Diplomarbeit

Lernen im komprimierten Raum

Alexander Fabisch
Matrikelnummer 2197753

Erstgutachter:	Prof. Dr. Frank Kirchner
Zweitgutachter:	Dr.-Ing. Thomas Röfer
Betreuer:	Dr.-Ing. Yohannes Kassahun

Universität Bremen
Fachbereich 3

Bremen, 20. Juli 2012

Erklärung

Ich versichere, die Diplomarbeit oder den von mir zu verantwortenden Teil einer Gruppenarbeit[1] ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 20. Juli 2012

_____

Alexander Fabisch

[1] Bei einer Gruppenarbeit muss die individuelle Leistung deutlich abgrenzbar und bewertbar sein und den Anforderungen der Absätze 1 und 2 §13 der DPO - Allgemeiner Teil - entsprechen.

Inhaltsverzeichnis

1. Einleitung

In dieser Arbeit wird ein Verfahren zur Beschleunigung des Trainings großer künstlicher neuronaler Netze entwickelt und anhand verschiedener Probleme aus den Bereichen überwachtes Lernen und Reinforcement Learning [86] geprüft.

1.1. Motivation

Zu Beginn der Entwicklung künstlicher Intelligenz waren an diese große Erwartungen geknüpft. Simon und Newell [80] schreiben beispielsweise im Jahr 1958:

> „Es ist nicht mein Ziel Sie zu schockieren [...]. Aber die einfachste Art und Weise, auf die ich die Situation zusammenfassen kann, ist zu sagen, dass nun in dieser Welt Maschinen sind, die denken, die lernen und die erschaffen. Darüber hinaus wird deren Fähigkeit, diese Dinge zu tun, sich rasant entwickeln, bis - in einer absehbaren Zukunft - die Menge von Problemen, mit denen sie umgehen können, der Menge entsprechen wird, für die der menschliche Geist ausgelegt ist.“

Zudem wurden in dieser Veröffentlichung einige Vorhersagen für die nächsten zehn Jahre gemacht, die allerdings nicht eintraten. Unter anderem solle ein Computer in der Lage sein, den Schachweltmeister zu schlagen. Dieses Ziel wurde zwar nicht innerhalb der folgenden zehn Jahre, jedoch ungefähr nach 40 Jahren erreicht.

Heutzutage sind die Probleme der künstlichen Intelligenz - und insbesondere in dem Teilbereich maschinelles Lernen - viel komplexer. Durch die Forschung in der Robotik, dem Reinforcement Learning, der Bildverarbeitung und verwandter Disziplinen wird maschinelles Lernen immer häufiger mit der realen Welt konfrontiert. Einige Beispiele hierfür sind das Steuern von Autos [70], die Erkennung von Verkehrszeichen [82] oder handgeschriebenen Ziffern [52] und in Ansätzen der RoboCup [62]. Es sieht zwar bisher nicht so aus, als würde die künstliche Intelligenz innerhalb der nächsten Jahre ein ähnliches Niveau wie die natürliche Intelligenz erreichen, allerdings sind zumindest bei manchen Problemen mit realen Bilddaten bereits Systeme entwickelt worden, die ähnlich gut oder besser als Menschen sind. So können beispielsweise neuronale Netze bereits besser als Menschen Verkehrszeichen erkennen [83].

Reale Probleme weisen häufig die folgenden Charakteristika auf. Die *Datenmengen in der realen Welt sind immens.* Kameras liefern beispielsweise Tausende bis Millionen von Pixel pro Bild. Bei robotischen Systemen kommen häufig noch viele weitere Sensoren hinzu, die Informationen über Lage, Geschwindigkeit oder Gelenkwinkel liefern. Viele *Daten sind allerdings mit geringem Informationsverlust komprimierbar.* Dies liegt daran, dass entweder viele Datenkomponenten häufig keine Informationen liefern oder redundante Informationen bereitstellen, das heißt „benachbarte“ Daten

sind häufig ähnlich und weisen deshalb eine hohe Korrelation auf. Bei Bildern sind benachbarte Pixel beispielsweise häufig ähnlich und dies kann zum Beispiel bei komprimierten Bildformaten ausgenutzt werden. Zudem sind Sensordaten oft *verrauscht, fehlerbehaftet und ungenau* [88, Seite 3-4]. Bei Daten von Brain-Computer-Interfaces (BCIs) [49] ist das Rauschen beispielsweise oft deutlich größer als das Signal, das klassifiziert werden soll. Ferner sind nicht nur viele Sensordaten verfügbar, sondern auch noch sehr viele Trainingsbeispiele. Zum einen ist die Menge der Beispiele beim überwachten Lernen oft nur durch die zur Verfügung stehende menschliche Arbeitszeit beim Kennzeichnen der Daten beschränkt, zum anderen werden allerdings auch sehr viele Beispiele benötigt. Ein mathematischer Grund dafür ist der sogenannte *Curse of Dimensionality* für Lernverfahren [74, Seite 739]. Intuitiver lässt sich dies allerdings anhand von Bildern verdeutlichen: ein Objekt kann in einem Bild rotiert, verschoben, skaliert oder verzerrt werden, der Hintergrund oder die Beleuchtung können sich ändern und dennoch sollte es als derselbe Gegenstand erkannt werden. Dafür werden normalerweise sehr viele Trainingsbeispiele benötigt oder es wird menschliche Arbeitszeit in die Entwicklung von Methoden zur Berechnung invarianter Eigenschaften investiert.

Um mit realen Daten umgehen zu können, wurden in den letzten Jahren vor allem Varianten neuronaler Netze mit vielen Schichten [7, 22, 50] genutzt. Diese sollen effizienter für den Umgang mit großen Datenmengen sein und eher dem menschlichen Gehirn ähneln [6]. Einige neuronale Netze, wie zum Beispiel Higher-Order Neural Networks [81] und Convolutional Neural Networks [50], sollen in der Lage sein, auch mit weniger Trainingsbeispielen invariante Eigenschaften von Klassen zu lernen.

Dennoch stellen große Datenmengen für jedes Lernverfahren ein Problem während der Trainingsphase dar. Oft dauert das Training Tage, Wochen oder Monate. Dieses Problem kann teilweise mit Hardware gelöst werden: durch den Einsatz von vielen Rechenkernen, Rechnern oder Grafikkarten [22] können viele Berechnungen parallel durchgeführt werden. In einigen Fällen reicht dies bereits. In dieser Arbeit wird allerdings eine weitere Methode untersucht, die die Lerndauer verkürzen soll und ohne großen Entwicklungsaufwand auf viele neuronale Netze übertragen werden kann. Als Testplattform wurde dazu ein spezielles neuronales Netz ausgewählt: das Multilayer Perceptron (MLP).

Bei dem Multilayer Perceptron sind die Neuronen in verschiedenen Schichten angeordnet und ein Signal kann jeweils nur in eine Richtung durch alle Schichten geleitet werden. Es kann zur Klassifikation und Regression eingesetzt werden. Insbesondere kann es als Funktionsapproximator zur Lösung von Aufgaben aus dem Bereich des Reinforcement Learning eingesetzt werden. Das Multilayer Perceptron generiert Modelle, die schnell ausgewertet werden können. Das Lernen von komplexen Aufgaben mittels eines Multilayer Perceptrons dauert hingegen sehr lange, da durch die hohe Anzahl von Eingabekomponenten die Anzahl der zu optimierenden Gewichte sehr groß wird und der Suchraum somit ebenfalls (siehe Abbildung 1.1). Dies verursacht ein Problem, das als *Curse of Dimensionality* für Optimierungsverfahren bezeichnet wird (siehe Abbildung 1.2). Der Begriff „Curse of Dimensionality“ wurde von Bellman [5] eingeführt. Auf Optimierungsverfahren lässt sich das Problem wie folgt übertragen: ein Optimierungsalgorithmus sollte in der Lage sein, in einem n-dimensionalen Einheitswürfel ein Optimum zu finden. Ein sehr grobes, globales Suchverfahren könnte

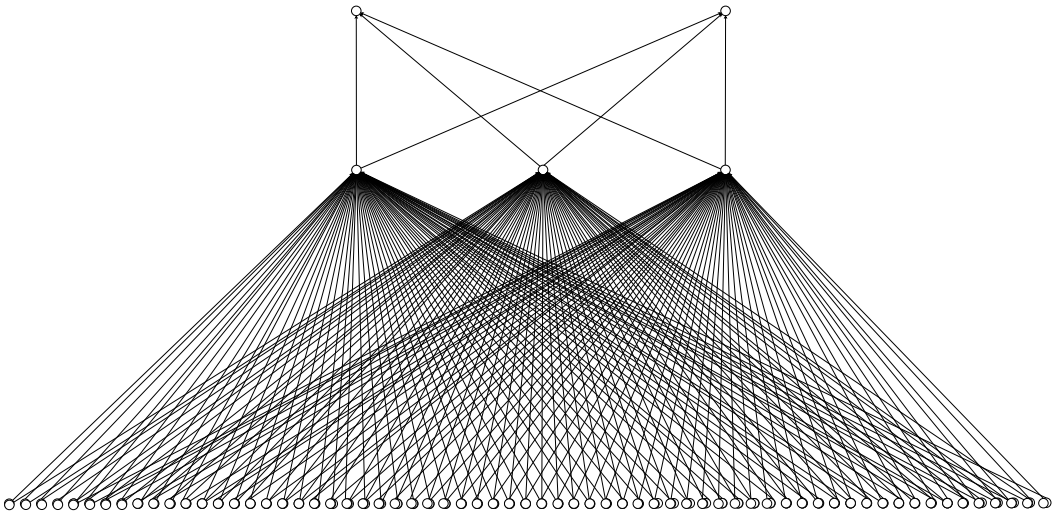


Abbildung 1.1.: Einfaches Multilayer Perceptron mit großem Eingabevektor. Durch die Größe des Eingabevektors steigt die Anzahl der Verbindungen und somit auch die Anzahl der Gewichte des MLP stark an. In diesem Fall sind es insgesamt 201 Verbindungen.

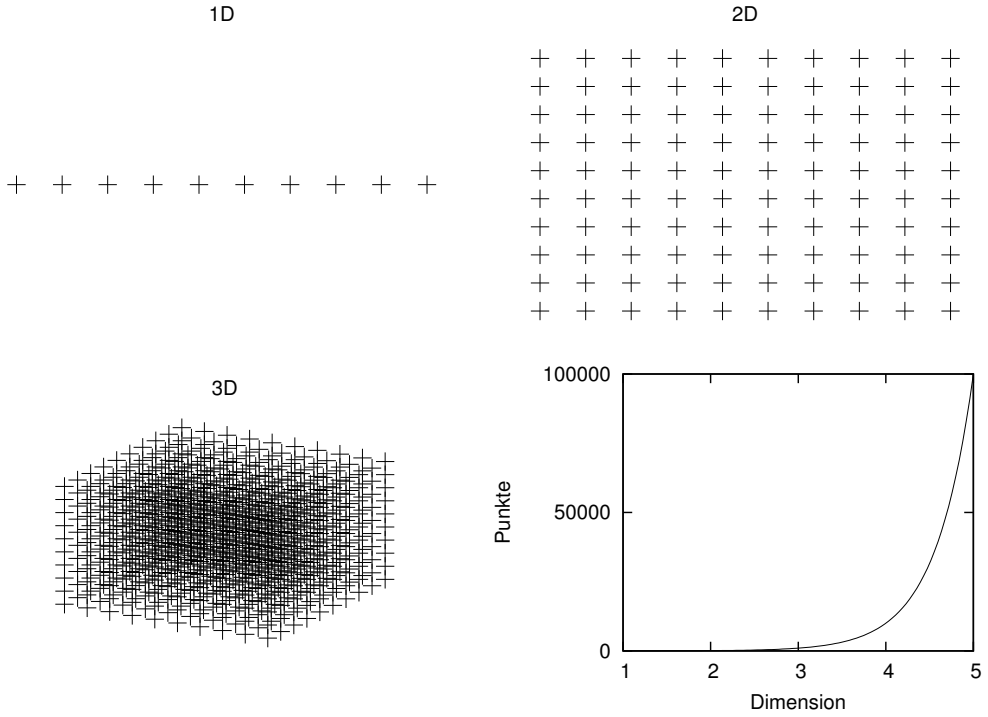


Abbildung 1.2.: Visualisierung des *Curse of Dimensionality*: die Anzahl der Punkte mit gleichmäßigen Abständen im Einheitswürfel steigt mit der Dimension exponentiell an.

beispielsweise in einer Dimension 10 Werte in einem Abstand von 0,1 durchprobieren, um ein Optimum grob zu approximieren. Würde man dieses Verfahren auf zwei Dimensionen erweitern, so müsste man zunächst in der ersten Dimension 10 Werte im Abstand 0,1 und für jeden dieser Werte 10 Werte in der zweiten Dimension durchprobieren. Dies wären bereits 10^2 Werte. Für n Dimensionen sind dies 10^n verschiedene Werte. Die zu prüfenden Werte steigen also bereits bei einem sehr groben, naiven Verfahren exponentiell an. Da bei einem neuronalen Netz durch die Ableitung die ungefähre Richtung zum nächsten Optimum in den meisten Fällen vorgegeben ist, ist die Auswirkung allerdings nicht so stark.

In dieser Diplomarbeit soll ein Verfahren entwickelt werden, bei dem die Gewichte in einer komprimierten Form repräsentiert werden, sodass der Suchraum eine geringere Dimension hat. Jedem Neuron wird dabei eine gewichtete Summe von orthogonalen Funktionen - ähnlich einer Fourier-Transformation - zugeordnet. Aus dieser gewichteten Summe sollen die Gewichte generiert werden. Die Gewichte der Funktionen stellen die Parameter dar, die optimiert werden sollen. Diese Methode ist als zusätzliches Werkzeug auf verschiedene Lernverfahren anwendbar.

1.2. Ähnliche Arbeiten

Die in dieser Arbeit entwickelte Idee zur Komprimierung neuronaler Netze ist nicht vollständig neu. In diesem Abschnitt werde ich kurz die zugrunde liegenden Arbeiten vorstellen.

Koutník u. a. [45] motivieren die Komprimierung von Gewichten eines neuronalen Netzes durch die Suche nach dem neuronalen Netz, das am besten zuvor ungesehene Daten vorhersagt, das heißt nach dem am besten generalisierenden neuronalen Netz. Nach dem Prinzip der *Minimum Description Length* (MDL) sei es dazu erforderlich, das *einfachste* neuronale Netz zu finden, das einen geringen Fehler auf den Trainingsdaten hat. Dabei wurde Einfachheit durch eine geringe Beschreibungslänge und eine somit geringe Kolmogorov-Komplexität definiert. Die Kolmogorov-Komplexität misst eigentlich die Länge eines Programms.

Schmidhuber [76, 77] verfolgte bereits vorher eine ähnlich Strategie, allerdings arbeitete er tatsächlich noch mit einer sogenannten universellen *Network Encoding Language* (NEL), die ein neuronales Netz erzeugt. Die Generalisierungsleistung ist hierbei in einigen Testproblemen gut [77]. Der große Nachteil eines solchen Verfahrens ist allerdings, dass die Funktion, die das neuronale Netz aus der komprimierten Repräsentation erzeugt, nicht kontinuierlich ist und somit viele Optimierungsverfahren für die effiziente Suche in dem komprimierten Raum ungeeignet sind. Dies ist einer der Gründe, die zur Komprimierung der Gewichte eines neuronalen Netzes durch Fourierreihenkoeffizienten geführt haben [45]. Dieser Ansatz soll eine kontinuierliche Funktion liefern, die vom komprimierten Suchraum in den Gewichtsraum abbildet. Das heißt eine kleine Veränderung der Koeffizienten hat eine kleine Änderung aller Gewichte zur Folge. Dadurch soll über die Anzahl der Koeffizienten eine einfache Kontrolle der Kolmogorov-Komplexität gewährleistet werden. Ein Ziel dieser Arbeit war die Reduktion der Trainingsdauer.

Koutník u. a. [45] gehen davon aus, dass die Gewichte eines neuronalen Netzes miteinander korrelieren, was bei neuronalen Netzen, die beispielsweise für Bildverarbeitung benötigt werden, der Fall sein könne. Allerdings wird hier auch bereits vorgeschlagen Gewichte verschiedener Neuronen durch verschiedene Koeffizienten zu komprimieren, da nicht bei allen Lernproblemen die Gewichte des kompletten neuronalen Netzes miteinander korrelieren. Genau dieser Ansatz wird hier verfolgt. Die Gewichte eingehender Verbindungen jedes Neurons werden einzeln komprimiert.

Koutník u. a. [44] haben das von ihnen entwickelte Verfahren später verfeinert, sodass die Gewichte eines vollständig verbundenen rekurrenten neuronalen Netzes zunächst durch eine diskrete Kosinustransformation komprimiert werden und dann der Optimierungsalgorithmus Cooperative Synapse Neuroevolution (CoSyNE) [32] das neuronale Netz optimiert. Das Verfahren ist nicht auf vollständig verbundene rekurrente neuronale Netze beschränkt, da andere Netze, wie zum Beispiel Feedforward-Netze, Spezialfälle dieses Typs darstellen. Die Arbeiten von Koutník u. a. [44, 45] beschränken sich zunächst auf Reinforcement Learning.

Für Feedforward-Netze kann allerdings beim überwachten Lernen eine Ableitung berechnet werden, die die Anwendung schnellerer Optimierungsverfahren ermöglicht. Dies werde ich in dieser Arbeit tun. Dabei stütze ich mich auf das Backpropagation-Verfahren zur Berechnung des Gradienten eines normalen, allgemeinen Feedforward--Netzes, welches durch Rumelhart u. a. [73] entwickelt wurde. Die exakte zweite Ableitung (Hesse-Matrix) von normalen, allgemeinen Feedforward-Netzen wurde durch Bishop [9] hergeleitet. Die Herleitung der Hesse-Matrix mit komprimierten Gewichten in dieser Arbeit orientiert sich stark an der Herleitung aus dieser Veröffentlichung.

1.3. Ziele

Das Ziel dieser Arbeit ist die Entwicklung einer Erweiterung des Multilayer Perceptrons um die erwähnte komprimierte Gewichtsrepräsentation und die Herleitung eines angepassten Backpropagation-Verfahrens und einer exakten Hesse-Matrix.

Im Vergleich zu Koutník u. a. [44] wird hier keine Lösung für rekurrente neuronale Netzwerke entwickelt, sondern nur für Feedforward-Netze. Allerdings werden hier anstatt Kosinusfunktionen beliebige orthogonale Funktionen zugelassen und neben einem erweiterten Backpropagation-Verfahren auch die Hesse-Matrix angegeben, sodass Verfahren zur Optimierung der Gewichte verwendet werden können, die auf diese Ableitungen angewiesen sind. Dadurch kann die Lerndauer beim überwachten Lernen reduziert werden.

Zur Evaluierung dieser Methode soll das entwickelte Lernverfahren in verschiedenen Domänen getestet werden, wobei insbesondere herausgestellt werden soll, dass die Trainingsdauer durch eine Gewichtskomprimierung abnimmt. Dies soll anhand von Problemen aus dem Bereich des überwachten Lernens und des Reinforcement Learning gezeigt werden. Im Reinforcement Learning wird das MLP jeweils als Funktionsapproximator eingesetzt. Wenn der Aktionsraum diskret ist, soll dabei die Nutzenfunktion $Q(s, a)$, die den Nutzen einer Aktion a in einem Zustand s angibt, approximiert werden. Wenn der Aktionsraum kontinuierlich ist, wird das MLP eine Strategie $\pi(s)$, die einem Zustand s eine Aktion zuordnet, direkt lernen. Durch die Anwendung auf ver-

schiedene Probleme soll gezeigt werden, dass insbesondere bei Problemen, die eine große Anzahl von Gewichten in der Eingabeschicht benötigen, eine Beschleunigung der Trainingsphase feststellbar ist.

1.4. Überblick

In Kapitel 2 werden die Grundlagen neuronaler Netze und des MLP zusammengefasst und es wird begründet, warum das MLP für den genannten Einsatzzweck geeignet ist. In Kapitel 3 werden die mathematischen Grundlagen für das Lernen von MLPs im komprimierten Gewichtsraum hergeleitet. Es werden eine Lösung für das Backpropagation-Verfahren allgemeiner Feedforward-Netze angegeben und die Hesse-Matrix hergeleitet. Die theoretischen Grundlagen schließen in Kapitel 4 mit einer Bewertung der Eignung einiger beliebter Optimierungsverfahren zum Lernen von neuronalen Netzen ab.

In zwei verschiedenen Domänen wird das entwickelte Lernverfahren dann geprüft. Zunächst wird untersucht, ob ein komprimiertes neuronales Netz mit dem sehr schweren, aber künstlichen Benchmark „Two Spirals“ zurechtkommt (Abschnitt 5.1). Danach wird der erste reale Datensatz untersucht. Bei diesem sollen handschriftliche Ziffern erkannt werden (Abschnitt 5.2). Es folgt eine Betrachtung der Eignung zur Klassifikation von EEG-Daten (Abschnitte 5.3 und 5.4). Im Bereich Reinforcement Learning wurden verschiedene Benchmarks ausgewählt. Diese Probleme erfordern sowohl den Umgang mit kontinuierlichem Zustands- und diskretem Aktionsraum als auch mit kontinuierlichem Zustands- und Aktionsraum. Die Betrachtung der Lernprobleme umfasst

- eine Beschreibung des Problems (Datensatz oder Reinforcement-Learning-Umgebung),
- eine Beschreibung der Methode die zur Lösung des Problems entwickelt wurde,
- eine Auswertung der Ergebnisse,
- gegebenenfalls einen Vergleich zu anderen veröffentlichten Ergebnissen
- und eine Zusammenfassung der Erkenntnisse, die aus der Anwendung gezogen werden können.

Diese Diplomarbeit wird mit einer Zusammenfassung der Ergebnisse und einem Ausblick auf zukünftige Arbeiten und mögliche Anwendungen der entwickelten Methode abgeschlossen (Kapitel 7).

2. Künstliche Neuronale Netze

Künstliche neuronale Netze werden zur Funktionsapproximation genutzt. Sie sollten ursprünglich Prozesse in Gehirnen nachahmen. Die Grundlagen wurden unter anderem durch folgende Personen entwickelt: McCulloch und Pitts [58] untersuchten als erste künstliche Neuronen, Hebb [36] beschrieb die nach ihm benannte Hebb'sche Lernregel, wodurch zum ersten Mal künstliche Neuronen zum Lernen eingesetzt wurden, Rosenblatt [72] entwickelte das Perzeptron, ein sehr einfaches künstliches neuronales Netz, das zum Beispiel nicht in der Lage war, die XOR-Funktion darzustellen. Das Lernen allgemeiner Feedforward-Netze ist durch die von Rumelhart u. a. [73] entwickelte Backpropagation möglich.

Hier werden die für diese Arbeit wichtigen Grundlagen neuronaler Netze dargestellt. Dabei orientiere ich mich inhaltlich an Bishop [11], werde allerdings einige Änderungen vornehmen. Eine Übersicht über die verwendeten mathematischen Symbole ist in Anhang A zu finden.

2.1. Vorwärtspropagation

Die Eingabe eines neuronalen Netzes ist ein beliebiger Vektor $\mathbf{x} \in \mathbb{R}^D$ und die Ausgabe ein Vektor $\mathbf{y} \in \mathbb{R}^F$, wobei $D, F \in \mathbb{N}$. Das künstliche Neuron, der Grundbaustein jedes neuronalen Netzes, ist in Abbildung 2.1 (a) schematisch dargestellt. Jedes Neuron innerhalb eines Netzes hat beliebig viele Eingänge $x_1, \ldots, x_n$ und einen Ausgang, der wiederum mit beliebig vielen anderen Neuronen verbunden werden kann. Die Ausgabe eines Neurons wird durch Aufsummieren der durch w_{ji} gewichteten Eingaben x_i zum Wert

$$a_j = \sum_i w_{ji} x_i \tag{2.1.1}$$

und anschließender Transformation durch eine Aktivierungsfunktion g berechnet.

Die Approximation beliebiger Funktionen zur Klassifikation oder Regression wird durch das Anpassen der Gewichte des neuronalen Netzes vorgenommen. Dieser Vorgang wird als Training oder Lernen bezeichnet.

Man unterscheidet neuronale Netze grundsätzlich zwischen Feedforward-Netzen, die in dieser Arbeit behandelt werden, und rekurrenten Netzen, bei denen Zyklen in den Verbindungen vorkommen dürfen, sodass eine Art Gedächtnis realisierbar ist.

Eine populäre Variante der Feedforward-Netze ist das Multilayer Perceptron (MLP), das in Abbildung 2.1 (b) schematisch dargestellt ist. Bei dieser speziellen Form sind die Neuronen in Schichten organisiert, wobei aufeinander folgende Schichten jeweils vollständig miteinander verbunden sind. Dabei wird unterschieden zwischen der Eingabe-, der Ausgabeschicht und den versteckten Schichten (englisch: *hidden layers*), die zwischen der Eingabe- und Ausgabeschicht liegen. Zusätzlich zu den Verbindungen

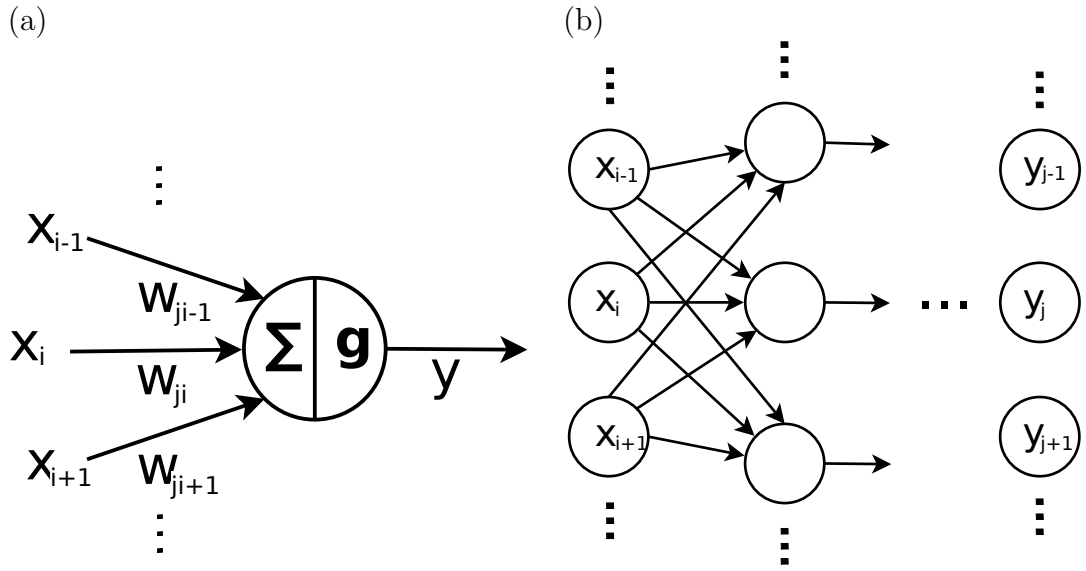


Abbildung 2.1.: (a) Künstliches Neuron. Das Neuron selbst ist durch den Kreis dargestellt, die Eingänge auf der linken Seite werden wie der Ausgang auf der rechten Seite durch Pfeile angedeutet. (b) Das Schema eines Multilayer Perceptrons. Der Eingabevektor $\mathbf{x} = (x_1, \ldots, x_D)^T$ wird auf den Ausgabevektor $\mathbf{y} = (y_1, \ldots, y_F)^T$ abgebildet.

Algorithmus 1 Vorwärtspropagation

Eingabe: $\mathbf{x} = (x_1, \ldots, x_D)^T$
Ausgabe: $\mathbf{y} = (y_1, \ldots, y_F)^T$
  for all Neuron j **do**
    $a_j \leftarrow \sum_i w_{ji} x_i$
    $z_j \leftarrow g(a_j)$
  end for
  for all Neuronen i in der Ausgabeschicht **do**
    $f \leftarrow$ Ausgabevektor-Index(i)
    $y_f \leftarrow z_i$
  end for
  return y

zur vorherigen Schicht kann es in jeder Schicht auch einen Bias geben. Das ist eine Verbindung, die immer die Eingabe 1 liefert.

Die Aktivierungsfunktion g eines Neurons in einer versteckten Schicht ist meistens eine logistische Funktion (auch Fermifunktion) oder Tangens Hyperbolicus [11, Seite 227, Abschnitt 5.1]. Die Aktivierungsfunktion der Ausgabeschicht hängt von dem Problem ab. Bei Regressionsproblemen wird in den meisten Fällen die Identitätsfunktion verwendet und bei Klassifikationsproblemen kann zum Beispiel die logistische Funktion verwendet werden, sodass die Ausgabe als Wahrscheinlichkeit einer Klassenzugehörigkeit interpretiert werden kann.

In Algorithmus 1 ist die Berechnung der Ausgabe eines allgemeinen künstlichen neuronalen Netzes schematisch dargestellt. Zu beachten ist hierbei, dass die Aktivierung jedes Neurons erst berechnet werden kann, wenn alle Eingabewerte x_i bekannt sind. Dies sind entweder direkt die Werte des Eingabevektors oder die Ausgaben anderer Neuronen.

2.2. Backpropagation

Rumelhart u. a. [73] haben eine Möglichkeit zum Berechnen des Gradienten der Fehlerfunktion nach dem Gewichtsvektor ($\nabla E_n(\mathbf{w})$) entwickelt, wodurch das Lernen von allgemeinen Feedforward-Netzen und somit auch MLPs ermöglicht wurde. Das Verfahren wird hier kurz dargestellt und in Abschnitt 3.1 erweitert. Der Begriff Backpropagation unterscheidet sich hier allerdings leicht von dem ursprünglichen. Hier bezeichnet er lediglich die Berechnung des Gradienten ohne die Anwendung eines Gradientenabstiegs zum Optimieren der Gewichte. Die Herleitung des Gradienten ist bei Rumelhart u. a. [73] oder Bishop [10, 11] nachzulesen und wird hier nicht beschrieben.

Bei einem Feedforward-Netz wird gelernt, indem die Gewichte angepasst werden. Das Ziel ist ein Gewichtsvektor $\mathbf{w}^*$, bei dem die Fehlerfunktion minimal ist. Zum Berechnen des Minimums können verschiedene Optimierungsverfahren angewandt werden. Eine Übersicht ist in Abschnitt 4 zu finden. Die meisten dieser Algorithmen benötigen den Gradienten $\nabla E_n(\mathbf{w})$, da dieser an Extrem- oder Sattelpunkten 0 ist. Optimierungsverfahren, die den Gradienten nutzen können, sind viel schneller als diejenigen, die dies nicht können.

Ein Beispiel für eine Fehlerfunktion eines einfachen neuronalen Netzes ist in Abbildung 2.2 zu sehen. Hier wurde ein MLP mit einem Neuron in der versteckten Schicht und ohne Bias optimiert ($y = w_2 \tanh w_1 x$). Die Trainingsdaten waren $\{(x_1 = 1, y_1 = 1), (x_2 = 2, y_2 = 2)\}$ und die Fehlerfunktion die halbierte Summe der quadratischen Fehler. Zu erkennen ist, dass mit dem hier verwendeten einfachen Gradientenabstieg und einer Lernrate von 0.1 nicht das globale Optimum links unten erreicht wird. Der Algorithmus, der in der Nähe von $\mathbf{w} = (0, 0)^T$ startet, bewegt sich stattdessen in Richtung des lokalen Minimums rechts unten. Außerdem wird dieses nicht erreicht, da die Schrittgröße fest ist und der Algorithmus dadurch zwischen zwei Punkten oszilliert.

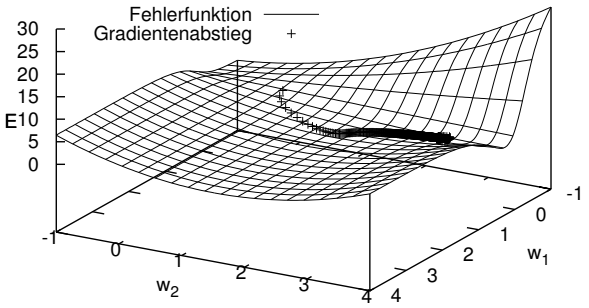


Abbildung 2.2.: Optimierung durch Gradientenabstieg. Die Optimierung startet in der Nähe von $\mathbf{w} = (0, 0)^T$ und läuft in ein lokales Minimum.

Backpropagation ist ein rekursives Verfahren zur Berechnung des Gradienten. Um das Verfahren anwenden zu können, wird ein Trainingsdatensatz

$$T = \{(x^{(1)}, t^{(1)}), \ldots, (x^{(N)}, t^{(N)})\}, \tag{2.2.2}$$

bestehend aus N Paaren von Eingabe und gewünschter Ausgabe benötigt.

Zur Ableitung der Fehlerfunktion $E(\mathbf{w})$ wird mehrfach die Kettenregel angewandt. Eine detaillierte Herleitung für die Summe der quadratischen Fehler ist bei Bishop [11] zu finden. Diese Fehlerfunktion ist definiert als

$$E = \sum_{n=1}^{N} E_n = \frac{1}{2} \sum_{n=1}^{N} \left|\left| y^{(n)} - t^{(n)} \right|\right|^2 = \frac{1}{2} \sum_{n=1}^{N} \sum_{f=1}^{F} \left(y_f^{(n)} - t_f^{(n)} \right)^2. \tag{2.2.3}$$

Im Folgenden wird für jedes Neuron j jeweils ein δ_j berechnet, das durch das Netz propagiert wird und mit dessen Hilfe die Ableitung berechnet werden kann. Es kann als Beitrag des Neurons j zu dem Fehler angesehen werden. Die folgenden Formeln beziehen sich auf ein bestimmtes Trainingsdatum n. Aus Gründen der Übersichtlichkeit wird deshalb der Index weggelassen. Nach der Summenregel für Ableitungen ist es möglich die Ableitungen der einzelnen Trainingsinstanzen aufzusummieren, um den Gradienten des gesamten Trainingsdatensatzes zu erhalten.

Für jedes Ausgabeneuron lässt sich δ_f durch

$$\frac{\partial E_n}{\partial a_f} = \delta_f = g'(a_f) \frac{\partial E_n}{\partial y_f} \tag{2.2.4}$$

berechnen. Dabei ist a_f die Aktivierung des Ausgabeneurons f. Wenn E_n die Summe der quadratischen Fehler ist, gilt

$$\frac{\partial E_n}{\partial y_f} = y_f - t_f \tag{2.2.5}$$

und für versteckte Neuronen gilt

$$\frac{\partial E_n}{\partial a_j} = \delta_j = g'(a_j) \sum_k w_{kj} \delta_k. \tag{2.2.6}$$

a_j ist die Aktivierung des Neurons j, das heißt die gewichtete Summe ohne Anwendung der Aktivierungsfunktion.

Die Ableitung lässt sich durch

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{2.2.7}$$

berechnen, wobei z_i die Ausgabe des Neurons i ist. Das Verfahren ist in Algorithmus 2 zusammengefasst.

Algorithmus 2 Backpropagation

Eingabe: $\mathbf{x}^{(n)} = (x_1, \ldots, x_D)^T, \mathbf{t}^{(n)} = (y_1, \ldots, y_F)^T$

Ausgabe: Gradient $\mathbf{g}^{(n)} = \left(\ldots, \frac{\partial E_n}{\partial w_{ji}}, \ldots\right)^T$

$\mathbf{y}^{(n)} \leftarrow$ Vorwärtspropagation$(\mathbf{x}^{(n)})$
for all Neuron j **do**
$$\delta_j \leftarrow \begin{cases} g'(a_j) \frac{\partial E_n}{\partial y_j} & \text{falls } j \text{ in der Ausgabeschicht ist} \\ g'(a_j) \sum_k w_{kj} \delta_k & \text{sonst} \end{cases}$$
end for
for all Neuronen-Paare (j, i) für die eine Verbindung von i zu j existiert **do**
$\frac{\partial E_n}{\partial w_{ji}} \leftarrow \delta_j z_i$
$r \leftarrow$ Gradient-Index(j, i)
$\mathbf{g}_r^{(n)} \leftarrow \frac{\partial E_n}{\partial w_{ji}}$
end for
return $\mathbf{g}^{(n)}$

2.3. Hesse-Matrix

Die zweiten partiellen Ableitungen werden in der sogenannten Hesse-Matrix angeordnet. Das ist eine symmetrische Matrix $\mathbf{H} = \nabla\nabla E(\mathbf{w}) \in \mathbb{R}^{K \times K}$. Die Hesse-Matrix wird nach Bishop [11, Seite 249, Abschnitt 5.4.] unter anderem für diese Aufgaben genutzt:

- Einige Optimierungsverfahren benötigen die zweite Ableitung, zum Beispiel das in Abschnitt 4 beschriebene Sequential Quadratic Programming.
- Die Hesse-Matrix ist die Grundlage eines schnellen Verfahrens zum Nachjustieren eines MLP [8] bei leichten Änderungen der Trainingsdaten.
- Die inverse Hesse-Matrix kann genutzt werden, um die am wenigsten signifikanten Gewichte zu bestimmen, wodurch das neuronale Netz verkleinert werden kann.

Hier wird kurz das von Bishop [9] entwickelte, exakte Verfahren zur Berechnung der Hesse-Matrix vorgestellt. Die Schritte des Verfahrens sind für jedes Trainingsbeispiel:

1. Vorwärtspropagation (siehe Algorithmus 1).
2. Berechnung der δ_j jedes Neurons j (Backpropagation, siehe Algorithmus 2).
3. Berechnung der $\gamma_{ji} = \frac{\partial a_j}{\partial a_i}$ jedes Neuronen-Paares (j, i).
4. Berechnung der $\beta_{ji} = \frac{\partial \delta_j}{\partial a_i}$ jedes Neuronen-Paares (j, i).
5. Berechnung der Komponenten der Hesse-Matrix.

Algorithmus 3 Hesse-Matrix

Eingabe: $\mathbf{x}^{(n)} = (x_1, \ldots, x_D)^T, \mathbf{t}^{(n)} = (y_1, \ldots, y_F)^T$

Ausgabe: Hesse-Matrix $\mathbf{H}^{(n)} = \begin{pmatrix} \cdots & \vdots & \cdots \\ \cdots & \frac{\partial^2 E_n}{\partial w_{ij} \partial w_{kl}} & \cdots \\ \cdots & \vdots & \cdots \end{pmatrix}$

$\mathbf{g}^{(n)} \leftarrow$ Backpropagation$(\mathbf{x}^{(n)}, \mathbf{t}^{(n)})$

for all Neuronen-Paare (j, i) **do**

$$\gamma_{ji} \leftarrow \begin{cases} 0 & \text{falls } j \text{ in einer Schicht unter } i \text{ ist} \\ 1 & \text{falls } i = j \\ \sum_k g'(a_k) w_{jk} \gamma_{ki} & \text{sonst} \end{cases}$$

end for

for all Neuronen-Paare (j, i) **do**

$$\beta_{ji} \leftarrow \begin{cases} \gamma_{ji} \left(g''(a_j) \frac{\partial E_n}{\partial y_j} + g'(a_j)^2 \frac{\partial^2 E_n}{\partial y_j^2} \right) & \text{falls } j \text{ ein Ausgabeneuron ist} \\ g''(a_j) \gamma_{ji} \sum_k w_{kj} \delta_k + g'(a_j) \sum_k w_{kj} \beta_{ki} & \text{sonst} \end{cases}$$

end for

for all Neuronen-Quadrupel (i, j, k, l), wobei eine Verbindung von i zu j und von k zu l existiert **do**

$\frac{\partial^2 E_n}{\partial w_{ij} \partial w_{kl}} \leftarrow z_j \delta_k g'(a_l) \gamma_{li} + z_j z_l \beta_{ki}$

$r \leftarrow$ Hesse-Matrix-Index(i, j)

$s \leftarrow$ Hesse-Matrix-Index(k, l)

$\mathbf{H}_{rs}^{(n)} \leftarrow \frac{\partial^2 E_n}{\partial w_{ij} \partial w_{kl}}$

$\mathbf{H}_{sr}^{(n)} \leftarrow \frac{\partial^2 E_n}{\partial w_{ij} \partial w_{kl}}$

end for

return $\mathbf{H}^{(n)}$

Die Schritte 1 und 2 sind bereits aus der normalen Backpropagation bekannt. Die γ_{ji} ergeben sich ähnlich der Vorwärtspropagation durch

$$\gamma_{ji} = \begin{cases} 0 & \text{falls } j \text{ in einer Schicht unter } i \text{ ist} \\ 1 & \text{falls } i = j \\ \sum_k g'(a_k) w_{jk} \gamma_{ki} & \text{sonst} \end{cases} \tag{2.3.8}$$

Die β_{ji} hingegen werden in einer Art Backpropagation berechnet. Wenn f ein Neuron der Ausgabeschicht ist, wird der Wert durch

$$\beta_{fi} = \gamma_{fi} \left(g''(a_f) \frac{\partial E_n}{\partial y_f} + g'(a_f)^2 \frac{\partial^2 E_n}{\partial y_f^2} \right) \tag{2.3.9}$$

berechnet.

Für alle weiteren Neuronen-Paare erfolgt die Berechnung rekursiv durch

$$\beta_{ji} = g''(a_j) \gamma_{ji} \sum_k w_{kj} \delta_k + g'(a_j) \sum_k w_{kj} \beta_{ki}. \tag{2.3.10}$$

Die Komponenten von $\mathbf{H}$ können mit Hilfe dieser Zwischenschritte als

$$\frac{\partial^2 E_n}{\partial w_{ij} \partial w_{kl}} = z_j \delta_k g'(a_l) \gamma_{li} + z_j z_l \beta_{ki} \tag{2.3.11}$$

berechnet werden.

Die Komplexität der Berechnung wird hauptsächlich durch die Anzahl der Berechnungen von Gleichung 2.3.11 bestimmt [9]. Da die Matrix die Dimension $K \times K$ hat, liegt der Aufwand in $O(K^2)$. Das Verfahren wird in Algorithmus 3 zusammengefasst.

2.4. Vereinfachung für das MLP

Das in Abschnitt 2.2 vorgestellte Backpropagation-Verfahren funktioniert für allgemeine Feedforward-Netze. Für MLPs lässt sich die Implementierung stark vereinfachen.

