MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Das universelle Approximationstheorem für neuronale Netze“

verfasst von / submitted by

Eva Hüll, BEd

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Education (MEd)

Wien, 2021 / Vienna 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 199 504 520 02
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Lehramt Sek (AB) UF Chemie UF Mathematik
Betreut von / Supervisor:	Doz. Dr. Franz Embacher

Inhaltsverzeichnis

Einleitung **1**

1 Künstliche Intelligenz und Deep Learning **3**
1.1 Machine Learning, Representation Learning und Deep Learning 3
1.2 Künstliche neuronale Netze 4
1.3 Die historischen Entwicklungen im Bereich künstlicher neuronaler Netze . 7
1.3.1 Kybernetik 7
1.3.2 Konnektionismus 11
1.3.3 Die dritte Welle - Deep Learning 14
1.4 Der Lernalgorithmus im Machine Learning 17
1.4.1 Überwachtes und unüberwachtes Lernen 18

2 Neuronale Feedforward-Netze **19**
2.1 Die Funktionsweise eines Feedforward-Netzes 19
2.1.1 Aufbau eines Feedforward-Netzes 19
2.1.2 Das mathematische Modell eines neuronalen Netzes 20
2.1.3 Typische Aktivierungsfunktionen in neuronalen Netzen 22
2.2 Lernen in Feedforward-Netzen 26
2.2.1 Das Gradientenabstiegsverfahren 27
2.2.2 Das Verfahren der Backpropagation 29

3 Das universelle Approximationstheorem für Feedforward-Netze **35**
3.1 Das universelle Approximationstheorem für Feedforward-Netze mit einer Zwischenschicht 35
3.2 Folgerungen und weiterführende Betrachtungen 44
3.2.1 Graphische Veranschaulichung der Approximation durch ein ReLU-Netzwerk 44
3.2.2 Weitere Versionen des universellen Approximationstheorems 47
3.2.3 Grenzen der Tiefe beziehungsweise Weite für eine erfolgreiche Approximation 51
3.2.4 Die Approximationsmöglichkeiten von Feedforward-Netzen mit der ReLU-Aktivierungsfunktion 52
3.2.5 Mögliche Probleme und wie sie behoben werden können 60

4 Fazit **65**

Literaturverzeichnis **67**

Abbildungsverzeichnis **77**

5 Anhang **79**
5.1 Mathematische Definitionen und Sätze für die Beweise aus Kapitel 3 . . . 79

Zusammenfassung **85**

Abstract **87**

Einleitung

Am 4. April 2021 wurde in der österreichischen Tageszeitung *Der Standard* ein Artikel mit der Schlagzeile „Künstliche Intelligenz komponiert ‚neuen' Nirvana-Song "([69]) veröffentlicht. Bei der dabei erwähnten „Künstlichen Intelligenz " handelt es sich um ein, für diese Anwendung vermutlich sehr großes, künstliches neuronales Netzwerk. Künstliche neuronale Netze begegnen uns mittlerweile in den unterschiedlichsten Bereichen unseres Alltags. Typische Beispiele sind die sogenannten Chatbots, die auf diversen Webseiten, wie unter anderem bei Onlineshops, zu finden sind und in Echtzeit auf unsere Fragen antworten können. Ebenso steckt auch hinter der Sprachsteuerung von Google Maps, die beim Autofahren äußerst nützlich ist, ein neuronales Netz.

Diese Arbeit beschäftigt sich mit der Frage, welche Funktionen durch künstliche neuronale Netzwerke realisiert werden können. Dazu wird das universelle Approximationstheorem vorgestellt und bewiesen. Außerdem werden weitere Varianten des Theorems aufgezeigt und ein paar praktische Betrachtungen, die beim Aufbau und erfolgreichem Training des Netzes helfen sollen, gegeben.

