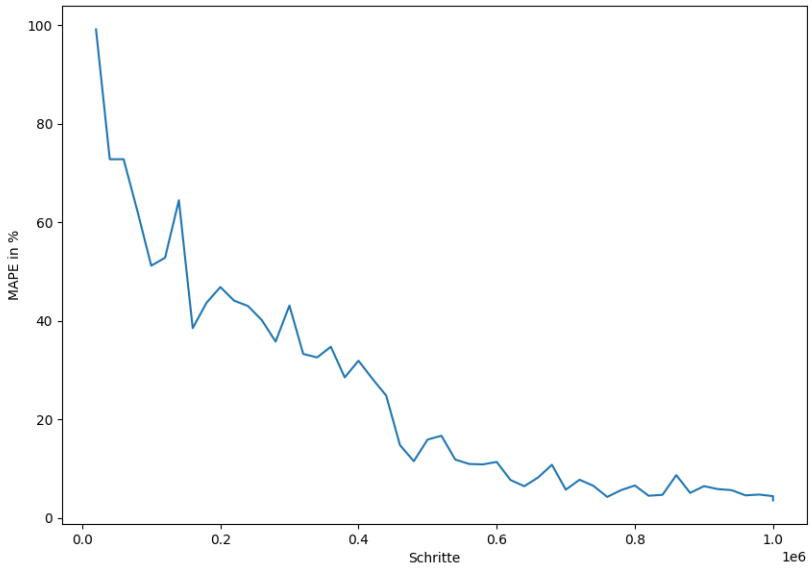

Abbildung A.2: MAPE des Modells zu den optimalen Belohnungen während des Trainings (Wirkleistungsmarkt)

A.1.3 Vergleich der Ergebnisse

Trainingszeit

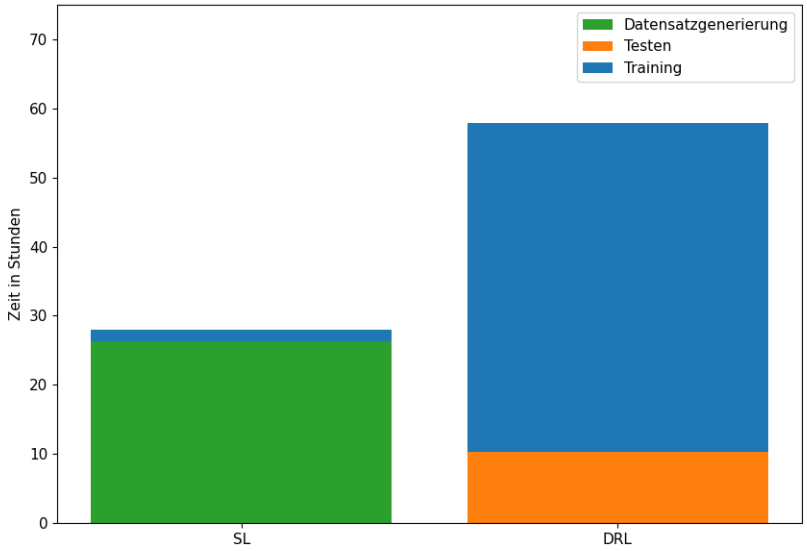

Abbildung A.3: Trainingszeit der Modelle (Wirkleistungsmarkt)