Der Eingabevektor $\mathbf{x}$ entspricht dem Aktivierungsvektor der 0. Schicht $a^{(0)}$. Der Ausgabevektor der 0. Schicht $z^{(0)}$ ist $\mathbf{x}$ erweitert mit dem Bias. Die letzte Komponente von $z^{(0)}$ ist also immer 1. In jeder Schicht l kann dann durch

$$\underbrace{a^{(l+1)}}_{J^{(l)} \times 1} = \underbrace{w^{(l)}}_{J^{(l)} \times I^{(l)}} \cdot \underbrace{z^{(l)}}_{I^{(l)} \times 1} \tag{2.4.12}$$

die Aktivierung der nächsten berechnet werden. Diese Matrizenmultiplikation lässt sich unter der Annahme, dass ein Bias verwendet wird, detaillierter darstellen als

$$\begin{pmatrix} a_1^{(l+1)} \\ \vdots \\ a_{J^{(l)}}^{(l+1)} \end{pmatrix} = \begin{pmatrix} w_{11}^{(l)} & \dots & w_{1I^{(l)}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{J^{(l)}1}^{(l)} & \dots & w_{J^{(l)}I^{(l)}}^{(l)} \end{pmatrix} \cdot \begin{pmatrix} z_1^{(l)} \\ \vdots \\ z_{I^{(l)}-1}^{(l)} \\ 1 \end{pmatrix} \tag{2.4.13}$$

Danach muss die Ausgabe um den Bias zu

$$z^{(l+1)} = \begin{pmatrix} g(a_1^{(l+1)}) \\ \vdots \\ g(a_{J^{(l)}}^{(l+1)}) \\ 1 \end{pmatrix}. \tag{2.4.14}$$

ergänzt werden. Ein Sonderfall ist hierbei die Ausgabeschicht, in der kein Bias hinzugefügt werden muss.

Bei der Backpropagation müssen pro Schicht drei weitere Vektoren berechnet werden: ein Fehlervektor $e^{(l)}$ zur Akkumulation der δ der höheren Schicht, ein Ableitungsvektor $g'^{(l)}$, der die Ableitungen der Aktivierungsfunktionen enthält, und ein Vektor $\delta^{(l)}$.

Abhängig davon, ob l die Ausgabeschicht ist, ergibt sich

$$e^{(l)} = \begin{cases} \left(\frac{\partial E_n}{\partial y_0}, \dots, \frac{\partial E_n}{\partial y_F}\right)^T & \text{falls } l \text{ die Ausgabeschicht ist} \\ (w^{(l)})^T \cdot \delta^{(l+1)} & \text{sonst} \end{cases} \tag{2.4.15}$$

Wenn die Summe quadratischer Fehler (SSE) als Fehlerfunktion verwendet wird, ist $e^{(l)}$ in der Ausgabeschicht als $y-t$ zu berechnen. Der Vektor $g'^{(l)}$ ist bei linearer Aktivierungsfunktion in allen Komponenten konstant (abhängig vom Proportionalitätsfaktor der Funktion), bei $tanh(x)$ lässt er sich berechnen durch

$$g'^{(l)} = \left(1 - (z_1^{(l)})^2, 1 - (z_2^{(l)})^2, \ldots\right)^T. \tag{2.4.16}$$

Durch komponentenweise Multiplikation von $g'^{(l)}$ und $e^{(l)}$ wird

$$\delta^{(l)} = \left(g_1'^{(l)} \cdot e_1^{(l)}, g_2'^{(l)} \cdot e_2^{(l)}, \ldots\right)^T \tag{2.4.17}$$

berechnet.

Mit Hilfe der $\delta^{(l)}$ können dann die Ableitungen der Gewichtsmatrizen der einzelnen Schichten als

$$V^{(l)} = \delta^{(l+1)} \cdot (z^{(l)})^T \tag{2.4.18}$$

berechnet werden, wobei $(V^{(l)})_{ji} = \frac{\partial E_n}{\partial (w^{(l)})_{ji}}$.

Die einzelnen Ableitungen können dann in einem Gradienten $\nabla E_n(\mathbf{w})$ angeordnet werden, was bei den meisten Optimierungsverfahren erforderlich ist.

2.5. Eignung des MLP zum Lernen komplexer Aufgaben

Als „komplex“ werden in dieser Arbeit Aufgaben im Bereich des maschinellen Lernens bezeichnet, bei denen die Dimension der Eingabevektoren groß ist. Bei einer hohen Anzahl von Eingabekomponenten ist auch die Anzahl der optimierbaren Parameter groß. Bei einem MLP gilt dies, da jede Eingabe eine Verbindung mit einem Gewicht zu jedem Neuron in der ersten versteckten Schicht hat. Relevante Kriterien zur Auswahl eines Lernverfahrens für komplexe Probleme sind:

- Hypothesenraum: Das Lernverfahren sollte möglichst viele Hypothesen darstellen und erlernen können.
- Generalisierungsfähigkeit: Es sollte nicht zu starkem Overfitting [74, Seite 705] kommen und es sollten auch bisher unbeobachtete Daten vorhersagbar sein.
- Effizienz des Berechnung: Wenn das Training abgeschlossen ist, sollte das generierte Modell schnell die Vorhersage für eine neue Instanz berechnen. Das Negativ-Beispiel ist hier k-Nearest-Neighbor. Dieses Verfahren muss immer alle N Trainingsinstanzen in die Berechnung einbeziehen und die Vorhersage liegt deshalb in der Aufwandsklasse $O(N)$ [74, Seite 738-739].
- Trainingsdauer: Probleme mit großen Dimensionen benötigen auch viele Trainingsdaten, wodurch wiederum die Trainingsdauer steigt. Deshalb sollte das Training möglichst schnell sein.

Es folgt eine Vorstellung relevanter Lernverfahren und die Begründung der Auswahl des MLP für diese Arbeit.

Support Vector Machines (SVM) haben den Vorteil, dass die zu optimierende Kostenfunktion nur ein Optimum hat, sodass die Lösung der Optimierung nicht von initialen Bedingungen abhängt, wie dies bei anderen Lernverfahren der Fall ist [15]. Vapnik [89, Seite 130] schreibt im Gegensatz dazu über die Optimierung von neuronalen Netzen: „Die Qualität der Lösung hängt von vielen Faktoren ab, insbesondere von der Initialisierung der Gewichte." Nach Bengio und LeCun [7] können SVMs als „flache Architekturen" angesehen werden. Es sei zwar bewiesen, dass einige Typen von flachen Architekturen prinzipiell in der Lage sind, jede Funktion mit beliebiger Genauigkeit anzunähern, allerdings seien „tiefe Architekturen" besser geeignet, um einige Klassen von Funktionen effizient darzustellen. Dies seien vor allem die für die Lösung komplexer Fragestellungen der künstlichen Intelligenz interessanten Funktionen.

Künstliche neuronale Netze sind bereits mit einer versteckten Schicht und einer ausreichenden Anzahl von versteckten Neuronen universelle Funktionsapproximatoren [39]. Sie sind also prinzipiell in der Lage alle möglichen Funktionen beliebig genau zu approximieren. In der Praxis ist es allerdings häufig schwierig für ein Problem eine geeignete Topologie zu finden. Eine der einfachsten Formen neuronaler Netze sind MLPs. Durch neuere Hardware-Entwicklungen können wesentlich größere neuronale Netze mit größeren Datensätzen trainiert werden. Zum Beispiel stehen durch den Einsatz von GPU-Programmierung mehrere hundert oder tausend Rechenkerne, die parallel rechnen können, preisgünstig zur Verfügung. Ciresan u. a. [22] haben beispielsweise normale MLPs mit bis zu neun versteckten Schichten und mehreren Millionen Gewichten genutzt, um den MNIST-Datensatz [53] mit 60.000 Trainingsbeispielen zu lernen. Der Datensatz besteht aus Bildern mit 784 Pixeln, in denen Ziffern erkannt werden müssen.

Eine ähnliche Art neuronaler Netze sind die sogenannten Convolutional Neural Networks (CNNs). Sie sind besonders geeignet für zweidimensionale Daten, wie zum Beispiel Bilder. Hier wird gegenüber einem normalen neuronalen Netz die Anzahl der Parameter reduziert, indem sich Neuronen Gewichte teilen. Nach LeCun und Bengio [50] hätten CNNs gegenüber „unstrukturierten Netzen" den Vorteil einer eingebauten Invarianz. Dies hätte zum Beispiel bei Bildern den Vorteil, dass leichte Rotationen oder Translationen keine Schwierigkeit darstellen. CNNs bestehen zum Teil wie MLPs aus aufeinander folgenden, vollständig verbundenen Schichten.

Hinton u. a. [38] haben eine Methode zur Konstruktion von neuronalen Netzen mit mehreren versteckten Schichten entwickelt, sogenannten Deep Belief Networks (DBNs). Die Entwicklung ist motiviert durch die Tatsache, dass neuronale Netze mit vielen versteckten Schichten schwieriger zu optimieren sind. Dies liegt daran, dass die Gewichte in den ersten Schichten eines tiefen neuronalen Netzes normalerweise nicht gut angepasst werden. Bei DBNs werden zunächst Restricted Boltzmann Machines (RBMs) unüberwacht auf einem Datensatz trainiert und dann durch eine Verkettung dieser ein neuronales Netz zusammengebaut. Zuletzt kann das so konstruierte Netz mit Backpropagation überwacht trainiert werden.

MLPs, CNNs oder DBNs sind für große Datensätze mit großen Eingabevektoren und nichtlinearen Zielfunktionen gut geeignet. Ein Grund dafür ist nach Bengio und LeCun [7], dass die erlernten Modelle im Vergleich zu einem durch SVM generier-

ten Modell komplexe nichtlineare Funktionen effizient darstellen können und deshalb beispielsweise schnell Vorhersagen für neue Daten berechnen können. Das MLP stellt ein einfaches neuronales Netz dar, das sehr weit verbreitet ist und teilweise sogar in CNNs und DBNs wiederzufinden ist. Deshalb wird die in dieser Arbeit entwickelte Methode, die für ein MLP ausgelegt ist, nicht darauf beschränkt, sondern kann auch auf andere neuronale Netze angewandt werden. Sie kann als allgemeines Werkzeug zur Vereinfachung des Optimierungsproblems von neuronalen Netzen betrachtet werden.

3. Lernen im komprimierten Raum

Um die Lerngeschwindigkeit von Feedforward-Netzen zu erhöhen, wird in dieser Arbeit eine Methode entwickelt, bei der der Gewichtsvektor $\mathbf{w} \in \mathbb{R}^K$ durch L Parameter approximiert wird, wobei möglichst $L \ll K$ aber mindestens $L < K$, sodass der Suchraum für Optimierungsalgorithmen eine geringere Dimension hat. Optimiert wird dann nicht mehr im Gewichtsraum, sondern in einem komprimierten Parameterraum, wodurch nicht mehr jede mögliche Kombination von Gewichten des neuronalen Netzes erlernt werden kann. Das Ziel ist jedoch ein Verfahren zu entwickeln, das bei $L = K$ den Hypothesenraum möglichst nicht verkleinert.

3.1. Komprimierung des Modells

Der Gewichtsvektor w_j eines Neurons, das durch den Index j bestimmt wird, wird in dem Verfahren, das hier entwickelt wird, durch eine Gewichtsfunktion f_{w_j} approximiert. Diese Gewichtsfunktion ist definiert als

$$f_{w_j} \quad : \quad [0,1] \to \mathbb{R} \tag{3.1.1}$$

$$f_{w_j}(t_i) \quad = \quad \sum_{m=1}^{M_j} \alpha_{jm} \Phi_m(t_i). \tag{3.1.2}$$

Um den Wert der i-ten Komponente von w_j zu berechnen, muss jeder Komponente i ein $t_i \in [0,1]$ zugeordnet werden, mit dem die Gewichtsfunktion parametrisiert wird. Dies geschieht durch eine Nummerierung der Eingänge des Neurons j von 1 bis I und mit Hilfe der Gleichung

$$t_i = \frac{\text{Index}(i) - 1}{I - 1}. \tag{3.1.3}$$

M_j ist die Anzahl der Parameter des Neurons j, die optimiert werden können, das heißt $\sum_j M_j = L$. Prinzipiell kann M_j für alle Neuronen unterschiedlich sein. Ich werde in den Anwendungen allerdings für die Neuronen einer Schicht die gleiche Parameteranzahl verwenden. Um eine Komprimierung zu erreichen muss M_j kleiner sein als die Anzahl der eingehenden Verbindungen des Neurons j.

Hier wird zunächst vorausgesetzt, dass $\Phi_1, \ldots, \Phi_{M_j}$ paarweise orthogonale Funktionen auf dem Intervall $[-1,1]$ sind, das heißt für beliebige $k, l \in \{1, \ldots, M_j\}$ gilt

$$\int_{-1}^{1} \Phi_k(t)\Phi_l(t)\, dt = 0 \qquad \text{[18, Seite 328, Abschnitt 5.3.7.5.]}. \tag{3.1.4}$$

In Abschnitt 3.5 wird allerdings gezeigt, dass die Funktionen nicht unbedingt orthogonal sein müssen. In dieser Arbeit werden die Unterschiede verschiedener Mengen von Basisfunktionen aber nicht ausführlich untersucht.

Algorithmus 4 Gewichtsgenerierung

Eingabe: $\alpha = \left(\ldots, \alpha_{j1}, \ldots, \alpha_{jM_j}, \ldots\right)$, $\Phi = \left(\Phi_1, \ldots, \Phi_{\max_j M_j}\right)$
Ausgabe: Gewichtsvektor $\mathbf{w}$ des neuronalen Netzes
  for all Neuron j **do**
    for all Neuron i, das zu j eine Verbindung hat **do**
      $t_i \leftarrow \frac{\text{Index}(i)-1}{I-1}$
      $w_{ji} \leftarrow \sum_{m=1}^{M_j} \alpha_{jm} \Phi_m(t_i)$
      $r \leftarrow$ Gewichtsvektor-Index(j, i)
      $\mathbf{w}_r \leftarrow w_{ji}$
    end for
  end for
  return $\mathbf{w}$

Eine Menge von orthogonalen Funktionen, die zur Approximation verwendet werden können, ist zum Beispiel

$$\Phi = \{1 = \cos(0\pi t), \cos(1\pi t), \cos(2\pi t), \ldots\}. \tag{3.1.5}$$

Diese wird auch von Koutník u. a. [44] verwendet, allerdings werden dort alle Gewichte des gesamten neuronalen Netzes durch eine einzige Gewichtsfunktion generiert.

In Algorithmus 4 ist das vorgestellte Verfahren zur Generierung der Gewichte aus den Parametern zusammengefasst. Der Algorithmus muss nur ausgeführt werden, wenn sich die Parameter während des Lernvorgangs ändern. Wenn das neuronale Netz trainiert ist, entsteht durch die Komprimierung kein zusätzlicher Aufwand. Die Werte der orthogonalen Funktionen können zudem vorberechnet werden, da sie sich nicht mehr ändern.

Bei einem Multilayer Perceptron mit D Eingaben, einer versteckten Schicht mit E Neuronen, F Ausgabeneuronen und jeweils vollständigen Verbindungen zwischen den Schichten werden insgesamt $D \cdot E + E \cdot F = (D + F) \cdot E = K$ Gewichte benötigt, die bei einer Komprimierung der ersten Schicht durch dieses Verfahren zum Beispiel durch $M \cdot E + E \cdot F = (M + F) \cdot E = L$ Parameter approximiert werden können, wobei die Anzahl der verwendeten Parameter pro Neuron M hier als gleich angenommen wird. Diese Approximation kann vor allem für große Eingabevektoren mit stark korrelierten Komponenten von Vorteil sein, da in diesem Fall eine starke Komprimierung erreicht werden kann ($M \ll D$). In diesem Fall wird die Dimension des zu optimierenden Parametervektors deutlich kleiner.

Betrachtet man den Vektor, dessen Komponenten die Gewichte eines Neurons sind, so ist klar, dass dieser prinzipiell jedes Element aus $\mathbb{R}^I$ sein kann:

$$\begin{pmatrix} w_{j1} \\ w_{j2} \\ \vdots \\ w_{jI} \end{pmatrix} \in \mathbb{R}^I. \tag{3.1.6}$$

Die vorgestellte Gewichtsgenerierung kann auch so formuliert werden, dass dieser Gewichtsvektor durch eine Linearkombination diskretisierter orthogonaler Funktionen berechnet wird:

$$\begin{pmatrix} w_{j1} \\ w_{j2} \\ \vdots \\ w_{jI} \end{pmatrix} = \alpha_{j1} \begin{pmatrix} \Phi_1(t_1) \\ \Phi_1(t_2) \\ \vdots \\ \Phi_1(t_I) \end{pmatrix} + \alpha_{j2} \begin{pmatrix} \Phi_2(t_1) \\ \Phi_2(t_2) \\ \vdots \\ \Phi_2(t_I) \end{pmatrix} + \ldots + \alpha_{jM_j} \begin{pmatrix} \Phi_{M_j}(t_1) \\ \Phi_{M_j}(t_2) \\ \vdots \\ \Phi_{M_j}(t_I) \end{pmatrix}. \quad (3.1.7)$$

Einen Vektor $(\Phi_m(t_1), \Phi_m(t_2), \ldots, \Phi_m(t_I))^T$, der eine Diskretisierung der kontinuierlichen Funktion Φ_m an I Stellen ist, wird hier als $\vec{\Phi}_m$ bezeichnet. In Gleichung 3.1.7 kann man deutlich erkennen, dass die Vektoren $\vec{\Phi}_1, \ldots, \vec{\Phi}_{M_j}$ eine Basis des $\mathbb{R}^I$ darstellen, genau dann wenn sie linear unabhängig sind und $M_j = I$. Dies würde bedeuten, dass mit I Parametern alle möglichen Gewichte darstellbar wären. Durch die Reduktion der Parameter wird allerdings die Dimension des Raums der darstellbaren Gewichte kleiner.

Eine wichtige Eigenschaft für eine sinnvolle Komprimierung ist die lineare Unabhängigkeit der Vektoren $\vec{\Phi}_1, \ldots, \vec{\Phi}_{M_j}$. Ist diese nicht gegeben, so ist mindestens ein Parameter α_{jm} überflüssig.

Die Orthogonalität der Funktionen $\Phi_1, \ldots \Phi_{M_j}$ garantiert, dass keine Funktion Φ_m als Linearkombination der anderen dargestellt werden kann. Ähnliches lässt sich auf die Diskretisierung der Funktionen übertragen: wenn I ausreichend groß ist, sind die Vektoren $\vec{\Phi}_1, \ldots, \vec{\Phi}_{M_j}$ annähernd orthogonal und sind dadurch ebenfalls linear unabhängig. Dies ist allerdings nur eine Approximation, die besser wird, je mehr Eingangsgewichte die Neuronen haben.

Unkomprimierte Netze sind ein Spezialfall dieses Verfahrens, bei dem $I = M_j$ und $\vec{\Phi}_1, \ldots, \vec{\Phi}_{M_j}$ die kanonischen Einheitsvektoren sind.

3.2. Backpropagation

Hier wird das Backpropagation-Verfahren für allgemeine Feedforward-Netze erweitert. Um die Gewichte des Feedforward-Netzes anpassen zu können, müssen die Parameter α_{jm} des Neurons j angepasst werden. Dies geschieht durch Optimierungsalgorithmen, wobei einige dieser Algorithmen dazu die Ableitungen $\frac{\partial E_n}{\partial a_{jm}}$ für alle j und alle $m \in \{1, \ldots, M_j\}$ benötigen.

Durch das normale Backpropagation-Verfahren ist bereits die partielle Ableitung $\frac{\partial E_n}{\partial w_{ji}}$ für jedes Gewicht w_{ji} bekannt. Die einzelnen Gewichte w_{ji} sind hier wiederum Funktionen der Parameter $\alpha_{j1}, \ldots, \alpha_{jM}$. Mit Hilfe der Kettenregel für partielle Ab-

Algorithmus 5 Backpropagation mit komprimierten Gewichten

Eingabe: $\mathbf{x}^{(n)} = (x_1, \ldots, x_D)^T, \mathbf{t}^{(n)} = (y_1, \ldots, y_F)^T$

Ausgabe: Gradient $\overline{\mathbf{g}}^{(n)} = \left(\ldots, \frac{\partial E_n}{\partial \alpha_{jm}}, \ldots\right)^T$

```
g^(n) ← Backpropagation(x^(n), t^(n))
for all Neuron j do
  for m = 1 to M_j do
    ∂E_n/∂α_jm ← Σ_k Φ_m(t_k) ∂E_n/∂w_jk
    r ← Gradient-Index(j, m)
    ḡ_r^(n) ← ∂E_n/∂α_jm
  end for
end for
return ḡ^(n)
```

leitungen[1] können die partiellen Ableitungen nach den Parametern berechnet werden als

$$\frac{\partial E_n}{\partial \alpha_{jm}} = \sum_{k=1}^{K} \frac{\partial E_n}{\partial w_{jk}} \frac{\partial w_{jk}}{\partial \alpha_{jm}}, \tag{3.2.8}$$

$$\text{wobei} \qquad \frac{\partial w_{jk}}{\partial \alpha_{jm}} = \frac{\partial}{\partial \alpha_{jm}} \left(\sum_{m'=1}^{M_j} \alpha_{jm'} \Phi_{m'}(t_k) \right) = \Phi_m(t_k). \tag{3.2.9}$$

In Algorithmus 5 wird das Verfahren für allgemeine Feedforward-Netze zusammengefasst. In Abschnitt 3.4 wird ein vereinfachtes Verfahren für MLPs dargestellt.

3.3. Hesse-Matrix

Im Folgenden wird die Hesse-Matrix eines allgemeinen Feedforward-Netzes mit komprimierten Gewichten hergeleitet. Das resultierende Verfahren ähnelt dem durch Bishop [9] entwickelten Verfahren.

In der Herleitung wird der folgende Operator verwendet, der angibt in welchen Schichten sich zwei Neuronen relativ zueinander befinden. Eine höhere Schicht ist dabei näher an der Ausgabeschicht.

$$\geq_l \quad : \quad \text{Neuron} \times \text{Neuron} \to \{\text{wahr}, \text{falsch}\} \tag{3.3.10}$$

$$i \geq_l j \quad = \quad \begin{cases} \text{wahr} & \text{falls } i \text{ in der selben oder einer Schicht über } j \text{ ist} \\ \text{falsch} & \text{sonst} \end{cases} \tag{3.3.11}$$

Nach dem Schwarzschen Vertauschungssatz[1] gilt in diesem Fall

$$\frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}} = \frac{\partial^2 E_n}{\partial \alpha_{im} \partial \alpha_{jm'}}, \tag{3.3.12}$$

[1] Siehe Anhang C.

sodass es genügt, alle Ableitungen, bei denen $i \geq_l j$ gilt, explizit zu berechnen. Von dieser Annahme gehe ich im Folgenden aus.

Die zweite partielle Ableitung lässt sich zunächst umschreiben zu

$$\frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}} = \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial \alpha_{im}} \right) \tag{3.3.13}$$

$$= \frac{\partial}{\partial \alpha_{jm'}} \left(\sum_k \frac{\partial E_n}{\partial w_{ik}} \frac{\partial w_{ik}}{\partial \alpha_{im}} \right) \tag{3.3.14}$$

$$= \sum_k \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial w_{ik}} \frac{\partial w_{ik}}{\partial \alpha_{im}} \right) \tag{3.3.15}$$

$$= \sum_k \left[\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial w_{ik}} \right) \underbrace{\frac{\partial w_{ik}}{\partial \alpha_{im}}}_{\Phi_m(t_k)} + \frac{\partial E_n}{\partial w_{ik}} \underbrace{\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial w_{ik}}{\partial \alpha_{im}} \right)}_{0} \right] \tag{3.3.16}$$

$$= \sum_k \left[\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial w_{ik}} \right) \Phi_m(t_k) \right]. \tag{3.3.17}$$

Dabei geht k über alle Neuronen, die zu i eine Verbindung haben.

Der unbekannte Term aus Gleichung 3.3.17 kann weiter aufgelöst werden zu

$$\frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial w_{ik}} = \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial a_i} \frac{\partial a_i}{\partial w_{ik}} \right) \tag{3.3.18}$$

$$= \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial a_i} \right) \frac{\partial a_i}{\partial w_{ik}} + \frac{\partial E_n}{\partial a_i} \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_i}{\partial w_{ik}} \right) \tag{3.3.19}$$

$$= \frac{\partial}{\partial \alpha_{jm'}} \left(\sum_l \frac{\partial E_n}{\partial a_l} \frac{\partial a_l}{\partial a_i} \right) \frac{\partial a_i}{\partial w_{ik}} + \frac{\partial E_n}{\partial a_i} \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_i}{\partial w_{ik}} \right) \tag{3.3.20}$$

$$= \underbrace{\sum_l \left[\underbrace{\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial E_n}{\partial a_l} \right)}_{\beta_{ljm'}} \underbrace{\frac{\partial a_l}{\partial a_i}}_{w_{li} g'(a_i)} + \underbrace{\frac{\partial E_n}{\partial a_l}}_{\delta_l} \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_l}{\partial a_i} \right) \right]}_{\beta_{ijm'}} \underbrace{\frac{\partial a_i}{\partial w_{ik}}}_{z_k} \tag{3.3.21}$$

$$+ \underbrace{\frac{\partial E_n}{\partial a_i}}_{\delta_i} \underbrace{\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_i}{\partial w_{ik}} \right)}_{\text{Gleichung 3.3.45}}. \tag{3.3.22}$$

Dabei geht l über alle Neuronen, zu denen i die Ausgabe sendet.

Zur Berechnung der $\beta_{ijm'}$ muss $\frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial a_l}{\partial a_i}\right)$ bestimmt werden. Dies geschieht durch

$$\begin{aligned}
\frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial a_l}{\partial a_i}\right) &= \frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial a_l}{\partial z_i}\frac{\partial z_i}{\partial a_i}\right) && (3.3.23)\\
&= \frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial a_l}{\partial z_i}\right)\frac{\partial z_i}{\partial a_i} + \frac{\partial a_l}{\partial z_i}\frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial z_i}{\partial a_i}\right) && (3.3.24)\\
&= \frac{\partial w_{li}}{\partial \alpha_{jm'}}\frac{\partial z_i}{\partial a_i} + w_{li}\frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial z_i}{\partial a_i}\right) && (3.3.25)\\
&= \frac{\partial w_{li}}{\partial \alpha_{jm'}}g'(a_i) + w_{li}\frac{\partial g'(a_i)}{\partial \alpha_{jm'}} && (3.3.26)\\
&= \begin{cases} 0 & \text{falls } j \neq l \wedge j \geq_l l \\ \Phi_{m'}(t_i)g'(a_i) & \text{falls } j = l \text{ (linker Summand)} \\ w_{li}\frac{\partial g'(a_i)}{\partial \alpha_{jm'}} & \text{sonst (rechter Summand)} \end{cases} && (3.3.27)\\
&= w_{li}\frac{\partial g'(a_i)}{\partial \alpha_{jm'}}. && (3.3.28)
\end{aligned}$$

Der letzte Schritt ist möglich aufgrund der Annahme $i \geq_l j$ und weil l sich in der Schicht über i befindet. Demzufolge befindet sich l auch in einer Schicht über j.

Weiterhin gilt

$$\begin{aligned}
\frac{\partial}{\partial \alpha_{jm'}}\left(\frac{\partial a_l}{\partial a_i}\right) &= w_{li}\frac{\partial g'(a_i)}{\partial \alpha_{jm'}} = w_{li}\frac{\partial g'(a_i)}{\partial a_i}\frac{\partial a_i}{\partial \alpha_{jm'}} && (3.3.29)\\
&= w_{li}g''(a_i)\frac{\partial a_i}{\partial \alpha_{jm'}} = w_{li}g''(a_i)\frac{\partial a_i}{\partial \alpha_{jm'}} && (3.3.30)\\
&= w_{li}g''(a_i)\,\gamma_{ijm'}. && (3.3.31)
\end{aligned}$$

Falls j sich in einer Schicht unter i befinden sollte, lässt sich $\gamma_{ijm'}$ rekursiv berechnen durch

$$\begin{aligned}
\frac{\partial a_i}{\partial \alpha_{jm'}} &= \sum_r \frac{\partial a_i}{\partial z_r}\frac{\partial z_r}{\partial \alpha_{jm'}} && (3.3.32)\\
&= \sum_r \frac{\partial a_i}{\partial z_r}\frac{\partial z_r}{\partial a_r}\frac{\partial a_r}{\partial \alpha_{jm'}} && (3.3.33)\\
&= \underbrace{\sum_r w_{ir}g'(a_r)\underbrace{\frac{\partial a_r}{\partial \alpha_{jm'}}}_{\gamma_{rjm'}}}_{\gamma_{ijm'}}, && (3.3.34)
\end{aligned}$$

wobei alle Neuronen r direkt nach i verbunden sind.

Damit lassen sich alle γ_{ijm} rekursiv berechnen mit

$$\gamma_{ijm} = \frac{\partial a_i}{\partial \alpha_{jm}} = \begin{cases} 0 & (1) \text{ falls } j \neq i \wedge j \geq_l i \\ \sum_r \Phi_m(t_r)z_r & (2) \text{ falls } j = i \\ \sum_r w_{ir}g'(a_r)\,\gamma_{rjm} & (3) \text{ sonst} \end{cases} \quad (3.3.35)$$

Demnach lassen sich die γ_{ijm} durch eine Art Vorwärtspropagation berechnen. Dabei wird bei der ersten Gewichtsschicht gestartet mit dem Fall (2) und der Wert wird jeweils an die nächste Schicht weitergeleitet, wo die Werte aller Neuronen mit Hilfe von (3) aufsummiert werden und weitere Werte nach (2) hinzugefügt werden. Dies wiederholt sich bis in die Ausgabeschicht.

Der Rekursionsanfang für β_{fjm} bei einem Ausgabeneuron f lässt sich berechnen durch

$$\begin{aligned}
\beta_{fjm} &= \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial E_n}{\partial a_f} \right) = \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial E_n}{\partial y_f} \frac{\partial y_f}{\partial a_f} \right) && (3.3.36)\\
&= \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial E_n}{\partial y_f} \right) \frac{\partial y_f}{\partial a_f} + \frac{\partial E_n}{\partial y_f} \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial y_f}{\partial a_f} \right) && (3.3.37)\\
&= \frac{\partial}{\partial y_f} \left(\frac{\partial E_n}{\partial y_f} \right) \frac{\partial y_f}{\partial a_f} \frac{\partial a_f}{\partial \alpha_{jm}} \frac{\partial y_f}{\partial a_f} + \frac{\partial E_n}{\partial y_f} \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial y_f}{\partial a_f} \right) && (3.3.38)\\
&= \frac{\partial^2 E_n}{\partial y_f^2} \frac{\partial a_f}{\partial \alpha_{jm}} g'(a_f)^2 + \frac{\partial E_n}{\partial y_f} \frac{\partial g'(a_f)}{\partial \alpha_{jm}} && (3.3.39)\\
&= \frac{\partial^2 E_n}{\partial y_f^2} \frac{\partial a_f}{\partial \alpha_{jm}} g'(a_f)^2 + \frac{\partial E_n}{\partial y_f} \frac{\partial g'(a_f)}{\partial a_f} \frac{\partial a_f}{\partial \alpha_{jm}} && (3.3.40)\\
&= \frac{\partial a_f}{\partial \alpha_{jm}} \left(\frac{\partial^2 E_n}{\partial y_f^2} g'(a_f)^2 + \frac{\partial E_n}{\partial y_f} g''(a_f) \right) && (3.3.41)\\
&= \gamma_{fjm} \left(\frac{\partial^2 E_n}{\partial y_f^2} g'(a_f)^2 + \frac{\partial E_n}{\partial y_f} g''(a_f) \right). && (3.3.42)
\end{aligned}$$

Somit können alle weiteren β_{ijm} durch

$$\begin{aligned}
\beta_{ijm} &= \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial E_n}{\partial a_i} \right) = \sum_k \beta_{kjm} w_{ki} g'(a_i) + \delta_k \frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_k}{\partial a_i} \right) && (3.3.43)\\
&= g''(a_i) \gamma_{ijm} \sum_k \delta_k w_{ki} + g'(a_i) \sum_k \beta_{kjm} w_{ki} && (3.3.44)
\end{aligned}$$

in einer Art Backpropagation berechnet werden, wie man anhand von Gleichung 3.3.21 sieht. k geht hier über alle Neuronen, zu denen i die Ausgabe sendet. Die erste Summe entspricht der aus der normalen Backpropagation (siehe Abschnitt 2.2) und muss nicht erneut berechnet werden.

Der letzte unbekannte Term lässt sich leicht umschreiben zu

$$\frac{\partial}{\partial \alpha_{jm'}} \left(\frac{\partial a_i}{\partial w_{ik}} \right) = \frac{\partial z_k}{\partial \alpha_{jm'}} = \frac{\partial z_k}{a_k} \frac{a_k}{\partial \alpha_{jm'}} = g'(a_k) \gamma_{kjm'}. \tag{3.3.45}$$

Danach ist die zweite Ableitung durch Einsetzen der Gleichungen 3.3.21 in 3.3.17 komplett berechenbar als

$$\frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}} = \sum_k \Phi_m(t_k) \left(\beta_{ijm'} z_k + \delta_i g'(a_k) \gamma_{kjm'} \right). \tag{3.3.46}$$

Erneut geht k über alle Neuronen, die zu i eine Verbindung haben.

In Algorithmus 6 ist das Verfahren zusammengefasst.

Algorithmus 6 Hesse-Matrix mit komprimierten Gewichten

Eingabe: $\mathbf{x}^{(n)} = (x_1, \ldots, x_D)^T, \mathbf{t}^{(n)} = (y_1, \ldots, y_F)^T$

Ausgabe: $\overline{\overline{\mathbf{H}}}^{(n)} = \begin{pmatrix} \cdots & \vdots & \cdots \\ \cdots & \frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}} & \cdots \\ \cdots & \vdots & \cdots \end{pmatrix}$

$\mathbf{g}^{(n)} \leftarrow \text{Backpropagation}(\mathbf{x}^{(n)}, \mathbf{t}^{(n)})$
for all Neuronen-Paare (i, j) **do**
  for all $m = 1$ **to** M_j **do**

$$\gamma_{ijm} = \begin{cases} 0 & (1) \text{ falls } j \neq i \wedge j \geq_l i \\ \sum_r \Phi_m(t_r) z_r & (2) \text{ falls } j = i \\ \sum_r w_{ir} g'(a_r) \gamma_{rjm} & (3) \text{ sonst} \end{cases}$$

  end for
end for
for all Neuronen-Paare (i, j) **do**
  for all $m = 1$ **to** M_j **do**

$$\beta_{ijm} \leftarrow \begin{cases} \gamma_{ijm} \left(\frac{\partial^2 E_n}{\partial y_i^2} g'(a_i)^2 + \frac{\partial E_n}{\partial y_i} g''(a_i) \right) & (1) \text{ falls } i \text{ in der Ausgabeschicht ist} \\ g''(a_i) \gamma_{ijm} \sum_k \delta_k w_{ki} + g'(a_i) \sum_k \beta_{kjm} w_{ki} & (2) \text{ sonst} \end{cases}$$

  end for
end for
for all Neuronen-Paare (i, j) mit $i \geq_l j$ **do**
  for all $m = 1$ **to** M_j **do**
    for all $m' = 1$ **to** M_i **do**
      $\frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}} \leftarrow \sum_k \Phi_m(t_k) (\beta_{ijm'} z_k + \delta_i g'(a_k) \gamma_{kjm'})$
      $r \leftarrow \text{Hesse-Matrix-Index}(i, m)$
      $s \leftarrow \text{Hesse-Matrix-Index}(j, m')$
      $\overline{\overline{\mathbf{H}}}^{(n)}_{rs} \leftarrow \frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}}$
      $\overline{\overline{\mathbf{H}}}^{(n)}_{sr} \leftarrow \frac{\partial^2 E_n}{\partial \alpha_{jm'} \partial \alpha_{im}}$
    end for
  end for
end for
return $\overline{\overline{\mathbf{H}}}^{(n)}$

3.4. Vereinfachung für das MLP

Die in den Abschnitten 2.2 und 3.2 vorgestellten Backpropagation-Verfahren funktionieren für allgemeine Feedforward-Netze. Für MLPs lässt sich das in Abschnitt 2.4 entwickelte vereinfachte Verfahren für eine komprimierte Gewichtsrepräsentation erweitern, sodass hier ebenfalls ein vereinfachtes Verfahren entsteht.

Um die Generierung der Gewichte effizient zu implementieren, sollte die Anzahl der Parameter pro Neuron M_j in jeder Schicht l einheitlich sein ($M^{(l)}$). Dann können die Parameter ebenso wie die Gewichte pro Schicht in einer Matrix

$$\alpha^{(l)} = \begin{pmatrix} \alpha_{1,1} & \dots & \alpha_{1,M^{(l)}} \\ \vdots & \ddots & \vdots \\ \alpha_{J^{(l)},1} & \dots & \alpha_{J^{(l)},M^{(l)}} \end{pmatrix} \tag{3.4.47}$$

angeordnet werden. $I^{(l)}$ ist die Anzahl der Neuronen in Schicht l inklusive Bias und $J^{(l)}$ ist die Anzahl der Neuronen in Schicht $l+1$ ohne Bias. Die Werte der orthogonalen Funktionen lassen sich vorher berechnen und können ebenfalls pro Schicht in einer Matrix

$$\Phi^{(l)} = \begin{pmatrix} \Phi_1(t_1) & \dots & \Phi_1(t_{I^{(l)}}) \\ \vdots & \ddots & \vdots \\ \Phi_{M^{(l)}}(t_1) & \dots & \Phi_{M^{(l)}}(t_{I^{(l)}}) \end{pmatrix} \tag{3.4.48}$$

angeordnet werden.

Die Gewichtsmatrix zwischen Schicht l und $l+1$ bezeichne ich als $w^{(l)}$. Die Gewichte zwischen Schicht l und $l+1$ lassen sich leicht durch die Matrizenmultiplikation

$$\underbrace{w^{(l)}}_{J^{(l)} \times I^{(l)}} = \underbrace{\alpha^{(l)}}_{J^{(l)} \times M^{(l)}} \cdot \underbrace{\Phi^{(l)}}_{M^{(l)} \times I^{(l)}} \tag{3.4.49}$$

berechnen.

Die Ableitungen nach den Parametern lassen sich dann mit

$$\nu^{(l)} = V^{(l)} \cdot (\Phi^{(l)})^T, \tag{3.4.50}$$

basierend auf den in Abschnitt 2.4 hergeleiteten Matrizen $V^{(l)}$, berechnen, sodass

$$(\nu^{(l)})_{jm} = \frac{\partial E_n}{\partial (\alpha^{(l)})_{jm}}. \tag{3.4.51}$$

Die einzelnen Komponenten können dann in einem Gradienten $\nabla E_n(\alpha)$ angeordnet werden.

3.5. Komprimierung der Daten

Ich werde in diesem Abschnitt zeigen, dass eine Komprimierung der Eingabe einer Schicht eines MLPs das Gleiche ist wie die Komprimierung der Gewichte einer Schicht und darüber den Zusammenhang zum Compressed Sensing [25, 20] herstellen.

Ausgangspunkt des Compressed Sensing ist die Gleichung

$$y = \Phi x, \tag{3.5.52}$$

wobei $x \in \mathbb{R}^N$, $\Phi \in \mathbb{R}^{M \times N}$, $y \in \mathbb{R}^M$ und $M < N$. Durch die Multiplikation mit der Matrix Φ werden die Daten auf den Unterraum $\mathbb{R}^M$ von $\mathbb{R}^N$ abgebildet.

Dabei werden in der Regel Daten betrachtet, die dünn besetzt sind (englisch: *sparse*). Das heißt die meisten Einträge von x sollten 0 sein. Die Mindestvoraussetzung ist allerdings, dass die Daten komprimierbar sind. Wären alle Daten von x wichtig, würden dabei Informationen verloren gehen und eine Rekonstruktion von x aus y wäre sehr ungenau.

Die wichtigste Erkenntnis des Compressed Sensing wird zum Beispiel durch Candès und Romberg [20] dargestellt: wenn x dünn besetzt ist, könne sogar eine zufällig generierte Matrix Φ verwendet werden und durch die Lösung des Optimierungsproblems $\min ||g||_{l_1}$ mit der Bedingung $\Phi g = y$ könne x mit großer Wahrscheinlichkeit eindeutig rekonstruiert werden, wenn M eine bestimmte Mindestgröße habe. Dabei sei es egal in welcher Basis x dünn besetzt ist. Die Norm $||g||_{l_1}$ sei definiert als $\sum_{m=1}^{M} |g_m|$. Eine ähnliche Aussage gelte für komprimierbare x. Da die Rekonstruktion des Vektors x aus den komprimierten Daten für das Lernen nicht benötigt wird, entfällt bei dieser Anwendung die Notwendigkeit einer Lösung des Optimierungsproblems.

Für komprimierbare Daten wie Bilder oder EEG-Daten reicht es demnach aus, eine zufällige Matrix zur Komprimierung der Daten zu verwenden. Allerdings müssen diese Matrizen die sogenannte *Restricted Isometry Property* (RIP) erfüllen [4]. Beispiele für solche Matrizen sind die, deren Komponenten aus der Standardnormalverteilung [20]

$$\Phi_{mi} \sim \mathcal{N}(0, 1), \tag{3.5.53}$$

einer skalierten Normalverteilung [4]

$$\Phi_{mi} \sim \mathcal{N}(0, \frac{1}{M}), \tag{3.5.54}$$

einer Bernoulli-Verteilung [4]

$$\Phi_{mi} = \begin{cases} +\frac{1}{\sqrt{M}} & \text{mit Wahrscheinlichkeit } \frac{1}{2} \\ -\frac{1}{\sqrt{M}} & \text{mit Wahrscheinlichkeit } \frac{1}{2} \end{cases}, \tag{3.5.55}$$

oder einer ähnlichen Verteilung [4]

$$\Phi_{mi} = \begin{cases} +\sqrt{\frac{3}{M}} & \text{mit Wahrscheinlichkeit } \frac{1}{6} \\ 0 & \text{mit Wahrscheinlichkeit } \frac{2}{3} \\ -\sqrt{\frac{3}{M}} & \text{mit Wahrscheinlichkeit } \frac{1}{6} \end{cases} \tag{3.5.56}$$

gezogen werden. Ich verwende in dieser Arbeit die Verteilung $\mathcal{N}(0, \frac{1}{M})$, wenn keine andere angegeben wird.

Compressed Sensing wurde bereits früher mit Lernverfahren kombiniert. Zum Beispiel haben Calderbank u. a. [19] Compressed Sensing in Kombination mit SVMs zur

Klassifikation dünn besetzter Daten eingesetzt und haben eine asymptotische obere Schranke für die Verringerung der Korrektklassifikationsrate angegeben. Maillard und Munos [56] verknüpfen Compressed Sensing mit Regression. Die Komprimierung wird dabei sogar zur Verbesserung der Generalisierungsfähigkeit des Lernverfahrens eingesetzt.

Normalerweise sieht das hier entwickelte Verfahren vor, dass in jeder Schicht l aus der Parametermatrix $\alpha^{(l)}$ und der Matrix $\Phi^{(l)}$, in der die Werte der orthogonalen Funktionen gespeichert werden, die Gewichtsmatrix $w^{(l)}$ generiert wird. Allerdings ist es auch möglich den Ausgabevektor $z^{(l)}$ zu komprimieren und $\alpha^{(l)}$ als Gewichtsmatrix zu betrachten, wie eine Umstellung zu

$$\underbrace{w^{(l)}}_{J^{(l)}\times I^{(l)}} z^{(l)} = (\underbrace{\alpha^{(l)}}_{J^{(l)}\times M^{(l)}} \underbrace{\Phi^{(l)}}_{M^{(l)}\times I^{(l)}}) \underbrace{z^{(l)}}_{I^{(l)}\times 1} = \alpha^{(l)}(\Phi^{(l)} z^{(l)}) = \alpha^{(l)} \underbrace{z'^{(l)}}_{M^{(l)}\times 1} \tag{3.5.57}$$

zeigt. Die Komprimierung $\Phi^{(l)} z^{(l)} = z'^{(l)}$ entspricht der Gleichung 3.5.52. Bisher wurden für $\Phi^{(l)}$ nur orthogonale Basisfunktionen zugelassen. In diese Kategorie fallen zum Beispiel die in Abschnitt 3.1 genannten Kosinus-Funktionen und Wavelets. Durch das Compressed Sensing und diese Gleichheit kann begründet werden, dass auch die genannten zufällig generierten Matrizen verwendet werden können.

Ein weiterer Vorteil dieser Betrachtungsweise ist eine geringere Komplexität der Gewichtserzeugung und der Backpropagation, wenn nur die erste Schicht komprimiert werden soll. Da $z^{(0)}$ der Eingabevektor x - gegebenenfalls mit einem Bias - ist, reicht es hier aus, für das Training die Vektoren $z'^{(0)}$ einmal zu generieren. Danach kann die Parametermatrix $\alpha^{(l)}$ als Gewichtsmatrix der ersten Schicht des neuronalen Netzes verwendet werden. Dadurch fallen der Aufwand zur Generierung der Gewichte nach Anpassung der Parameter (Gleichung 3.4.49) und die Berechnung der Ableitungen nach den Parametern aus den Ableitungen nach den Gewichten in der Backpropagation (Gleichung 3.4.50) weg, sodass die Ableitungen in der Schicht l berechnet werden durch

$$\begin{aligned}
\nu^{(l)} &= V^{(l)} \cdot (\Phi^{(l)})^T && (3.5.58)\\
&= \delta^{(l+1)} \cdot (z^{(l)})^T \cdot (\Phi^{(l)})^T && (3.5.59)\\
&= \delta^{(l+1)} \cdot (\Phi^{(l)} \cdot z^{(l)})^T && (3.5.60)\\
&= \delta^{(l+1)} \cdot (z'^{(l)})^T. && (3.5.61)
\end{aligned}$$

Hier wird davon ausgegangen, dass der Rechenaufwand einer Matrizenmultiplikation zweier Matrizen $A \in \mathbb{R}^{P\times Q}$ und $B \in \mathbb{R}^{Q\times R}$ in $O(PQR)$ liegt. Der Rechenaufwand einer Gewichtsgenerierung würde demnach normalerweise in $O(J^{(0)}M^{(0)}I^{(0)})$ liegen. Der Rechenaufwand der Ableitungsberechnung in der ersten Schicht nach den Komponenten von $\alpha^{(0)}$ würde ebenfalls in $O(J^{(0)}M^{(0)}I^{(0)})$ liegen. Durch die Komprimierung der Trainingsdaten entfällt die Gewichtsgenerierung und der Aufwand zur Ableitung der ersten Schicht liegt nur noch in $O(J^{(0)}I^{(0)})$.

Eine nützliche Eigenschaft bei der Komprimierung der Daten ist, dass nach abgeschlossenem Training aus der Matrix $\Phi^{(0)}$ und den Parametern $\alpha^{(0)}$ wieder die Gewichtsmatrix $w^{(0)}$ generiert werden kann, sodass die Komprimierung der Daten und die Berechnung der Aktivierung der ersten Schicht in einer Matrizenmultiplikation stattfinden, wodurch kein zusätzlicher Aufwand durch die Datenkomprimierung während der Auswertung des Modells entsteht.

Das hier entwickelte Verfahren unterscheidet sich vom Compressed Sensing und anderen Datenkomprimierungsverfahren, da die Möglichkeit besteht die Gewichte aller Schichten zu Komprimieren. Wenn allerdings nur die erste Gewichtsschicht komprimiert wird, sind die Komprimierung der Daten und die Komprimierung der Gewichte durch Gleichung 3.5.52 gleich.

4. Bewertung von Optimierungsalgorithmen

Die Optimierungsverfahren für neuronale Netze sind kein Schwerpunkt dieser Arbeit. Dennoch ist es zum Erreichen einer zu aktuellen Lernverfahren vergleichbaren Leistungsfähigkeit unerlässlich, einen guten Optimierungsalgorithmus zu wählen. Ein normaler Gradientenabstieg ist meistens viel zu langsam und das Ergebnis hängt stark von der Wahl der Lernrate ab. Deshalb werden in diesem Abschnitt einige Optimierungsalgorithmen untersucht. Detailliertere Übersichten über für normale neuronale Netze geeignete Optimierungsverfahren haben LeCun u. a. [51] und Bishop [10] zusammengestellt. An diesen werde ich mich im Folgenden orientieren. Es existieren auch einige spezialisierte Lernverfahren für neuronale Netze, wie zum Beispiel Resilient Backpropagation (Rprop) [69] oder Quickprop [29], die nach dem Neural Network FAQ von Sarle [63] allerdings nicht an die Leistungen von Optimierungsalgorithmen wie Levenberg-Marquardt oder Conjugate Gradient herankommen. Aus diesem Grund wird hier nicht in Erwägung gezogen, diese für eine komprimierte Gewichtsrepräsentation einzusetzen.

Die meisten Verfahren, die hier erwähnt werden, setzen voraus, dass der gesamte Trainingsdatensatz zur Verfügung steht, sodass ein *Batch Learning* durchgeführt werden kann. Das heißt, es wird mit der Fehlerfunktion des gesamten Datensatzes gearbeitet. Der Levenberg-Marquardt-Algorithmus ist hier eine Sonderform. Er benötigt zwar die Fehlerwerte einzelner Trainingsdaten, ist allerdings auch ein Batch-Verfahren. Im Gegensatz dazu existieren auch Verfahren, die als *Stochastic Learning* oder *Online Learning* [51, Abschnitt 1.4.1] bezeichnet werden. Diese benötigen für die Gewichtsanpassung jeweils nur ein Trainingsdatum. Hier tritt ein praktisches Problem auf: da wesentlich häufiger Anpassungen der Parameter vorgenommen werden, müssen die Gewichte auch häufiger berechnet werden, wodurch zusätzlicher Rechenaufwand entsteht. Dieser zusätzliche Aufwand würde von der Größe des Trainingsdatensatzes N abhängen. Angenommen, dass für das Batch Learning und das Online Learning gleich oft Gradienten oder Fehlerwerte jedes einzelnen Trainingsdatums berechnet werden, dann werden die Gewichte beim Online Learning normalerweise nach jeder einzelnen Gradientenberechnung angepasst, beim Batch Learning allerdings nur nach der Berechnung aller N Gradienten. Beim Online Learning steigt die Anzahl der Gewichtsberechnungen also um den Faktor N. Deshalb ist das Online Learning eigentlich für eine indirekte Gewichtsrepräsentation ungeeignet. Wenn nur die erste Gewichtsschicht komprimiert werden muss, können, wie in Abschnitt 3.5 beschrieben, stattdessen die Daten selbst während des Trainings komprimiert werden, sodass der zusätzliche Aufwand der Komprimierung entfällt.

4.1. Ziel der Optimierung

Das Ziel eines Optimierungsalgorithmus ist die Optimierung, also entweder die Minimierung oder Maximierung, einer Zielfunktion. Bei den hier betrachteten künstlichen neuronalen Netzen soll in den meisten Fällen eine Fehlerfunktion E minimiert werden. Es wird also ein Gewichtsvektor $\mathbf{w}^* \in \mathbb{R}^K$ gesucht, sodass

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}}\ E(\mathbf{w}). \tag{4.1.1}$$

K ist die Dimension des Suchraums und bestimmt maßgeblich den Aufwand der Optimierung. Wenn die Gewichte des neuronalen Netzes komprimiert werden, wird stattdessen der Fehler abhängig von dem Parametervektor α minimiert. Bei dieser Art von Optimierungsproblem müssen keine Randbedingungen erfüllt sein. Ausnahmen bilden dabei die in Abschnitt 6.4 betrachteten Probleme aus dem Bereich Reinforcement Learning, bei denen eine andere Zielfunktion optimiert werden muss.

Die Fehlerfunktionen von mehrschichtigen neuronalen Netzen weisen einige Eigenschaften auf, die die Optimierung erschweren:

- **Nicht separierbar:** „Wir bezeichnen eine Zielfunktion $f : x \to f(x)$ als separierbar nach Koordinate i, wenn der optimale Wert der i. Koordinate nicht von der Wahl der anderen Koordinaten abhängt. Wir nennen eine Zielfunktion separierbar, wenn sie nach jeder Koordinate separierbar ist. [...] Separierbare Funktionen sind nicht anfällig für den Curse of Dimensionality.“ [35]

  In Abbildung 4.1 ist die Fehlerfunktion eines sehr einfachen neuronalen Netzes zu sehen und bereits hier wird deutlich, dass die beiden Koordinaten des Parametervektors w nicht separierbar sind, da sich der optimale Wert von w_2 mit w_1 verändert. Besonders deutlich ist der Unterschied bei wechselndem Vorzeichen von w_1. Diese Eigenschaft hängt von der Topologie des neuronalen Netzes, den Aktivierungsfunktionen, der Fehlerfunktion und von dem Trainingsdatensatz ab, allerdings gehe ich hier davon aus, dass der Effekt praktisch immer auftritt.

- **Schlecht konditioniert:** „Der Begriff schlecht konditioniert bezieht sich auf Situationen, in denen verschiedene Variablen oder verschiedene Richtungen im Suchraum stark unterschiedliche Auswirkungen auf den Wert der Zielfunktion haben. [..] Wir können eine Funktion schlecht konditioniert nennen, wenn sich für Punkte mit ähnlichen Funktionswerten die minimale Verschiebung dieser Punkte zur Verbesserung des Funktionswertes um einen bestimmten Betrag um Größenordnungen unterscheidet.“ [35]

  Dies ist eine problematische Eigenschaft neuronaler Netze, bei denen Verfahren zur Optimierung eingesetzt werden, die auf Backpropagation basieren [63, Part 2: How does ill-conditioning affect NN training?].

- **Nicht konvex:** Die Fehlerfunktion von neuronalen Netzen mit mehreren Schichten ist meistens nicht-konvex mit vielen lokalen Minima und flachen Regionen [51].

Diese Eigenschaften gelten nicht nur für normale Feedforward-Netze, sondern sind auch auf Feedforward-Netze mit komprimierten Gewichten übertragbar, da die Gewichte nur eine gewichtete Summe von Komponenten des Parametervektors α sind.

4.2. Initialisierung

Damit die Optimierungsalgorithmen, insbesondere die auf Ableitungen basierenden, funktionieren, ist es wichtig, dass ein guter Startpunkt verwendet wird, sodass zum Beispiel keine großen numerischen Fehler in den Ableitungen auftreten. LeCun u. a. [51] erläutern diese Problematik genauer. Wenn alle Gewichte 0 sind, ist an Gleichung 2.2.6 abzulesen, dass die partielle Ableitung an den meisten Stellen 0 ist und somit auf Ableitung basierende Verfahren fehlschlagen. LeCun u. a. [51] empfehlen deshalb, die Gewichte zufällig zu initialisieren.

Gleiches gilt für komprimierte Gewichte. In diesem Fall muss die generierende Funktion unterschiedliche Gewichte liefern.Wie genau die Gewichte initialisiert werden, scheint egal zu sein. Um möglichst unterschiedliche Gewichte zu erhalten, sollten zum Beispiel bei orthogonalen Kosinusfuntkionen die hochfrequenten Anteile einen Vorfaktor mit hohem Betrag haben. Deshalb werden in dieser Arbeit die initialen Parameter eines Neurons j aus einer Normalverteilung mit dem Erwartungswert 0 und der Standardabweichung $\sigma = \frac{1}{3}(\frac{m}{M_j} + \frac{1}{3})^{1,05}$ gezogen.

4.3. Übersicht über die Optimierungsverfahren

Optimierungsalgorithmen werden hier grob in drei Kategorien unterteilt:

(1) **Ableitungsfreie Optimierungsverfahren (Blackbox-Optimierung):** Bei einigen Problemen kann es sein, dass gar keine Ableitung der Zielfunktion berechnet werden kann oder diese nicht sinnvoll ist. Dies ist zum Beispiel beim Reinforcement Learning mit kontinuierlichem Zustands- und Aktionsraum der Fall. Eine genaue Beschreibung hierzu ist in Abschnitt 6.4 zu finden.

(2) **Optimierung auf Grundlage der 1. Ableitung:** Da die genaue Berechnung der 2. Ableitung (Hesse-Matrix) oft teuer ist, werden in dieser Kategorie oft Näherungsverfahren für diese eingesetzt. Es wird also nur ein Gradient benötigt. Durch den Gradienten kann die Richtung eines lokalen Minimums berechnet werden.

(3) **Optimierung mit der 1. und 2. Ableitung:** Ähnlich dem klassischen Newton-Verfahren nutzen diese Algorithmen alle Informationen zur Berechnung des Optimums. Beim überwachten Lernen und beim Reinforcement Learning mit kontinuierlichem Zustands- und diskretem Aktionsraum sind die erste und zweite Ableitung berechenbar, da ein numerischer Fehler berechnet werden kann. Mit Hilfe der Hesse-Matrix wird die Schrittweite bis zum nächsten lokalen Minimum abgeschätzt.

Verfahren	Typ	Besonderheiten
CMA Evolution Strategies	(1)	geeignet für nicht-separierbare, schlecht konditionierte Funktionen
Conjugate Gradient	(2)	linearer Speicherverbrauch $O(L)$
Levenberg-Marquardt	(2), (3)	optimiert für Summe quadratischer Fehler
Sequential Quadratic Programming	(2), (3)	kann Hesse-Matrix approximieren
Stochastic Gradient Descent	(2)	geeignet für große Datensätze und linearer Speicherverbrauch $O(L)$

Tabelle 4.1.: Eine Auswahl beliebter Optimierungsverfahren. Die Spalte *Typ* gibt die Kategorie des Optimierungsalgorithmus nach Abschnitt 4.3 an.

Eine kurze Übersicht über die hier betrachteten Optimierungsalgorithmen ist in Tabelle 4.1 zu sehen. Im Folgenden werden die Optimierungsalgorithmen beschrieben und bewertet.

4.4. Conjugate Gradient

Nach LeCun u. a. [51] hat Conjugate Gradient (CG) den Vorteil, dass der Speicherverbrauch während der Optimierung linear in der Anzahl der Parameter ist, da die Hesse-Matrix nicht berechnet oder approximiert wird. Dadurch eignet sich CG für große neuronale Netze mit vielen Parametern [63, Part 2: What are conjugate gradients, Levenberg-Marquardt, etc.?].

Die hier verwendete Variante stammt aus der Bibliothek ALGLIB [14]. Das Verfahren benötigt nur die Fehlerfunktion und den Gradienten.

Die Richtung der Optimierung ρ_t im Schritt t entspricht nicht, wie bei einem normalen Gradientenabstieg, dem negativen Gradienten, sondern wird durch

$$\rho_t = -\nabla E(w_t) + \beta_t \rho_{t-1} \tag{4.4.2}$$

berechnet [51]. Die vorherige Richtung wird in die Berechnung einbezogen, wodurch die Richtung sich langsamer verändert. Zudem verwendet CG ein Linesearch-Verfahren. Es wird also in der errechneten Richtung eine gute Schrittlänge gesucht.

4.5. Levenberg-Marquardt

Der Levenberg-Marquardt-Algorithmus (LMA) [54, 57] eignet sich speziell zur Optimierung von Problemen, deren Fehlerfunktion die Summe quadratischer Fehler ist. Diese wird bei Neuronalen Netzen oft verwendet. Das Verfahren benötigt die Fehler der einzelnen Trainingsdaten und eine Jacobi-Matrix, in der jede Zeile den Gradienten eines Trainingsdatums repräsentiert. Speicher in der Größenordnung von $N \cdot L$ wird also für die Jacobi-Matrix benötigt, wobei L die Anzahl der optimierbaren Parameter und N die Größe des Trainingsdatensatzes sind. Der Rechenaufwand in jedem

Optimierungsschritt ist in $O(L^3)$ [51]. LMA ist demzufolge für wenige Parameter effizient und außerdem im Gegensatz zum einfachen Gradientenabstieg gut für schlecht konditionierte Fehlerfunktionen geeignet, da der Algorithmus Informationen über die (approximierte) zweite Ableitung nutzt [63].

Die Hesse-Matrix ist normalerweise nicht erforderlich, sondern wird approximiert. In der hier verwendeten Variante aus der Bibliothek ALGLIB ist es allerdings auch möglich, die exakte Hesse-Matrix anzugeben, wodurch auch beliebige Fehlerfunktionen optimiert werden können [13].

4.6. Sequential Quadratic Programming

Sequential Quadratic Programming (SQP) benötigt mindestens die erste Ableitung der Fehlerfunktion nach den Parametern und möglichst die zweite Ableitung. Diese kann aber auch angenähert werden. Die Stärke liegt darin, dass sehr wenige Funktionsauswertungen zur Konvergenz notwendig sind. Allerdings bleibt das Verfahren häufig in lokalen Minima oder Sattelpunkten hängen.

In dieser Arbeit wird eine aus der Software GNU Octave [27] übernommene SQP-Implementierung verwendet. Allerdings beachtet diese Implementierung im Gegensatz zum Original keine Randbedingungen der Parameter. Die Implementierung approximiert ein nichtlineares Optimierungsproblem in jedem Schritt durch ein quadratisches Optimierungsproblem der Form

$$\min_{x'} 0.5 x'^T H x' + x'^T g, \tag{4.6.3}$$

wobei H die Hesse-Matrix und g der Gradient der Fehlerfunktion sind. Danach wird in der sich daraus ergebenden Richtung p nach der optimalen Schrittlänge α mit Hilfe eines Linesearch-Verfahrens gesucht, sodass

$$\min_{\alpha} f(x + \alpha p). \tag{4.6.4}$$

Der Algorithmus terminiert, wenn entlang der errechneten Richtung kein besserer Funktionswert mehr zu finden ist. SQP nutzt ebenso wie LMA die zweite Ableitung der Fehlerfunktion und sollte deshalb besser mit schlecht konditionierten Fehlerfunktionen zurechtkommen [63].

Die teuerste Operation innerhalb des Algorithmus ist das Invertieren der Hesse-Matrix zur quadratische Optimierung. Der Aufwand dafür ist bei der hier verwendeten Matrix-Bibliothek Eigen [33] kubisch in der Anzahl der Parameter, das heißt er liegt in $O(L^3)$.

4.7. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) ist eine Variante des normalen Gradientenabstiegs. Es handelt sich hierbei im Gegensatz zu allen anderen hier vorgestellten Optimierungsalgorithmen um ein Online-Learning-Verfahren und nicht um ein Batch-Learning-Verfahren. Die Parameter werden jeweils anhand des Gradienten eines Trai-

ningsdatums angepasst. Wenn unkomprimierte Gewichte optimiert werden, wird also in jedem Schritt der Gewichtsvektor $\mathbf{w}$ durch

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta \nabla E^{(n)}(\mathbf{w}_t) \tag{4.7.5}$$

angepasst, wobei η eine Lernrate ist. In dieser Arbeit wird die Lernrate über den Verlauf der Optimierung verkleinert nach der Formel

$$\eta = \frac{1}{2000 + \frac{t}{6N}}, \tag{4.7.6}$$

wobei t die Anzahl der Veränderungen der Parameter und N die Größe des Trainingsdatensatzes sind.

Ich werde die Trainingsdaten zur Anpassung der Parameter in zufälliger Reihenfolge auswählen. Die Auswahlwahrscheinlichkeit eines Trainingsdatums ist hierbei nicht, wie in der Empfehlung von LeCun u. a. [51], proportional zu dem Fehler des Trainingsdatums im letzten Durchgang, sondern zu der Norm des Gradienten.

SGD ist „normalerweise viel schneller“ und „führt oft zu besseren Lösungen“ [51] als der normale Gradientenabstieg. Insbesondere eigne sich das Verfahren nach LeCun u. a. [52] für große Datensätze und viele optimierbare Parameter und sei hier schneller als LMA, BFGS und CG.

Um eine Vergleichbarkeit zu den anderen Optimierungsalgorithmen herzustellen, spreche ich in dieser Arbeit im Zusammenhang mit SGD von einer Iteration, wenn N Gewichtsanpassungen vorgenommen wurden, wobei N die Größe des Trainingsdatensatzes ist. Allerdings wird in jeder Iteration nicht notwendigerweise für jedes Trainingsdatum eine Gewichtsanpassung vorgenommen, da andere mehrfach verwendet werden können.

4.8. Evolution Strategies with Covariance Matrix Adaption

Evolution Strategies ist ein ableitungsfreier Optimierungsalgorithmus. Die Erweiterung Covariance Matrix Adaption beschleunigt vor allem bei nicht-separierbaren Funktionen die Optimierung um einige Größenordnungen [34]. Zudem kommt CMA-ES gut mit schlechter Konditionierung zurecht [35].

Eine modifizierte Variante, die mehrmals neu gestartet wird und dabei die Populationsgröße erhöht, ist IPOP-CMA-ES [3]. IPOP-CMA-ES schneidet im Vergleich zu anderen CMA-ES-Varianten und ableitungsfreien Optimierungsalgorithmen meistens sehr gut ab [2]. Aus diesem Grund wird hier diese Variante zum Trainieren des Multilayer Perceptrons verwendet. Die in dieser Arbeit verwendete Implementierung ist eine Portierung der ANSI-C-Implementierung von Nikolaus Hansen.[1]

Das in Abbildung 2.2 vorgestellte Optimierungsproblem, bei dem ein sehr einfaches neuronales Netz optimiert wird, ist in Abbildung 4.1 sichtbar besser durch CMA-ES gelöst. CMA-ES ist gegenüber Verfahren, die die Ableitung nutzen allerdings deutlich

[1] CMA-ESpp: `https://github.com/AlexanderFabisch/CMA-ESpp`.

(a) (b)

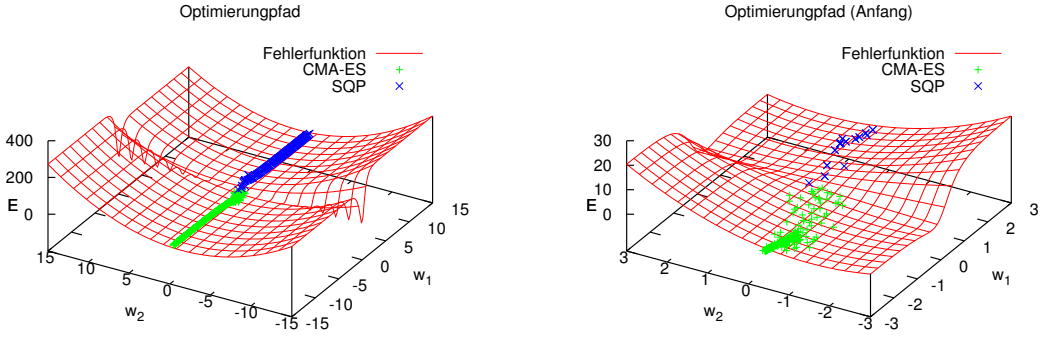


Abbildung 4.1.: (a) Anwendung von Evolution Strategies mit Covariance Matrix Adaption und Sequential Quadratic Programming auf das in Abbildung 2.2 vorgestellte Optimierungsproblem. Die Fehlerfunktion ist symmetrisch. Beide Algorithmen finden eine ähnlich gute Lösung. Bei einem Fehler unter 10^{-10} wurde die Optimierung abgebrochen. CMA-ES benötigt deutlich mehr Funktionsauswertungen und bewegt sich mit kleineren „Schritten“. (b) Dies ist vor allem am Anfang (Mitte) sichtbar.

Symbol	Bedeutung	Standard-Wert
λ	Populationsgröße	$4 + \lfloor 3 \ln N \rfloor$
σ_0	initiale Schrittgröße	10
-	Populationswachstumsfaktor nach Neustart	2

Tabelle 4.2.: Parameter von IPOP-CMA-ES. N ist die Dimension des Suchraums.

langsamer und sollte deshalb nur verwendet werden, wenn keine Ableitung berechnet werden kann.

Im Gegensatz zu den anderen Algorithmen gibt es hier einige Parameter, die eingestellt werden müssen und einen großen Einfluss auf die Optimierungsdauer und das Ergebnis haben können. Für die meisten gibt es in der verwendeten Implementierung allerdings empfohlene Werte. Die wichtigsten Parameter und die hier verwendeten Standard-Werte sind in Tabelle 4.2 zu sehen. Alle nicht genannten Parameter wurden gegenüber dem empfohlenen Standard-Wert der Implementierung nicht verändert. Alle Abweichungen von den hier genannten Standard-Werten sind in den entsprechenden Abschnitten aufgeführt.

4.9. Praxistauglichkeit

Vorab werden hier die vorgestellten Optimierungsverfahren anhand einfacher Probleme miteinander vergleichen. Zunächst werden hierzu ein sehr einfacher Datensatz und dann einige Benchmarks der Bibliothek FANN [59] verwendet.

Die Topologie der Netze bei dem ersten Problem ist sehr einfach gehalten. Ich verwende im Folgenden die Topologie-Notation D-H-...-F, die die Anzahl der Neuronen in den einzelnen Schichten ohne Berücksichtigung des Bias angibt. Die Gewichte sind

(a) (b)

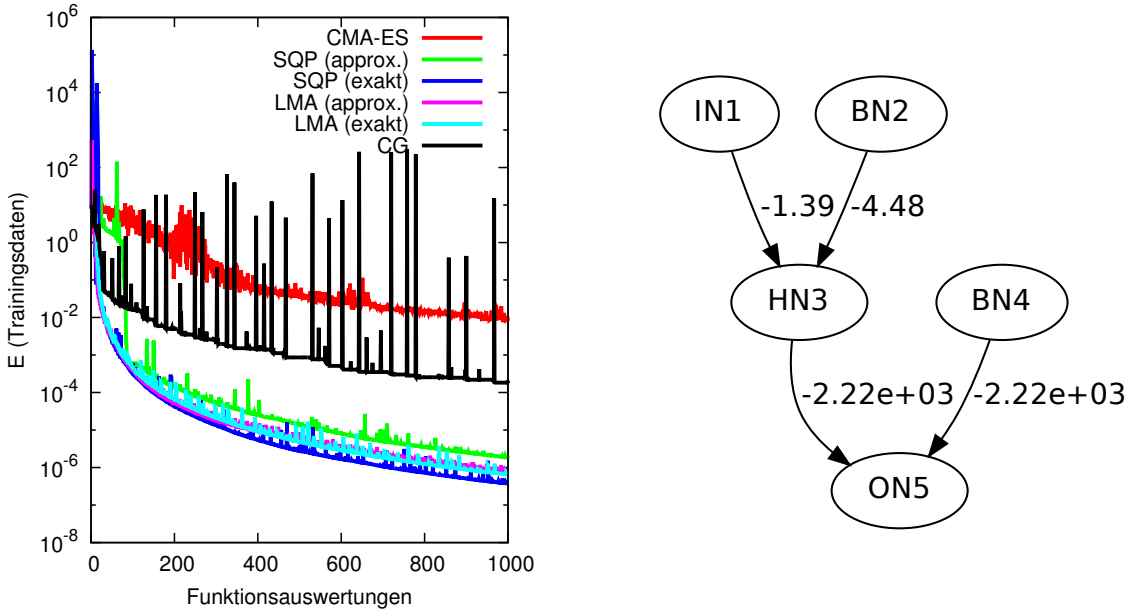


Abbildung 4.2.: (a) Bei einer versteckten Schicht liegen LMA und SQP mit exakter Hesse-Matrix nach Funktionsauswertungen vorn, allerdings benötigt LMA etwas weniger Gradienten- und Hesse-Matrix-Auswertungen. (b) Die Topologie des optimierten MLP (1-1-1 mit Bias).

mit dem in Abschnitt 3 vorgestellten Verfahren komprimiert. Dazu werden in jeder Gewichtsschicht zwei Parameter zur Approximation verwendet. Es findet tatsächlich keine Komprimierung statt, da die Anzahl der Parameter nicht geringer ist, als die der Gewichte. Der Trainingsdatensatz ist

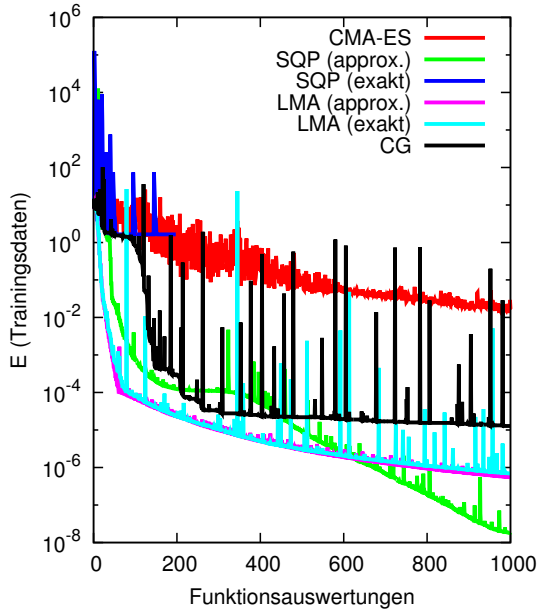
$$T = \{(1;1),(2;2),(3;2,5),(4;2,75),(5;2,875),(6;2,9375),(7;2,96875)\}. \quad (4.9.7)$$

Es werden drei verschiedenen Topologien verwendet: je ein Multilayer Perceptron mit einer versteckten Schicht (Abbildung 4.2), zwei versteckten Schichten (Abbildung 4.3) und drei versteckten Schichten (Abbildung 4.4). Auf der linken Seite der Abbildungen (a) werden jeweils die Fehlerwerte bei den einzelnen Funktionsauswertungen dargestellt. Hierbei ist zu beachten, dass die Funktionsauswertungen allein kein Maß für die Eignung der Algorithmen sind. Auswertungen des Gradienten und der Hesse-Matrix sind ebenso in die Bewertung mit einzubeziehen. Auf der rechten Seite sind jeweils die Topologien der Netze zu sehen. Die Kantenbeschriftungen sind jeweils die besten gefundenen Gewichte. Die Knotenbeschriftungen geben den Typ des Neurons an (IN: Eingabeneuron, BN: Bias, HN: normales Neuron, ON: Ausgabeneuron).

Es fällt auf, dass bei einer versteckten Schicht die Verfahren, die die exakte Hesse-Matrix nutzen, am besten abschneiden, bei mehreren versteckten Schichten allerdings eher die Verfahren, die die Hesse-Matrix approximieren.

Dieses Problem ist sehr einfach und dennoch wird bereits klar, dass die Auswertung der exakten Hesse-Matrix für Batch-Verfahren keinen großen Vorteil bringt. Bei größeren Datensätzen wird dies noch deutlicher, da der Aufwand zur Berechnung der

(a)



(b)

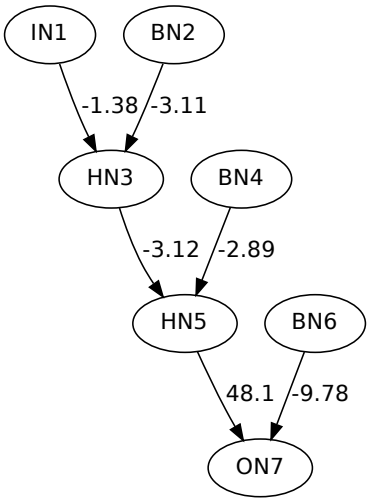
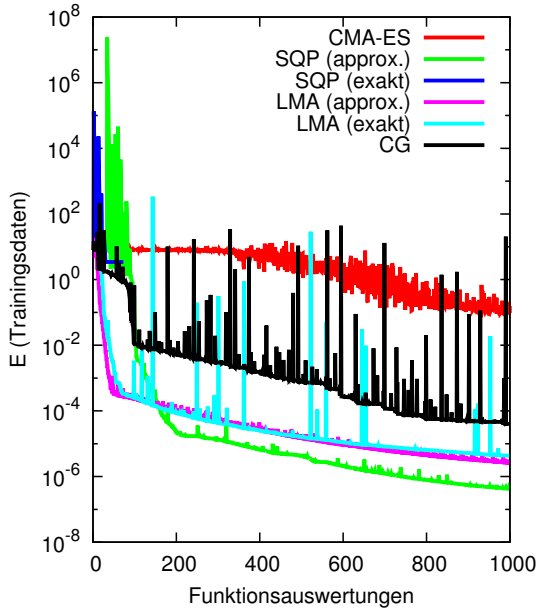


Abbildung 4.3.: (a) Hier schlägt SQP mit exakter Hesse-Matrix fehl. Mit der Approximation funktioniert es allerdings deutlich besser. LMA funktioniert in beiden Varianten ähnlich gut. (b) Die Topologie des optimierten MLP (1-1-1-1 mit Bias).

(a)



(b)

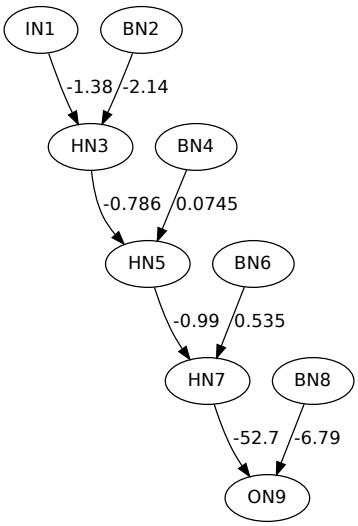


Abbildung 4.4.: (a) Hier unterscheidet sich das Ergebnis kaum von dem aus Abbildung 4.3. (b) Die Topologie des optimierten MLP (1-1-1-1-1 mit Bias).

exakten Hesse-Matrix viel größer ist. Alleine die Berechnung der Hesse-Matrix dauert teilweise länger als mehrere komplette Iterationen von Optimierungsalgorithmen mit approximierter Hesse-Matrix. Aus diesem Grund werde ich die exakte Hesse-Matrix in späteren Experimenten nicht mehr verwenden.

Die folgenden Benchmarks der Bibliothek FANN [59] wurden eingesetzt, um die Algorithmen bei komplizierteren Problemen gegenüberzustellen:

- das Two Spirals Problem [48],
- das Parity 8 Problem, welches eine Generalisierung des XOR-Problems ist,
- das Mushroom Data Set [75][2]
- und das Pima Indians Diabetes Data Set[3].

LMA stellte sich bei den hier genannten Datensätzen als zuverlässigstes Optimierungsverfahren heraus. LMA konvergiert am schnellsten beim Mushroom Data Set (Topologie 125-1-2 mit Bias), bleibt im Gegensatz zu SQP nicht in lokalen Minima beim Parity 8 Problem (Topologie 8-16-1 mit Bias) hängen und erlernt durchschnittlich mehr korrekte Vorhersagen bei den Testdaten des Pima Indians Data Set (Topologie 8-4-2 mit Bias) und beim Two Spirals Problem (Topologie 2-20-10-2 mit Bias) als alle anderen vorgestellten Optimierungsalgorithmen.

Demzufolge wird hier zuerst versucht, LMA ohne die approximierte Hesse-Matrix einzusetzen. Wenn die Anzahl der Parameter sehr groß ist, werden allerdings CG oder SGD genutzt, da Speicher- und Zeitkomplexität in jeder Iteration dieser Optimierungsalgorithmen nur linear von der Anzahl der Parameter abhängen. Wenn der Gradient nicht berechnet werden kann, wird IPOP-CMA-ES eingesetzt.

[2] Beschreibung unter `http://archive.ics.uci.edu/ml/datasets/Mushroom`.
[3] Beschreibung unter `http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes`.

5. Anwendung: Überwachtes Lernen

Beim überwachten Lernen ist ein Trainingsdatensatz

$$T = \{(x^{(1)}, t^{(1)}), \ldots, (x^{(N)}, t^{(N)})\} \tag{5.0.1}$$

gegeben, wobei $x^{(1)}, \ldots, x^{(N)} \in X$ und $t^{(1)}, \ldots, t^{(N)} \in Y$ und die Zuordnung durch eine unbekannte Funktion $f : X \to Y$ generiert wurde [74]. Ein Lernverfahren soll die Funktion f aus den Trainingsdaten rekonstruieren. Das heißt, es soll den Fehler auf den Trainingsdaten minimieren und bisher ungesehene Daten korrekt vorhersagen. Ist die Menge Y endlich, so handelt es sich um ein Klassifikationsproblem, ansonsten um ein Regressionsproblem [74]. f wird hier durch ein neuronales Netz approximiert.

In diesem Kapitel werden verschiedene Klassifikationsprobleme vorgestellt und das Verhalten eines MLP mit komprimierten Gewichten bei der Lösung der Aufgaben untersucht.

5.1. Two Spirals

5.1.1. Datensatz

Ich untersuche hier einen Benchmark der Bibliothek FANN (Fast Artificial Neural Network Library) [59] genauer. Der ursprünglich durch Lang und Witbrock [48] verwendete Datensatz enthält Koordinaten einer Ebene, die zu zwei verschiedenen, ineinander verschlungenen Spiralen gehören. Der Datensatz wurde bereits zum Testen vieler Lernverfahren eingesetzt. Der hier verwendete Datensatz enthält sowohl 193 Trainings- als auch 193 Testdaten (siehe Abbildung 5.1). Dadurch kann ein mögliches Overfitting gemessen werden. Beide Datensätze sind regelmäßig auf den beiden Spiralen verteilt, sodass auf einer Spirale jeweils abwechselnd Punkte des Test- und des Trainingsdatensatzes liegen. Die Eingaben sind jeweils auf das Intervall $[0, 1]$ beschränkt. Die Klasse wird hier durch die Ausgaben -1 oder 1 repräsentiert. Sobald die Ausgabe des MLP größer als 0 ist, wird die Eingabe der Klasse 1 zugeordnet, ansonsten der Klasse -1.

Es handelt sich hierbei eigentlich nicht um eine Aufgabe für die ein MLP mit komprimierten Gewichten sehr viel besser als eines mit unkomprimierten Gewichten geeignet wäre, da normalerweise keine starke Korrelation der verschiedenen Gewichte zu erwarten ist. Das Problem wird hier zum Vergleich der Backpropagation-Verfahren für komprimierte und normale MLP verwendet und als Testfall der Implementierung.

5.1.2. Methode

Osowski u. a. [65] haben das Problem erfolgreich mit einem MLP mit der Topologie 2-50-1 gelöst. Dabei wurden durchschnittlich ungefähr 650 Iterationen des Optimierungsalgorithmus benötigt. Zur Optimierung wurden die Algorithmen BFGS und

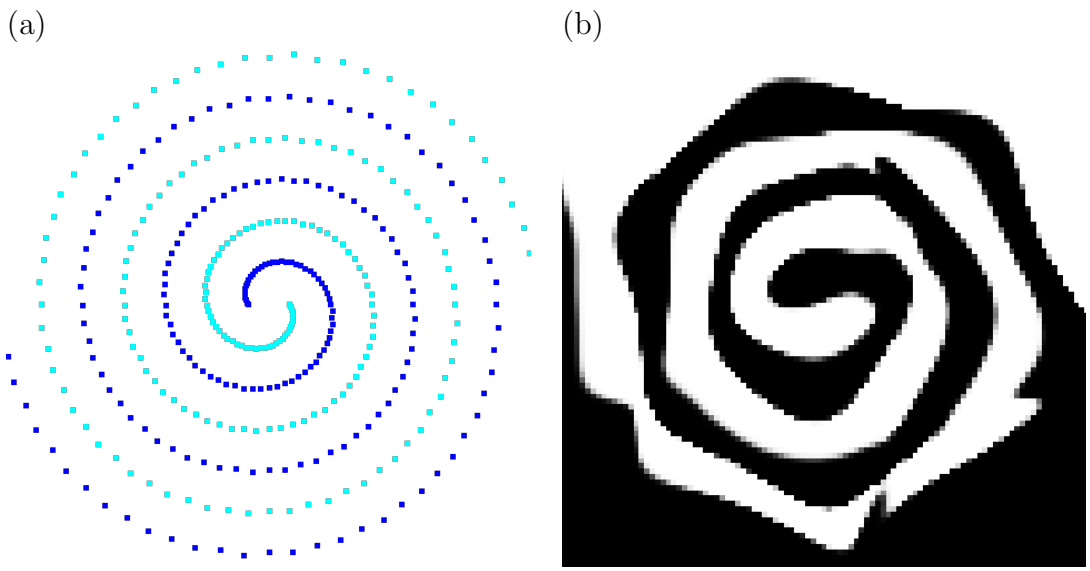


Abbildung 5.1.: (a) Two-Spirals-Datensatz. Die Farbe gibt die Klasse an. (b) Ein Modell, das durch ein MLP erlernt wurde.

Conjugate Gradient eingesetzt. In der Beschreibung der CMU Neural Networks Benchmark Collection [90] wird darauf hingewiesen, dass ein Netzwerk mit der Topologie 2-20-10-1 und Bias erfolgreich auf den Datensatz angewandt wurde. Zum Lernen mit Backpropagation und Gradientenabstieg wurden 13.900 Epochen benötigt. Deshalb werden hier zunächst verschiedene Topologien mit zwei versteckten Schichten ohne Komprimierung untersucht, um eine geeignete Topologie zu bestimmen.

Ich verwende in jeder Schicht Tangens Hyperbolicus als Aktivierungsfunktion und die halbierte Summe quadratischer Fehler (SSE) als Fehlerfunktion.

In diesem Versuch wurden orthogonale Kosinus-Funktionen (siehe Gleichung 3.1.5) zur Komprimierung verwendet. Die Parameter wurden mit dem in Abschnitt 4.2 beschriebenen Verfahren initialisiert. Die Gewichte für unkomprimierte neuronale Netze wurden aus so initialisierten Parametern generiert (für jede Topologie 3-21-21 Parameter). Durch dieses Vorgehen ist die Auswertung nicht durch die Initialisierung verfälscht.

Ausreichend gut funktioniert bei diesem Problem nur der Optimierungsalgorithmus Levenberg-Marquardt (LMA) ohne die exakte Hesse-Matrix. Sequential Quadratic Programming mit approximierter und exakter Hesse-Matrix lernen den Trainingsdatensatz nicht, das heißt die Korrektklassifikationsrate liegt fast immer unter 60 %, und die Berechnung der exakten Hesse-Matrix ist generell deutlich langsamer als eine Iteration des normalen LMA. Mit Conjugate Gradient können zwar ähnlich gute Werte für die Korrektklassifikationsrate erzielt werden, allerdings liegt die Rechenzeit pro Experiment hier zwischen 40 und 90 Sekunden, da ein Vielfaches der Iterationen bis zum Erfüllen der Abbruchbedingungen der Optimierung benötigt wird. Der Vorteil des linearen Speicherverbrauchs von Conjugate Gradient kann in dieser Anwendung nicht ausgenutzt werden, da die Anzahl der Parameter zu gering ist. Gleiches gilt für Stochastic Gradient Descent.