Lösungsqualität Aktionen

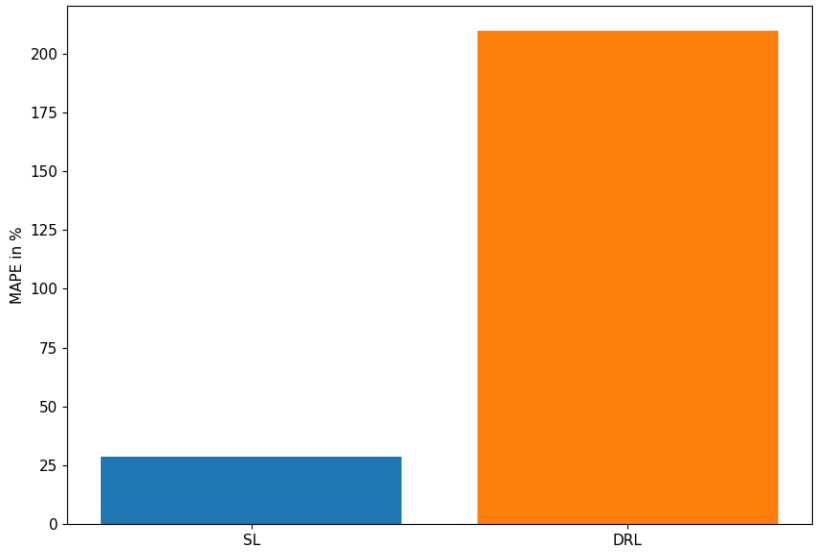

Abbildung A.4: MAPE zu den optimalen Belohnungen (Wirkleistungsmarkt)

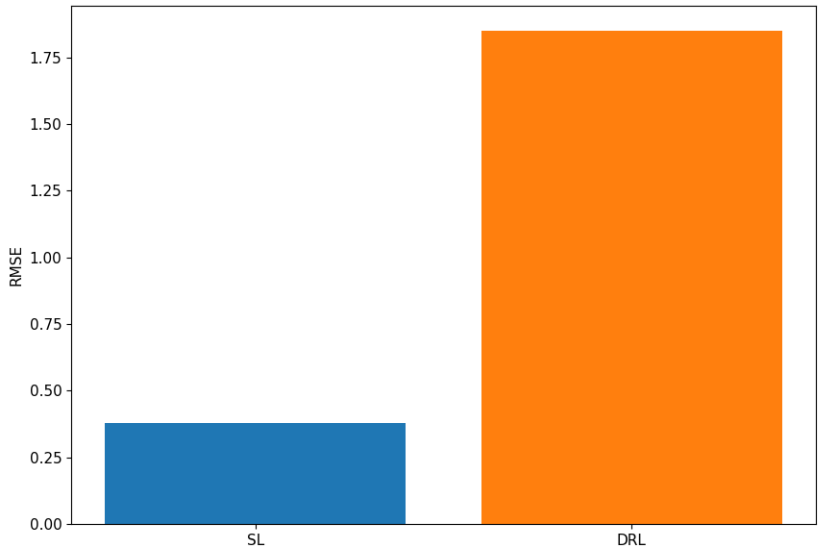

Abbildung A.5: RMSE zu den optimalen Belohnungen (Wirkleistungsmarkt)