Topologie	2-10-5-1	2-10-10-1	2-20-10-1	**2-20-20-1**
Gewichte	91	151	281	501
Iterationen (LMA)	7818,36	1249,61	756,55	277,3
Trainingsdauer	36,235 s	9,864 s	15,453 s	18,696 s
SSE (Training)	12,477	0,214	0	0
SSE (Test)	37,437	11,634	7,543	5,962
Korrekte Vorhersagen (Test)	170,41	186,8	188,88	189,69
Korrektklassifikationsrate (Test)	88,295 %	96,788 %	97,865 %	98,285 %
Beste Vorhersage (Test)	175,87	189,49	190,85	191,55
Topologie	**2-20-20-1**			
Komprimierung	3-6-1	3-6-6	3-12-12	3-21-21
Parameter	181	186	312	501
Iterationen (LMA)	309,91	619,91	376,67	274,48
Trainingsdauer	3,901 s	7,8 s	10,338 s	19,273 s
SSE (Training)	0	0	0	0
SSE (Test)	11,045	14,267	10,893	13,484
Korrekte Vorhersagen (Test)	186,95	185,26	187	185,65
Accuracy (Test)	96,865 %	95,99 %	96,891 %	96,192 %
Beste Vorhersage (Test)	188,68	188,73	189,82	188,99

Tabelle 5.1.: Ergebnisse der Experimente. Alle Werte sind auf 3 Nachkommastellen genau und über 100 Experimente gemittelt. In der oberen Tabellenhälfte sind die Ergebnisse für unkomprimierte Gewichte zu sehen, in der unteren Hälfte für komprimierte Gewichte. Die Notation für die Komprimierung wird in Anhang B erläutert.

Die Optimierung durch LMA wurde abgebrochen nach 10.000 Iterationen oder wenn

$$|E_{SSE}^{(t)} - E_{SSE}^{(t-1)}| \leq 10^{-18} \max\{E_{SSE}^{(t-1)}, E_{SSE}^{(t)}, 1\}, \tag{5.1.2}$$

wobei $t \geq 1$ die Iteration angibt und $E_{SSE}^{(t)}$ den Fehler in Iteration t. Mit jeder Konfiguration wurden 100 Testläufe durchgeführt.

5.1.3. Ergebnisse

In Tabelle 5.1 sind die Ergebnisse der Tests zusammengefasst. Zum Vergleich wurden die Anzahl der Iterationen von LMA, der Trainingsfehler, der Testfehler, die korrekten Vorhersagen auf dem Testdatensatz (von maximal 193) und die sich daraus ergebende Korrektklassifikationsrate über die Tests gemittelt. Zudem wurde neben der Anzahl der korrekt klassifizierten Testdaten am Ende der Optimierung auch die beste im Verlauf der Optimierung erreichte Anzahl korrekt klassifizierter Testdaten angegeben. Das Maß Iterationen ist kein gutes Kriterium zum Vergleich der Effizienz eines Lernverfahrens, da diese, abhängig von der Topologie und der Parameteranzahl, unterschiedlich lange dauern. Aus diesem Grund wurde die durchschnittliche Trainingsdauer angegeben. Die Zeit ist etwas verfälscht, da in jeder Iteration der Fehler auf dem Testdatensatz berechnet und aufgezeichnet wurde. Dies wird in realen Anwendungen meistens nicht

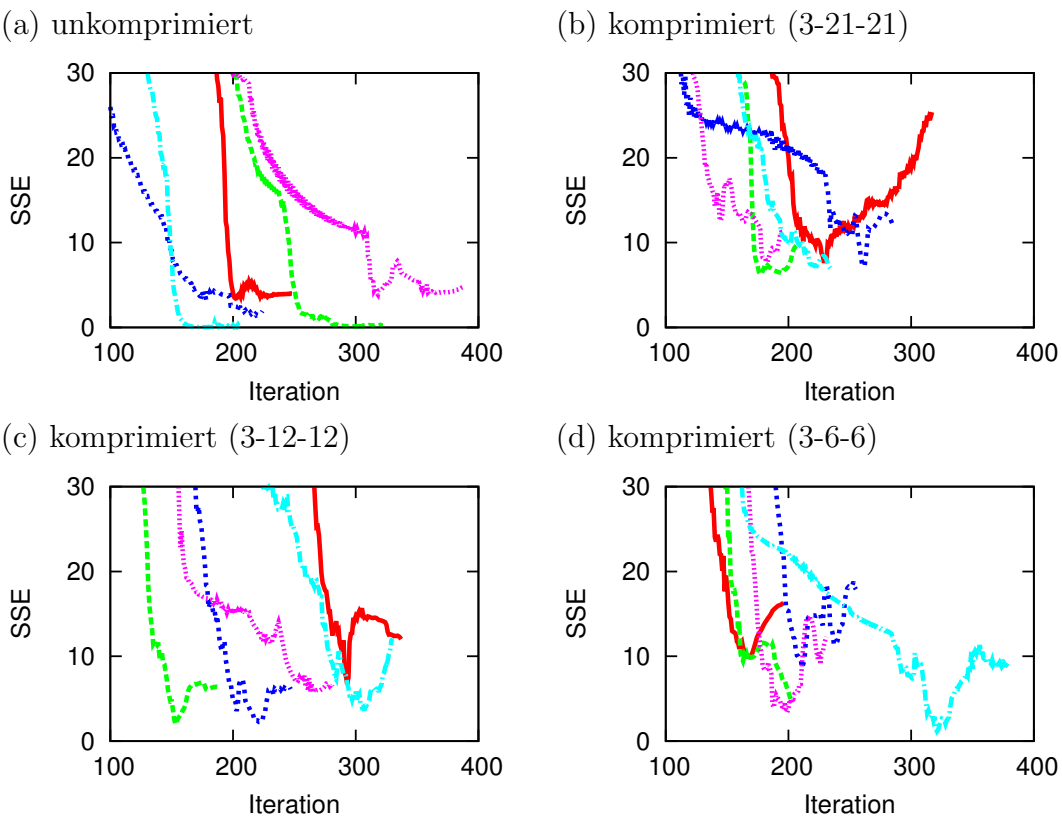


Abbildung 5.2.: Die Summe der quadratischen Fehler (SSE) auf den Testdaten gegen Ende der Optimierung. Die hier verwendete Topologie ist 2-20-20-1. Es ist der Testfehler von fünf verschiedenen Durchläufen zu sehen.

getan. Die Zeit wurde auf einem Lenovo T510 (siehe Anhang F, Rechnerkonfiguration 4) gemessen.

Durch Hinzufügen von mehr versteckten Knoten wird die Lösung in weniger LMA-Iterationen gefunden. Zudem steigt auch die Korrektklassifikationsrate leicht an. Ab der Topologie 2-10-10-1 ist das Problem gut lösbar. Der Fehler auf den Trainingsdaten ist fast 0 und durchschnittlich werden über 95 % der Testdaten korrekt vorhergesagt. Die am besten geeignete, getestete Topologie ist 2-20-20-1. Hier werden am wenigsten Iterationen benötigt, um eine Lösung zu finden, und der Fehler auf den Testdaten ist am geringsten. Allerdings wird der Rechenaufwand bei einer größeren Anzahl von versteckten Knoten auch größer, wodurch die Laufzeit bis zum Optimierungsabbruch steigt.

Durch die Komprimierung der Gewichte ohne Parameterreduktion wird der Suchraum so transformiert, dass zwar durchschnittlich ähnlich schnell eine Lösung gefunden wird, diese allerdings nicht ganz so gut ist. Durch Reduktion der Parameter tritt ein ähnlicher Effekt auf, wie bei dem Entfernen von versteckten Neuronen: die Anzahl der benötigten Iterationen steigt leicht. Allerdings nimmt auch insgesamt die Laufzeit ab, da weniger Parameter optimiert werden müssen.

Aus der Abbildung 5.2 ist abzuleiten, dass gegen Ende der Optimierung Overfitting ein Problem darstellt. Aus diesem Grund habe ich überprüft, ob sich das Over-

Topologie	2-10-5-1	2-10-10-1	2-20-10-1	2-20-20-1
Beste Vorhersage	175,87	189,49	190,85	191,55
Letzte Vorhersage	170,41	186,8	188,88	189,69
Topologie	2-20-20-1			
Komprimierung	3-6-1	3-6-6	3-12-12	3-21-21
Beste Vorhersage	188,68	188,73	189,82	188,99
Letzte Vorhersage	186,95	185,26	187	185,65

Tabelle 5.2.: Vergleich der durchschnittlich besten und letzten korrekten Vorhersagen auf dem Testdatensatz.

fitting stärker bei einer komprimierten Gewichtsrepräsentation bemerkbar macht. In Tabelle 5.2 ist der Vergleich zwischen der im Durchschnitt besten Vorhersage und der Vorhersage nach Ende der Optimierung zu sehen. Bei einer großen Anzahl von Parametern ist der Effekt des Overfitting bei diesem Problem etwas stärker ausgeprägt mit einer komprimierten Gewichtsrepräsentation. Allerdings wird hier auch stärker, als eigentlich nötig gewesen wäre, optimiert und ohne das Overfitting ist die Vorhersage ohne Komprimierung immer noch besser. Eine allgemeine Aussage darüber, ob Overfitting bei einer Komprimierung verstärkt auftritt, lässt sich hier nicht ableiten. Dieser Aspekt muss bei weiteren Experimenten beachtet werden.

5.1.4. Erkenntnisse

Für die Summe quadratischer Fehler ist der Levenberg-Marquardt-Algorithmus ohne die exakte Hesse-Matrix normalerweise am besten zur Optimierung eines neuronalen Netzes geeignet, wenn als Kriterium Korrektklassifikationsrate zugrunde gelegt wird. CG und SGD sind weitere Optionen für eine große Anzahl von optimierbaren Parametern.

Ein Problem, bei dem eine Gewichtskomprimierung eingesetzt wird, sollte eine möglichst große Anzahl von Eingabekomponenten haben, die miteinander korrelieren. Dies ist beispielsweise bei EEG-Daten oder Bildern der Fall. Der Benchmark Two Spirals wird durch ein normales MLP genauer gelöst.

Die Implementierung des MLP und der Gewichtskomprimierung funktioniert für diesen Benchmark und es kann deshalb im Folgenden davon ausgegangen werden, dass kein gravierender Implementierungsfehler vorhanden ist.

5.2. Ziffernerkennung

5.2.1. Datensatz

Der MNIST-Datensatz [53] wird seit einigen Jahren als Standard-Benchmark für Klassifikationsverfahren eingesetzt. Der Datensatz besteht aus 60.000 Trainings- und 10.000 Testdaten. Das Ziel ist die Erkennung von handschriftlichen Ziffern in Bildern mit 28x28 Pixeln. Einige Beispielbilder sind in Abbildung 5.3 zu sehen. Da im Laufe der Jahre einige sehr spezialisierte Verfahren auf diesen Datensatz angewendet wurden, ist

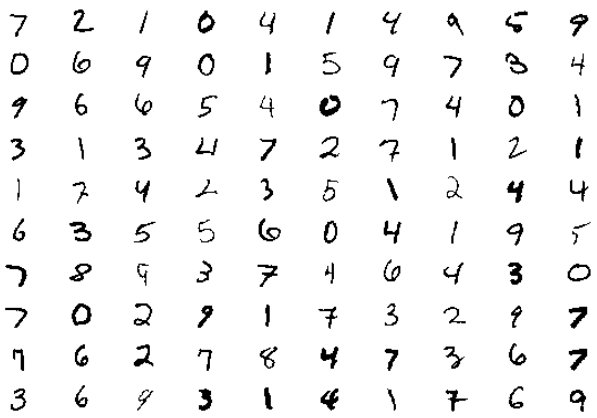


Abbildung 5.3.: Die ersten 100 Bilder des MNIST-Testdatensatz ohne Grauabstufungen dargestellt.

Topologie (MLP)	Korrektklassifikationsrate	Gewichte	Publikation
784-500-300-10	98,47 %	545.810	nicht publiziert, Quelle: LeCun und Cortes [53]
784-800-10	98,4 %	636.010	Simard u. a. [79]
784-500-150-10	97,05 %	469.160	LeCun u. a. [52]
784-300-100-10	96,95 %	266.610	LeCun u. a. [52]
784-1000-10	95,5 %	795.010	LeCun u. a. [52]
784-300-10	95,3 %	238.510	LeCun u. a. [52]

Tabelle 5.3.: Beste mit Backpropagation trainierte neuronale Netze ohne Vorverarbeitung oder Generierung von zusätzlichen Trainingsdaten.

mein Ziel hier nicht das Erreichen der höchsten Korrektklassifikationsrate. Stattdessen soll demonstriert werden, dass ein sehr einfaches neuronales Netz in der Lage ist, mit Hilfe der Komprimierung eine relativ hohe Korrektklassifikationsrate zu erreichen.

Zur Vergrößerung des Trainingsdatensatzes wurden bei anderen Ergebnissen verschiedene Verzerrungen der Bilder generiert und es wurden verschiedene Methoden zur Vorverarbeitung angewendet. Dies wird hier nicht getan und dabei allerdings die Komprimierung des Netzes nicht als Vorverarbeitungsschritt, sondern als Modifikation des Lernverfahrens betrachtet. Die Ergebnisse werden zunächst nur mit normalen neuronalen Netzen verglichen, die mit Backpropagation oder darauf basierenden Optimierungsalgorithmen trainiert wurden. Die besten Ergebnisse dieser neuronaler Netze ohne Vorverarbeitung und Generierung weiterer Trainingsdaten sind in Tabelle 5.3 zu sehen. Der Wertebereich der Eingaben ist normalerweise $[0, 255]$. In den folgenden Experimenten wurde dieser jedoch auf $[0, 1]$ skaliert.

5.2.2. Methode

Da es sich hierbei um ein Problem mit mehr als zwei Klassen handelt, verwende ich in der letzten Schicht die Softmax-Aktivierungsfunktion [10, Seite 237-240]

$$y_f = g(a_f) = \frac{\exp a_f}{\sum_{f'=1}^{F} \exp a_{f'}}, \tag{5.2.3}$$

deren Ausgabe in jeder Komponente y_f zwischen 0 und 1 liegt und als Wahrscheinlichkeit für die Angehörigkeit zur f-ten Klasse angesehen werden kann. Dementsprechend hat ein neuronales Netz zur Ziffernerkennung zehn Ausgabekomponenten. Die Implementierung dieser Aktivierungsfunktion wird in Abschnitt D.2 beschrieben. Um den Gradienten einfacher berechnen zu können, verwende ich die Cross-Entropy-Fehlerfunktion

$$E_{CE}^{(n)} = \sum_{f=1}^{F} t_f^{(n)} \ln y_f^{(n)}. \tag{5.2.4}$$

Da die Fehlerfunktion nicht die Summe quadratischer Fehler ist, kann LMA nicht eingesetzt werden. Zur Optimierung hat sich Stochastic Gradient Descent als geeignet erwiesen, da es nicht in lokalen Minima oder flachen Regionen hängen bleibt und die Zeitkomplexität jedes Optimierungsschritts in $O(L)$ liegt.

Abbildung 5.3 lässt vermuten, dass die meisten Bilder komprimierbar sind. Sie sind sogar dünn besetzt, wenn die Farben invertiert werden, sodass der Hintergrund schwarz ist und somit die entsprechenden Stellen in den Eingabevektoren 0 sind. Damit müssten nach Abschnitt 3.5 zufällige Matrizen zur Komprimierung geeignet sein. Aus diesem Grund werden die einzelnen Komponenten der Komprimierungsmatrizen Φ durch Ziehen eines Wertes aus der Wahrscheinlichkeitsverteilung $\mathcal{N}(0; 0{,}01)$ generiert. Hier werden nur die Gewichte der ersten Schicht komprimiert.

5.2.3. Ergebnisse

Zum Testen einer geeigneten Komprimierung wurden zunächst kleine neuronale Netze mit verschieden starken Komprimierungen geprüft und der Fehler im Verlauf der Optimierung beobachtet. In Abbildung 5.4 (a), (b) sind die Korrektklassifikationsraten für Trainings- und Testdaten im Verlauf der Optimierung eines MLP mit der Topologie 784-10-10 und verschiedenen Komprimierungen dargestellt. Hier wird deutlich, dass eine Komprimierung einen Vorteil bringt. Das beste Ergebnis ist hier bereits mit $M = 256$ Parametern pro Neuron erreichbar. In Abbildung 5.4 (c), (d) wurden verschiedene Komprimierungen eines MLP mit der Topologie 784-100-10 getestet. Mit mehr als 256 Parametern ist auch hier keine Verbesserung auf den Testdaten mehr festzustellen.

Zuletzt wurden einige Testläufe mit MLPs mit zwei versteckten Schichten durchgeführt. Dafür wurden MLPs mit den Topologien 784-200-50-10, 784-150-50-10 und 784-100-50-10 ausgewählt. In Abbildung 5.4 (e), (f) sind die Korrektklassifikationsraten für Trainings- und Testdaten im Verlauf der Optimierung zu sehen. Die besten dabei erreichten Ergebnisse für die Korrektklassifikationsrate auf den Testdaten sind in Tabelle 5.4 zu sehen. Alle drei getesteten neuronalen Netze könnten in Tabelle 5.3

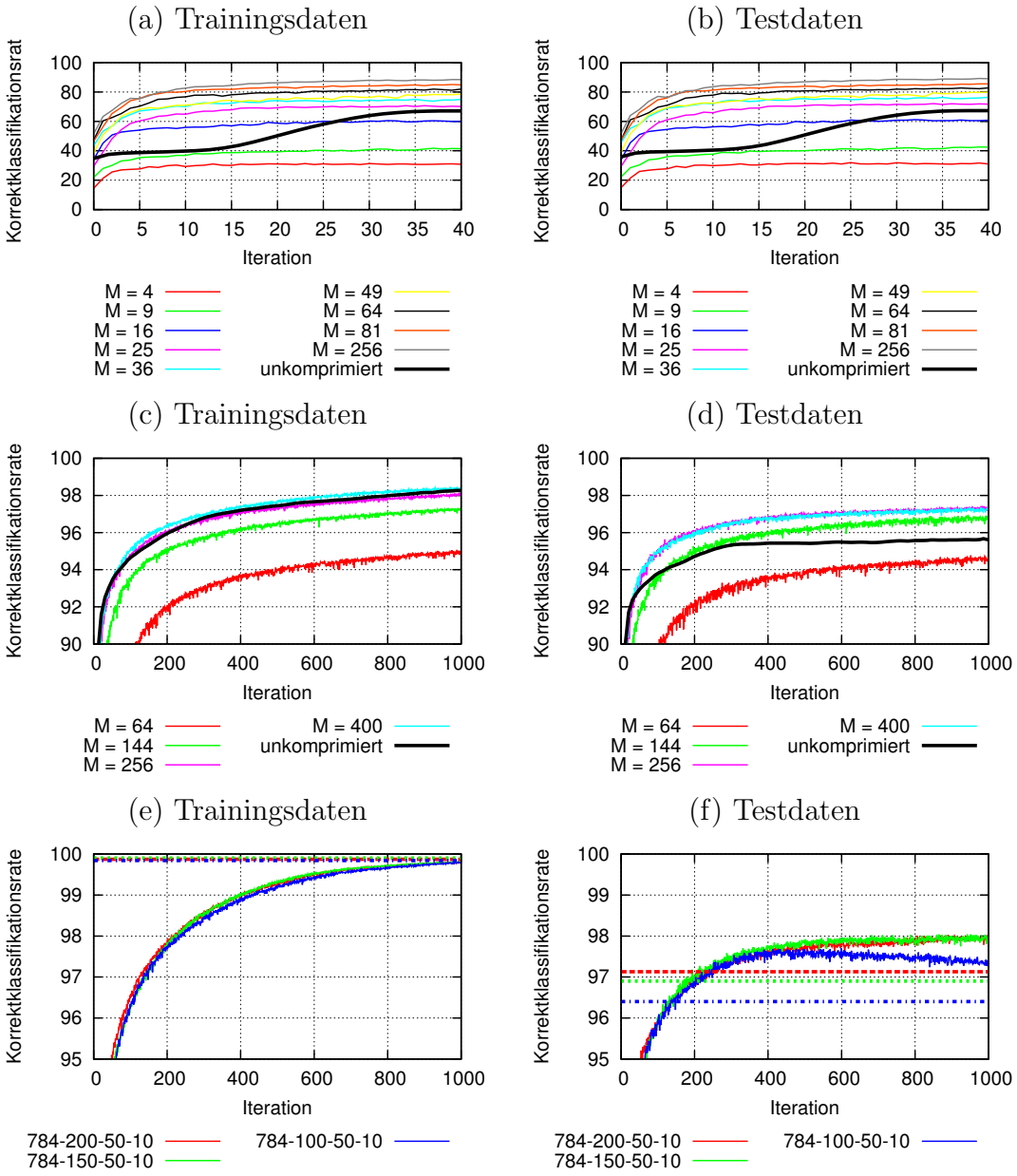


Abbildung 5.4.: (a) Prozentuale Korrektklassifikationsrate auf Trainingsdaten während des Trainings eines MLP mit 10 versteckten Knoten bei unterschiedlichen Komprimierungen und (b) Korrektklassifikationsrate auf Testdaten. (c) Korrektklassifikationsrate auf Trainingsdaten während des Trainings eines MLP mit 100 versteckten Knoten bei unterschiedlichen Komprimierungen und (d) Korrektklassifikationsrate auf Testdaten. (e) Korrektklassifikationsrate auf Trainingsdaten während des Trainings der besten getesteten Topologien (die gestrichelten Linien geben das beste Ergebnis mit unkomprimierten Gewichten an) und (f) Korrektklassifikationsrate auf Testdaten.

Topologie	Korrektklassifikationsrate	Parameter	Gewichte
MLP 784-200-50-10	98,07 %	61.960	167.560
MLP 784-150-50-10	98,04 %	46.610	125.810
MLP 784-100-50-10	97,7 %	31.260	84.060

Tabelle 5.4.: Die nach Korrektklassifikationsrate auf den Testdaten des MNIST-Datensatz besten ermittelten Ergebnisse mit komprimierten neuronalen Netzen.

an dritter Stelle eingetragen werden, obwohl die komprimierten Netze deutlich weniger Parameter haben als die beiden Besten. Die Korrektklassifikationsrate mit denselben Topologien ohne Komprimierung ist zudem um ca. 1 % schlechter. Bei komprimierten Netzen ist zwar der Fehler auf den Trainingsdaten etwas geringer, allerdings ist der Fehler auf den Testdaten größer. Training im komprimierten Raum hat hier also eine bessere Generalisierung zur Folge. Zudem ist gegenüber dem besten Ergebnis, bei dem eine Korrektklassifikationsrate von 98,47 % erreicht wurde, die Anzahl der optimierbaren Parameter bei der Topologie 784-100-50-10 ca. 17,5 mal geringer und dennoch ist die Korrektklassifikationsrate nur um 0,77 % gesunken. Zu den Ergebnissen anderer Publikationen wurden zwar keine Angaben über die Anzahl der benötigten Iterationen der Optimierungsverfahren gemacht, allerdings wird eine so starke Parameterreduktion sehr wahrscheinlich die Trainingsdauer reduzieren. Die Topologie 784-100-50-10 ist allerdings nicht optimal. Die Anzahl der Knoten in der ersten versteckten Schicht sollte etwas höher sein, um das auftretende, leichte Overfitting zu vermeiden.

Die besten Ergebnisse auf diesem Datensatz mit und ohne Vergrößerung des Trainingsdatensatzes durch Verzerrungen konnten allerdings mit Convolutional Neural Networks erzielt werden. Das beste Ergebnis ohne Verzerrungen wurde von Jarrett u. a. [40] erreicht und liegt bei 99,47 % Korrektklassifikationsrate auf den Testdaten. Hierbei wurde allerdings nicht nur mit Backpropagation und Stochastic Gradient Descent trainiert, sondern ein unüberwachtes Training der ersten Schichten durchgeführt, sodass diese bessere Merkmale vom Originalbild extrahieren.

Um zu prüfen, ob CNNs ein ähnliches Ergebnis mit dem hier verwendeten Optimierungsverfahren erreichen können, habe ich ein Convolutional Neural Network implementiert und dieses zum Lernen des MNIST-Datensatzes eingesetzt. Ich habe dabei die Architektur von Simard u. a. [79] übernommen: die Eingabe wird auf die Größe 29x29 erweitert, die erste Schicht ist ein Convolutional Layer mit sechs sogenannten Feature Maps und einem trainierbaren Filter der Größe 5x5 und einem Bias, die zweite Schicht ist wieder ein Convolutional Layer mit 50 Feature Maps und ebenfalls einem Filter der Größe 5x5 und einem Bias, darauf folgt eine vollständig verbundene Schicht mit 100 Neuronen und einem Bias und die Ausgabeschicht ist genauso wie bei den hier verwendeten MLPs aufgebaut. Die sechs Feature Maps in der ersten Schicht haben jeweils 13x13 Komponenten, die durch Anwendung der zugehörigen Filter auf die Eingabe und anschließende Transformation durch eine Aktivierungsfunktion (hier: Tangens Hyperbolicus) entstehen. Dabei werden die Filter immer um 2 Pixel verschoben. Die 50 Feature Maps in der zweiten Schicht bestehen aus je 5x5 Komponenten und für jede Feature Map aus der vorherigen Schicht gibt es einen anderen Filter. Die Ausgaben der Filter werden dabei addiert. Das CNN hat insgesamt nur 134.066

Abbildung 5.5.: Ausgaben der Neuronen eines CNN. Das hier dargestellte neuronale Netz besteht aus zwei Convolutional Layers, einer vollständig verbundenen Schicht und einer Ausgabeschicht, die ebenfalls vollständig mit der vorherigen verbunden ist. Die hier vorhergesagte, wahrscheinlichste Klasse stimmt bei dieser Instanz nicht mit der tatsächlichen Klasse überein und ist hellrot dargestellt. Die zweitwahrscheinlichste Klasse ist die echte Klasse und ist dunkelgrün dargestellt.

Gewichte. Diese Anzahl ist so gering, da alle Komponenten einer Feature Map sich die Gewichte der Filter teilen. Diese Methode zur Reduktion der Gewichte kann auch als eine Art der Gewichtskomprimierung angesehen werden.

In Abbildung 5.5 ist der Aufbau des verwendeten CNN zu erkennen. Dargestellt wurden hier die Ausgaben der einzelnen Neuronen des neuronalen Netzes nach der Vorwärtspropagation. Die einzelnen Feature Maps wurden hier zweidimensional gezeichnet. Das Originalbild ist zumindest in der ersten Schicht noch gut erkennbar, sodass eine einzelne Feature Map hier auch als Komprimierung des Originalbildes angesehen werden kann. In der ersten Schicht gibt es insgesamt 1014 Neuronen, in der zweiten 1250 und in der dritten 100. Ohne Verzerrungen und Vorverarbeitung habe ich mit diesem CNN eine Korrektklassifikationsrate von 98,85 % erreicht. Es wurden dabei nur 14 Iterationen bis zur besten Korrektklassifikationsrate benötigt. Das Ergebnis ist besser als bei normalen MLPs, kommt allerdings nicht an die Ergebnisse von LeCun u. a. [52] (99,05 %) oder Jarrett u. a. [40] (99,47 %) heran. Jarrett u. a. [40] haben, wie bereits erwähnt, ein komplizierteres Lernverfahren verwendet und LeCun u. a. [52] haben eine spezialisiertere Architektur verwendet. Simard u. a. [79] geben kein Ergeb-

nis ohne Verzerrungen an. Für Bilder sind CNNs in der Regel besser geeignet. Das Training ist hier viel schneller und die Korrektklassifikationsrate auf den Testdaten ist höher.

5.2.4. Erkenntnisse

Eine Komprimierung der Gewichte kann die Generalisierungsfähigkeit gewöhnlicher MLPs auf Bilddatensätzen steigern und die Anzahl der zur Optimierung benötigten Parameter stark reduzieren.

Gerade für Bilder gibt es mit CNNs allerdings schon lange untersuchte, schnelle und robuste Verfahren, die eine spezialisiertere und besser geeignetere Form der Parameterreduktion implementieren.

5.3. P300-Speller

5.3.1. Datensatz

Ich werde in diesem Abschnitt eine Methode zum Buchstabieren mit Hilfe eines Brain-Computer Interfaces (BCI) untersuchen. Das zugrunde liegende Verfahren wurde durch Farwell und Donchin [30] und durch Donchin u. a. [24] beschrieben. Es werden an verschiedenen Stellen des Kopfes einer Person Hirnspannungen gemessen. Dies wird als Elektroenzephalografie (EEG) bezeichnet. Währenddessen soll sich die Person auf einen Buchstaben konzentrieren und ihr werden in definierten Abständen in zufälliger Reihenfolge Zeilen und Spalten einer Buchstabenmatrix (siehe Abbildung 5.6 (a)) gezeigt. Wenn die Spalte oder Zeile, in der sich der aktuelle Buchstabe befindet, aufleuchtet, ist bei den meisten Personen ein sogenanntes P300-Potenzial messbar. Das ist eine Komponente eines ereigniskorrelierten Potenzials (EKP) oder Event-Related Potential (ERP). Der Name ist darauf zurückzuführen, dass diese etwa 300 ms nach dem Reiz messbar ist. Das zugrunde liegende Prinzip wird als *Oddball Paradigm* bezeichnet. Dieses Prinzip besagt, dass seltene Ereignisse in einer Reihe von Ereignissen dieses P300-Potenzial im menschlichen Hirn auslösen. In diesem Fall sind die seltenen Ereignisse die Beleuchtungen der Zeile und Spalte, in denen der richtige Buchstabe vorkommt, im Gegensatz zu allen anderen Zeilen- oder Spaltenbeleuchtungen. Um den richtigen Buchstaben zu bestimmen, müssen die P300-Potenziale erkannt werden. Der Buchstabe, der in der Reihe und der Spalte, die die P300-Potenziale ausgelöst haben, vorkommt, ist der, auf den sich die Person konzentriert hat. Da das Rauschen in den gemessenen Hirnspannungen sehr groß ist, werden für jeden einzelnen Buchstaben mehrmals alle Zeilen und Spalten beleuchtet, wodurch das Rauschen herausgerechnet werden kann.

Im Rahmen der BCI Competition III [12] wurden mehrere Datensätze von verschiedenen BCIs zur Verfügung gestellt, anhand derer verschiedene Methoden zur Signalverarbeitung und Klassifikation verglichen werden sollten. Der Wettbewerb wurde bereits im Jahr 2005 beendet. Der hier verwendete Datensatz II [47] erfordert die Unterscheidung von 36 verschiedenen Zeichenklassen. Diese sind in Abbildung 5.6 (a) zu sehen.

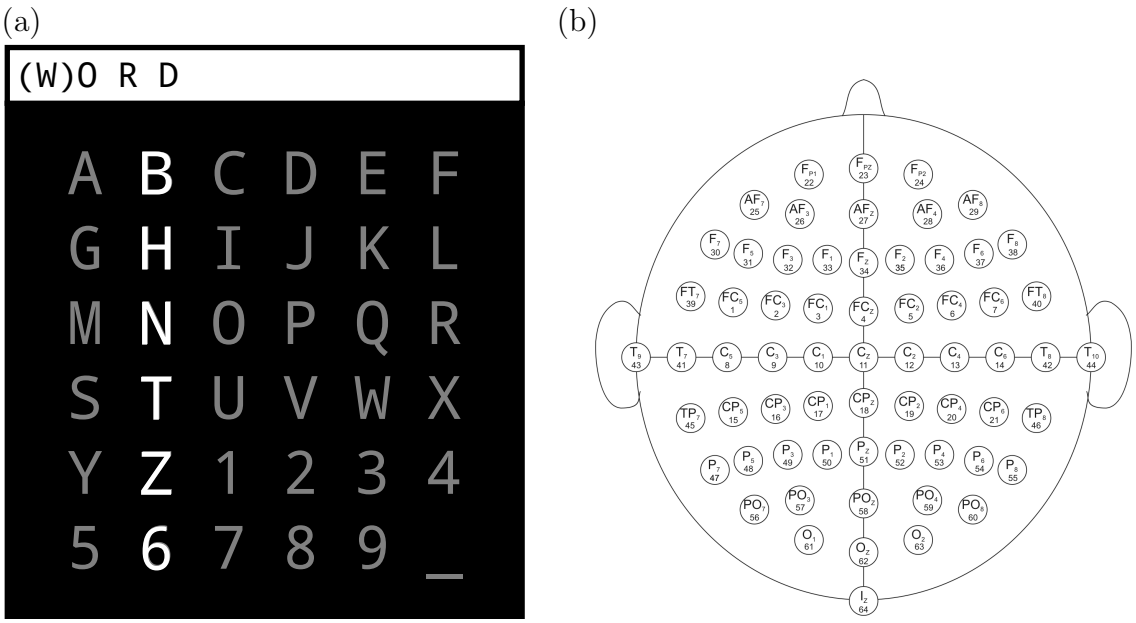


Abbildung 5.6.: (a) Buchstabenmatrix. Das zu buchstabierende Wort ist oben zu sehen. Es werden in zufälliger Reihenfolge alle 6 Spalten und 6 Zeilen beleuchtet. (b) Kanalbelegung (Zeichnung übernommen von Krusienski und Schalk [47]). Es werden an 64 verschiedenen Stellen Hirnspannungen gemessen.

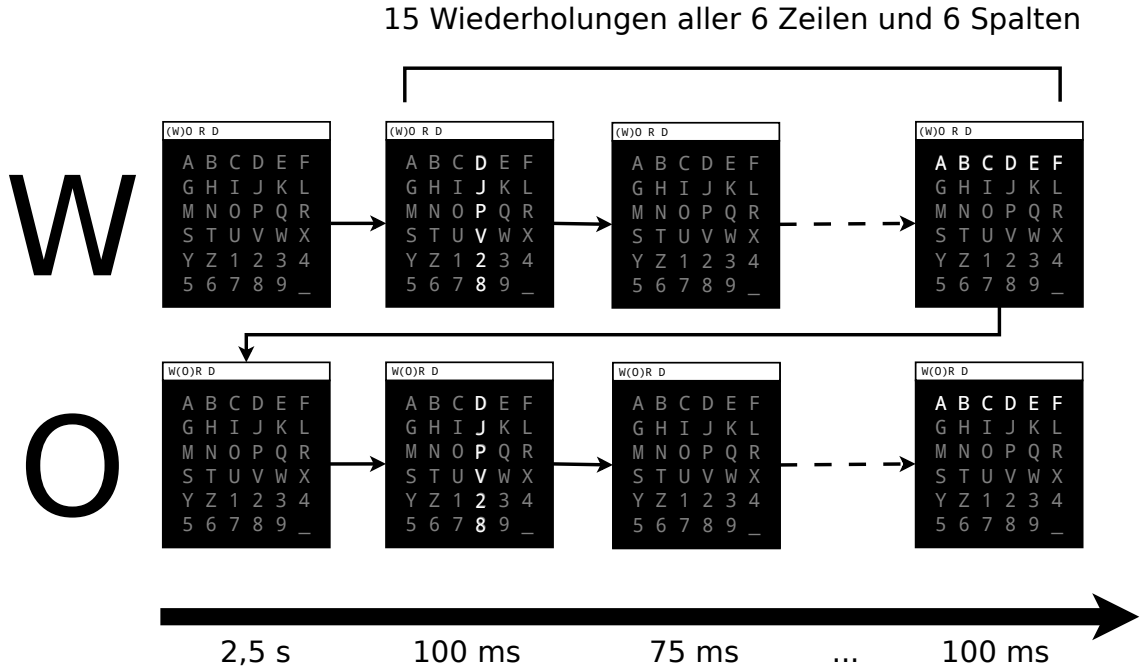
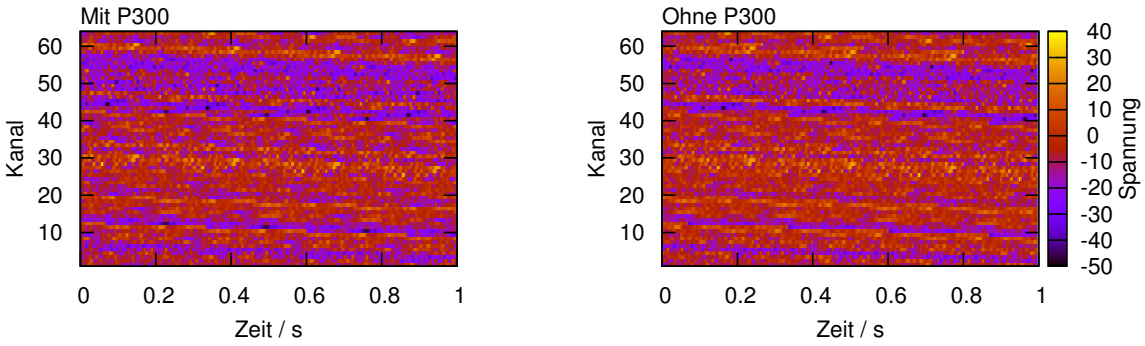


Abbildung 5.7.: Zeitlicher Verlauf der Bildschirmanzeige während der Aufzeichnung der Daten.

(a)



(b)

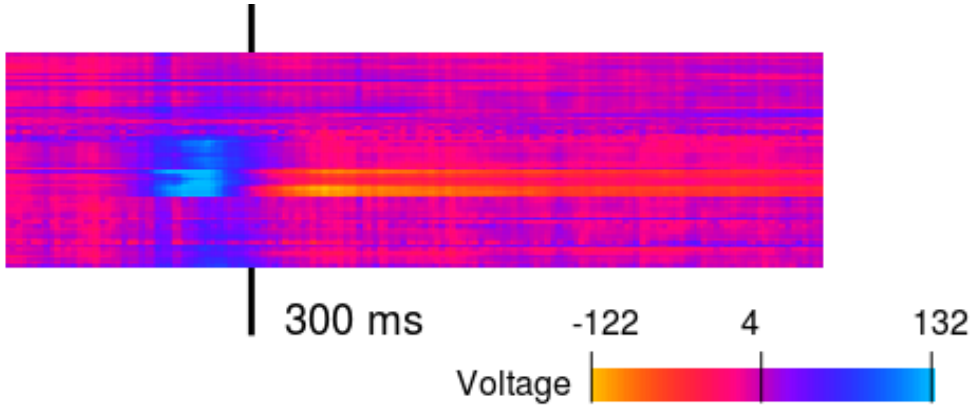


Abbildung 5.8.: (a) Beispiele für Trainingsdaten mit und ohne P300-Potenzial. (b) EEG-Signal zwischen 0 und 1 s nach dem Reiz. Hierbei werden die Zeit durch die x-Achse, der Kanal durch die y-Achse und die Spannung durch die Farbe angegeben. Auf den mittleren Kanälen sind sehr starke Schwankungen zu erkennen. Es handelt sich dabei nicht um ein P300-Potenzial. Die Einheit der Spannung wird nicht in der Beschreibung des Datensatzes angegeben. Die Angaben sind aber vermutlich in Mikrovolt.

Die aufgenommenen Daten stammen von zwei verschiedenen Personen. Es wurden pro Person Trainingsdaten für 85 Buchstaben und Testdaten für 100 Buchstaben aufgenommen. Es wurden Gehirnaktivitäten an 64 verschiedenen Stellen des Kopfes gemessen (siehe Abbildung 5.6 (b)).

Für jeden Buchstaben wurden in einer sogenannten Buchstaben-Epoche 15 mal die zwölf verschiedenen Zeilen und Spalten in zufälliger Reihenfolge beleuchtet. Vor und nach jeder Epoche wurde die komplette Matrix für 2,5 s nicht beleuchtet. Die Beleuchtung einer Zeile oder Spalte dauerte jeweils 100 ms und darauf folgte eine Pause von 75 ms. In der Pause am Ende einer Buchstaben-Epoche konnte sich die Person auf den nächsten Buchstaben konzentrieren. Der aktuelle Buchstabe wurde jeweils durch Klammern über der Buchstabenmatrix markiert. In Abbildung 5.7 ist diese Abfolge zusammengefasst.

Während des gesamten Experiments wurden die Hirnspannungen mit den Frequenzen 0,1 bis 60 Hz von 64 Kanälen mit 240 Hz aufgezeichnet. Die zur Verfügung gestellten Daten enthalten zu jeder Buchstaben-Epoche jeweils die gemessenen Spannungen in jedem Kanal und die angezeigten Zeilen oder Spalten über den gesamten Zeitraum der Epoche. Die Reihenfolge der Buchstaben während der Aufzeichnung ist in den Trainings- und Testdaten nicht bekannt.

Vorhergesagt werden muss zu jeder angezeigten Zeile oder Spalte, ob diese den Zielbuchstaben enthält. Dabei ist klar, dass jeweils eine Spalte und eine Zeile den aktuellen Buchstaben enthalten. Durch die Kombination von der Zeile und der Spalte, in denen das P300-Potenzial erkannt wurde, kann der Zielbuchstabe ermittelt werden.

Das Klassifikationsproblem ist aus folgenden Gründen sehr schwer:

- Durch Zufall wird nur eine Korrektklassifikationsrate von $\frac{1}{36} \approx 2{,}8\ \%$ erreicht.
- Die Daten sind stark verrauscht, sodass Overfitting ein Problem darstellt und eine Unterscheidung vielen Menschen kaum möglich ist (siehe Abbildung 5.8 (a)).
- Aufeinander folgende Trainingsinstanzen überschneiden sich, sodass in einer Trainingsinstanz ohne P300 das P300-Potenzial einer folgenden Instanz erkennbar sein kann. Die Lokalisierung muss also genau sein.
- Teilweise unterliegen die EEG-Daten starken Schwankungen, die nicht in Zusammenhang mit den P300-Potenzialen stehen (siehe Abbildung 5.8 (b)).

5.3.2. Vorbereitender Test

Um zu zeigen, dass bei einem stark verrauschten Signal, wie zum Beispiel dem EEG-Signal, die Nutzung einer hohen Abtastrate das Erkennen schwacher Signale durch ein MLP oder ein lineares Modell ermöglicht, wird vorab ein leichteres Experiment mit künstlich generierten Daten vorgestellt.

In diesem Test setzen sich die generierten Daten aus einem Signal $sin(2\pi x - x_0)$, das zufällig um $x_0 \in [0; 0,2]$ verschoben wurde, möglicherweise einem künstlichen P300-Potenzial $\exp \frac{-(2\pi x-5)^2}{0,5}$ und gleichverteiltem Rauschen aus dem Intervall $[-5; 5]$ zusammen. Das Rauschen ist hier bewusst größer gewählt als der Ausschlag durch das P300-Potenzial und die Amplitude des Signals. Die künstlichen P300-Potenziale sind hier nicht an der selben Stelle wie echte P300-Potenziale. Für das hier eingesetzte Klassifikationsverfahren macht dies keinen Unterschied.

Ein MLP soll lernen, zwischen Signalen mit P300 und ohne P300 zu unterscheiden. Für verschiedene Abtastraten, die an 100 bis 500 Stellen das Signal im Intervall $[0, 1]$ messen, wurden jeweils 1000 Test- und Trainingsdaten generiert. In Abbildung 5.9 (a) und (b) sind die einzelnen und kombinierten Komponenten der Trainingsdaten dargestellt und zwei Trainingsinstanzen mit Rauschen aus unterschiedlichen Klassen (mit P300 und ohne P300) zu sehen.