Lösungsqualität Zielfunktion

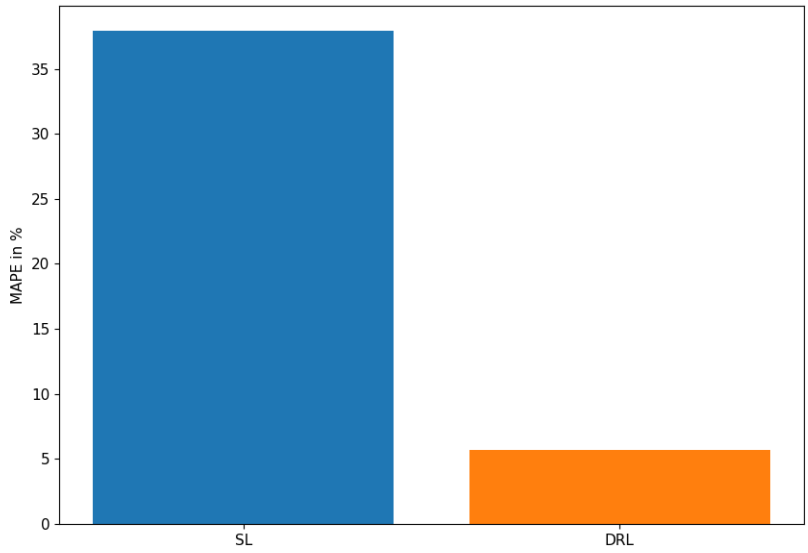

Abbildung A.6: MAPE zu den optimalen Zielfunktionswerten

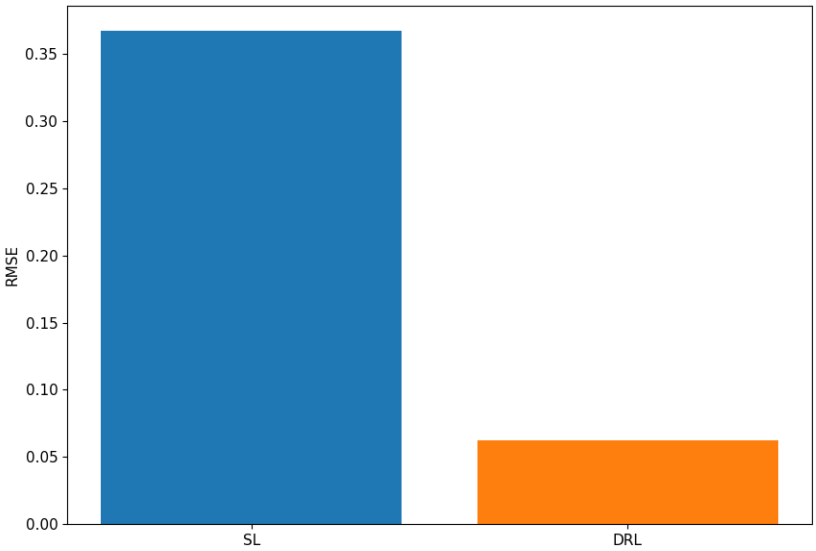

Abbildung A.7: RMSE zu den optimalen Zielfunktionswerten

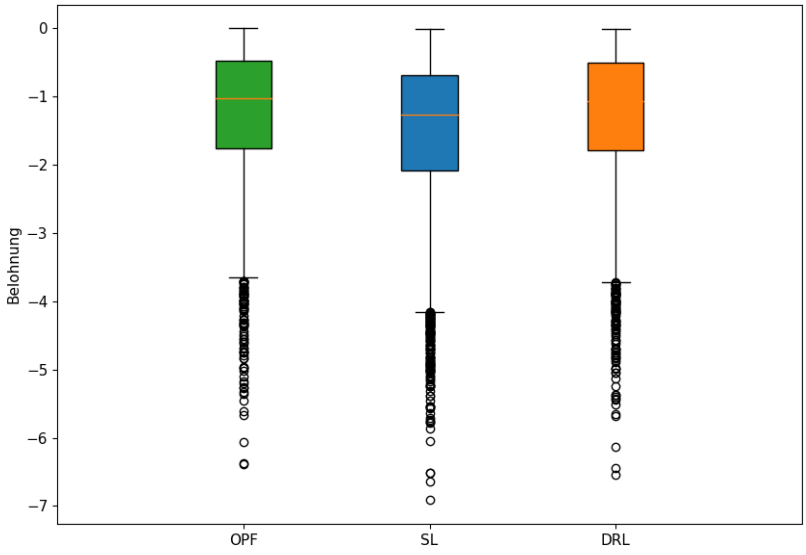

Abbildung A.8: Boxplots der Zielfunktionswerte inklusive Ausreißer (Wirkleistungsmarkt)

Im Gegensatz zu Abbildung 5.12 wurden hier die ungültigen Lösungen herausgefiltert.