Ich vergleiche, wie gut ein MLP mit verschiedenen Abtastraten des Signals zurechtkommt und welchen Einfluss eine Komprimierung der Gewichte bei der höchsten Abtastrate hat. Die Gewichte eines MLP ohne versteckte Schicht mit 500 unkomprimierten Gewichten nach dem Training sind in Abbildung 5.9 (c) zu sehen. Ich

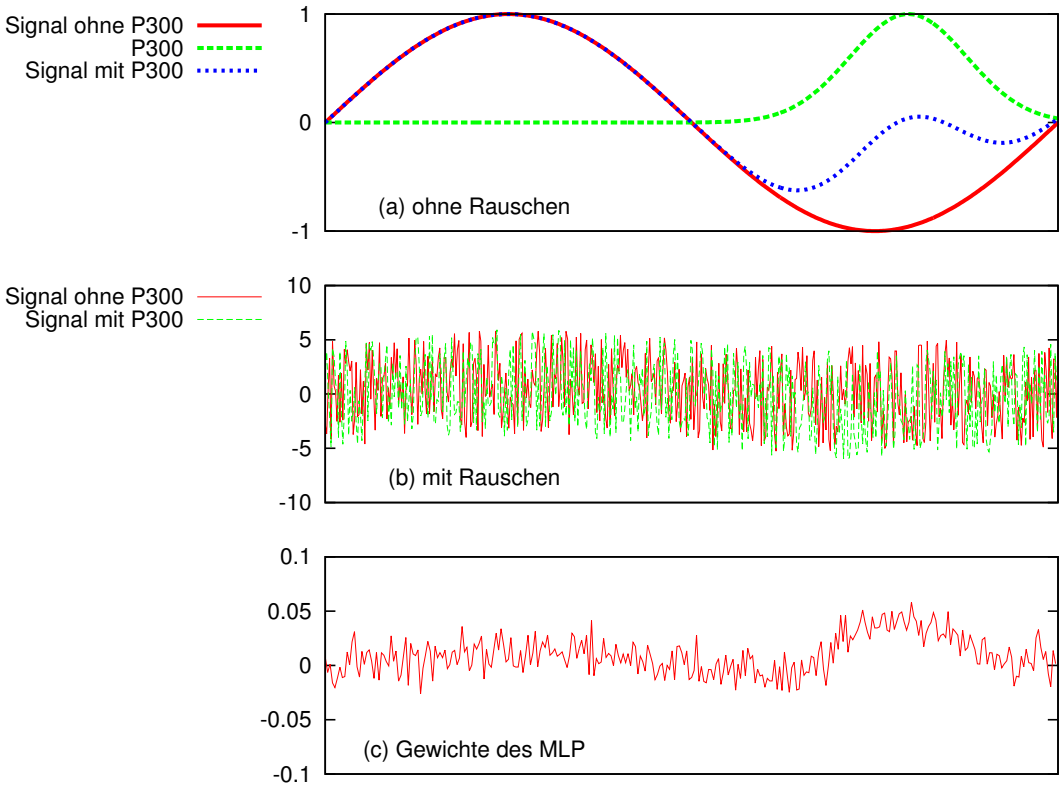


Abbildung 5.9.: Künstlich generiertes P300-Potenzial (a) ohne Rauschen und (b) mit Rauschen (abgetastet an 500 Stellen) und (c) die erlernten Gewichte eines unkomprimierten MLP mit 500 Gewichten.

vermute, dass die Gewichte des neuronalen Netzes durch eine Reduktion der Komprimierungsparameter weniger anfällig für das Rauschen in den Trainingsdaten sind, da eine geglättete Gewichtsfunktion entsteht. Demzufolge könnte der Fehler auf den Trainingsdaten zwar steigen, wenn die Anzahl der Parameter reduziert wird, allerdings sollte der Fehler auf den Testdaten geringer werden. Des Weiteren vermute ich, dass durch eine geringere Abtastrate eine Klassifikation erschwert wird, da das Rauschen schwieriger herausgerechnet werden kann.

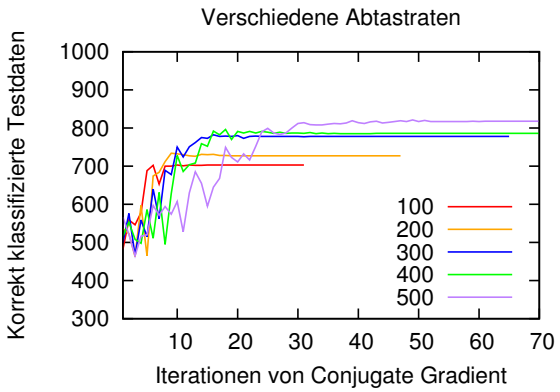
Um diese Vermutungen zu überprüfen, wurden Trainings- und Testdatensätze mit 100, 200, 300, 400 und 500 Abtastungen generiert. Jeder Datensatz enthält 1000 Instanzen, wobei auf jede Instanz mit der Wahrscheinlichkeit 0,5 ein simuliertes P300-Potenzial addiert wurde. Zunächst wurden mit diesen Trainingsdatensätzen unkomprimierte Single Layer Perceptrons (SLP, ein MLP ohne versteckte Schicht) ohne Bias trainiert. Danach wurden auf dem Trainingsdatensatz mit 500 Abtastungen komprimierte SLPs mit 100, 200, 300, 400 und 500 Parametern trainiert. Dazu wurden maximal 100 Iterationen des Optimierungsalgorithmus Conjugate Gradient durchgeführt.

In Tabelle 5.5 ist das vermutete Ergebnis abzulesen. Mit steigender Anzahl der Abtastungen sinken Trainings- und Testfehler, allerdings findet ein leichtes Overfitting statt. Durch die Reduktion der Parameter hingegen wird das Rauschen der Trainings-

unkomprimierte SLPs, variable Abtastungen					
Abtastungen	100	200	300	400	500
Iterationen	**31**	47	65	79	100
Trainingsfehler (SSE)	327	230	171	120	**92**
Korrekte klassifizierte Trainingsdaten	770	857	910	970	**985**
Testfehler (SSE)	406	370	337	348	342
Korrekte klassifizierte Testdaten	703	727	778	786	818
komprimierte SLPs, variable Parameteranzahl					
Parameter	100	200	300	400	500
Iterationen	57	77	100	100	100
Trainingsfehler (SSE)	162	141	123	110	**92**
Korrekte klassifizierte Trainingsdaten	916	942	956	973	**985**
Testfehler (SSE)	**195**	222	259	294	342
Korrekte klassifizierte Testdaten	**890**	873	845	833	818

Tabelle 5.5.: Vergleich verschiedener Abtastraten und Komprimierungen. Es werden die auf ganze Zahlen gerundeten Werte der letzten Iteration angegeben.

(a)



(b)

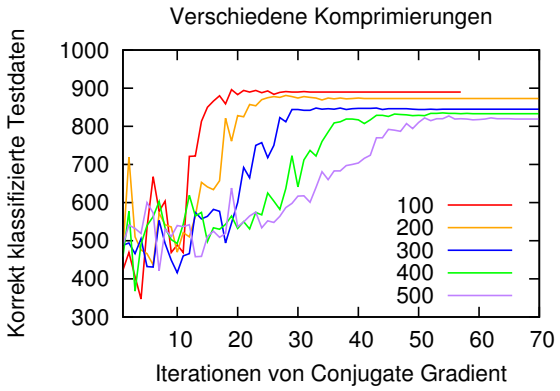


Abbildung 5.10.: Anzahl der korrekt klassifizierten Testdaten im Verlauf der Optimierung bei (a) verschiedenen Abtastraten und (b) verschiedenen Parameteranzahlen mit 500 Abtastungen des Signals.

daten weniger mitgelernt. Dadurch steigt zwar der Trainingsfehler wieder an, aber der Testfehler sinkt weiter.

Ein Nachteil der Komprimierung ist hier allerdings, dass der Optimierungsalgorithmus durch die Komprimierung mehr Iterationen bis zur Konvergenz benötigt. Mit der Anzahl der Parameter nimmt allerdings auch dieser Effekt ab, wie in Abbildung 5.10 zu erkennen ist. Zu erklären ist dies mit der stärkeren Korrelation der Parameter durch die indirekte Gewichtsrepräsentation. Ein Parameter beeinflusst nicht nur ein Gewicht, wie dies bei einem normalen MLP der Fall ist, sondern potenziell alle Gewichte. Ich vermute allerdings, dass die Lerngeschwindigkeit bei starker Komprimierung auch die eines normalen MLP unterschreitet. Tatsächlich lässt sich bei diesem Problem die Anzahl der Parameter auf 15 reduzieren und in diesem Fall wird mit 87,5 % in 10 Iterationen schneller eine höhere Korrektklassifikationsrate auf den Testdaten erreicht als bei dem unkomprimierten SLP mit 500 Abtastungen.

Da hier das Experiment nur mit künstlichen Daten durchgeführt wurde, ist dies zwar ein Hinweis darauf, dass eine Gewichtskomprimierung gut für reale EEG-Daten geeignet ist, aber noch keine Garantie.

5.3.3. Methode

Die Erkennung von Buchstaben erfordert ein Klassifikationsverfahren. Dabei muss aus den Spannungen der 64 Kanäle in dem Zeitraum vom Einsetzen des Reizes bis zu einer Sekunde danach abgeleitet werden, ob der gesuchte Buchstabe in der aktuellen Zeile oder Spalte ist. Das neuronale Netz erhält also

$$D = \overbrace{64}^{\text{Kanäle}} \cdot \underbrace{1\ \text{s}}_{\text{Dauer}} \cdot \overbrace{240\ \text{Hz}}^{\text{Abtastrate}} = 15.360 \tag{5.3.5}$$

Eingaben. Die Dimension der Ausgabe ist $F = 1$. Die Größe des Trainingsdatensatzes ist

$$N = \overbrace{85}^{\text{Zeichen}} \cdot \underbrace{15}_{\text{Wiederholungen}} \cdot \overbrace{12}^{\text{Zeilen und Spalten}} = 15.300 \tag{5.3.6}$$

und die des Testdatensatzes entsprechend 18.000. Die Aktivierungsfunktion in der Ausgabeschicht des neuronalen Netzes ist Tangens Hyperbolicus, die Ausgabe liegt also in $[-1, 1]$, wobei Werte unter null für „kein P300“ und über null für „P300“ stehen. Nachdem die 180 Instanzen für die Zeilen- und Spaltenanzeigen der einzelnen Buchstabenepochen klassifiziert wurden, kann durch Summieren der Ausgaben des neuronalen Netzes für jede Spalte und jede Zeile ein Punktwert errechnet werden. Dieser gibt an, wie wahrscheinlich es ist, dass der gesuchte Buchstabe in der entsprechenden Zeile oder Spalte vorkommt. Durch Kombination der Zeile und Spalte mit dem höchsten Punktwert kann das gesuchte Zeichen bestimmt werden. Selbst wenn die Korrektklassifikationsrate für einzelne P300-Potenziale niedrig ist, kann die Korrektklassifikationsrate für Zeichen durch die 15 Wiederholungen gut sein. Im Folgenden werden die Wiederholungen auch als Trials bezeichnet.

Ein Single Layer Perceptron (SLP) ist zur Klassifikation der P300 ausreichend. Die besten Ergebnisse können mit dem Optimierungsalgorithmus LMA erreicht werden.

Eine Vorverarbeitung der Daten bringt Vorteile. Hier wurden zwei verschiedene Arten der Vorverarbeitung zugelassen:

1. Anwendung eines Tiefpassfilters (*Finite Impulse Response*, FIR) 31. Ordnung mit einem Hamming-Fenster und einer Grenzfrequenz von 10 Hz.

   Durch eine Glättung der Daten verringert der Tiefpassfilter das Rauschen.

2. Unterabtastung (englisch: *Downsampling*) mit dem Konvertierungsfaktor 11.

   Nach der Anwendung des Filters, kann mit geringem Informationsverlust eine Unterabtastung durchgeführt werden, da in einem Kanal aufeinander folgende Daten sich wenig unterscheiden. Die Kombination beider Verfahren wird als *Decimation* bezeichnet. Nach der Unterabtastung hat eine Instanz 1344 Komponenten.

Die Komprimierung wurde bei den Experimenten mit Hilfe der CUBLAS-Bibliothek [64] auf der GPU implementiert, wodurch die benötigte Zeit für die Komprimierung mehr als halbiert wurde.

5.3.4. Ergebnisse

Zur Bestimmung einer geeigneten Komprimierung wurden einige Versuche mit genau 15 Iterationen des Optimierungsverfahrens durchgeführt. Dabei wurden die SSE auf Trainings- und Testdaten, die Korrektklassifikationsrate auf Trainings- und Testdaten und die Laufzeit aufgezeichnet (siehe Abbildung 5.11). Es wird zunächst keine Vorverarbeitung des Trainingsdatensatzes durchgeführt.

Anhand der Synchronität des Verlaufes der Fehler auf den Trainings- und den Testdaten kann man erkennen, dass ein SLP gut generalisiert. Für Komprimierungen mit mehr als 1600 Parametern reichen die 15 Iterationen von LMA nicht aus, um das SLP zu trainieren. Allerdings ist die Trainingsdauer bereits mit 3200 Parametern und 15 Iterationen länger als 25 Minuten. Bei den optimalen, hier gefundenen Komprimierungen dauert das Training weniger als fünf Minuten. Die Dauer des Trainings eines unkomprimierten Netzes bis zur Konvergenz, das heißt bis sich der SSE zwischen zwei Iterationen nicht mehr ändert, liegt bei ungefähr 24 Stunden. Dabei wird allerdings auch nur eine Korrektklassifikationsrate von 20 % auf den Testdaten erreicht.

In den darauf folgenden Experimenten wurde die Optimierung abgebrochen, wenn eine der folgenden Bedingungen erfüllt war:

$$(1) \quad |E_{SSE}^{(t)} - E_{SSE}^{(t-1)}| \leq 0,001 \max\{E_{SSE}^{(t-1)}, E_{SSE}^{(t)}, 1\}, \tag{5.3.7}$$

$$(2) \quad t \geq 20, \tag{5.3.8}$$

wobei $t \geq 1$ die Iteration angibt und $E_{SSE}^{(t)}$ den Fehler in Iteration t.

Die fünf besten Ergebnisse aus dem Wettbewerb und die besten hier erzielten Ergebnisse sind in Tabelle 5.6 zu sehen. Die Qualität des Verfahrens wird hierbei gemessen in der Korrektklassifikationsrate auf den Buchstaben mit 15 Trials. Zudem wurden Werte für 5 Trials angegeben. Beide Werte sind über zwei Probanden gemittelt.

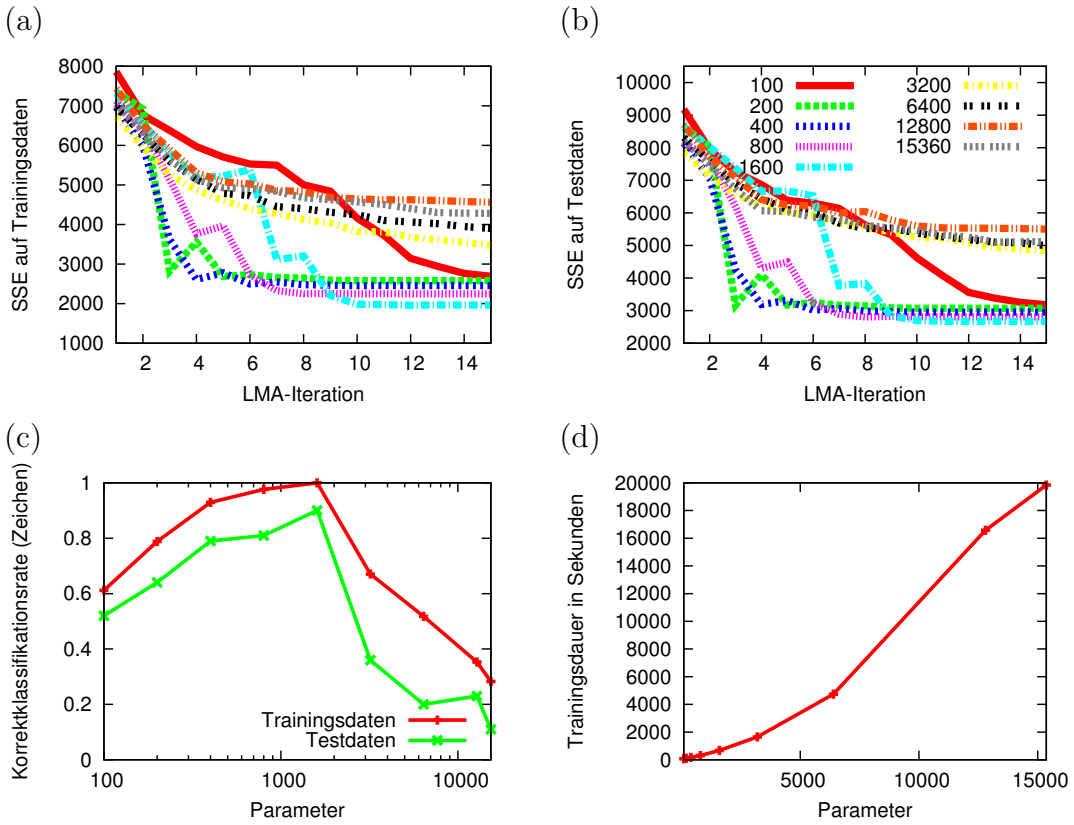


Abbildung 5.11.: SSE auf den (a) Trainingsdaten und (b) Testdaten im Optimierungsverlauf. (c) Korrektklassifikationsrate und (d) Trainingsdauer (Rechnerkonfiguration 3, Anhang F) nach 15 Iterationen mit verschiedenen Komprimierungen.

Korrektklassifikationsrate 15 Trials	5 Trials	Klassifikation	Vorverarbeitung
BCI Competition III			
96,5 %	**73,5 %**	Ensemble von SVMs	Bandbreitenfilter, Downsampling, Kanal-Auswahl
90,5 %	55,0 %	Bagging mit SVMs	Bandbreitenfilter, Entfernung von Augenbewegungsartefakten, Downsampling, Kanal-Auswahl
90 %	59,5 %		Filter, Downsampling, Detrending, PCA, t-Statistik
89,5 %	53,5 %	Gradient Boosting	Detrending, Decimation
87,5 %	57,5 %	Bagging mit LDA	Bandbreitenfilter, Kanal-Auswahl, Downsampling
SLPs			
97 %	59,5 %	SLP (801 Parameter)	Decimation
96 %	58,5 %	SLP (801 Parameter)	Tiefpassfilter
94 %	53 %	SLP (unkomprimiert)	Decimation
88 %	45 %	SLP (1201 Parameter)	-

Tabelle 5.6.: Beste Ergebnisse der BCI Competition III [12] und die besten hier erzielten Ergebnisse.

Auffällig ist, dass bei den meisten Ergebnissen aus dem Wettbewerb wesentlich mehr Vorverarbeitungsmethoden angewandt wurden. Vier der ersten fünf Verfahren nutzen zudem eine Kombination aus mehreren Klassifikatoren.

Komplett ohne Vorverarbeitung, aber mit der Komprimierung der Parameter konnte hier bereits eine Korrektklassifikationsrate von 88 % erreicht werden. Mit Hilfe des Tiefpassfilters kann die Genauigkeit deutlich gesteigert werden, sodass das beste Ergebnis des Wettbewerbs von Rakotomamonjy und Guigue [67] fast erreicht wird. Mit anschließendem Downsampling wird das Ergebnis sogar übertroffen.

Mit 5 Trials konnte hier allerdings maximal eine Korrektklassifikationsrate von 59,5 % erreicht werden. Damit wurde nur das zweitbeste Ergebnis des Wettbewerbs erreicht. Mit nur einem Trial fällt die Korrektklassifikationsrate sogar unter 20 %.

Die Trainingsdauer, inklusive Einlesen und Komprimierung des Trainings- und Testdatensatzes, liegt bei den vier getesteten Varianten zwischen zwei und drei Minuten. Das ist eine enorme Beschleunigung gegenüber einem unkomprimierten SLP. Bei den Verfahren aus dem Wettbewerb liegen keine Informationen über die Lerngeschwindigkeit vor.

5.3.5. Erkenntnisse

Die hier betrachteten EEG-Daten weisen zwei wichtige Eigenschaften auf, die den Einsatz der hier entwickelten Methode begünstigen: sie sind komprimierbar und sie sind stark verrauscht.

Die Komprimierung der Parameter eines SLP steigert die Korrektklassifikationsrate bei EEG-Daten enorm. Das Rauschen kann hierbei durch die implizite Akkumulation der Eingabekomponenten herausgerechnet werden.

Bei einer so starken Reduktion der Parameter, wie sie hier möglich war, können aufwändige Optimierungsalgorithmen wie LMA eingesetzt werden und es werden trotz indirekter Gewichtsrepräsentation weniger Iterationen bis zur Konvergenz benötigt.

Vorverarbeitungsmethoden wie Tiefpassfilter und Unterabtastung haben ähnliche Effekte und können auch mit dem hier entwickelten Verfahren kombiniert werden.

LMA hat sich als besonders geeignet für die Klassifikation von P300-Potenzialen herausgestellt. Mit allen anderen in dieser Arbeit vorgestellten Optimierungsalgorithmen wurden viel schlechtere Ergebnisse erreicht.

5.4. Single-Trial-P300-Erkennung

5.4.1. Datensatz

Ebenso wie in Abschnitt 5.3 werde ich hier ein SLP zur Klassifikation von P300-Potenzialen einsetzen. Im Gegensatz dazu werden allerdings nicht öffentlich zugängliche Daten genutzt. Die Daten stammen von dem Projekt IMMI (Intelligentes Mensch-Maschine-Interface) des DFKI (Deutsches Forschungszentrum für künstliche Intelligenz) Robotics Innovation Center und der Universität Bremen.

In diesen Datensätzen müssen die P300-Potenziale ohne mehrfache Wiederholung erkannt werden. Den Probanden wurden dabei zwei Arten von visuellen Reizen gezeigt:

sogenannte „Standards“ und „Targets“. Das Verhältnis von Standards zu Targets beträgt 8:1. Die Targets sollen P300-Potenziale auslösen. Insgesamt stehen hier von acht Probanden jeweils Datensätze aus drei Aufnahmen zur Verfügung. Dabei werde ich pro Proband die ersten beiden Datensätze für das Training und den letzten Datensatz zum Testen verwenden. Die Aufnahme der EEG-Daten erfolgte über ein actiCap-System der Firma Brain Products GmbH. Das System tastet 136 Kanäle mit 5000 Hz ab. Insgesamt wurden hier 62 Kanäle zur Klassifikation verwendet. Eine Instanz besteht also eigentlich aus 310.000 Komponenten. Allerdings werde ich hier einige Vorverarbeitungsmethoden, die in dem Projekt verwendet werden, übernehmen:

1. Die Daten eines Kanals wurden jeweils standardisiert, das heißt es wurden der Mittelwert abgezogen und auf die Standardabweichung 1 normalisiert.

2. Es wurde eine Decimation auf 25 Hz durchgeführt.

3. Anschließend wurde ein Tiefpassfilter mit der Grenzfrequenz 4 Hz angewandt.

Somit besteht eine Instanz nur noch aus 1550 Komponenten.

Die besonderen Schwierigkeiten der Datensätze sind die geringe Anzahl von Trainingsdaten (weniger als 1000) im Vergleich zur Dimension der Eingaben und die ungleiche Verteilung der Beispiele für die beiden Klassen.

5.4.2. Methode

Ich werde wieder ein SLP mit unterschiedlichen Komprimierungsstufen zur Klassifikation verwenden. Die Aktivierungsfunktion ist erneut Tangens Hyperbolicus. Targets werden durch 1 und Standards durch -1 dargestellt. Die Summe der quadratischen Fehler wird durch LMA minimiert. Die Matrix Φ zur Komprimierung der Gewichte wird durch die Verteilung aus Gleichung 3.5.56 generiert.

Zum direkten Vergleich werde ich auch Support Vector Machines trainieren. Dieses Klassifikationsverfahren wird in dem Projekt IMMI üblicherweise verwendet. Ich werde dabei die Eingabe durch eine auf die gleiche Art generierte Matrix Φ komprimieren. Ich verwende die Bibliothek LIBSVM [21]. Der Kernel ist linear und der Parameter C wurde für jeden Probanden separat aus der Menge $\{10^x | x \in \{-5, \ldots, 5\}\}$ bestimmt. Normalerweise gehört die Zeit zur Bestimmung von C und anderen Metaparametern eines Lernverfahrens zur Trainingsdauer. Das ist in dem folgenden Vergleich allerdings nicht der Fall. Dadurch würde der Vorteil von SVMs in der Trainingsdauer verloren gehen.

Des Weiteren habe ich die im Projekt IMMI entwickelte Referenzmethode zum Vergleich herangezogen. Dabei werden neben den bereits erwähnten Vorverarbeitungsmethoden folgende Schritte ausgeführt:

1. Anwendung von xDAWN [71], einem Verfahren das unüberwacht den von den P300-Potenzialen am meisten betroffenen Unterraum der Daten extrahiert.

2. Kleine Abschnitte der Daten werden durch Geraden lokal approximiert. Die Steigungen der Geraden werden als Merkmal extrahiert.

3. Die Merkmale werden normalisiert. Es wird also wieder der Mittelwert abgezogen und durch die Standardabweichung geteilt.

4. Der optimale Metaparameter C der SVMs wird durch Kreuzvalidierung bestimmt.

5. Die Klassifikation erfolgt durch SVMs. Instanzen verschiedener Klassen werden unterschiedlich gewichtet: Standards werden mit 1 und Targets mit 5 gewichtet, sodass die falsche Klassifikation von Targets eher vermieden wird.

Alle drei hier untersuchten Varianten sind in Abbildung 5.12 (a) skizziert.

Ein geeignetes Maß für die Bewertung der Qualität eines Klassifikators auf einem Datensatz mit zwei ungleich verteilten Klassen ist die *balancierte Korrektklassifikationsrate* (Balanced Accuracy [17]), die durch

$$\text{Balanced Accuracy} = \frac{\frac{TP}{TP+FN} + \frac{TN}{TN+FP}}{2} \tag{5.4.9}$$

berechnet wird, wobei TP True Positives, FN False Negatives, TN True Negatives und FP False Positives bedeuten. Das Maß garantiert, dass auch bei einer ungleichen Verteilung der Klassen ein Klassifikator, der immer dieselbe Klasse vorhersagt, die schlechteste mögliche Bewertung von 0,5 erhält. Dies wäre bei der normalen Korrektklassifikationsrate nicht der Fall.

5.4.3. Ergebnisse

Mit 20 verschiedenen Komprimierungsstufen wurden jeweils zehn Experimente pro Proband durchgeführt. Dabei variieren zwischen den Experimenten nur die Komprimierungsmatrix Φ und die Initialisierung der Parameter der SLPs. Gemessen wurden die Trainingsdauer und die balancierte Korrektklassifikationsrate. Die Ergebnisse sind in Abbildung 5.12 (b) zusammengefasst. Auf der x-Achse ist für SLPs die Anzahl der zu optimierenden Parameter und für SVMs die Anzahl der Komponenten einer Instanz nach der Komprimierung angegeben. Rechnerkonfiguration 4 (siehe Anhang F) wurde bei den Experimenten mit SLPs und SVMs verwendet. Bei der Referenzmethode wurde Rechnerkonfiguration 2 verwendet. Die durchschnittliche Trainingsdauer von 14,87 Sekunden ist deshalb nicht direkt vergleichbar. Dennoch ist hier erkennbar, dass komprimierte SLPs deutlich schneller sind.

Die Referenzmethode erreicht hier mit Abstand die beste Korrektklassifikationsrate. Allerdings ist die Trainingsdauer auch vergleichsweise hoch und es werden spezialisiertere Vorverarbeitungsmethoden eingesetzt.

Unkomprimierte SLPs und SVMs erreichen im Mittel eine fast identische Korrektklassifikationsrate. Wenn bestimmte Datensätze betrachtet werden, sind allerdings manchmal SLPs und manchmal SVMs überlegen. Aufgrund des kubischen Aufwands jeder Iteration von LMA ist die Lerndauer für SLPs bei geringer Komprimierung deutlich länger als bei SVMs.

Die Korrektklassifikationsrate steigt etwas bei leicht komprimierten SLPs gegenüber unkomprimierten. Bei SVMs hingegen wird die Korrektklassifikationsrate durch die

Komprimierung etwas schlechter. Eventuell ist hier das Ergebnis besser, wenn der Parameter C für jede Komprimierungsstufe angepasst wird. Dies würde aber die Trainingsdauer erhöhen.

Bis zu einer Komprimierung zu etwa 232 Parametern (ca. 15 % der Anzahl der Gewichte) fällt die Korrektklassifikationsrate sowohl bei SVMs als auch bei SLPs nur leicht ab. Bereits bei einer Komprimierung mit 388 Parametern (ca. 25 %) ist die Trainingsdauer von SVMs und SLPs fast identisch, aber die Korrektklassifikationsrate des SLPs ist auf dem Niveau unkomprimierter SVMs.

5.4.4. Erkenntnisse

Eine Komprimierung von SLPs zur Erkennung von P300-Potenzialen kann die Trainingsdauer stark reduzieren, ohne dabei die Zuverlässigkeit der Vorhersagen zu verringern.

Mit einem stark komprimierten SLP kann die Lerngeschwindigkeit von SVMs erreicht werden.

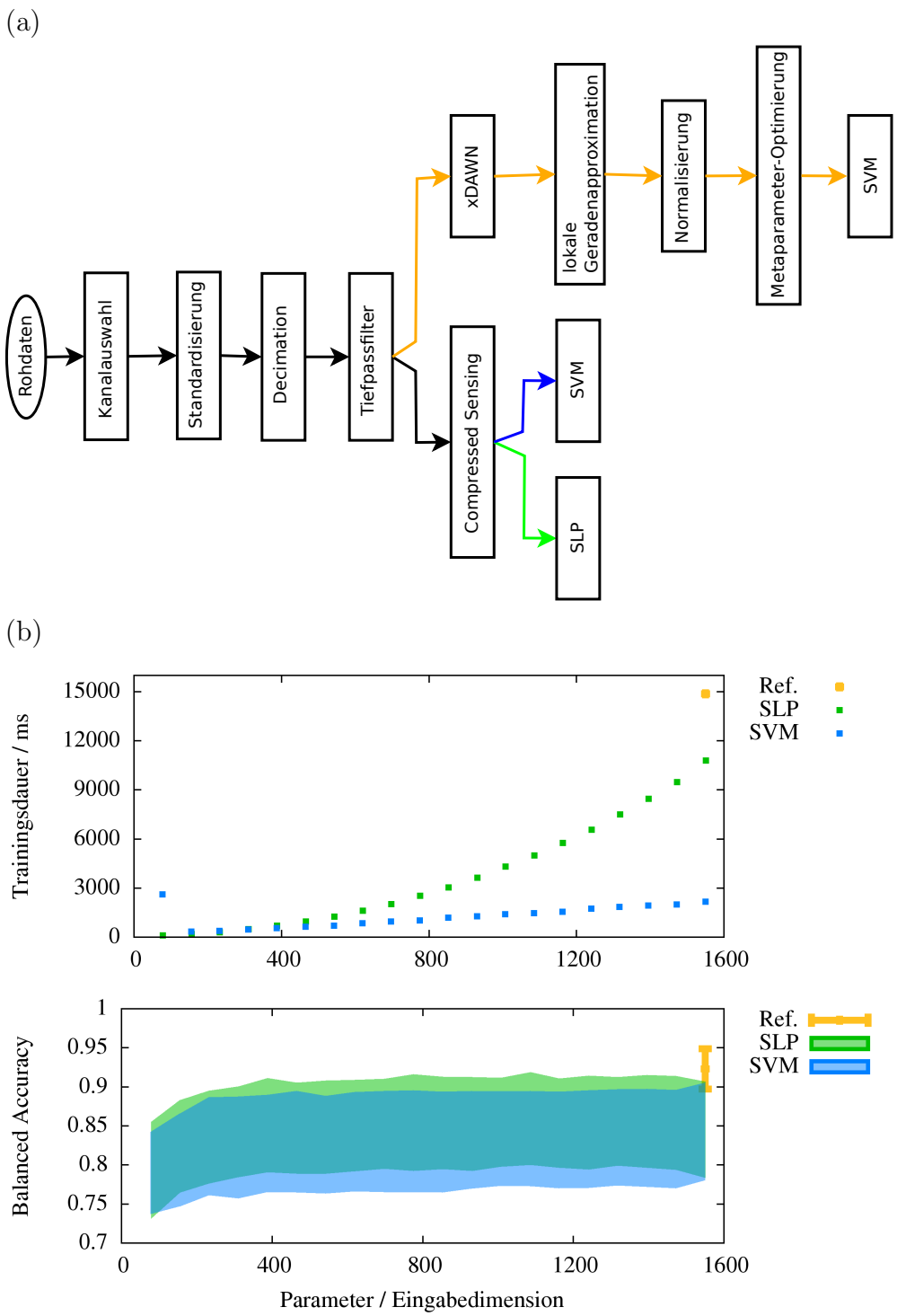


Abbildung 5.12.: (a) Drei verschiedene Methoden zur Klassifikation von P300-Potenzialen. (b) Balanced Accuracy auf dem Testdatensatz und Trainingsdauer mit verschiedenen Komprimierungen. Die Werte sind über acht Probanden und jeweils zehn Experimente gemittelt. Bei der Balanced Accuracy wurde das Intervall $[\mu-\sigma, \mu+\sigma]$ gekennzeichnet. Die Farben der Intervalle sind transparent, sodass bei Überschneidungen andere Farben entstehen. Die Balanced Accuracy der Referenzmethode wurde nur über die acht Probanden gemittelt.

6. Anwendung: Reinforcement Learning

Reinforcement Learning (bestärkendes Lernen) ist ein Teilgebiet des maschinellen Lernens. Hier werden Funktionsapproximatoren benötigt, wenn der Zustandsraum kontinuierlich ist. Funktionsapproximatoren sind Regressionsverfahren und deshalb können auch neuronale Netze eingesetzt werden.

In vielen Anwendungen wurden bereits neuronale Netze zu diesem Zweck verwendet. Zum Lernen eines Backgammon-Computergegners, der auf Expertenniveau spielt, wurde von Tesauro [87] ein MLP verwendet. Riedmiller [68] hat den Algorithmus Neural Fitted Q Iteration (NFQ) auf Grundlage eines MLP entwickelt und auf das Mountain-Car-Problem und Single Pole Balancing angewendet. Die erstaunlichste Anwendung von NFQ ist aber wahrscheinlich das Steuern eines echten Autos [70].

Da das Ziel des Reinforcement Learnings häufig das Erlernen einer Strategie eines Agenten in realen oder komplexen, simulierten Umgebungen ist, sollte die Anzahl der Aktionen, die ein Agent in dieser Welt durchführt, reduziert werden. Dadurch wird bei realen Systemen der Hardware-Verschleiß reduziert oder bei simulierten Systemen die Rechenzeit verringert. Die Verringerung der Rechenzeit einer Iteration des Optimierungsverfahrens ist hier also nicht so wichtig wie die Verringerung der Iterationen.

6.1. Grundlagen

Ich werde in diesem Abschnitt die den hier betrachteten Problemen zugrunde liegenden Eigenschaften erläutern. Dies ist keine allgemeine Beschreibung des Reinforcement Learnings, sondern speziell auf diese Probleme zugeschnitten. Ich gehe beispielsweise davon aus, dass die Probleme alle in Episoden ausführbar sind und nicht kontinuierlich weiterlaufen.

6.1.1. Agent-Environment-Interface

Im Reinforcement Learning wird das Verhalten eines Agenten in einer Umgebung erlernt. Dabei muss der Agent seine Umwelt zunächst erkunden. Eine Übersicht über die Interaktion von Umwelt und Agent ist in Abbildung 6.1 zu sehen. Der Index $t \in \mathbb{N}$ bezieht sich hierbei immer auf einen Zeitpunkt. Die Zeitskala ist diskret.

Diese Beschreibung ist an Sutton und Barto [86, Abschnitt 3.1] angelehnt. Der Agent nimmt zu jedem Zeitpunkt t innerhalb einer Episode den Zustand der Umwelt s_t aus der Menge aller möglichen Zustände S wahr. Er führt eine Aktion a_t aus, wobei a_t ein Element der Menge aller möglichen Aktionen A ist, und erhält einen *Reward* r_t. Der Reward ist zum Beispiel eine Bewertung des Nachfolgezustandes s_{t+1}, in dem sich

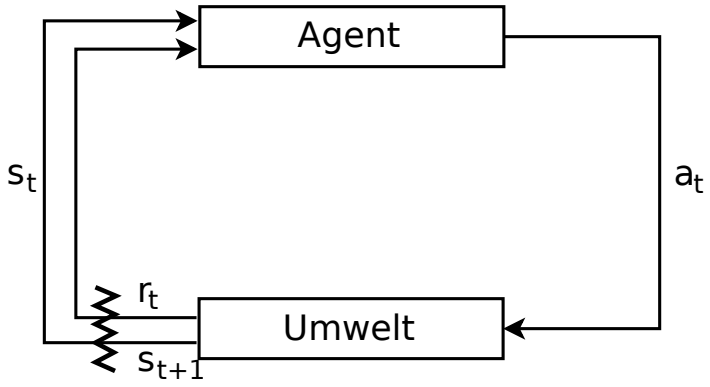


Abbildung 6.1.: Agent-Environment-Interface nach Sutton und Barto [86, Abschnitt 3.1].

der Agent im nächsten Zeitschritt befindet. Dadurch entsteht vom Start bis zum Ende einer Episode eine sogenannte Trajektorie

$$(s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T). \tag{6.1.1}$$

Eine Episode endet, wenn ein Endzustand erreicht wurde oder eine maximale Anzahl von Schritten durchgeführt wurde.

Der Agent wählt die Aktion auf Grundlage einer Strategie π. Das Ziel des Reinforcement Learning ist das Erlernen einer optimalen Strategie π^*. Eine optimale Strategie maximiert die Nutzenfunktion (englisch: value function) $V^*(s)$ oder $Q^*(s, a)$. Diese wiederum gibt den in dem Zustand s oder in dem Zustand s nach Ausführung der Aktion a erwarteten *Return* R an. Der Return ist eine Zusammenfassung des Rewards bis zum Ende der Episode.

Um die optimale Strategie zu erlernen, muss der Agent den *Reward* über die gesamte Episode optimieren. Der Agent muss den erwarteten *Return* $E(R_t)$ maximieren. Es existieren verschiedene Modelle zur Berechnung des Returns aus den einzelnen Rewards. Zu unterscheiden ist dabei zwischen Modellen mit begrenztem und unendlichem Horizont (oder bis zum Ende der Episode). In vielen Algorithmen (beispielsweise NFQ [68]) wird der Reward bis zum Ende der Episode beachtet, allerdings mit einer durch einen Faktor $0 \leq \gamma \leq 1$ abnehmenden Gewichtung [86, Abschnitt 3.3]:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{6.1.2}$$

6.1.2. Modell der Umwelt

Die meisten Probleme im Reinforcement Learning sind sogenannte Markov-Entscheidungsprobleme (Markov Decision Process, MDP). Eine Voraussetzung dafür ist, dass es jeweils keine anderen Faktoren gibt, die den Nachfolgezustand s' und den Reward r bestimmen, als den aktuellen Zustand s, die ausgeführte Aktion a und den Zufall.

Das heißt die Wahrscheinlichkeitsverteilung des Nachfolgezustandes ist nur von (s, a) abhängig. Es gilt also nach Sutton und Barto [86, Abschnitt 3.3]:

$$P(s_{t+1} = s', r_t = r | a_t, s_t, r_{t-1}, a_{t-1}, s_{t-1}, \ldots, r_0, a_0, s_0) \tag{6.1.3}$$
$$= P(s_{t+1} = s', r_t = r | a_t, s_t). \tag{6.1.4}$$

Diese Annahme wird als Markov-Annahme bezeichnet und ist in realen Umgebungen meistens eine Approximation. Des Weiteren gibt es Probleme, bei denen von dieser Annahme ausgegangen wird, obwohl sie falsch ist. Dies ist beispielsweise bei einem sogenannten *Partially Observable Markov Decision Process* (POMDP) der Fall. Hier kann der tatsächliche Zustand gar nicht vollständig wahrgenommen werden. Das zugrunde liegende Problem erfüllt dann eventuell die Markov-Annahme, allerdings ist sie aus der Perspektive des Agenten nicht mehr gültig.

In einem *endlichen* Markov-Entscheidungsproblem können oft das Zustandsübergangsmodell

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a) \tag{6.1.5}$$

und das Belohnungsmodell

$$\mathcal{R}^a_{ss'} = E(r_t | s_t = s, a_t = a, s_{t+1} = s') \tag{6.1.6}$$

angegeben werden, wodurch sich leicht mit Hilfe von Dynamic Programming die optimale Nutzenfunktion V^* berechnen lässt [86, Kapitel 4].

Ich werde hier davon ausgehen, dass das Modell der Umwelt nicht bekannt ist (*Model-free* Reinforcement Learning), sodass der Agent die Umwelt in verschiedenen Episoden erkunden muss, bevor er eine gute Lösung finden kann.

6.1.3. Nutzenfunktion und optimale Strategie

Normalerweise lernen Algorithmen deterministische Strategien $\pi : S \to A$, die in einem Zustand s eine bestimmte Aktion a auswählen. Es ist allerdings auch möglich eine nichtdeterministische Strategie $\pi : S \times A \to \mathbb{R}$ zu lernen, die Aktionen zufällig mit Hilfe einer erlernten Wahrscheinlichkeitsverteilung auswählen.

Der Algorithmus NFQ, der in Abschnitt 6.2 genutzt wird, lernt die Nutzenfunktion der optimalen Strategie $Q^* : S \times A \to \mathbb{R}$, die einem Zustands-Aktions-Paar (s, a) einen erwarteten Return zuordnet:

$$Q^*(s, a) = E_{\pi^*}(R_t | s_t = s, a_t = a). \tag{6.1.7}$$

Daraus kann wiederum die optimale deterministische Strategie π^* abgeleitet werden durch

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s, a). \tag{6.1.8}$$

Das funktioniert allerdings nur für einen endlichen Aktionsraum A.

6.1.4. Aktionsauswahl

Während des Lernens muss ein Agent zum einen die Umgebung erkunden (*Exploration*), um eine gute Strategie zu finden, zum anderen muss er allerdings die aktuelle Approximation der besten Strategie ausnutzen (*Exploitation*), um die Nutzenfunktion besser abschätzen zu können und somit möglicherweise eine gefundene Strategie als schlecht beurteilen zu können.

Ein Agent, der nur die aktuelle Approximation der besten Strategie nutzt, wird als *greedy* (gierig) bezeichnet. Der Agent erkundet seine Umgebung in einer deterministischen Welt nur, wenn die Werte der Nutzenfunktion optimistisch initialisiert wurden, da er dann feststellt, dass die Werte der Nutzenfunktion auf dem abgelaufenen Pfad schlechter sind als angenommen und dann einen anderen Pfad ablaufen wird. Dies geschieht so lange bis ein Pfad gefunden wurde, wo die Nutzenfunktion bessere Werte als die der Initialisierung liefert oder alle Pfade durchlaufen wurden. Dann sollte der Agent den besten Pfad gefunden haben. Ein Agent, der nicht exploriert, wird nicht mit einer dynamischen Umgebung zurechtkommen.

Um mehr zu explorieren, kann eine Softmax-Aktionsauswahl (englisch: softmax action selection) [86, Abschnitt 2.3] verwendet werden. Diese wählt die verschiedenen Aktionen abhängig von dem approximierten Wert von $Q(s, a)$ zufällig aus. Aktionen mit einem höheren Wert werden mit einer höheren Wahrscheinlichkeit ausgewählt. Ein Parameter $\tau > 0$, die sogenannte Temperatur, gibt dabei den Grad der Zufälligkeit an. Bei niedrigen Temperaturen ist die Wahrscheinlichkeit groß, die Aktion mit dem höchsten Q-Wert auszuwählen. Die Strategie ist also fast *greedy*. Durch diese Methode werden „sinnvolle“ Explorationen bevorzugt, da eine als sehr schlecht bewertete Aktion nur selten ausgewählt wird, die besten Aktionen allerdings relativ häufig. Die Wahrscheinlichkeit zur Auswahl einer Aktion wird über die Boltzmann-Verteilung

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}} \tag{6.1.9}$$

berechnet. Wenn die Q-Funktion gut angenähert ist, werden seltener schlechte Aktionen ausgewählt.

Zu unterscheiden sind in diesem Zusammenhang Agenten, die auf Grundlage der aktuell verwendeten Strategie ihre Approximation der Nutzenfunktion anpassen (On-Policy) und deshalb in der Regel eine annähernd greedy Strategie verfolgen sollten, und Agenten, die eine stärker explorierende Strategie im Training einsetzen, währenddessen aber die Nutzenfunktion der optimalen Strategie erlernen (Off-Policy).

6.2. Probleme mit kontinuierlichem Zustandsraum und diskretem Aktionsraum

6.2.1. Neural Fitted Q Iteration

Neural Fitted Q Iteration (NFQ) [68] basiert auf dem allgemeinen Verfahren *Fitted Q Iteration* [28], dessen Hauptfunktion in Algorithmus 7 schematisch dargestellt ist. NFQ nutzt ein MLP zur Approximation der Nutzenfunktion $Q(s, a)$. Die Besonderheit von

Algorithmus 7 Fitted Q Iteration

Eingabe: Nutzenfunktion $Q_k : S \times A \to \mathbb{R}$, Trajektorientupel $\mathcal{T} = ((s_1, a_1, r_1, s'_1), \ldots)$
Ausgabe: Nutzenfunktion $Q_{k+1} : S \times A \to \mathbb{R}$

```
Trainingsdatensatz T_0 ← {}
for all (s_i, a_i, r_i, s'_i) ∈ T do
    x^(i) ← (s_i, a_i)
    t^(i) ← r_i + γ max_a' Q_k(s'_i, a')
    T_{i+1} ← T_i ∪ {(x^(i), t^(i))}
end for
Q_{k+1} ← Regression mit dem Trainingsdatensatz T_|T|
return Q_{k+1}
```

NFQ ist, dass alle während des Trainings durchgeführten Zustandsübergänge gespeichert werden, das heißt es existiert eine Datenstruktur, die alle Tupel (s_t, a_t, r_t, s_{t+1}) enthält. Dadurch sollen die Generalisierungseffekte eines MLP genutzt werden ohne die Nachteile eines globalen Funktionsapproximators zu haben: normalerweise würde durch eine Anpassung an neue Daten die Gefahr bestehen, bisherige Anpassungen zu verlernen. Durch die Verwendung aller Daten zum Trainieren des MLP soll dies vermieden werden. Allerdings handelt man sich damit auch eine hohe Trainingsdauer aufgrund der enormen Größe des Trainingsdatensatzes ein. Der Algorithmus ist daher geeignet für Anwendungen, bei denen es wichtig ist, mit möglichst wenigen Interaktionen mit der Umwelt auszukommen.

Das Training erfolgt jeweils nach Ende einer Episode als Batch Learning. Dadurch können spezialisierte Verfahren zum Lernen verwendet werden. Riedmiller [68] verwendet dazu *Resilient Backpropagation* (Rprop) [69]. Der Trainingsdatensatz wird vorher mit Hilfe der Tupel (s_t, a_t, r_t, s_{t+1}) neu konstruiert. Die Eingabe sind dabei jeweils s_t und a_t und die erwartete Ausgabe ist $r_t + \gamma \max_{a'} Q_k(s_{t+1}, a'))$. Dabei gibt k die Iteration und somit auch die zuletzt abgeschlossene Episode an. Während der Trainingsepisoden verfolgt der Algorithmus eine *greedy* Strategie, das heißt es wird immer die nach der aktuellen Approximation von Q beste Aktion ausgeführt. Da durch die zufällige Initialisierung der Gewichte des MLP nicht gewährleistet werden kann, dass die approximierte Q-Funktion optimistisch initialisiert ist, werde ich in dem im Folgenden betrachteten Problem eine Softmax-Aktionsauswahl verwenden.

6.2.2. Probleme des MLP als Funktionsapproximator

Bei dem Einsatz eines MLP als Funktionsapproximator muss einiges beachtet werden. Die Generalisierungseffekte eines MLP wurden durch Boyan und Moore [16] als Grund für die schlechte Leistung in verschiedenen Testproblemen identifiziert. Whiteson u. a. [91] ermitteln aus demselben Grund verschiedene Konfigurationen eines MLP in Kombination mit dem Reinforcement-Learning-Verfahren Sarsa als unbrauchbar für das Mountain-Car-Problem. Dieses Problem wird durch NFQ gelöst.

Nach Riedmiller [68] ist der Kern des Problems, dass eine Gewichtsanpassung in einem neuronalen Netz globale Auswirkungen hat. Riedmiller [68] reduziert den Nachteil der Generalisierung durch die Verwendung aller ausgeführten Zustandsübergänge zum

Trainieren des MLP. Diese Technik wurde bereits durch Lin [55] unter dem Namen *Experience Replay* eingesetzt. Allerdings tritt dabei ein weiteres Problem auf: durch die nicht selektive Verwendung aller Daten kann es zu sogenanntem Übertraining (Overtraining) [55] kommen. Wenn dieselben Daten zu oft für das Training verwendet werden, könnten dadurch bestimmte Bereiche der Eingabedaten schneller genauer erlernt werden und andere Eingabebereiche werden in späteren Iterationen eventuell kaum Gewicht haben. Dies kann vor allem bei *On-Policy*-Verfahren der Fall sein. Da NFQ mit der hier verwendeten Methode zur Aktionsauswahl ein *Off-Policy*-Verfahren ist, stellt dies hier allerdings nur ein geringes Problem dar.

Gabel und Riedmiller [31] beschreiben ein weiteres Problem: *Policy Degradation*. Auch bei *Off-Policy*-Verfahren tritt dieses auf. Oft können mit neuronalen Netzen sehr schnell gute Strategien gefunden werden, allerdings degenerieren diese häufig wieder, da die Konvergenz bei der Verwendung von Funktionsapproximatoren nicht garantiert ist.

Das Problem der Policy Degradation wird durch NFQ nicht gelöst. Gabel und Riedmiller [31] nennen Ansätze zur Lösung dieses Problems. Eine Möglichkeit sei demnach das Prüfen jeder Strategie durch Testepisoden. Die Anzahl der Testepisoden müsse in stochastischen Umgebungen allerdings oft sehr hoch sein. Deshalb wird ein neues Verfahren entwickelt, das als Monitored Q Iteration (MQI) bezeichnet wird. In diesem Verfahren wird zu jeder Strategie ein Fehlermaß berechnet, mit dessen Hilfe der Zeitpunkt zum Abbrechen des Lernvorgangs bestimmt werden kann und gute Strategien gespeichert werden. Allerdings ist dies nicht der Schwerpunkt dieser Arbeit. Aus diesem Grund werde ich zum Vergleich, wie dies durch Riedmiller [68] getan wird, nur die Zeit bis zum Erlernen der ersten und besten erfolgreichen Strategie und die Qualität dieser Strategien heranziehen.

6.3. Mountain Car

6.3.1. Umgebung

Beim Mountain-Car-Problem muss der Agent ein Auto steuern, das sich in einer Umgebung mit zwei Bergen und einem Tal befindet (Abbildung 6.2). Das Ziel liegt auf einem der beiden Berge. Allerdings hat das Auto nicht genug Leistung, um vom Tal direkt den Gipfel zu erreichen, sodass es zunächst Geschwindigkeit aufbauen muss, indem es von dem anderen Berg herunterrollt. Der Agent muss also lernen zunächst in die falsche Richtung zu fahren. Es existieren unterschiedliche Beschreibungen des Problems. Ich orientiere mich hier an Whiteson u. a. [91].

Der Zustand des Agenten $s = (x_t, \dot{x}_t)^T$ setzt sich aus den kontinuierlichen Komponenten Position ($-1,2 \leq x_t \leq 0,6$) und Geschwindigkeit ($-0,07 \leq \dot{x}_t \leq 0,07$) zusammen. Der Aktionsraum ist diskret und enthält drei Elemente: das Auto ist in der Lage rückwärts zu beschleunigen ($a = -1$), nicht zu beschleunigen ($a = 0$) oder vorwärts zu beschleunigen ($a = 1$). Der Zustandsübergang wird wie folgt beschrieben:

$$s_{t+1} = \begin{pmatrix} x_t + \dot{x}_{t+1} \\ \dot{x}_t + 0,001a_t - 0,0025\cos(3x_t) \end{pmatrix} \tag{6.3.10}$$

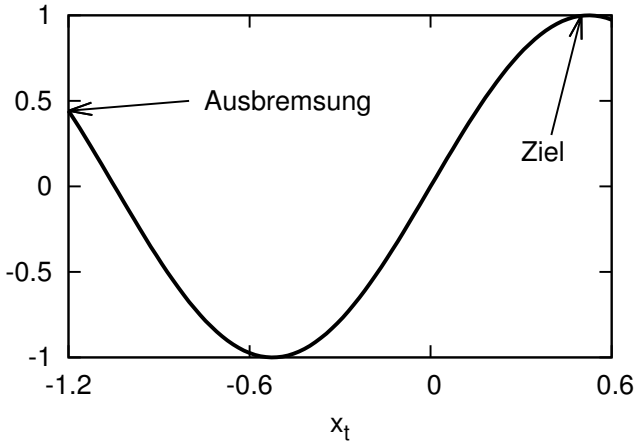


Abbildung 6.2.: Mountain-Car-Umgebung. Die Höhe wird durch $\sin(3x_t)$ berechnet.

Beide Komponenten sind dabei auf ihre Wertebereiche beschränkt. Wenn der Wagen die Position -1,2 erreicht, wird die Geschwindigkeit auf 0 gesetzt. Anfangs wird der Zustand innerhalb der gegebenen Grenzen zufällig initialisiert. Eine Episode wird entweder nach 5000 Schritten oder nach Erreichen einer Position $x_t \geq 0{,}5$ beendet. Der Reward ist in jedem normalen Schritt -1 und im Ziel 0.

6.3.2. Methode

Es wurde jeweils, wie auch bei Riedmiller [68], ein MLP mit der Topologie 3-5-5-1 verwendet, allerdings mit einem Bias in jeder Schicht. Die Aktivierungsfunktionen sind in den versteckten Schichten Tangens Hyperbolicus und in der Ausgabeschicht die Identität. Die für das MLP verwendete Fehlerfunktion ist die Summe quadratischer Fehler. Zur Optimierung der Parameter wurde sowohl bei komprimierten als auch bei unkomprimierten Gewichten LMA verwendet. Die Komprimierung erfolgte mit orthogonalen Kosinusfunktionen. Die Optimierung wurde abgebrochen, wenn

- der Gradient zu klein wird ($|g| < 0{,}001$) oder
- $|E_{SSE}^{(t)} - E_{SSE}^{(t-1)}| \leq 0{,}0001 \max\{E_{SSE}^{(t-1)}, E_{SSE}^{(t)}, 1\}$, wobei t den Optimierungsschritt angibt.

Zur Berechnung des Returns wurde der Diskontierungsfaktor $\gamma = 1{,}0$ gewählt. Die Trainingstrajektorien wurden durch Softmax-Aktionsauswahl auf Grundlage der aktuell approximierten Q-Funktion generiert. Dabei wurde die Temperatur $\tau = 0{,}01$ verwendet. Die Länge der Trainingstrajektorien war nie größer als 100 Zustandsübergänge, um die Anzahl der Trainingsdaten zu reduzieren.

6.3.3. Ergebnisse

Riedmiller [68] hat den Algorithmus Neural Fitted Q Iteration unter anderem in einer Mountain-Car-Umgebung getestet. Die dabei verwendete Umgebung wurde von

Schoknecht und Riedmiller [78] entwickelt und weicht stark von der in Abschnitt 6.3.1 beschriebenen ab. Ein direkter Vergleich ist also nicht möglich.

Das Ergebnis der Experimente ist in Tabelle 6.1 zu sehen. Ich bezeichne die NFQ-Variante mit komprimierter Gewichtsrepräsentation als CNFQ-n (Compressed Neural Fitted Q Iteration). Gemessen wurden die Anzahl der Episoden und der Zyklen (Zustandsübergänge) bis zum Erlernen der ersten erfolgreichen und der besten Strategie. Eine Strategie gilt als erfolgreich, wenn diese in 1000 Testepisoden mit zufälligem Startpunkt immer das Ziel erreicht. Zudem wurde die Qualität der Strategie gemessen durch Berechnung des arithmetischen Mittels, des Medians und der Standardabweichung des Returns in 1000 Testepisoden. In Abbildung 6.4 sind Strategien dargestellt, die in dieser Umgebung erfolgreich waren.

Die Anzahl der Episoden und Zyklen zum Erlernen der besten erfolgreichen Strategie nimmt mit der Anzahl der Parameter ab, unterschreitet allerdings erst mit der stärksten getesteten Komprimierung die Anzahl der bei NFQ benötigten Zyklen. Ein ähnlicher Effekt wurde bereits durch Koutník u. a. [45] entdeckt. Dort wurde festgestellt, dass die Lösung für die Umgebung *Single Pole Balancing* durch Suchen im sechsdimensionalen Gewichtsraum schneller gelöst wird als durch Suchen im fünf- oder sechsdimensionalen Parameterraum. Mit zwei bis vier Parametern wurde die Lösung allerdings noch schneller gefunden.

Des Weiteren ist hier ein leichter Zusammenhang zwischen der Anzahl der benötigten Zyklen zum Finden der besten Strategie und der Qualität dieser Strategie zu erkennen: je mehr Zyklen benötigt werden, desto besser ist die Strategie. Die Un-

	MLP 3-5-5-1	NFQ	CNFQ-1	CNFQ-2	CNFQ-3
	Komprimierung	keine	4-6-6	4-3-3	4-1-1
	Parameter	56	56	38	26
	Erste erfolgreiche Strategie				
	Episode	58,1	64,4	79,7	52,6
	Zyklen	4980,8	5462,9	6900,8	4420,5
Return	μ	-77,6	-65,8	-99,7	-107,8
Return	σ	54,6	42,2	92,2	121,3
Return	Median	-70,6	-63,4	-67,3	-71
	Beste Strategie (in 500 Episoden)				
	Episode	166,8	172,4	167,6	**141,8**
	Zyklen	14251	14753	14454,6	**12143,9**
Return	μ	-55	**-52,9**	-54,1	-55,5
Return	σ	33,4	31,9	33,2	33,6
Return	Median	-55,3	**-53,9**	-54,6	-56

Tabelle 6.1.: Vergleich von NFQ mit normalem MLP und mit komprimierten Gewichten. Alle Werte wurden über 20 Experimente gemittelt und auf eine Nachkommastelle genau angegeben. Verglichen werden die Trainingsdauer in der Umgebung gemessen in Episoden und akkumulierten Zyklen (Zustandsübergänge) und die Qualität der erlernten Strategie anhand des über 1000 Testepisoden beobachteten Returns.

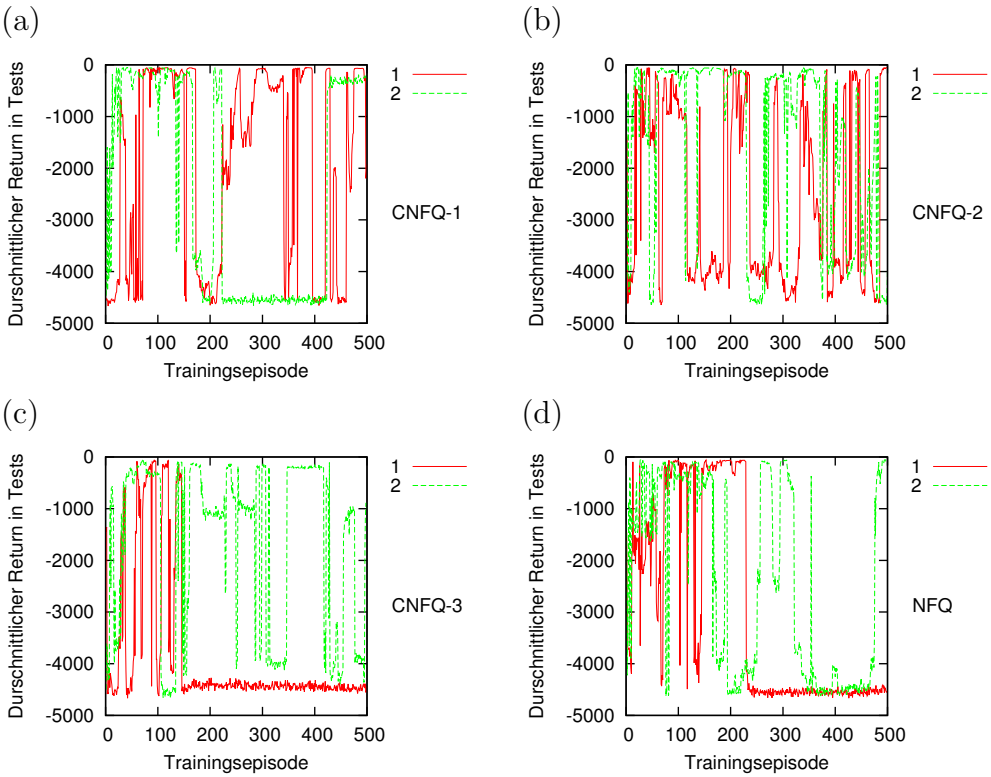


Abbildung 6.3.: Return beim Mountain-Car-Problem im Verlauf der Optimierung mit unterschiedlichen Funktionsapproximatoren. Für jeden Funktionsapproximator wurden jeweils zwei Versuche mit 500 Trainingsepisoden dargestellt. Nach jeder Trainingsepisode wurde der Return über 1000 Testepisoden gemittelt.

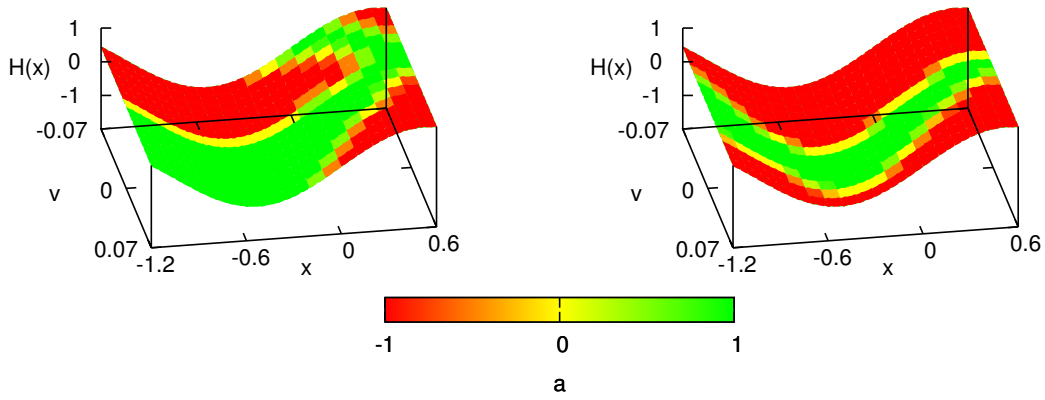


Abbildung 6.4.: Erfolgreiche Strategien in der Mountain-Car-Umgebung. Die Oberfläche stellt den Zustandsraum dar, wobei $H(x)$ die Höhe an der Position x ist und $v = \dot{x}$. Die Farbe zeigt die in dem Zustand gewählte Beschleunigung a an.

Funktionsapproximator	Lernverfahren	Veröffentlichung	Optimale Schrittanzahl
MLP (komprimiert)	CNFQ-n	hier	53-57
MLP	NFQ	hier	56
Tile Coding	Sarsa	[91]	56*
Neuronales Netz	NEAT	[91]	56*
RBF	Q-Learning	[46]	56
Tile Coding	Q-Learning	[46]	68
RBF	Sarsa	[26]	75*
RBF	LSPI	[26]	76

Tabelle 6.2.: Vergleich der besten gefundenen Strategien für das Mountain-Car-Problem aus verschiedenen Veröffentlichungen. Die Anzahl der Schritte wurde auf eine ganze Zahl gerundet. Mit * gekennzeichnete Werte wurden aus Abbildungen extrahiert.

terschiede in der Qualität der Strategien sind allerdings nicht signifikant bei einem angestrebten Signifikanzniveau von 1 %.

Abbildung 6.3 zeigt den Verlauf des durchschnittlichen Returns in den Testepisoden während des Trainings. Es werden meistens sehr schnell gute Strategien erlernt, die einen hohen durchschnittlichen Return haben, allerdings können diese innerhalb einer Trainingsepisode komplett verlernt werden. Dieser *Policy Degradation* genannte Effekt ist bei der Verwendung von NFQ zu erwarten (siehe Abschnitt 6.2.2). Eine gute Strategie kann allerdings durch Testepisoden erkannt werden.

6.3.4. Vergleich zu anderen Funktionsapproximatoren

Das Mountain-Car-Problem wurde bereits für verschiedene Algorithmen und Funktionsapproximatoren als Benchmark eingesetzt. Pyeatt und Howe [66] verwenden Entscheidungsbäume, Boyan und Moore [16] setzen Locally Weighted Regression ein, Sutton [85] nutzt Tile Coding, Kretchmar und Anderson [46] vergleichen Radial Basis Functions (RBF) und Tile Coding und Whiteson u. a. [91] vergleichen SARSA mit Tile Coding, einem Single Layer Perceptron und einem Multilayer Perceptron mit NeuroEvolution of Augmenting Topologies (NEAT). Dennoch findet man in diesen Veröffentlichungen wenig Daten, die einen Vergleich der Qualität der ermittelten Strategien ermöglichen. Allerdings wurden einige Ergebnisse im Rahmen eines NIPS-Workshops [26] veröffentlicht.

Häufig wurden leichte Variationen des Problems verwendet. Bei dem NIPS-Workshop [26] wurden beispielsweise nur 50 unterschiedliche Startzustände generiert die in den Trainingsepisoden verwendet wurden und die Geschwindigkeit wurde bei allen auf 0 gesetzt. Zudem war die untere Grenze der Position -1,1 und die Anzahl der Schritte auf 300 pro Episode beschränkt.

Aus Tabelle 6.2 ist die Qualität der erlernten Strategien aus den Publikationen, die mir bekannt sind, zu entnehmen. Das Maß ist hierbei die durchschnittliche Anzahl der Schritte zum Erreichen des Ziels bei zufälliger Initialisierung. Durch den in Tabelle 6.1 angegebenen durchschnittlichen Return lässt sich leicht die Schrittanzahl berechnen,

da eine erfolgreiche Strategie in jedem normalen Schritt den Reward -1 erhält und im letzten Schritt den Reward 0. Bei den im Rahmen des NIPS-Workshop [26] ermittelten Werten ist zu beachten, dass die Geschwindigkeit immer mit 0 initialisiert wird.

Die Qualität der besten durch NFQ oder CNFQ-n erlernten Strategien ist vergleichbar mit denen anderer Lernverfahren. Allerdings konvergieren Verfahren, die auf Funktionsapproximatoren basieren, die eine lokale Anpassung der Q-Funktion ermöglichen, wie Tile Coding oder RBF, in der Praxis. Dadurch sind dort keine weiteren Testepisoden nötig.

6.3.5. Erkenntnisse

Eine Komprimierung der Gewichte kann eine Beschleunigung des Lernverfahrens NFQ bewirken. Hier wurde dies allerdings nicht eindeutig nachgewiesen. Um den Nachweis zu liefern, müsste ein komplexeres Problem betrachtet werden.

NFQ ist allerdings rechnerisch aufwändig. Die Durchführung der Experimente dauerte zum Beispiel insgesamt über drei Wochen mit Rechnerkonfiguration 4 (siehe Anhang F). Deshalb werde ich diese Idee hier nicht weiter verfolgen, sondern stattdessen ein anderes Reinforcement-Learning-Verfahren untersuchen.

6.4. Probleme mit kontinuierlichem Zustands- und Aktionsraum

6.4.1. Kontinuierlicher Aktionsraum

Ein kontinuierlicher oder sehr großer Aktionsraum erschwert das Reinforcement Learning. Es ist nicht mehr möglich durch Prüfen des Q-Wertes aller Aktionen die beste Aktion zu finden. Eine Möglichkeit, mit Problemen dieser Art umzugehen sind neuroevolutionäre Methoden. Dabei wird die Strategie $\pi : S \rightarrow A$ direkt durch ein neuronales Netz dargestellt und ein ableitungsfreies Optimierungsverfahren wird zur Anpassung des Netzes verwendet. Dabei wird die Tauglichkeit einer Strategie nach jeder Episode ermittelt und dieser Wert zur Optimierung verwendet. Es wird also keine Backpropagation mehr verwendet. Zu unterscheiden ist hierbei zwischen Verfahren, die die Topologie verändern (zum Beispiel NEAT [84] und EANT [43]) und solchen, die nur die Parameter anpassen. In dieser Arbeit werden nur die Parameter optimiert. Dazu wird das Optimierungsverfahren IPOP-CMA-ES verwendet. Ein sehr ähnliches Verfahren namens CMA-NeuroES wurde durch Heidrich-Meisner und Igel [37] entwickelt und anhand verschiedener Pole-Balancing-Probleme getestet. CMA-NeuroES unterscheidet sich zu dem hier verwendeten Verfahren dadurch, dass dabei CMA-ES anstatt IPOP-CMA-ES verwendet wird und die Gewichte direkt angepasst werden. Im Folgenden werde ich allerdings auch die hier verwendete CMA-ES-Variante durch CMA-ES abkürzen.

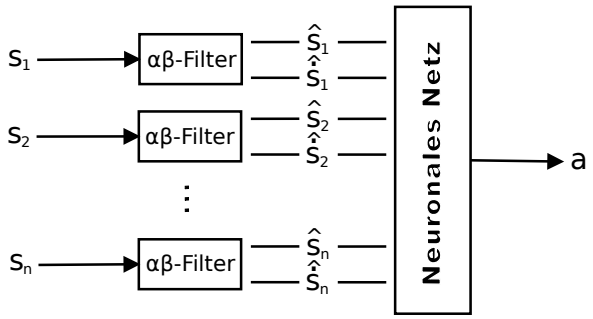


Abbildung 6.5.: Zur Zustandsschätzung in teilweise beobachtbaren oder verrauschten Umgebungen durch α-β-Filter erweitertes neuronales Netz [42].

6.4.2. Unvollständige Zustandsinformationen

Das hier entwickelte Verfahren zur Gewichtskomprimierung wurde nur für das Multilayer Perceptron implementiert, welches ein spezielles Feedforward-Netz ist. Es gibt also normalerweise keine Möglichkeit, etwas in dem neuronalen Netz zwischen zwei Vorwärtspropagationen zu speichern.

Eine übliche Modifikation des Benchmarks Pole Balancing ist, dass der Agent nicht mehr den gesamten Zustand wahrnimmt. Die Geschwindigkeitskomponenten werden dabei weggelassen. Ohne die Geschwindigkeitskomponenten ist es allerdings nicht möglich, eine erfolgreiche Strategie zu erlernen. Der Entscheidungsprozess ist nicht mehr vollständig beobachtbar und erfüllt somit nicht mehr die Markov-Annahme.

Koutník u. a. [44] verwenden vollständig verbundene, rekurrente neuronale Netze (*Fully Connected Recurrent Neural Networks - FCRNNs*). Diese realisieren einen Speicher, indem jedes Neuron seine Ausgabe an jedes andere Neuron sendet. Bereits durchlaufene Neuronen speichern diese Ausgabe bis zur nächsten Vorwärtspropagationen. Auf diese Weise ist das Problem lösbar.

Bei Feedforward-Netzen ist das Erlernen einer erfolgreichen Strategie allerdings auch möglich, wenn neben dem aktuellen auch der vorherige Zustand als Eingabe des Netzes verwendet wird, da das Netz dann die Geschwindigkeiten durch die Differenz der beiden Zustände approximieren kann.

Bessere Schätzungen der Geschwindigkeiten und der aktuellen Position kann aber ein sogenannter α-β-*Filter* berechnen. Der α-β-Filter ist eine Vereinfachung des Kalman-Filters [88, Seite 40 ff.], einem Verfahren zur Zustandsschätzung. Die Schätzungen der Filter können als Eingabe des neuronalen Netzes zur Berechnung der Aktion verwendet werden (siehe Abbildung 6.5). Dieses Verfahren wurde von Kassahun u. a. [42] entwickelt und als Augmented Neural Network with Kalman Filter (ANKF) bezeichnet. Hierbei ist allerdings zu beachten, dass für jeden α-β-Filter ein weiterer Parameter durch den Optimierungsalgorithmus angepasst werden muss. Diese Parameter werden hier nicht komprimiert, allerdings wäre dies möglich.

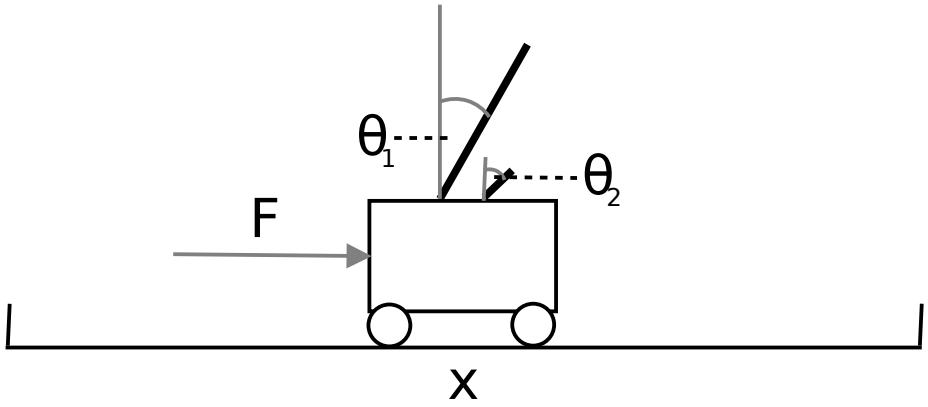


Abbildung 6.6.: Double Pole Balancing. Beim Single Pole Balancing fehlt der zweite Stab. Beide Stäbe können sich nicht berühren.

6.5. Pole Balancing

6.5.1. Umgebung

In der folgenden Beschreibung der Umgebung werden die physikalischen Einheiten weggelassen, da sie für diese Arbeit nicht wichtig sind. Alle Winkelangaben sind hier im Bogenmaß. Die Beschreibung der Umgebung wurde von Wieland [92] übernommen.[1]

Pole Balancing ist ein Standard-Problem im Reinforcement Learning. Hierbei sind N (hier: einer oder zwei) Stäbe auf einem Wagen montiert und müssen balanciert werden, indem eine Kraft $F \in [-10, 10]$ auf den Wagen ausgewirkt wird. Die Stäbe können nur an einem Gelenk und der Wagen nur in einer Dimension bewegt werden. Der Zustand des Systems kann durch die Position x und Geschwindigkeit $\dot{x}$ des Wagens und die Winkel θ_i und die Winkelgeschwindigkeiten $\dot{\theta}_i$ der Stäbe beschrieben werden. In Abbildung 6.6 ist die Umgebung für $N = 2$ schematisch dargestellt. Eine Strategie gilt als erfolgreich, wenn die Stäbe 100.000 Schritte balanciert werden, das heißt die Position des Wagens darf maximal 2,4 vom Startpunkt entfernt sein und die Beträge der Winkel der Stabgelenke dürfen nie 0,62 überschreiten. Alle Komponenten des Zustands $s = (x, \dot{x}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2)^T$ sind initial 0 bis auf $\theta_1 = 0{,}07$, wobei die letzten beiden Komponenten beim Single Pole Balancing nicht Teil des Zustands sind.

Ich verwende hier die durch Wieland [92] entwickelten Dynamik-Gleichungen:

$$\ddot{x} = \frac{F - \mu_c \operatorname{sgn} \dot{x} + \sum_{i=1}^{N} \tilde{F}_i}{M + \sum_{i=1}^{N} \tilde{m}_i} \tag{6.5.11}$$

$$\ddot{\theta}_i = -\frac{3}{4l_i} \left(\ddot{x} \cos\theta_i + g \sin\theta_i + \frac{\mu_{pi} \dot{\theta}_i}{m_i l_i} \right) \tag{6.5.12}$$

[1] Ich verwende in dieser Arbeit die unter `http://sourceforge.net/projects/eant-project` im Rahmen einer Veröffentlichung von Kassahun u. a. [42] bereitgestellte Implementierung dieser Umgebung.

Parameter	M	m_1	l_1	m_2	l_2	μ_c	μ_{p1}, μ_{p2}	g
Wert	1	0,1	0,5	0,01	0,05	0,0005	0,000002	-9,8

Tabelle 6.3.: Parameter für Single und Double Pole Balancing.

$$\tilde{F}_i = m_i l_i \dot{\theta}_i^2 \sin\theta_i + \frac{3}{4} m_i \cos\theta_i \left(\frac{\mu_{pi}\dot{\theta}_i}{m_i l_i} + g \sin\theta_i \right) \tag{6.5.13}$$

$$\tilde{m}_i = m_i \left(1 - \frac{3}{4}\cos^2\theta_i \right) \tag{6.5.14}$$

Dabei ist x die Position des Wagens, θ_i ist der Winkel des i-ten Stabes, N ist die Anzahl der Stäbe auf dem Wagen, g ist die Beschleunigung durch die Erdanziehung, m_i und l_i sind Masse und halbe Länge des i-ten Stabes, M ist die Masse des Wagens, μ_c ist der Reibungskoeffizient des Wagens auf dem Boden, μ_{pi} ist der Reibungskoeffizient des Drehgelenks des i-ten Stabs, F ist die an den Wagen angelegte Kraft, $\tilde{F}_i$ ist die effektive Kraft des i-ten Stabes auf den Wagen und $\tilde{m}_i$ ist die effektive Masse des i-ten Stabes.

Zur Berechnung eines Zustandsübergangs wird ein numerisches Integrationsverfahren benötigt. Hier wird das Runge-Kutta-Verfahren vierter Ordnung [18, Seite 931 f., Abschnitt 19.4.1.2] verwendet. Durch Integration der oben aufgeführten Gleichungen für $\ddot{x}$ und $\ddot{\theta}_i$ zum Zeitpunkt t lassen sich $\dot{x}$ und $\dot{\theta}_i$ des Folgezustands zum Zeitpunkt $t+1$ ermitteln und durch Integration von $\dot{x}$ und $\dot{\theta}_i$ zum Zeitpunkt t wiederum x und θ zum Zeitpunkt $t+1$. Zwischen den Zeitpunkten t und $t+1$ liegt jeweils der Zeitraum $\tau = 0{,}02$ s. In Tabelle 6.3 werden die hier verwendeten Parameter der Umgebung zusammengefasst.

Bei einer üblichen Erweiterung des Pole Balancing sind die Geschwindigkeitskomponenten des tatsächlichen Zustands nicht beobachtbar. Dadurch ist die Markov-Annahme aus Sicht des Agenten nicht mehr erfüllt. Ich werde auch diese Variante untersuchen.

6.5.2. Methode

Die Gewichte der Funktionsapproximatoren werden hier nach dem in Abschnitt 3.1 beschriebenen Verfahren generiert. Dazu werden die dort aufgeführten orthogonalen Kosinusfunktionen verwendet.

Pole Balancing mit Geschwindigkeiten: Zur Lösung dieses Problems habe ich, wie dies auch durch Koutník u. a. [44] getan wurde, die Strategie $\pi : S \to A$ durch ein neuronales Netz repräsentiert und die Parameter durch ein ableitungsfreies Optimierungsverfahren angepasst. Die Topologien der neuronalen Netze sind in Abbildung 6.7 zu sehen. Eine erfolgreiche Strategie wird schneller erlernt, wenn kein Bias vorhanden ist, sodass dieser weggelassen wurde. Die verwendete Aktivierungsfunktion ist die Identität, sodass die ausgewählte Aktion jeweils eine gewichtete Summe der Zustandskomponenten ist.

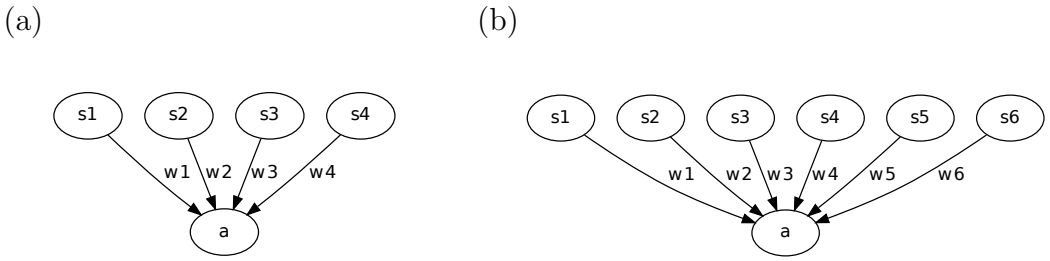


Abbildung 6.7.: Topologie der neuronalen Netze zur Repräsentation der Strategie beim (a) Single Pole Balancing und (b) Double Pole Balancing.

Der verwendete Optimierungsalgorithmus ist IPOP-CMA-ES und die zu minimierende Funktion ist $E(\alpha) = -t$, wobei t hier die Anzahl der Schritte bis zum Ende einer Episode ist und α der Parameter-Vektor des neuronalen Netzes ist. Die Optimierung wurde abgebrochen, sobald eine Strategie die Stäbe erfolgreich für 100.000 Schritte balancieren konnte. Ein Neustart mit verdoppelter Populationsgröße fand jeweils statt, wenn nach 1000 Episoden keine erfolgreiche Strategie gefunden wurde. Beim Single Pole Balancing ist die initiale Schrittgröße für CMA-ES $\sigma_0 = 100$ und beim Double Pole Balancing $\sigma_0 = 10$. Der Wert 100 liefert beim Single Pole Balancing deutlich bessere Ergebnisse als 10. Eine genaue Untersuchung der Parameter von CMA-ES wurde hier allerdings nicht durchgeführt.

Pole Balancing ohne Geschwindigkeiten: Die Konfiguration ist hier ähnlich. Allerdings müssen die Geschwindigkeiten für die neuronalen Netze errechenbar sein. Dazu werden zwei Ansätze geprüft. Im ersten Ansatz wird die Eingabe des neuronalen Netzes aus dem aktuellen und dem letzten Zustand (s_t und s_{t-1}) zusammengesetzt und in dem zweiten Ansatz wird für jede wahrnehmbare Zustandskomponente ein α-β-Filter verwendet, der die Position und Geschwindigkeit dieser Komponente schätzt, und die Ausgabe des Filters als Eingabe des Netzes verwendet. Da die Positionen nicht verrauscht sind, sollte die Schätzung der Position durch den α-β-Filter bei einer Lösung exakt der Eingabe entsprechen. Die Filter haben jeweils einen Parameter, der optimiert werden muss, sodass beim Single Pole Balancing zwei und beim Double Pole Balancing drei weitere Parameter optimiert werden müssen. Die initiale Schrittweite für CMA-ES ist in allen Fällen $\sigma_0 = 10$.

6.5.3. Ergebnisse

Pole Balancing mit Geschwindigkeiten: Zum Vergleich verschiedener Komprimierungen wurden pro Konfiguration 1000 Tests durchgeführt. Für Konfigurationen, bei denen es möglich war eine erfolgreiche Strategie zu erlernen, gab es unter 1000 Tests keinen einzigen, bei dem diese nicht erlernt wurde. Deshalb wurde hier nur die Lerngeschwindigkeit als Vergleichskriterium herangezogen, wozu die Anzahl der benötigten Episoden gemessen wurden. In den Tabellen 6.4 und 6.5 sind der durchschnittliche Wert μ, der minimale Wert, der maximale Wert, der Median und die Standardab-

	Gewichte	unkomprimiert	komprimiert		
	Parameter	4	4	3	2	1
Episoden	μ	25	13	9	5	**2**
	σ	20,261	9,917	7,696	4,894	2,207
	Minimum	1	1	1	1	1
	Maximum	137	59	54	44	20
	Median	20	11	8	4	**2**

Tabelle 6.4.: Ergebnisse für Single Pole Balancing mit Geschwindigkeiten.

	Gewichte	unkomprimiert	komprimiert	
	Parameter	6	6	5
Episoden	μ	279	212	**195**
	σ	195,951	142,457	186,686
	Minimum	23	11	17
	Maximum	1562	1340	1423
	Median	234	181	**148**

Tabelle 6.5.: Ergebnisse für Double Pole Balancing mit Geschwindigkeiten.

weichung σ der 1000 Testdurchläufe angegeben. Betrachtet werden hierbei nur komprimierte Konfigurationen, bei denen die Anzahl der Parameter höchstens die der Gewichte des neuronalen Netzes ist und bei denen es möglich ist, eine Lösung zu finden.

Die Unterschiede zwischen dem unkomprimierten MLP und der besten Komprimierung sind bei beiden Umgebungen nach dem z-Test signifikant mit einem Signifikanzniveau von 0,01. Durch Reduktion der Parameter wird der Optimierungsvorgang also erheblich beschleunigt. Insbesondere das Ergebnis beim Single Pole Balancing zeigt, dass die Lösung sehr einfach repräsentiert werden kann. Das Ergebnis deckt sich mit dem von Koutník u. a. [44], die dies damit begründen, dass die Gewichte nur gleich und positiv sein müssen, damit eine Strategie erfolgreich ist.

Pole Balancing ohne Geschwindigkeiten: Wieder wurden 1000 Tests pro Konfiguration ausgeführt. Nur Konfigurationen, bei denen keine Fehlschläge auftraten, werden in den Tabellen 6.6 und 6.7 verglichen. Die Parameteranzahl bei der Verwendung von α-β-Filtern setzt sich aus den unkomprimierten Parametern der α-β-Filter und den Parametern des neuronalen Netzes zusammen.