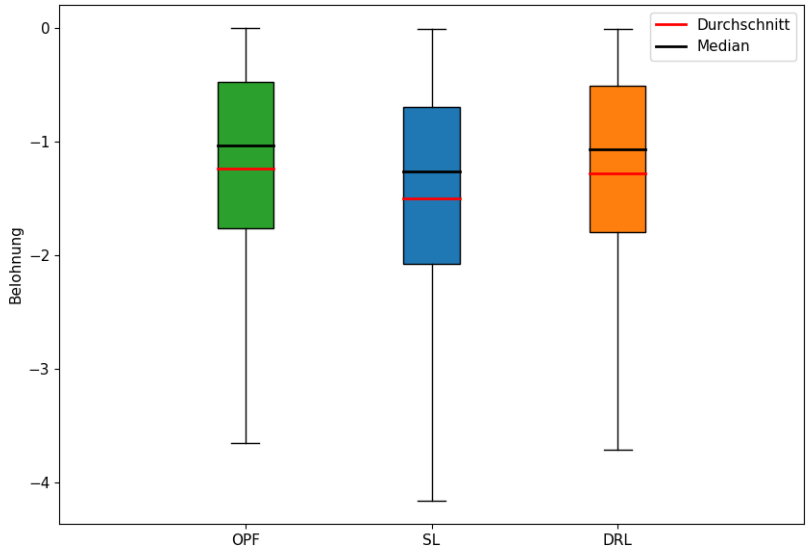

Abbildung A.9: Boxplots der Zielfunktionswerte (Wirkleistungsmarkt)

Im Gegensatz zu Abbildung 5.11 wurden hier die ungültigen Lösungen herausgefiltert.

Einhaltung der Nebenbedingungen

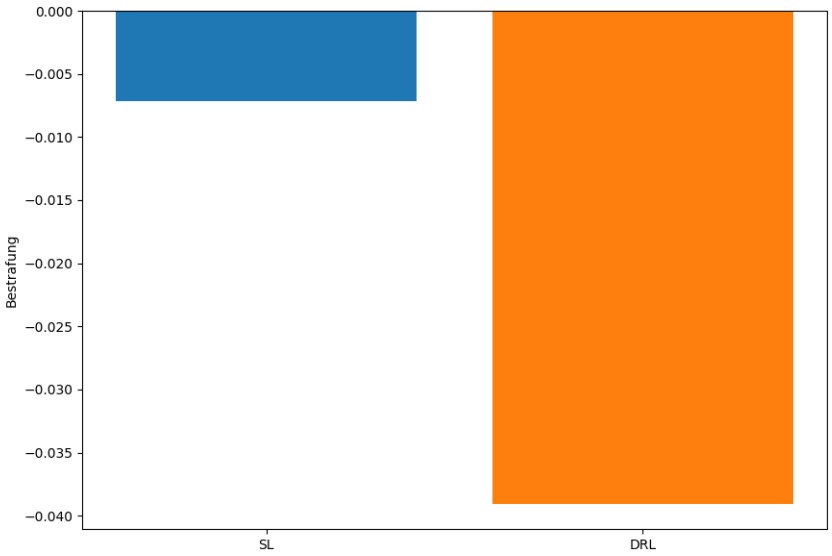

Abbildung A.10: Durchschnittliche Bestrafungen (Wirkleistungsmarkt)