Die Berechnung der Geschwindigkeiten durch α-β-Filter funktioniert in beiden Umgebungen, die Verwendung des aktuellen und vorherigen Zustands als Eingabe des neuronalen Netzes funktioniert mit den getesteten Topologien nur beim Single Pole Balancing. Auch mit einer versteckten Schicht, funktioniert dies beim Double Pole Balancing nicht.

Die Unterschiede zwischen dem unkomprimierten MLP und der besten Komprimierung sind nur beim Single Pole Balancing nach dem z-Test signifikant mit einem Signifikanzniveau von 0,01. Eine Komprimierung beschleunigt also nur beim Single Pole Balancing das Erlernen einer erfolgreichen Strategie.

MLP-Eingabe		s_t, s_{t-1}		Ausgabe der α-β-Filter			
Gewichte		unkomprimiert	komprimiert	unkomprimiert	komprimiert		
Parameter		4	4	2+4	2+4	2+3	2+2
Episoden	μ	195	170	30	20	**14**	32
	σ	203,399	128,357	14,473	11,635	9,871	16,250
	Minimum	10	1	1	1	1	1
	Maximum	1466	1277	92	86	62	91
	Median	146	152	28	19	**13**	31

Tabelle 6.6.: Ergebnisse für Single Pole Balancing ohne Geschwindigkeiten.

MLP-Eingabe		Ausgabe der α-β-Filter		
Gewichte		unkomprimiert	komprimiert	
Parameter		3+6	3+6	3+5
Episoden	μ	422	485	**420**
	σ	232,094	303,577	300,847
	Minimum	31	3	18
	Maximum	1804	2808	1672
	Median	378	433	**344**

Tabelle 6.7.: Ergebnisse für Double Pole Balancing ohne Geschwindigkeiten.

6.5.4. Vergleich zu anderen Lernverfahren

Ich vergleiche die Ergebnisse mit denen anderer Methoden anhand der Anzahl der benötigten Episoden zum Erlernen einer erfolgreichen Strategie beim Single Pole Balancing (SPB) und Double Pole Balancing (DPB). Die Ergebnisse sind in Tabelle 6.8 zu sehen. Die durch Kassahun [41] eingeführte Abkürzung DOF steht hierbei für *Discrete Orthogonal Functions* und umschließt das in Abschnitt 3.1 entwickelte Verfahren zur Gewichtskomprimierung.

Beim Single Pole Balancing mit Geschwindigkeiten konnte mit einer guten Wahl des CMA-ES-Parameters σ_0 das bisher beste Ergebnis von Koutník u. a. [44] reproduziert werden. Dort wurde genau dieselbe Topologie und Komprimierung verwendet, allerdings ein anderes Optimierungsverfahren. Die Lösung ist hier, wie bereits erwähnt, durch eine Komprimierung mit einem Parameter sehr einfach repräsentierbar. Beim Double Pole Balancing ist das hier ermittelte Ergebnis zwar besser als das bisher beste von Koutník u. a. [44], allerdings ist es nicht möglich die Signifikanz festzustellen, da die Standardabweichung dort nicht veröffentlicht wurde.

Beim Single Pole Balancing ohne Geschwindigkeiten ist das hier ermittelte Resultat deutlich besser als das aus bisherigen Veröffentlichungen. Die Ausnahme bilden hier die Ergebnisse von Kassahun [41], wo ein fast identisches Verfahren verwendet wurde, allerdings wurde eine andere Fitness-Funktion verwendet und nach Aussage des Autors ein Parameter für alle α-β-Filter genutzt, sodass beim SPB ein und beim DPB zwei Parameter weniger optimiert wurden, als bei dem hier entwickelten Verfahren. Zudem wurde dort der Parameter für die Filter zusammen mit den Gewichten des neuronalen Netzes komprimiert. In dieser Arbeit wurden hingegen die Parameter für die α-β-Filter unkomprimiert optimiert. Beim Double Pole Balancing ohne Geschwindigkeiten war die Verbesserung durch die Komprimierung nicht signifikant. Dies deckt sich mit dem

Pole Balancing mit Geschwindigkeiten			
Methode	Quelle	SPB	DPB
EP	Gomez u. a. [32]	-	307.200
AHC	Gomez u. a. [32]	189.500	-
PGRL	Gomez u. a. [32]	28.779	-
Q-MLP	Gomez u. a. [32]	2.056	-
SARSA-CABA	Gomez u. a. [32]	965	-
NEAT	Gomez u. a. [32]	743	3.600
SARSA-CMAC	Gomez u. a. [32]	540	-
CNE	Gomez u. a. [32]	352	22.100
SANE	Gomez u. a. [32]	302	12.600
ESP	Gomez u. a. [32]	289	3.800
RWG	Gomez u. a. [32]	199	474.329
CoSyNE	Gomez u. a. [32]	98	954
CMA-NeuroES	Heidrich-Meisner und Igel [37]	91	585
CoSyNE und DCT	Koutník u. a. [44]	2	258
CMA-ES	hier	25	279
CMA-ES und DOF	hier	2	195

Pole Balancing ohne Geschwindigkeiten			
Methode	Quelle	SPB	DPB
SARSA-CABA	Gomez u. a. [32]	15.617	-
SARSA-CMAC	Gomez u. a. [32]	13.562	-
Q-MLP	Gomez u. a. [32]	11.331	-
RWG	Gomez u. a. [32]	8.557	-
NEAT	Gomez u. a. [32]	1.523	6.929
SANE	Gomez u. a. [32]	1.212	262.700
CNE	Gomez u. a. [32]	724	76.906
ESP	Gomez u. a. [32]	589	7.374
CMA-NeuroES	Heidrich-Meisner und Igel [37]	192	860
CoSyNE und DCT	Koutník u. a. [44]	151	3.421
CoSyNE	Gomez u. a. [32]	127	1.249
CMA-ES, ANKF und DOF	Kassahun [41]	12	480
CMA-ES und ANKF	Kassahun u. a. [42]	-	302
CMA-ES und ANKF	hier	30	422
CMA-ES, ANKF und DOF	hier	14	420

Tabelle 6.8.: Vergleich von verschiedenen Lernverfahren anhand der Anzahl der Episoden, die zum Erlernen einen erfolgreichen Strategie beim Pole Balancing benötigt werden. Die Werte aus dieser Diplomarbeit sind über 1000 Experimente gemittelt, die Werte von Koutník u. a. [44] über 100 Experimente und alle anderen Werte über 50 Experimente.

Ergebnis von Kassahun [41]. Bei Koutník u. a. [44] war das komprimierte FCRNN sogar langsamer als ein normales FCRNN. Dennoch ist die hier verwendete Methode mit α-β-Filtern und CMA-ES, ob mit oder ohne Komprimierung, eines der aktuell schnellsten Verfahren zum Lösen von DPB ohne Geschwindigkeiten. Ein besseres Ergebnis konnte nur durch Kassahun u. a. [42] mit CMA-ES und α-β-Filtern erzielt werden. Allerdings wurden dort die Parameter der α-β-Filter nicht optimiert, sondern vorher festgelegt.

Beim Pole Balancing mit Geschwindigkeiten sind Verfahren, die die Gewichte eines neuronalen Netzes komprimieren im Reinforcement Learning zurzeit die besten. Insbesondere beim Single Pole Balancing ist eine weitere Verbesserung kaum möglich. Beim Pole Balancing ohne Geschwindigkeiten ist ein Verfahren mit α-β-Filtern normalen rekurrenten Netzen überlegen und beim SPB kann auch hier noch eine Verbesserung der Lerngeschwindigkeit durch eine Komprimierung der Parameter erreicht werden.

6.5.5. Erkenntnisse

CMA-ES ist ein effizientes Verfahren zur Optimierung komprimierter Gewichte.

Selbst bei sehr einfachen Problemen kann eine Komprimierung der Gewichte einen Vorteil bringen. Eine Voraussetzung dafür ist, dass die Lösung tatsächlich einfach repräsentierbar ist. Dies war zum Beispiel beim Single Pole Balancing mit Geschwindigkeiten der Fall.

Um die Stärken der Komprimierung herauszustellen, ist allerdings ein komplexeres Problem nötig.

6.6. Oktopus-Arm

6.6.1. Umgebung

Bei diesem Benchmark muss ein Oktopus-Arm gesteuert werden, sodass entweder Nahrung zu einem Mund geführt wird oder ein Punkt oder mehrere Punkte berührt werden. Besonderheiten dieses Benchmarks gegenüber dem vorherigen sind der mehrdimensio-

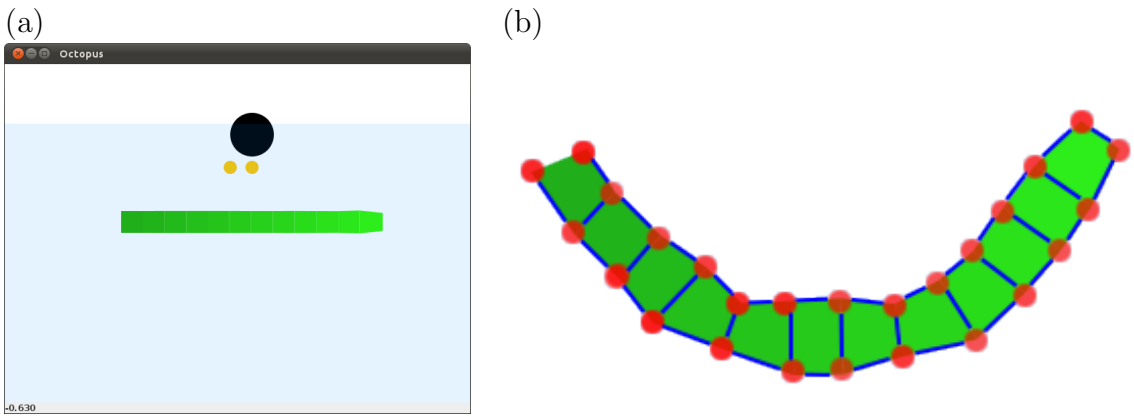


Abbildung 6.8.: (a) Oktopus-Arm und die Umgebung bei einem Nahrungsproblem. (b) Die Massepunkte (rote Punkte) und Muskeln (blaue Linien) des Arms.

nale Aktionsraum und die hohe Dimension des Zustandsraums. Der Oktopus-Arm besteht aus n beweglichen, zusammenhängenden Kammern, von denen jede drei Muskeln hat. Demzufolge hat der Aktionsraum die Dimension $3n$. Der Arm befindet sich im Wasser und deshalb wirkt auf ihn neben der Gravitationskraft auch die entgegengesetzte Auftriebskraft. Der Raum, in dem sich der Arm bewegt, ist zweidimensional. Alle Objekte sind Massepunkte, das heißt sie haben keine Ausdehnung. Der Arm selbst besteht aus $2n + 2$ Massepunkten. Diese sind in Abbildung 6.8 (b) rot markiert. Die Seiten der Kammern sind Muskeln, die zusammengezogen werden können. Sie sind in Abbildung 6.8 (b) blau markiert. Die von der linken Seite aus ersten beiden Massepunkte des Arms können sich nur gegenüberliegend auf einer Kreisbahn bewegen, sodass der Arm auf dieser Seite verankert ist. Der Zustand der Verankerung kann durch einen Winkel und eine Winkelgeschwindigkeit bestimmt werden. Jeder weitere Massepunkt hat eine Position, die aus x- und y- Komponente besteht und eine Geschwindigkeit in x- und y-Richtung, sodass der Zustandsraum die Dimension $8n+2$ hat. Bei einer sogenannten Nahrungsaufgabe kommen pro Nahrungsstück zwei Positions- und zwei Geschwindigkeitskomponenten hinzu.

Ich verwende hier eine Implementierung dieser Umgebung, die bei einem Reinforcement Learning Workshop im Rahmen der ICML 2006 verwendet wurde. Auf der dazugehörigen Internetseite [61] sind detailliertere Beschreibungen der Konfigurationsdateien und der Physik der Umgebung zu finden.

Die Umgebung kann in dieser Implementierung durch eine Konfigurationsdatei angepasst werden. Dabei können zwei verschiedene Aufgabentypen eingestellt werden. Bei einer Nahrungsaufgabe muss der Arm einen oder mehrere Massepunkte in einen Mund schieben (siehe Abbildung 6.8 (a)) und bei einer Zielaufgabe müssen ein Punkt oder eine Abfolge von Punkten durch den Arm berührt werden. Die Schwierigkeit bei einer Zielaufgabe ist, dass die Ziele nicht in dem Zustandsvektor angegeben sind, sodass nicht errechnet werden kann, wie weit der Arm von dem Ziel entfernt ist. Es kann lediglich über den Reward festgestellt werden, ob ein Ziel erreicht wurde. Hier werden zwei verschiedene Aufgaben betrachtet.

Aufgabe 1: Ich werde hier eine in Abbildung 6.8 (a) dargestellte Nahrungsaufgabe lösen. Der Arm besteht hierbei aus 12 Kammern und es gibt zwei Nahrungsstücke. Demzufolge hat ein Zustandsvektor 106 Komponenten und ein Aktionsvektor hat 36 Komponenten. Eine Episode läuft maximal für 1000 Schritte oder bis beide Nahrungsstücke in den Mund (schwarzer Kreis in Abbildung 6.8 (a)) geschoben wurden. Pro Schritt, in dem keines der beiden Nahrungsstücke (orange Kreise) in den Mund befördert wurde, ist der Reward -0,01. Wenn das rechte Nahrungsstück in den Mund geschoben wird, ist der Reward 7 und beim linken ist er 5. Daraus ergibt sich, dass der minimale Return ohne Diskontierung -10 ist und das theoretische Maximum 12 ist. Praktisch wird das Maximum nicht erreicht werden, da sich dazu die Nahrungsstücke beim Start bereits im Mund befinden müssten. Die verwendete Konfigurationsdatei ist in Anhang E.1 zu sehen.

Aufgabe 2: Bei dieser Aufgabe muss der Arm 20 Punkte in einer festgelegten Reihenfolge berühren. An welcher Stelle die Berührung stattfindet, ist dabei nicht wichtig.

Die Punkte sind zu vier Gruppen mit jeweils fünf Punkten zusammengefasst. Der Arm ist wie in Aufgabe 1 konfiguriert. Der Zustandsraum hat 98 Komponenten. Eine Episode läuft maximal für 300 Schritte. In jedem Schritt, in dem kein Ziel erreicht wurde, ist der Reward -0,01. Der Reward für ein Ziel kann 0,1, 0,3, 0,5 oder 1 sein. Der Return liegt zwischen -3 und 9,5. In Anhang E.2 ist die entsprechende Konfigurationsdatei zu finden.

6.6.2. Methode

Zur Lösung dieser Probleme wurde das gleiche Verfahren wie für Pole Balancing mit Geschwindigkeiten verwendet: die Strategie wird durch ein neuronales Netz repräsentiert, das durch CMA-ES optimiert wird. Das MLP wird auch hier mit orthogonalen Kosinusfunktionen komprimiert. Hier wird allerdings ein MLP mit einer versteckten Schicht verwendet. In dieser befinden sich zehn Neuronen. Die Aktivierungsfunktion ist in der versteckten Schicht Tangens Hyperbolicus. Da die Muskeln mit einem Wert zwischen 0 und 1 angesteuert werden müssen, ist die Aktivierungsfunktion in der Ausgabeschicht die logistische Funktion $g(a_f) = \frac{1}{1+\exp(-a_f)}$. Der Return, der hier einfach der in einer Episode akkumulierte Reward ist, soll durch CMA-ES maximiert werden.

6.6.3. Ergebnisse

Um die Lerngeschwindigkeit der verschiedenen Komprimierungen direkt vergleichen zu können, wurde für jedes Experiment der durchschnittliche Return in allen Episoden berechnet. Je höher dieser ist, desto schneller wird eine gute Strategie erlernt. Zudem wurde der maximalen Return aller Episoden ermittelt, um abschätzen zu können, wie gut die beste mögliche Strategie bei den betrachteten Komprimierungen jeweils war. Mit jeder Komprimierung wurden 20 Experimente durchgeführt. Für jede Komprimierung wurden der Durchschnitt und die Standardabweichung des durchschnittlichen Returns und das Maximum des maximalen Returns in allen Experimenten angegeben. Aus der Standardabweichung σ wurde der Standardfehler $SE = \frac{\sigma}{\sqrt{n}}$ berechnet, der eine Abschätzung der Genauigkeit des empirischen Mittelwerts angibt.

Aufgabe 1: In Tabelle 6.9 sind die Ergebnisse für Aufgabe 1 zu sehen. Abbildung 6.9 (a) zeigt den Verlauf des über alle Experimente gemittelten Returns für ausgewählte Konfigurationen. Der beste Return in allen Experimenten war 11,80. Durch manuelle Kontrolle des Arms mit sechs kombinierbaren, aber nicht dosierbaren möglichen Aktionen konnte ich nach ungefähr 200 Episoden nur den Return 11,74 erreichen.

Die Tabelle zeigt, dass eine Repräsentation der Gewichte durch orthogonale Kosinusfunktionen alleine einen Vorteil bringt. Dies scheint also für dieses Problem besonders günstig zu sein. Außerdem verkürzt eine Verringerung der Parameter die Lerndauer. Es werden weniger Episoden bis zum Lernen einer guten Strategie benötigt, was in Abbildung 6.9 (a) zu sehen ist.

Topologie	106-10-36 mit Bias						
Komprimierung	keine	107-11	80-11	40-11	20-11	10-11	5-11
Parameter	1466	1466	1196	796	596	496	446
	Return in 900 Episoden						
Durchschnitt	-4,52	2,49	1,89	2,06	2,28	2,73	**3,6**
Standardabweichung	1,02	0,82	1,32	0,86	0,6	1,3	1,08
Standardfehler	0,23	0,18	0,3	0,19	0,13	0,29	0,24
Maximum	11,71	11,73	11,72	11,80	11,72	11,73	11,72

Tabelle 6.9.: Ergebnisse von Aufgabe 1.

Topologie	98-10-36 mit Bias						
Komprimierung	keine	99-11	80-11	40-11	20-11	10-11	5-11
Parameter	1386	1386	1196	796	596	496	446
	Return in 5000 Episoden						
Durchschnitt	-0,82	**0,51**	0,45	0,04	0,01	0,39	-0,14
Standardabweichung	0,54	0,58	0,5	0,78	0,8	0,85	0,8
Standardfehler	0,12	0,13	0,11	0,17	0,18	0,19	0,18
Maximum	8,67	8,73	8,78	8,34	8,63	8,72	8,65

Tabelle 6.10.: Ergebnisse von Aufgabe 2.

Aufgabe 2: Die Ergebnisse der zweiten Aufgabe sind in Tabelle 6.10 aufgeführt. Eine Übersicht über den Verlauf des Returns bei der besten und schlechtesten Konfigurationen ist in Abbildung 6.9 (b) zu sehen.

Bei dieser Aufgabe habe ich durch manuelle Kontrolle maximal den Return 7,23 erreichen können, was deutlich schlechter als bei den erlernten Strategien ist. Anhand des maximalen Returns der verschiedenen Komprimierungen ist abzulesen, dass es prinzipiell bei allen Komprimierungen möglich ist, eine ähnlich gute Strategie zu erlernen.

Auch hier hat von allen geprüften Konfigurationen das MLP ohne Komprimierung mit Abstand den geringsten durchschnittlichen Return. Demzufolge ist auch für diese Aufgabe eine Repräsentation der Gewichtsfunktion durch orthogonale Kosinusfunktionen gut geeignet. Dass jedoch stärkere Komprimierungen einen geringeren durchschnittlichen Return zur Folge haben, habe ich nicht erwartet. Die Optimierung wird hier also durch die Reduktion der Parameter erschwert. Der Optimierungsalgorithmus bleibt eher in lokalen Minima oder flachen Regionen der Fitnessfunktion hängen. Andererseits ist dies auch ein Vorteil, wenn schnell eine sehr gute Region gefunden wird, denn dann verlässt der Optimierungsalgorithmus diese seltener. Die besten durchschnittlichen Returns mit Werten von 2,13 bis 2,43 wurden bei den Komprimierung 10-11, 20-11 und 40-11 erreicht. Allerdings gab es bei diesen Konfigurationen auch sehr viele schlechte durchschnittliche Returns, sodass diese Konfigurationen im Durchschnitt schlechter sind.

Es macht bei diesem Problem zwar Sinn, die Gewichtsfunktion durch eine gewichtete Summe von Kosinusfunktionen darzustellen, aber es ist besser die Anzahl der Parameter nicht zu reduzieren.

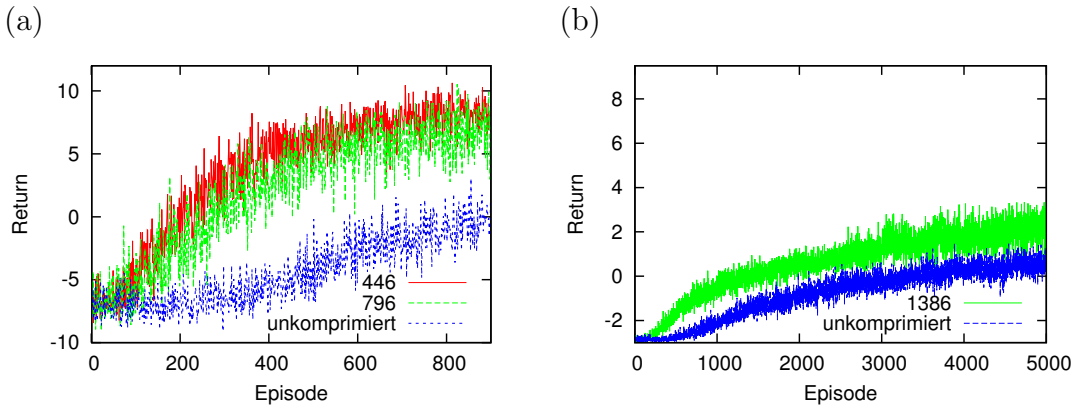


Abbildung 6.9.: Entwicklung des Returns (a) in Aufgabe 1 und (b) in Aufgabe 2. An jeder Stelle sind die Werte über alle mit der entsprechenden Konfiguration durchgeführten Experimente gemittelt. Verglichen werden hier unterschiedliche Komprimierungen. In der Legende ist die Anzahl der zur Komprimierung verwendeten Parameter angegeben.

6.6.4. Erkenntnisse

Anhand des Oktopus-Arms kann gezeigt werden, dass eine Komprimierung oder zumindest eine alternative Repräsentation der Gewichte eines MLP für komplexe Probleme Vorteile bringen kann.

Das hier entwickelte Verfahren könnte insbesondere auf realen robotischen Systemen eingesetzt werden, da dort ebenfalls häufig viele Zustandsinformationen zur Verfügung stehen, viele Aktionen möglich sind und oft eine nichtlineare Strategie erforderlich ist, die durch ein MLP repräsentiert werden könnte.

7. Ergebnis

7.1. Zusammenfassung der Ergebnisse

In dieser Arbeit wurde ein Verfahren zur Komprimierung der Gewichte allgemeiner Feedforward-Netze vorgestellt und gezeigt, wie der Gradient und die Hesse-Matrix nach den Komprimierungsparametern zu berechnen sind. Das vorgestellte Verfahren ist nach dem Trainieren nicht langsamer als ein normales neuronales Netz, da die Gewichte nach dem Training nicht mehr neu berechnet werden müssen.

Es hat sich herausgestellt, dass die Komprimierung der Eingabe einer Schicht bei einem MLP identisch zu einer Komprimierung der Gewichte dieser Schicht ist. Bei einem SLP ist also sogar die Komprimierung aller Gewichte identisch zur Komprimierung der Eingabe.

Eine andere mögliche Interpretation dieser Art der Komprimierung ist, dass vor einer komprimierten Schicht eine weitere Schicht mit festen Gewichten und ohne Aktivierungsfunktion eingeschoben wird.

Eine Komprimierung der Gewichte eines neuronalen Netzes kann aus verschiedenen Gründen Sinn machen:

- Wenn die Anzahl der Parameter deutlich geringer als die Anzahl der Gewichte ist, wird die Anzahl der Iterationen, die zum Finden einer Lösung nötig sind, geringer sein.

- Wenn weniger Parameter optimiert werden, können eher Optimierungsalgorithmen mit hoher Speicher- und Zeitkomplexität, wie zum Beispiel LMA, angewandt werden.

- Wenn das Rauschen in den Daten groß ist, kann es bei einem unkomprimierten MLP eher zu Overfitting auf dem Trainingsdatensatz kommen. Durch eine Komprimierung kann diese Gefahr reduziert werden.

Wie für alle Lernverfahren gilt auch für diese Erweiterung das No-Free-Lunch-Theorem [93], welches besagt, dass es keine a priori besseren Lernverfahren gibt, sondern die Qualität eines Lernverfahrens nur anhand eines zugrunde liegenden Problems ermittelt werden kann. Jedes Verfahren ist für einige Probleme gut geeignet und für andere nicht.

Zu beachten beim Einsatz dieses Verfahrens ist, dass die Daten oder die Gewichte des neuronalen Netzes eine Bedingungen erfüllen müssen, um die genannten Effekte nutzen zu können: sie müssen in irgendeiner Basis miteinander korrelieren, das heißt teilweise redundant, komprimierbar oder sogar dünn besetzt sein, damit sie komprimierter repräsentiert werden können.

Es gibt auch Gründe die gegen eine Komprimierung sprechen können:

- Durch die indirekte Gewichtsrepräsentation wird das Optimierungsproblem in den meisten Fällen schwieriger. Demzufolge muss normalerweise eine starke Komprimierung möglich sein, damit die Anzahl der benötigten Iterationen des Optimierungsalgorithmus reduziert wird.

- Bei einer Komprimierung von mehr als nur der ersten Schicht eines MLP kommt durch die Erweiterung des Backpropagation-Verfahrens und die Generierung der Gewichte zusätzlicher Aufwand hinzu.

Der erste Punkt ist eingeschränkt gültig. Wie anhand des Oktopus-Arms gezeigt werden konnte, gibt es Probleme, bei denen eine indirekte Repräsentation der Gewichte auch ohne eine Komprimierung vorteilhaft ist. Es ist allerdings schwer, solch eine Repräsentation ohne Experimente zu finden. Wenn nur die erste Schicht komprimiert wird, können stattdessen die Daten komprimiert werden und der zusätzliche Aufwand für Gewichtsgenerierung und Backpropagation entfällt. Bei der Verwendung von Neuroevolution entfällt der zusätzliche Aufwand für Backpropagation, da keine Ableitung berechnet wird.

7.2. Ausblick

In dieser Diplomarbeit sind einige Aspekte des entwickelten Verfahrens nicht behandelt worden. Dazu gehört die Wahl der Basisfunktionen. Es ist nicht leicht zu bestimmen, wann orthogonale Funktionen und wann zufällig generierte Werte gewählt werden sollten und welche orthogonalen Funktionen oder welche Verteilung gewählt werden sollten. In dieser Arbeit wurden hauptsächlich orthogonale Kosinusfunktionen verwendet und bei sehr großen Eingabevektoren wurden zufällige Werte generiert. Anhand weiterer Datensätze und Probleme sollten diese Aspekte näher untersucht werden. Hierbei sollte unterschieden werden zwischen globalen Approximationen (zum Beispiel Kosinusfunktionen oder Gauß-Verteilung), bei denen die Änderung eines Parameters alle Gewichte eines Neurons ändert, und lokalen Approximationen (zum Beispiel Wavelets oder die in Gleichung 3.5.56 aufgeführte Verteilung).

Des Weiteren ist das hier entwickelte Verfahren nicht auf MLPs beschränkt, deshalb wäre es interessant dieses auf andere neuronale Netze zu übertragen:

- Sogenannte Higher-Order Neural Networks (HONN) sind nach Spirkovska und Reid [81] insbesondere dazu geeignet Invarianz gegenüber Rotationen, Translation und Skalierung zu lernen. Dabei ist die Anzahl der Gewichte allerdings sehr groß, weshalb Methoden erforderlich sind, die die Anzahl der Gewichte reduzieren, damit HONNs zum Beispiel für Bilder eingesetzt werden können. Artyomov und Yadid-Pecht [1] haben dazu bereits ein Verfahren entwickelt. Das hier entwickelte Komprimierungsverfahren könnte auch auf HONNs übertragen werden.

- Bei Convolutional Neural Networks werden die Gewichte zwar schon stark reduziert, allerdings gibt es auch dort eventuell noch Potenzial. Es können entweder die Parameter der Filter komprimiert werden oder die vollständig verbundenen Schichten, die in den höheren Schichten verwendet werden und denen eines MLP gleichen.

In dieser Diplomarbeit wurde davon ausgegangen, dass die Gewichte eines Neurons stark miteinander korrelieren und deshalb eine gemeinsame Komprimierung dieser Gewichte besser funktioniert. Es könnte jedoch auch Probleme geben, bei denen eine Komprimierung aller Gewichte in einer einzigen Funktion oder eine Komprimierung der Gewichte der Ausgänge eines Neurons besser geeignet sind. Eine weitere Alternative ist die schichtweise Komprimierung.

Hier wurde die Hesse-Matrix hergeleitet, ohne dass diese sinnvoll angewendet werden konnte. Es gibt jedoch mögliche Anwendungszwecke, die hier noch nicht besprochen wurden:

- Die Hesse-Matrix wird zur Optimierung bei Bayesian Neural Networks [11, Seite 277 ff.] benötigt. Bayesian Neural Networks sind eine Form regularisierter neuronaler Netze. Durch das Optimierungsverfahren soll Overfitting vermieden werden.

- Es gibt Varianten von SGD bei denen die Hesse-Matrix zur Beschleunigung der Optimierung eingesetzt werden kann. SGD bleibt normalerweise nicht in lokalen Minima hängen, da die Reihenfolge der zur Gewichtsanpassung verwendeten Instanzen zufällig ist. Dadurch eignet sich dieses Optimierungsverfahren besonders gut zur Verwendung der Hesse-Matrix, da mit der Hesse-Matrix sehr schnell lokale Minima erreicht werden können.

In dieser Arbeit wurden zwar bereits einige Anwendungen vorgestellt, allerdings gibt es noch sehr viele Probleme die eine große Anzahl von Eingabekomponenten haben und anhand derer dieses Verfahren getestet werden könnte. Beispiele dafür sind die Folgenden:

- Norb-Datensatz (`http://www.cs.nyu.edu/~ylclab/data/norb-v1.0`): Ein Datensatz bei dem Grauwertbilder klassifiziert werden sollen. Die Bilder bestehen aus 108×108 Pixel.

- CIFAR-10-Datensatz und CIFAR-100-Datensatz (`http://www.cs.utoronto.ca/~kriz/cifar.html`): Die Datensätze bestehen aus Farbbildern mit 32×32 Pixel, die in 10 beziehungsweise 100 Klassen unterteilt werden sollen.

- German Traffic Sign Recognition Benchmark (`http://benchmark.ini.rub.de`): Die Bilder in diesem Datensatz haben variable Größen. Die maximale Größe ist 250×250 Pixel. Die Bilder sind farbig.

Im Reinforcement Learning gibt es bisher wenige Benchmarks, bei denen eine sehr große Anzahl von Eingabekomponenten vorhanden ist. Deshalb eignen sich hier insbesondere reale Anwendungen mit autonomen Maschinen, wie zum Beispiel Robotern, zur Anwendung einer Gewichtskomprimierung.

A. Mathematische Symbole

In dieser Arbeit werden viele mathematische Symbole verwendet. Da die Bezeichnungen im Verlauf des Textes in der Regel konsistent sind, ist hier eine Übersicht über die Bedeutungen der einzelnen Zeichen.

Symbol	**Bedeutung**
$D \in \mathbb{N}$	Dimension des Eingabevektors, Anzahl der Neuronen in der Eingabeschicht
$d \in \{1, \ldots, D\}$	Index eines bestimmten Eingabeneurons
$F \in \mathbb{N}$	Dimension des Ausgabevektors, Anzahl der Neuronen in der Ausgabeschicht
$f \in \{1, \ldots, F\}$	Index eines bestimmten Ausgabeneurons
$\mathbf{x} \in \mathbb{R}^D$	Eingabevektor
x_i	i-te Komponente des Eingabevektors
$\mathbf{y} \in \mathbb{R}^F$	Ausgabevektor
y_i	i-te Komponente des Ausgabevektors
$K \in \mathbb{N}$	Anzahl aller Gewichte eines neuronalen Netzes
$\mathbf{w} \in \mathbb{R}^K$	Gewichtsvektor
w_{ji}	Gewicht der Verbindung von Neuron i zu Neuron j
$g : \mathbb{R} \to \mathbb{R}$	Aktivierungsfunktion
a_i	Aktivierung von i vor Anwendung von g
$z_i = g(a_i)$	Ausgabe des Neurons i
N	Größe des Trainingsdatensatzes
T	Trainingsdatensatz
$\mathbf{x}^{(n)}$	Eingabevektor einer Trainingsinstanz n
$\mathbf{t}^{(n)}$	Erwartete Ausgabe einer Trainingsinstanz n
E	Fehlerfunktion des gesamten Trainingsdatensatzes
E_n	Fehlerfunktion einer bestimmten Instanz $n \in \{1, \ldots, N\}$
$\mathbf{g}$	Gradient der Fehlerfunktion, ohne Komprimierung
$\mathbf{H}$	Hesse-Matrix der Fehlerfunktion, ohne Komprimierung
$\delta_i = \frac{\partial E_n}{\partial a_i}$	durch Backpropagation berechneter Fehler eines Neurons i
$\gamma_{ji} = \frac{\partial a_j}{\partial a_i}$	durch Vorwärtspropagation erhaltene Werte zur Berechnung von $\mathbf{H}$
$\beta_{ji} = \frac{\partial \delta_j}{\partial a_i}$	durch Backpropagation erhaltene Werte zur Berechnung von $\mathbf{H}$
$f_{w_j} : [0, 1] \to \mathbb{R}$	Funktion zur Approximation der Gewichte des Neurons j
$M_j \in \mathbb{N}$	Anzahl der Parameter zur Approximation der Gewichte w_{ji}
$m \in \{1, \ldots, M_j\}$	Index eines Parameters zur Gewichtsapproximation
α_{jm}	m-ter Parameter zur Komprimierung der Gewichte von Neuron j
$\Phi_m(t_i)$	m-te Funktion zur Komprimierung des Gewichtes von Neuron i
$I \in \mathbb{N}$	Anzahl der Eingabeneuronen i eines Neurons j
$L \in \mathbb{N}$	Anzahl der optimierbaren Parameter eines KNN
$\overline{\mathbf{g}}$	Gradient der Fehlerfunktion, mit Komprimierung
$\overline{\mathbf{H}}$	Hesse-Matrix der Fehlerfunktion, mit Komprimierung

Symbol	Bedeutung
$\gamma_{ijm} = \frac{\partial a_i}{\partial \alpha_{jm}}$	ähnlich der γ_{ji}, verwendet zur Berechnung von $\overline{\overline{\mathbf{H}}}$
$\beta_{ijm} = \frac{\partial}{\partial \alpha_{jm}} \left(\frac{\partial E_n}{\partial a_i} \right)$	ähnlich der β_{ji}, verwendet zur Berechnung von $\overline{\overline{\mathbf{H}}}$
$P(a\|b)$	bedingte Wahrscheinlichkeit von a unter der Voraussetzung b
A	Aktionsraum
$a_t \in A$	zum Zeitpunkt t ausgeführte Aktion
S	Zustandsraum
$s_t \in S$	Zustand zum Zeitpunkt t
$r_t \in \mathbb{R}$	Reward (Belohnung) nach Ausführung von a_t
R_t	Return (akkumulierter Reward) ab Zeitpunkt t
γ	Diskontierungsfaktor des Return-Modells
$Q(s,a) : S \times A \to \mathbb{R}$	Nutzenfunktion, die den erwarteten Return angibt
$\pi(s,a) : S \times A \to [0,1]$	probabilistische Strategie
$\pi(s) : S \to A$	deterministische Strategie
$\mathcal{N}(\mu, \sigma)$	Normalverteilungen mit Mittelwert μ und Standardabweichung σ

B. Abkürzungen

Die wichtigsten Abkürzungen werden hier mit ihrer Bedeutung aufgeführt.

Ich habe zur verkürzten Darstellung der Topologie eines MLP die folgende Notation verwendet

$$D - H_1 - H_2 - \ldots - F$$

Dabei ist D die Anzahl der Eingabekomponenten, $H_1, H_2, \ldots$ sind die Anzahlen der Knoten in den versteckten Schichten und F ist die Anzahl der Ausgabekomponenten. Zusätzlich muss hierbei angegeben werden, ob ein Bias in jeder Schicht verwendet wird oder nicht.

Analog habe ich die Komprimierung des MLP angeben durch die Notation

$$M^{(1)} - M^{(2)} - \ldots$$

Dabei ist $M^{(1)}$ die Anzahl der verwendeten Parameter zur Komprimierung der Gewichte von allen Eingabekomponenten zu einem der Neuronen in der ersten versteckten Schicht und $M^{(2)}$ die Anzahl der Parameter zur Komprimierung der Gewichte von allen Knoten der ersten versteckten Schicht zu einem Neuron der zweiten versteckten Schicht und so weiter.

Abkürzung	**Bedeutung**
ANKF	Augmented Neural Network with Kalman Filter
BCI	Brain-Computer-Interface
CE	Cross Entropy (Fehlerfunktion), hier: $E_{CE} = -\sum_{n=1}^{N}\sum_{f=1}^{F} t_f^{(n)} \ln y_f^{(n)}$
CG	Conjugate Gradient
CMA-ES	Covariance Matrix Adaption Evolution Strategies
CNN	Convolutional Neural Network
CoSyNE	Cooperative Synapse Neuroevolution
CUDA	Compute Unified Device Architecture
DBN	Deep Belief Network
DOF	Discrete Orthogonal Functions
DPB	Double Pole Balancing
EEG	Elektroenzephalografie
EKP	ereigniskorreliertes Potenzial
ERP	Event-Related Potential
FCRNN	Fully Connected Recurrent Neural Network
FIR	Finite Impulse Response (Filter)
GPU	Graphics Processing Unit
HONN	Higher-Order Neural Network
KNN	künstliches neuronales Netz
LMA	Levenberg-Marquardt-Algorithmus
MDL	Minimum Description Length

Abkürzung	Bedeutung
MLP	Multilayer Perceptron, Mehrschichtiges Perzeptron
MQI	Monitored Q Iteration
MSE	Mean Squared Error, hier: $E_{MSE} = \frac{2}{N} E_{SSE}$
NEL	Network Encoding Language
NFQ	Neural Fitted Q Iteration
POMDP	Partially Observable Markov Decision Process
RBF	Radial Basis Functions
RBM	Restricted Boltzman Machine
RIP	Restricted Isometry Property
Rprop	Resilient Backpropagation
SGD	Stochastic Gradient Descent
SLP	Single Layer Perceptron
SQP	Sequential Quadratic Programming
SPB	Single Pole Balancing
SSE	Sum of Squared Errors, hier: $E_{SSE} = \frac{1}{2} \sum_{n=1}^{N} \|\|y^{(n)} - t^{(n)}\|\|^2$
SVM	Support Vector Machines

C. Verwendete mathematische Regeln

Die folgenden mathematischen Regeln und Sätze werden in dieser Arbeit oftmals verwendet, ohne dass explizit darauf hingewiesen wird.

C.1. Ableitungen

Ableitungsregeln für partielle Ableitungen: Gegeben seien zwei Funktionen von x: $f(x), g(x)$.

$$\text{Summenregel:} \quad \frac{\partial f+g}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x} \tag{C.1.1}$$

$$\text{Produktregel:} \quad \frac{\partial f \cdot g}{\partial x} = \frac{\partial f}{\partial x} \cdot g + f \cdot \frac{\partial g}{\partial x} \tag{C.1.2}$$

Kettenregel für partielle Ableitungen: Gegeben sei eine Funktion $f(g_1, \ldots, g_n)$ mit $n \in \mathbb{N}$ und jede Funktion $g_1, \ldots, g_n$ ist selbst wiederum eine Funktion von x.

Die partielle Ableitung $\frac{\partial f}{\partial x}$ lässt sich nach der **Kettenregel für partielle Ableitungen** [18, Seite 413, Abschnitt 6.2.3.1.] wie folgt berechnen:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}. \tag{C.1.3}$$

Schwarzscher Vertauschungssatz: Gegeben sei eine Funktion $f(a_1, \ldots, a_n)$ mit $n \in \mathbb{N}$. Wenn die Ableitung an dem betrachteten Punkt stetig ist, spielt die Reihenfolge der zweiten Ableitung nach dem **Schwarzschen Vertauschungssatz** [18, Seite 411, Abschnitt 6.2.2.2.] keine Rolle, das heißt für beliebige $i, j \in \{1, \ldots, N\}$ gilt

$$\frac{\partial^2 f}{\partial a_i \partial a_j} = \frac{\partial^2 f}{\partial a_j \partial a_i}. \tag{C.1.4}$$

C.2. Matrizen

Für beliebige Matrizen $A, B \in \mathbb{R}^{n \times n}$ gilt

$$(AB)^T = B^T A^T \quad \text{[18, Seite 265, Abschnitt 4.1.5]} \tag{C.2.5}$$

C.3. Signifikanztest

Zum Testen der Signifikanz eines Ergebnisses verwende ich einen Zwei-Stichproben-z-Test [23]. Angenommen aus zwei Normalverteilungen wurden Stichproben der Größe n_1 und n_2 mit den Mittelwerten μ_1 und μ_2 und den Standardabweichungen σ_1 und σ_2 gezogen. Um zu zeigen, dass beide Stichproben mit einem Signifikanzniveau von α von unterschiedlichen Verteilungen erzeugt wurden, wird als Nullhypothese angenommen, dass die Differenz der tatsächlichen Mittelwerte 0 ist. Danach wird versucht, die Nullhypothese mit der Wahrscheinlichkeit $1 - \alpha$ zu widerlegen. Dazu muss zunächst ein z-Wert durch

$$z = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{C.3.6}$$

berechnet werden. Danach muss geprüft werden, ob der errechnete Wert den für das gewünschte Signifikanzniveau erforderlichen Wert überschreitet. Für $\alpha = 0,01$ muss zum Beispiel die Bedingung $z > 2,33$ erfüllt sein [18, Seite 1123, Abschnitt 21.17.1]. In diesem Fall wäre die Annahme, dass die Stichproben aus verschiedenen Verteilungen generiert wurde mit der Wahrscheinlichkeit 0,01 falsch.

D. Implementierung

D.1. Quellcode-Auszug

Mit Hilfe der Matrix-Bibliothek Eigen [33] lässt sich das Backpropagation-Verfahren elegant in Quellcode übertragen. Hier sind Auszüge aus meiner Implementierung zu sehen. Der Quellcode wurde zum Teil ebenfalls veröffentlicht unter `https://github.com/AlexanderFabisch/OpenANN`.

```
1  void generateWeights()
2  { // L − Anzahl der Schichten ohne Eingabeschicht
3    for (int l = 0; l < L; l++) weights[l] = parameters[l] ∗ orthogonalFunctions[l];
4  }
```

Listing D.1: Gewichtsgenerierung

```
1  Vector forwardPropagate(const Vector& x)
2  { // D − Dimension der Eingabe, U[l] − Anzahl der Neuronen (ohne Bias) in
       Schicht l aus {0,...,L}
3    activations[0] = x;
4    for (int d = 0; d < D; d++) outputs[0](d) = activations[0](d);
5    for (int l = 0, nl = 1; l <= L; l++, nl++) {
6      activations[nl] = weights[l] ∗ outputs[l];
7      for (int j = 0; j < U[nl]; j++)
8        if (nl <= L) outputs[nl](j) = tanh(activations[nl](j));
9        else outputs[nl](j) = activations[nl](j);
10   }
11   return outputs[L+1];
12 }
```

Listing D.2: Vorwärtspropagation

```
1  void backpropagate(const Vector& t)
2  {
3    deltas[L+1] = outputs[L+1] − t;
4    for (int l = L; l > 0; l−−) {
5      for (int j = 0; j < U[l]; j++) derivatives[l](j) = 1.0 − outputs[l](j)∗outputs[l](
           j);
6      errors[l] = weights[l].transpose() ∗ deltas[l+1];
7      for (int j = 0; j < U[l]; j++) deltas[l](j) = derivatives[l](j) ∗ errors[l](j);
8    }
```

```
9     for (int l = 0; l <= L; l++) weightDerivatives[l] = deltas[l+1] * outputs[l].
          transpose();
10    for (int l = 0; l < L; l++)
11      parameterDerivatives[l] = weightDerivatives[l] * orthogonalFunctions[l].
            transpose();
12  }
```

Listing D.3: Backpropagation

D.2. Aktivierungsfunktionen

Aktivierungsfunktionen sind ein kritischer Punkt in der Implementierung von künstlichen neuronalen Netzen. Hier treten leicht numerische Fehler durch eine naive Implementierung auf. Deshalb gebe ich hier die Implementierung zweier Aktivierungsfunktionen an.

Logistische Funktion: Die naive Implementierung würde $g(a_i) = \frac{1}{1+\exp(-a_i)}$ direkt berechnen. Bei Gleitkommazahlen einfacher Genauigkeit darf dann a_i nicht kleiner als -88 und bei Gleitkommazahlen doppelter Genauigkeit nicht kleiner als -709 sein. Ansonsten ist das Ergebnis der Aktivierungsfunktion `NaN`. Eine sicherere Implementierung ist die Folgende [63, How to avoid overflow in the logistic function?]:

```
1  double logistic(double x)
2  {
3    if(x < −45.0) return 0.0;
4    else if(x > 45.0) return 1.0;
5    else return 1.0/(1.0+exp(−x));
6  }
```

Listing D.4: Logistische Funktion

Softmax-Aktivierungsfunktion: Hier tritt das gleiche Problem auf. Dieses ist allerdings nicht so einfach zu lösen. Die naive Implementierung wäre:

```
1  Vector softmax(const Vector& a)
2  {
3    Vector y(a.rows());
4    for(int f = 0; f < y.rows(); f++) y(f) = std::exp(a(f));
5    return y / y.sum();
6  }
```

Listing D.5: Naive Softmax-Implementierung

Die Softmax-Funktion lässt sich allerdings auch anders darstellen. Es folgt der Beweis für die Gleichheit der folgenden beiden Repräsentationen der Softmax-Funktion. Sei $m \in \mathbb{R}$ eine beliebige Konstante. Zu zeigen ist, dass

$$y_f = \frac{\exp(a_f)}{\sum_{f'} \exp(a_{f'})} = \frac{\exp(a_f - m)}{\sum_{f'} \exp(a_{f'} - m)} \quad [60]. \tag{D.2.1}$$

Dieses Problem ist mit Rechenregeln für Logarithmen und Exponentialfunktionen lösbar.

$$\begin{aligned}
& y_f && = \frac{\exp(a_f)}{\sum_{f'} \exp(a_{f'})} && \text{(D.2.2)} \\
\Leftrightarrow \quad & \ln(y_f) && = \ln\left(\frac{\exp(a_f)}{\sum_{f'} \exp(a_{f'})}\right) && \text{(D.2.3)} \\
& && = \ln\left(\exp(a_f)\right) - \ln\left(\sum_{f'} \exp(a_{f'})\right) && \text{(D.2.4)} \\
& && = a_f - \ln\left(\sum_{f'} \exp(a_{f'} + m - m)\right) && \text{(D.2.5)} \\
& && = a_f - \ln\left(\sum_{f'} \exp(a_{f'} - m) \exp(m)\right) && \text{(D.2.6)} \\
& && = a_f - \left[\ln\left(\sum_{f'} \exp(a_{f'} - m)\right) + m\right] && \text{(D.2.7)} \\
& && = (a_f - m) - \ln\left(\sum_{f'} \exp(a_{f'} - m)\right) && \text{(D.2.8)} \\
\Leftrightarrow \quad & y_f && = \frac{\exp(a_f - m)}{\sum_{f'} \exp(a_{f'} - m)} && \text{(D.2.9)}
\end{aligned}$$

Wenn $m = \max_f a_f$ gewählt wird, liefert `std::exp` bei der maximalen Eingabe die Ausgabe 1. Theoretisch sind die Ergebnisse beider Implementierungen gleich. Bei üblichen Zahlenrepräsentationen werden mit dieser Implementierung allerdings einige kleinere Komponenten 0, sodass diese wegfallen. Die Anpassung ist leicht zu implementieren:

```
1 Vector softmax(const Vector& a)
2 {
3   Vector y(a.rows()); const double max = y.maxCoeff();
4   for(int f = 0; f < y.rows(); f++) y(f) = std::exp(a(f) - max);
5   return y / y.sum();
6 }
```

Listing D.6: Softmax-Implementierung

E. Konfiguration der Oktopus-Arm-Umgebung

E.1. Aufgabe 1

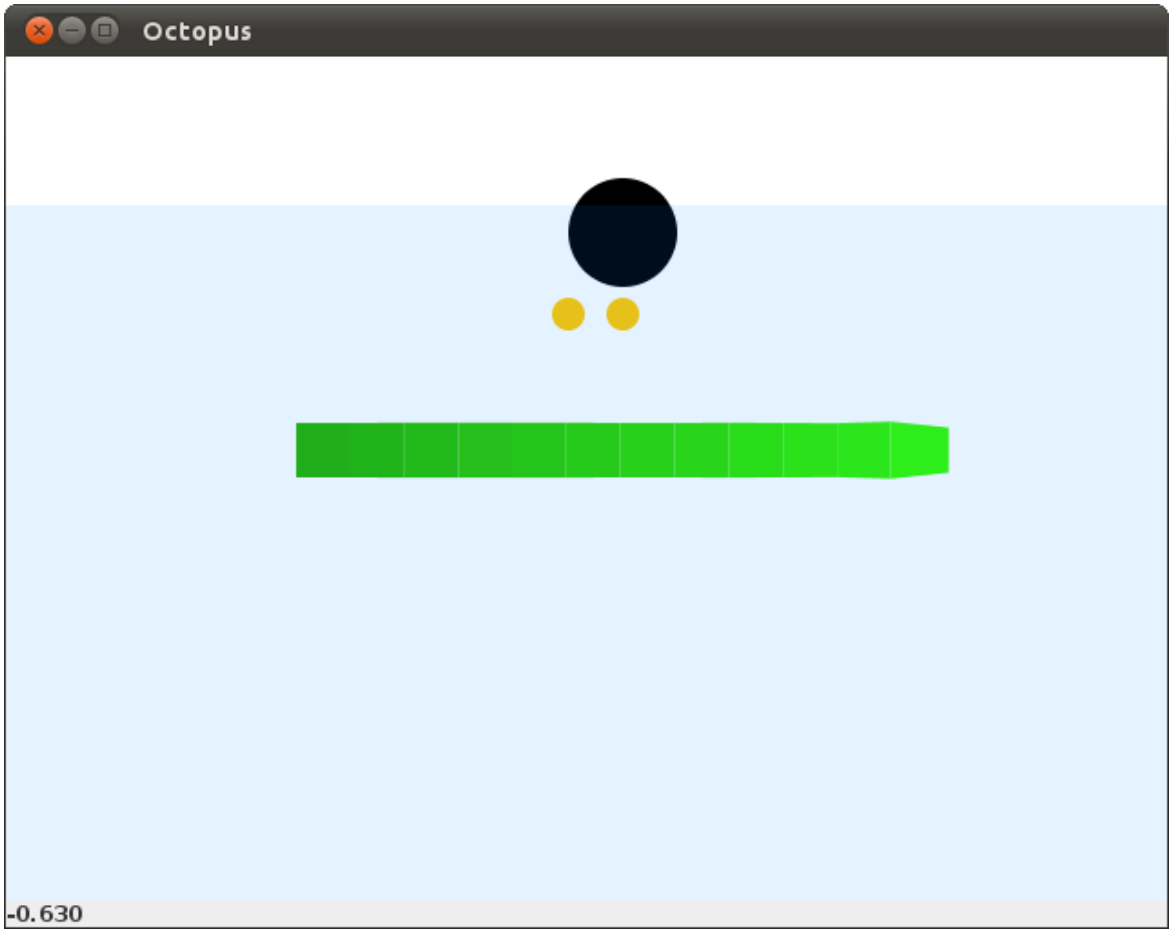


Abbildung E.1.: Umgebung in Aufgabe 1.

Listing E.1: settings-food.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <config>
3 <constants>
4     <frictionTangential>0.2</frictionTangential>
5     <frictionPerpendicular>1</frictionPerpendicular>
6     <pressure>10</pressure>
```

```
7     <gravity>0.01</gravity>
8     <surfaceLevel>5</surfaceLevel>
9     <buoyancy>0.08</buoyancy>
10    <muscleActive>0.1</muscleActive>
11    <musclePassive>0.05</musclePassive>
12    <muscleNormalizedMinLength>0.4</muscleNormalizedMinLength>
13    <muscleDamping>−0.3</muscleDamping>
14    <repulsionConstant>.04</repulsionConstant>
15    <repulsionPower>2.3</repulsionPower>
16    <repulsionThreshold>.7</repulsionThreshold>
17  </constants>
18  <environment>
19  <foodTask timeLimit="1000" stepReward="−0.01">
20    <mouth x="5" y="3.5" width="2" height="2" />
21    <food velocity="0␣0" position="5␣3" mass="1" reward="5" />
22    <food velocity="0␣0" position="6␣3" mass="2" reward="7" />
23  </foodTask>
24  <arm>
25    <nodePair>
26      <upper velocity='0␣0' position='0␣1' mass='1' />
27      <lower velocity='0␣0' position='0␣0' mass='1' />
28    </nodePair>
29    <nodePair>
30      <upper velocity='0␣0' position='1␣1' mass='0.9900990099' />
31      <lower velocity='0␣0' position='1␣0' mass='0.9900990099' />
32    </nodePair>
33    <nodePair>
34      <upper velocity='0␣0' position='2␣1' mass='0.9803921569' />
35      <lower velocity='0␣0' position='2␣0' mass='0.9803921569' />
36    </nodePair>
37    <nodePair>
38      <upper velocity='0␣0' position='3␣1' mass='0.9708737864' />
39      <lower velocity='0␣0' position='3␣0' mass='0.9708737864' />
40    </nodePair>
41    <nodePair>
42      <upper velocity='0␣0' position='4␣1' mass='0.9615384615' />
43      <lower velocity='0␣0' position='4␣0' mass='0.9615384615' />
44    </nodePair>
45    <nodePair>
46      <upper velocity='0␣0' position='5␣1' mass='0.9523809524' />
47      <lower velocity='0␣0' position='5␣0' mass='0.9523809524' />
48    </nodePair>
49    <nodePair>
50      <upper velocity='0␣0' position='6␣1' mass='0.8433962264' />
51      <lower velocity='0␣0' position='6␣0' mass='0.8433962264' />
52    </nodePair>
53    <nodePair>
54      <upper velocity='0␣0' position='7␣1' mass='0.8345794393' />
55      <lower velocity='0␣0' position='7␣0' mass='0.8345794393' />
56    </nodePair>
57    <nodePair>
58      <upper velocity='0␣0' position='8␣1' mass='0.8259259259' />
59      <lower velocity='0␣0' position='8␣0' mass='0.8259259259' />
60    </nodePair>
61    <nodePair>
```

```
62      <upper velocity='0 0' position='9 1' mass='0.8174311927' />
63      <lower velocity='0 0' position='9 0' mass='0.8174311927' />
64    </nodePair>
65    <nodePair>
66      <upper velocity='0 0' position='10 1' mass='0.7090909091' />
67      <lower velocity='0 0' position='10 0' mass='0.7090909091' />
68    </nodePair>
69    <nodePair>
70      <upper velocity='0 0' position='11 1' mass='0.7009009009' />
71      <lower velocity='0 0' position='11 0' mass='0.7009009009' />
72    </nodePair>
73    <nodePair>
74      <upper velocity='0 0' position='12 1' mass='0.7928571429' />
75      <lower velocity='0 0' position='12 0' mass='0.7928571429' />
76    </nodePair>
77  </arm>
78  </environment>
79  </config>
```

E.2. Aufgabe 2

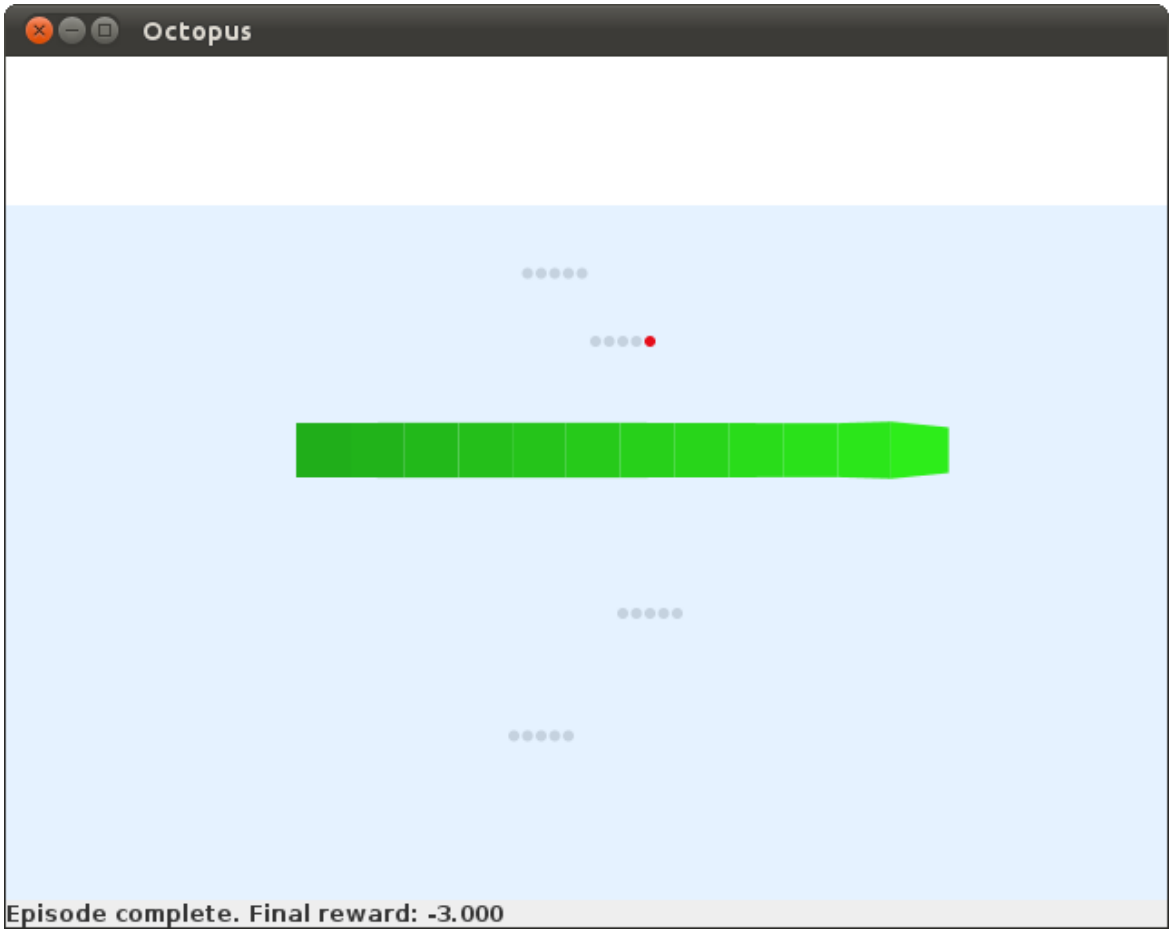


Abbildung E.2.: Umgebung in Aufgabe 2.

Listing E.2: settings-target.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
<constants>
    <frictionTangential>0.02</frictionTangential>
    <frictionPerpendicular>0.01</frictionPerpendicular>
    <pressure>10</pressure>
    <gravity>0.01</gravity>
    <surfaceLevel>5</surfaceLevel>
    <buoyancy>0.02</buoyancy>
    <muscleActive>0.1</muscleActive>
    <musclePassive>0.05</musclePassive>
    <muscleNormalizedMinLength>0.4</muscleNormalizedMinLength>
    <muscleDamping>-0.3</muscleDamping>
    <repulsionConstant>.04</repulsionConstant>
    <repulsionPower>2.3</repulsionPower>
    <repulsionThreshold>.7</repulsionThreshold>
</constants>
```

```
<environment>
<targetTask timeLimit="300" stepReward="−0.01">
    <sequence>
        <target position="6.5 2.5" reward="0.1" />
        <target position="6.25 2.5" reward="0.1" />
        <target position="6 2.5" reward="0.1" />
        <target position="5.75 2.5" reward="0.1" />
        <target position="5.5 2.5" reward="0.1" />
        <target position="4.25 3.75" reward="0.3" />
        <target position="4.5 3.75" reward="0.3" />
        <target position="4.75 3.75" reward="0.3" />
        <target position="5 3.75" reward="0.3" />
        <target position="5.25 3.75" reward="0.3" />
        <target position="7 −2.5" reward="0.5" />
        <target position="6.75 −2.5" reward="0.5" />
        <target position="6.5 −2.5" reward="0.5" />
        <target position="6.25 −2.5" reward="0.5" />
        <target position="6 −2.5" reward="0.5" />
        <target position="4 −4.75" reward="1.0" />
        <target position="4.25 −4.75" reward="1.0" />
        <target position="4.5 −4.75" reward="1.0" />
        <target position="4.75 −4.75" reward="1.0" />
        <target position="5 −4.75" reward="1.0" />
    </sequence>
</targetTask>
<arm>
  <nodePair>
    <upper velocity='0 0' position='0 1' mass='1' />
    <lower velocity='0 0' position='0 0' mass='1' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='1 1' mass='0.9900990099' />
    <lower velocity='0 0' position='1 0' mass='0.9900990099' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='2 1' mass='0.9803921569' />
    <lower velocity='0 0' position='2 0' mass='0.9803921569' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='3 1' mass='0.9708737864' />
    <lower velocity='0 0' position='3 0' mass='0.9708737864' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='4 1' mass='0.9615384615' />
    <lower velocity='0 0' position='4 0' mass='0.9615384615' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='5 1' mass='0.9523809524' />
    <lower velocity='0 0' position='5 0' mass='0.9523809524' />
  </nodePair>
  <nodePair>
    <upper velocity='0 0' position='6 1' mass='0.8433962264' />
    <lower velocity='0 0' position='6 0' mass='0.8433962264' />
  </nodePair>
  <nodePair>
```

```
73      <upper velocity='0.0' position='7.1' mass='0.8345794393' />
74      <lower velocity='0.0' position='7.0' mass='0.8345794393' />
75    </nodePair>
76    <nodePair>
77      <upper velocity='0.0' position='8.1' mass='0.8259259259' />
78      <lower velocity='0.0' position='8.0' mass='0.8259259259' />
79    </nodePair>
80    <nodePair>
81      <upper velocity='0.0' position='9.1' mass='0.8174311927' />
82      <lower velocity='0.0' position='9.0' mass='0.8174311927' />
83    </nodePair>
84    <nodePair>
85      <upper velocity='0.0' position='10.1' mass='0.7090909091' />
86      <lower velocity='0.0' position='10.0' mass='0.7090909091' />
87    </nodePair>
88    <nodePair>
89      <upper velocity='0.0' position='11.1' mass='0.7009009009' />
90      <lower velocity='0.0' position='11.0' mass='0.7009009009' />
91    </nodePair>
92    <nodePair>
93      <upper velocity='0.0' position='12.1' mass='0.7928571429' />
94      <lower velocity='0.0' position='12.0' mass='0.7928571429' />
95    </nodePair>
96  </arm>
97  </environment>
98  </config>
```

F. Rechnerkonfigurationen

Zur Durchführung der Experimente wurden in dieser Diplomarbeit die folgenden Rechner verwendet.

Rechner	**1. Fachbereich-Rechner**	**2. IMMI-Cluster**
Anzahl	23	14
CPU	Intel Core 2 6700	Intel Xeon X5650
Kerne	1	6
Taktfrequenz	2,66 GHz	2,66 GHz
Cache-Größe	4 MB	12 MB
Arbeitsspeicher	4 GB	48 GB
Festplatte	Western Digital WD2500YS	
Grafikkarte	Nvidia Quadro FX 560	

Rechner	**3. Desktop-Rechner**	**4. Notebook**
Anzahl	1	1
CPU	Intel Core i7-2600K	Intel Core i5-560M
Kerne	4	2
Taktfrequenz	3,4 GHz	2,66 GHz
Cache-Größe	8 MB	3 MB
Arbeitsspeicher	16 GB	3 GB
Festplatte	Samsung SSD 830	Hitachi HTS545032B9A300
Grafikkarte	Nvidia GeForce GTX 560 Ti	

Tabelle F.1.: Verwendete Rechnerkonfigurationen.

Literaturverzeichnis

[1] Artyomov, Evgeny ; Yadid-Pecht, Orly: Modified High-Order Neural Network for Invariant Pattern Recognition. In: *Pattern Recognition Letters* 26 (2005), Nr. 6, S. 843–851

[2] Auger, Anne ; Finck, Steffen ; Hansen, Nikolaus ; Ros, Raymond: BBOB 2010: Comparison Tables of All Algorithms on All Noisy Functions / INRIA. 2010 (RT-389). – Forschungsbericht

[3] Auger, Anne. ; Hansen, Nikolaus: A Restart CMA Evolution Strategy With Increasing Population Size. In: *Proceedings of the IEEE Congress on Evolutionary Computation* Bd. 2, IEEE Press, 2005, S. 1769–1776

[4] Baraniuk, Richard G. ; Davenport, Mark A. ; DeVore, Ronald A. ; Wakin, Michael B.: A Simple Proof of the Restricted Isometry Property for Random Matrices. In: *Constructive Approximation* (2008), Nr. 3, S. 253–263

[5] Bellman, Richard E.: *Dynamic Programming*. Princeton University Press, 1957. – ISBN 978-0-691-07951-6

[6] Bengio, Yoshua: Learning Deep Architectures for AI / Dept. IRO, Universite de Montreal. 2007 (1312). – Forschungsbericht

[7] Bengio, Yoshua ; LeCun, Yann: *Scaling Learning Algorithms towards AI*. S. 321–360. In: Bottou, L. (Hrsg.) ; Chapelle, O. (Hrsg.) ; Decoste, D. (Hrsg.) ; Weston, J. (Hrsg.): *Large-Scale Kernel Machines*, MIT Press, 2007

[8] Bishop, Christopher M.: A Fast Procedure for Retraining the Multilayer Perceptron. In: *International Journal of Neural Systems* 2 (1991), Nr. 3, S. 229–236

[9] Bishop, Christopher M.: Exact Calculation of the Hessian Matrix for the Multilayer Perceptron. In: *Neural Computation* 4 (1992), Nr. 4, S. 494–501

[10] Bishop, Christopher M.: *Neural Networks for Pattern Recognition*. Oxford University Press, 1996. – ISBN 0198538642

[11] Bishop, Christopher M.: *Pattern Recognition and Machine Learning*. Springer, 2007. – ISBN 0387310738

[12] Blankertz, Benjamin: *BCI Competition III Webpage*. 2005. – URL `http://www.bbci.de/competition/iii/`. – Zugriffsdatum: 29. Mai 2012. – Internetseite

[13] Bochkanov, Sergey ; Bystritsky, Vladimir: *Levenberg-Marquardt Algorithm for Multivariate Optimization.* – URL `http://www.alglib.net/optimization/levenbergmarquardt.php`. – Zugriffsdatum: 12. Oktober 2011. – Internetseite

[14] Bochkanov, Sergey ; Bystritsky, Vladimir: *Unconstrained Optimization: L-BFGS and CG.* – URL `http://www.alglib.net/optimization/lbfgsandcg.php`. – Zugriffsdatum: 13. Oktober 2011. – Internetseite

[15] Boser, Bernhard E. ; Guyon, Isabelle M. ; Vapnik, Vladimir N.: A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, ACM Press, 1992, S. 144–152

[16] Boyan, Justin A. ; Moore, Andrew W.: Generalization in Reinforcement Learning: Safely Approximating the Value Function. In: *Advances in Neural Information Processing Systems 7*, MIT Press, 1995, S. 369–376

[17] Brodersen, Kay H. ; Ong, Cheng S. ; Stephan, Klaas E. ; Buhmann, Joachim M.: The Balanced Accuracy and Its Posterior Distribution. In: *Proceedings of the 2010 20th International Conference on Pattern Recognition*, IEEE Computer Society, 2010, S. 3121–3124

[18] Bronstein, Ilja N. ; Semendjajew, Konstantin A. ; Musiol, Gerhard ; Mühlig, Heiner: *Taschenbuch der Mathematik.* 6. Auflage. Harri Deutsch, 2005. – ISBN 3-8171-2006-0

[19] Calderbank, Robert ; Jafarpour, Sina ; Schapire, Robert: Compressed Learning: Universal Sparse Dimensionality Reduction and Learning in the Measurement Domain / Princeton University. URL `http://dsp.rice.edu/files/cs/cl.pdf`. – Zugriffsdatum: 11. März 2012, 2009. – Forschungsbericht

[20] Candès, Emmanuel J. ; Romberg, Justin K.: Practical Signal Recovery from Random Projections. In: *Proceedings of SPIE Computational Imaging III*, SPIE Press, 2005, S. 76–86

[21] Chang, Chih-Chung ; Lin, Chih-Jen: LIBSVM: A Library for Support Vector Machines. In: *ACM Transactions on Intelligent Systems and Technology* 2 (2011), Nr. 3, S. 27:1–27:27

[22] Ciresan, Dan C. ; Meier, Ueli ; Gambardella, Luca M. ; Schmidhuber, Jürgen: Deep, Big, Simple Neural Nets for Handwritten Digit Recognition. In: *Neural Computation* 22 (2010), Nr. 12, S. 3207–3220

[23] Cochran, Susan D.: *2-Sample z-Tests.* 2004. – URL `http://www.stat.ucla.edu/~cochran/stat10/winter/lectures/lect21.html`. – Zugriffsdatum: 13. März 2012. – Vorlesungsunterlagen

[24] Donchin, Emanuel ; Spencer, Kevin M. ; Wijesinghe, Ranjith: The Mental Prosthesis: Assessing the Speed of a P300-Based Brain-Computer Interface. In: *IEEE Transactions on Rehabilitation Engeneering* 8 (2000), S. 174–179

[25] Donoho, David L.: Compressed sensing. In: *IEEE Transactions on Information Theory* 52 (2006), Nr. 4, S. 1289–1306

[26] Dutech, Alain ; Edmunds, Tim ; Kok, Jelle ; Lagoudakis, Michail ; Littman, Michael ; Riedmiller, Martin ; Russell, Brian ; Scherrer, Bruno ; Sutton, Richard ; Timmer, Stephan ; Vlassis, Nikos ; White, Adam ; Whiteson, Shimon: *NIPS Workshop: Reinforcement Learning Benchmarks and Bake-offs II*. 2005. – URL `http://www.cs.rutgers.edu/~mlittman/topics/nips05-mdp/bakeoffs05.pdf`. – Zugriffsdatum: 8. Mai 2012. – Beschreibungen der Workshop-Beiträge

[27] Eaton, John W.: *GNU Octave Manual*. Network Theory Limited, 2002. – ISBN 0-9541617-2-6

[28] Ernst, Damien ; Geurts, Pierre ; Wehenkel, Louis: Tree-Based Batch Mode Reinforcement Learning. In: *Journal of Machine Learning Research* 6 (2005), S. 503–556

[29] Fahlman, Scott E.: An Empirical Study of Learning Speed in Back-Propagation Networks. 1988. – Forschungsbericht

[30] Farwell, Lawrence A. ; Donchin, Emanuel: Talking off the Top of Your Head: Toward a Mental Prosthesis Utilizing Event-Related Brain Potentials. In: *Electroencephalography and Clinical Neurophysiology* 70 (1988), S. 510–523

[31] Gabel, Thomas ; Riedmiller, Martin: Reducing Policy Degradation in Neuro-Dynamic Programming. In: *ESANN*, 2006, S. 653–658

[32] Gomez, Faustino ; Schmidhuber, Jürgen ; Miikkulainen, Risto: Accelerated Neural Evolution through Cooperatively Coevolved Synapses. In: *Journal of Machine Learning Research* 9 (2008), S. 937–965

[33] Guennebaud, Gaël ; Jacob, Benoît u. a.: *Eigen v3*. 2010. – URL `http://eigen.tuxfamily.org`. – Zugriffsdatum: 29. Mai 2012. – Internetseite

[34] Hansen, Nikolaus ; Ostermeier, Andreas: Completely Derandomized Self-Adaptation in Evolution Strategies. In: *Evolutionary Computation* 9 (2001), Nr. 2, S. 159–195

[35] Hansen, Nikolaus ; Ros, Raymond ; Mauny, Nikolas ; Schoenauer, Marc ; Auger, Anne: PSO Facing Non-Separable and Ill-Conditioned Problems / INRIA. 2008 (RR-6447). – Forschungsbericht

[36] Hebb, Donald O.: *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press, 2002. – Nachdruck des Originals von 1949. – ISBN 0805843000

[37] Heidrich-Meisner, Verena ; Igel, Christian: Neuroevolution Strategies for Episodic Reinforcement Learning. In: *Journal of Algorithms* 64 (2009), Nr. 4, S. 152–168

[38] Hinton, Geoffrey E. ; Osindero, Simon ; Teh, Yee-Whye: A Fast Learning Algorithm for Deep Belief Nets. In: *Neural Computation* 18 (2006), Nr. 7, S. 1527–1554

[39] Hornik, Kurt ; Stinchcombe, Maxwell B. ; White, Halbert: Multilayer Feedforward Networks are Universal Approximators. In: *Neural Networks* 2 (1989), Nr. 5, S. 359–366

[40] Jarrett, Kevin ; Kavukcuoglu, Koray ; Ranzato, Marc'Aurelio ; LeCun, Yann: What is the Best Multi-Stage Architecture for Object Recognition? In: *Proceedings of the International Conference on Computer Vision (ICCV'09)*, IEEE Press, 2009, S. 2146–2153

[41] Kassahun, Yohannes: Evolving Augmented Neural Networks in Compressed Parameter Space. In: *4th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*, 2011. – Zur Veröffentlichung angenommen

[42] Kassahun, Yohannes ; de Gea, Jose ; Edgington, Mark ; Metzen, Jan H. ; Kirchner, Frank: Accelerating Neuroevolutionary Methods Using a Kalman Filter. In: *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, ACM, 2008, S. 1397–1404

[43] Kassahun, Yohannes ; Sommer, Gerald: Efficient Reinforcement Learning through Evolutionary Acquisition of Neural Topologies. In: *ESANN*, 2005, S. 259–266

[44] Koutník, Jan ; Gomez, Faustino J. ; Schmidhuber, Jürgen: Evolving Neural Networks in Compressed Weight Space. In: Pelikan, Martin (Hrsg.) ; Branke, Jürgen (Hrsg.): *GECCO*, ACM, 2010, S. 619–626

[45] Koutník, Jan ; Gomez, Faustino J. ; Schmidhuber, Jürgen: Searching for Minimal Neural Networks in Fourier Space. In: Baum, Eric (Hrsg.) ; Hutter, Marcus (Hrsg.) ; Kitzelnmann, Emanuel (Hrsg.): *Proceedings of The Third Conference on Artificial General Intelligence (AGI 2010)*, Atlantic Press, 2010, S. 61–66

[46] Kretchmar, R. M. ; Anderson, Charles W.: Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning. In: *International Conference on Neural Networks*, 1997, S. 834–837

[47] Krusienski, Dean ; Schalk, Gerwin: *Wadsworth BCI Dataset (P300 Evoked Potentials)*. 2004. – URL `http://www.bbci.de/competition/iii/desc_II.pdf`. – Zugriffsdatum: 19. April 2012. – Beschreibung des Datensatzes

[48] Lang, Kevin J. ; Witbrock, Michael J.: Learning to Tell Two Spirals Apart. In: Touretzky, David (Hrsg.) ; Hinton, Geoffrey (Hrsg.) ; Sejnowski, Terrence (Hrsg.): *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, 1988, S. 52–61

[49] Lebedev, Mikhail ; Nicolelis, Miguel A. L.: Brain–Machine Interfaces: Past, Present and Future. In: *Trends in Neurosciences* 29 (2006), Nr. 9, S. 536–546

[50] LeCun, Yann ; Bengio, Yoshua: *Convolutional Networks for Images, Speech and Time Series*. S. 255–258. In: Arbib, Michael A. (Hrsg.): *The Handbook of Brain Theory and Neural Networks*, MIT Press, 1995

[51] LeCun, Yann ; Bottou, Léon ; Orr, Genevieve B. ; Müller, Klaus-Robert: Efficient BackProp. In: Orr, Genevieve B. (Hrsg.) ; Müller, Klaus-Robert (Hrsg.): *Neural Networks: Tricks of the Trade*, Springer, 1998, S. 9–50

[52] LeCun, Yann. ; Bottou, Léon ; Bengio, Yoshua. ; Haffner, Patrick: Gradient-Based Learning Applied to Document Recognition. In: Haykin, Simon (Hrsg.) ; Kosko, Bart (Hrsg.): *Intelligent Signal Processing*, IEEE Press, 2001, S. 306–351

[53] LeCun, Yann ; Cortes, Corinna: *The MNIST Database of Handwritten Digits*. – URL `http://yann.lecun.com/exdb/mnist`. – Zugriffsdatum: 2. April 2012. – Internetseite

[54] Levenberg, Kenneth: A Method for the Solution of Certain Problems in Least Squares. In: *Quarterly of Applied Mathematics* 2 (1944), S. 164–168

[55] Lin, Long-Ji: Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. In: *Machine Learning* 8 (1992), Nr. 3, S. 293–321

[56] Maillard, Odalric-Ambrym ; Munos, Rémi: Compressed Least-Squares Regression. In: Bengio, Yoshua (Hrsg.) ; Schuurmans, Dale (Hrsg.) ; Lafferty, John (Hrsg.) ; Williams, Chris K. I. (Hrsg.) ; Culotta, Aron (Hrsg.): *Advances in Neural Information Processing Systems 22*. 2009, S. 1213–1221

[57] Marquardt, Donald: An Algorithm for Least-Squares Estimation of Nonlinear Parameters. In: *Journal of the Society for Industrial and Applied Mathematics* 11 (1963), Nr. 2, S. 431–441

[58] McCulloch, Warren S. ; Pitts, Walter: A Logical Calculus of the Ideas Immanent in Nervous Activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133

[59] Nissen, Steffen: Implementation of a Fast Artificial Neural Network Library (fann) / Department of Computer Science University of Copenhagen (DIKU). URL `http://leenissen.dk/fann/report/report.html`. – Zugriffsdatum: 8. Mai 2012, 2003 (31). – Forschungsbericht

[60] o. V.: *Implementation of a Softmax Activation Function for Neural Networks*. – URL `http://stackoverflow.com/questions/9906136`. – Zugriffsdatum: 10. April 2012. – Internetseite

[61] o. V.: *Reinforcement Learning Competitition and Benchmarking Event.* – URL http://www.cs.mcgill.ca/~dprecup/workshops/icml06.html. – Zugriffsdatum: 16. Januar 2012. – Internetseite

[62] o. V.: *RoboCup Objective.* – URL http://www.robocup.org/about-robocup/objective/. – Zugriffsdatum: 19. März 2012. – Internetseite

[63] o. V. ; Sarle, Warren S. (Hrsg.): *Neural Network FAQ.* 1997. – URL ftp://ftp.sas.com/pub/neural/FAQ.html. – Zugriffsdatum: 31. Oktober 2011. – Internetseite

[64] o. V.: *CUDA Toolkit 4.1 CUBLAS Library.* 2012. – URL http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf. – Zugriffsdatum: 17. April 2012. – Handbuch

[65] Osowski, Stanislaw ; Bojarczak, Piotr ; Stodolski, Maciej: Fast Second Order Learning Algorithm for Feedforward Multilayer Neural Networks and Its Applications. In: *Neural Networks* 9 (1996), Nr. 9, S. 1583–1596

[66] Pyeatt, Larry D. ; Howe, Adele E.: Decision Tree Function Approximation in Reinforcement Learning. In: *Proceedings of the 3rd International Symposium on Adaptive Systems: Evolutionary Computation and Probabilistic Graphical Models*, 2001, S. 70–77

[67] Rakotomamonjy, Alain ; Guigue, Vincent: BCI Competition III: Dataset II - Ensemble of SVMs for BCI P300 Speller. In: *IEEE Transactions on Biomedical Engeneering* 55 (2008), Nr. 3, S. 1147–1154

[68] Riedmiller, Martin: Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In: Gama, João (Hrsg.) ; Camacho, Rui (Hrsg.) ; Brazdil, Pavel (Hrsg.) ; Jorge, Alípio (Hrsg.) ; Torgo, Luís (Hrsg.): *16th European Conference on Machine Learning*, Springer, 2005, S. 317–328

[69] Riedmiller, Martin ; Braun, Heinrich: A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In: *IEEE International Conference on Neural Networks*, IEEE Press, 1993, S. 586–591

[70] Riedmiller, Martin ; Montemerlo, Michael ; Dahlkamp, Hendrik: Learning to Drive a Real Car in 20 Minutes. In: Howard, Daniel (Hrsg.) ; Rhee, Phill-Kyu (Hrsg.): *FBIT*, IEEE Computer Society, 2007, S. 645–650

[71] Rivet, Bertrand ; Souloumiac, Antoine ; Attina, Virginie ; Gibert, Guillaume: xDAWN Algorithm to Enhance Evoked Potentials: Application to Brain–Computer Interface. In: *IEEE Transactions on Biomedical Engeneering* 56 (2009), Nr. 8, S. 2035–2043

[72] Rosenblatt, Frank: The Perceptron - a Perceiving and Recognizing Automaton / Project PARA, Cornell Aeronautical Lab. 1957 (85-460-1). – Forschungsbericht

[73] Rumelhart, David E. ; Hinton, Geoffrey E. ; Williams, Ronald J.: Learning Representations by Back-propagating Errors. In: *Nature* 323 (1986), Nr. 6088, S. 533–536

[74] Russell, Stuart ; Norvig, Peter: *Artificial Intelligence: A Modern Approach*. 3. Auflage. Prentice Hall, 2009. – ISBN 0136042597

[75] Schlimmer, Jeffrey C.: *Concept Acquisition through Representational Adjustment*, Department of Information and Computer Science, University of California, Dissertation, 1987

[76] Schmidhuber, Jürgen: Discovering Solutions with Low Kolmogorov Complexity and High Generalization Capability. In: *International Conference on Machine Learning*, Morgan Kaufmann, 1995, S. 488–496

[77] Schmidhuber, Jürgen: Discovering Neural Nets with Low Kolmogorov Complexity and High Generalization Capability. In: *Neural Networks* 10 (1997), Nr. 5, S. 857–873

[78] Schoknecht, Ralf ; Riedmiller, Martin: Reinforcement Learning on Explicitly Specified Time Scales. In: *Neural Computing & Applications* 12 (2003), Nr. 2, S. 61–80

[79] Simard, Patrice Y. ; Steinkraus, Dave ; Platt, John C.: Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In: *Proceedings of the Seventh International Conference on Document Analysis and Recognition* Bd. 2, IEEE Computer Society, 2003, S. 958–963

[80] Simon, Herbert A. ; Newell, Allen: Heuristic Problem Solving: The Next Advance in Operations Research. In: *Operations Research* 6 (1958), Nr. 1, S. 1–10

[81] Spirkovska, Lilly ; Reid, Max B.: Robust Position, Scale, and Rotation Invariant Object Recognition Using Higher-Order Neural Networks. In: *Pattern Recognition* 25 (1992), Nr. 9, S. 975–985

[82] Stallkamp, Johannes ; Schlipsing, Marc ; Salmen, Jan ; Igel, Christian: The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In: *IEEE International Joint Conference on Neural Networks*, IEEE Press, 2011, S. 1453–1460

[83] Stallkamp, Johannes ; Schlipsing, Marc ; Salmen, Jan ; Igel, Christian: Man vs. Computer: Benchmarking Machine Learning Algorithms for Traffic Sign Recognition. In: *Neural Networks* 32 (2012), S. 323–332

[84] Stanley, Kenneth O. ; Miikkulainen, Risto: Evolving Neural Networks Through Augmenting Topologies. In: *Evolutionary Computation* 10 (2002), Nr. 2, S. 99–127

[85] Sutton, Richard S.: Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In: *Advances in Neural Information Processing Systems 8*, MIT Press, 1996, S. 1038–1044

[86] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. MIT Press, 1998. – ISBN 0262193981

[87] Tesauro, Gerald: Temporal Difference Learning and TD-Gammon. In: *Communications of the ACM* 38 (1995), Nr. 3, S. 58–68

[88] Thrun, Sebastian ; Burgard, Wolfram ; Fox, Dieter: *Probabilistic Robotics*. MIT Press, 2005. – ISBN 0262201623

[89] Vapnik, Vladimir N.: *The Nature of Statistical Learning Theory*. Springer, 1995. – ISBN 0-387-94559-8

[90] White, Matt ; Sejnowski, Terry ; Rosenberg, Charles ; Qian, Ning ; Gorman, R. P. ; Wieland, Alexis ; Deterding, David ; Niranjan, Mahesan ; Robinson, Tony: *Bench: CMU Neural Networks Benchmark Collection*. – URL `http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/0.html`. – Zugriffsdatum: 27. November 2011. – Internetseite

[91] Whiteson, Shimon ; Taylor, Matthew E. ; Stone, Peter: Critical Factors in the Empirical Performance of Temporal Difference and Evolutionary Methods for Reinforcement Learning. In: *Journal of Autonomous Agents and Multi-Agent Systems* 21 (2009), Nr. 1, S. 1–27

[92] Wieland, Alexis P.: Evolving Controls for Unstable Systems. In: Touretzky, David S. (Hrsg.) ; Elman, Jeffrey L. (Hrsg.) ; Sejnowski, Terrence J. (Hrsg.) ; Hinton, Geoffrey E. (Hrsg.): *Connectionist Models: Proceedings of the 1990 Summer School*. CA: Morgan Kaufmann, 1990, S. 91–102

[93] Wolpert, David H.: The Lack of A Priori Distinctions Between Learning Algorithms. In: *Neural Computation* 8 (1996), Nr. 7, S. 1341–1390