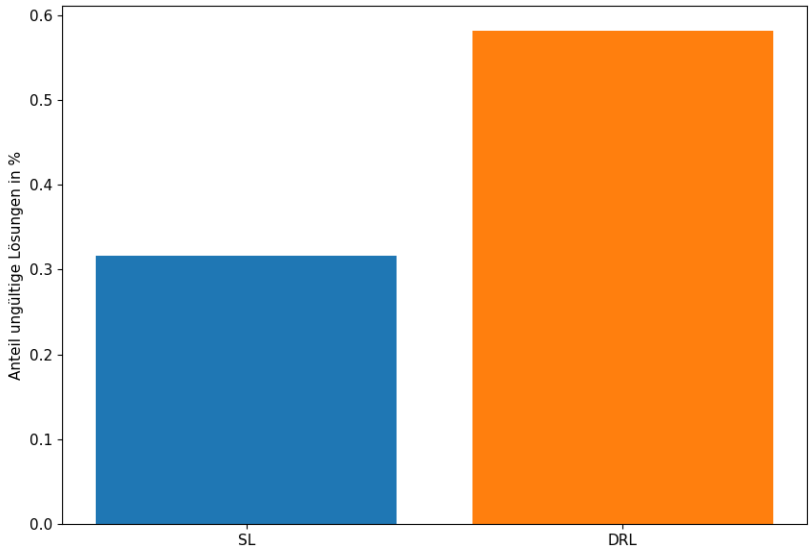

Abbildung A.11: Anteil an ungültigen Lösungen von den gesamten Lösungen (Wirkleistungsmarkt)

Abbildungsverzeichnis

Literaturverzeichnis

[1] Frank, S., Steponavice, I., Rebennack, S.: Optimal power flow: a bibliographic survey i. Energy systems **3**(3), 221–258 (2012)

[2] Pan, X., Zhao, T., Chen, M., Zhang, S.: Deepopf: A deep neural network approach for security-constrained dc optimal power flow. IEEE Transactions on Power Systems **36**(3), 1725–1735 (2021). doi:10.1109/TPWRS.2020.3026379

[3] Cao, D., Hu, W., Xu, X., Wu, Q., Huang, Q., Chen, Z., Blaabjerg, F.: Deep reinforcement learning based approach for optimal power flow of distribution networks embedded with renewable energy and storage devices. Journal of Modern Power Systems and Clean Energy **9**(5), 1101–1110 (2021). doi:10.35833/MPCE.2020.000557

[4] Baker, K.: Solutions of dc opf are never ac feasible. In: Proceedings of the Twelfth ACM International Conference on Future Energy Systems, pp. 264–268 (2021)

[5] Jo, T.: Machine Learning Foundations. Springer, Garosuro Cheongju, Korea (2021)

[6] Zhou, Z.-H.: Machine Learning. Springer, Singapur (2021)

[7] McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics **5**(4), 115–133 (1943)

[8] Bhattacharyya, S., Snasel, V., Hassanien, A.E., Saha, S., Tripathy, B.: Deep Learning: Research and Applications vol. 7. Walter de Gruyter GmbH & Co KG, Berlin and Boston (2020)

[9] Frochte, J..-V., Frochte, J.: Maschinelles Lernen : Grundlagen und Algorithmen in Python, 3., überarbeitete und erweiterte auflage edn. (2021)

[10] Mousavi, S.S., Schukat, M., Howley, E.: Deep reinforcement learning: an overview. In: Proceedings of SAI Intelligent Systems Conference, pp. 426–440. Springer, Galway, Irland (2018)

[11] Alpaydın, E..-V., Alpaydın, E.: Maschinelles Lernen, 3., aktualisierte und erweiterte auflage. edn. De Gruyter Studium, Wien (2022)

[12] Plaat, A.: Deep Reinforcement Learning, a textbook (2022). https://arxiv.org/abs/2201.02135

[13] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press, Cambridge and London (2018)

[14] Arulkumaran, K., Deisenroth, M.P., Brundage, M., Bharath, A.A.: Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine **34**(6), 26–38 (2017)

[15] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning (2019). https://arxiv.org/abs/1509.02971

[16] Zhao, T., Pan, X., Chen, M., Venzke, A., Low, S.H.: DeepOPF+: A Deep Neural Network Approach for DC Optimal Power Flow for Ensuring Feasibility. arXiv (2020). doi:10.48550/ARXIV.2009.03147. https://arxiv.org/abs/2009.03147

[17] Zamzam, A.S., Baker, K.: Learning optimal solutions for extremely fast ac optimal power flow. In: 2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), pp. 1–6 (2020). doi:10.1109/SmartGridComm47815.2020.9303008

[18] Kim, M., Kim, H.: Projection-aware deep neural network for dc optimal power flow without constraint violations. In: 2022 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), pp. 116–121 (2022). doi:10.1109/SmartGridComm52983.2022.9961047

[19] Huang, W., Pan, X., Chen, M., Low, S.H.: DeepOPF-V: Solving AC-OPF Problems Efficiently. arXiv (2021). doi:10.48550/ARXIV.2103.11793. https://arxiv.org/abs/2103.11793

[20] Velloso, A., Van Hentenryck, P.: Combining deep learning and optimization for preventive security-constrained dc optimal power flow. IEEE Transactions on Power Systems **36**(4), 3618–3628 (2021). doi:10.1109/TPWRS.2021.3054341

[21] Wu, H., Xu, Z.: Fast dc optimal power flow based on deep convolutional neural network. In: 2022 IEEE 5th International Electrical and Energy Conference (CIEEC), pp. 2508–2512 (2022). doi:10.1109/CIEEC54735.2022.9846143

[22] Yan, Z., Xu, Y.: Real-time optimal power flow: A lagrangian based deep reinforcement learning approach. IEEE Transactions on Power Systems **35**(4), 3270–3273 (2020). doi:10.1109/TPWRS.2020.2987292

[23] Zeng, L., Sun, M., Wan, X., Zhang, Z., Deng, R., Xu, Y.: Physics-constrained vulnerability assessment of deep reinforcement learning-based scopf. IEEE Transactions on Power Systems, 1–15 (2022). doi:10.1109/TPWRS.2022.3192558

[24] Sayed, A.R., Wang, C., Anis, H., Bi, T.: Feasibility constrained online calculation for real-time optimal power flow: A convex constrained deep reinforcement learning approach. IEEE Transactions on Power Systems, 1–13 (2022). doi:10.1109/TPWRS.2022.3220799

[25] Nie, J., Liu, Y., Zhou, L., Jiang, X., Preindl, M.: Deep reinforcement learning based approach for optimal power flow of microgrid with grid services implementation. In: 2022 IEEE Transportation Electrification Conference & Expo (ITEC), pp. 1148–1153 (2022). doi:10.1109/ITEC53557.2022.9813862

[26] Zhou, Y., Zhang, B., Xu, C., Lan, T., Diao, R., Shi, D., Wang, Z., Lee, W.-J.: A data-driven method for fast ac optimal power flow solutions via deep reinforcement learning. Journal of Modern Power Systems and Clean Energy **8**(6), 1128–1139 (2020)

[27] Guo, L., Guo, J., Zhang, Y., Guo, W., Xue, Y., Wang, L.: Real-time decision making for power system via imitation learning and reinforcement learning. In: 2022 IEEE/IAS Industrial and Commercial Power System Asia (I&CPS Asia), pp. 744–748 (2022). doi:10.1109/ICPSAsia55496.2022.9949821

[28] Wolgast, T.: Create Reinforcement Learning Environments of Pandapower Power System Models. `https://gitlab.com/thomaswolgast/mlopf`, Oldenburg (2023). Abgerufen: 02.03.2023

[29] Wolgast, T.: Deep Reinforcement Learning. `https://gitlab.com/thomaswolgast/drl`, Oldenburg (2023). Abgerufen: 02.03.2023

[30] Allwright, S.: RMSE Vs MAPE, Which Is the Best Regression Metric? `https://stephenallwright.com/rmse-vs-mape/`, Norwegen (2022). Abgerufen: 23.03.2023

[31] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms. In: International Conference on Machine Learning, pp. 387–395 (2014). Pmlr

[32] Bartz, E., Bartz-Beielstein, T., Zaefferer, M., Mersmann, O.: Hyperparameter Tuning for Machine and Deep Learning with R: A Practical Guide. Springer, Deutschland (2023)

Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichungen, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

K. Ortmann

Oldenburg, den 24. April 2023 Keno Ortmann