Im ersten Kapitel werden zunächst die Begriffe *künstliche Intelligenz, maschinelles Lernen* und *Deep Learning* definiert und im Anschluss erfolgt eine kurze Vorstellung künstlicher neuronaler Netzwerke, gefolgt von einem historischen Rückblick. Im zweiten Kapitel wird der Aufbau von Feedforward-Netzwerken betrachtet und welche Mathematik dahintersteckt. Weiters wird das bekannte Gradientenabstiegsverfahren, das zum Trainieren neuronaler Netze dient, vorgestellt sowie das Verfahren der Backprogagation zum Berechnen des Gradienten erklärt. Kapitel 3, das Herzstück der Arbeit, behandelt das universelle Approximationstheorem für Feedforward-Netze. Zunächst wird das Theorem für Netzwerke mit einer verdeckten Schicht formuliert und bewiesen. Im Anschluss gibt es eine kurze graphische Erklärung, gefolgt von weiteren Versionen des Theorems. Danach werden ein paar, auch insbesondere für die Praxis relevante, Betrachtungen bezüglich möglicher Approximationsgrenzen bei der Tiefe beziehungsweise Weite von Feedforward-Netzen, insbesondere solcher, die die ReLU-Aktivierungsfunktion aufweisen, ausgeführt. Im letzten Abschnitt des Kapitels werden mögliche Trainingsprobleme inklusive deren Lösungsansätze thematisiert. Am Ende folgt ein Fazit mit einem kleinen Ausblick bezüglich künftiger Entwicklungen.

1 Künstliche Intelligenz und Deep Learning

1.1 Machine Learning, Representation Learning und Deep Learning

Unter künstlicher Intelligenz (KI), manchmal auch artifizieller Intelligenz (*Artificial Intelligence*, AI), versteht man intelligentes Verhalten, das durch Computersysteme erzeugt wird, bei dem also weder Mensch noch Tier involviert sind. Sie stellt ein Fachgebiet der Informatik dar, ist interdisziplinär und umfasst viele Teilbereiche, wie unter anderem maschinelles Lernen (*Maschine Learning*). Bei diesem stehen Algorithmen, die sich selbstständig verändern können, im Fokus. Ein solches dynamisches Verhalten wird durch das Sammeln von Erfahrungen erzielt. Hierbei werden der Maschine Datensätze vorgesetzt, aus denen diese Gemeinsamkeiten und Zusammenhänge ableiten soll (vgl. [34], S. 2-39).

Ein Segment des maschinellen Lernens, das *Representation Learning*, beschäftigt sich mit der Frage nach einer geeigneten Darstellung der Daten. Die Grundidee beim Representation Learning ist, dass das Netzwerk selbst eine geeignete Aufbereitung der Daten lernen soll, um diese im nächsten Schritt einfacher verarbeiten, wie beispielsweise klassifizieren, zu können. Dazu werden Trainingsdaten vorgesetzt, anhand derer das Netzwerk wichtige Merkmale, auch *Features* genannt, identifizieren und davon ausgehend eine geeignete Repräsentation der Daten wählen soll. Dies wird oft mit Hilfe künstlicher neuronaler Netzwerke (KNNs) realisiert, deren Architektur und Funktionsweise sich von biologischen Neuronen und deren Vernetzungen ableitet. Ein äußerst wichtiger Faktor für einen erfolgreichen Lernprozess ist die Tiefe des Netzwerks. Ein vielschichtiges, also tiefes, Netzwerk bringt nämlich zwei Vorteile mit sich: Zum einen können gewisse Merkmale wiederverwendet werden, zum anderen können in tieferen Schichten zunehmend abstraktere Merkmale aus den Inputdaten abgeleitet werden (vgl. [10], S. 1798-1801).

Das Fachgebiet, das sich mit Netzwerkstrukturen, die viele Zwischenschichten (*Hidden Layers*) aufweisen, beschäftigt, heißt *Deep Learning*. Methoden in diesem Teilbereich des maschinellen Lernens verwenden dementsprechend erweiterte Algorithmen des Representation Learnings, um mehrere Darstellungen in unterschiedlichen Komplexitätsniveaus zu ermöglichen. Abstrakte Merkmale und Repräsentationen werden dabei durch Kombination einfacherer dargestellt. Sind die Inputdaten beispielsweise Bilder, so erhält das Netzwerk diese in Form eines Vektors mit den einzelnen Pixelwerten als Einträge. In der ersten Schicht werden nun primitive Merkmale, wie Kanten, gesucht und lokalisiert. Daraus werden in der darauffolgenden Schicht Motive abgeleitet, die wiederum in späteren Schichten zu Objekten, beispielsweise Autos oder Tieren, zusammengefügt werden und die im Idealfall auch tatsächlich im Inputbild vorhanden waren. In den letzten Jahren leistete Deep Learning einen beachtlichen Beitrag im Bereich des maschinellen Lernens

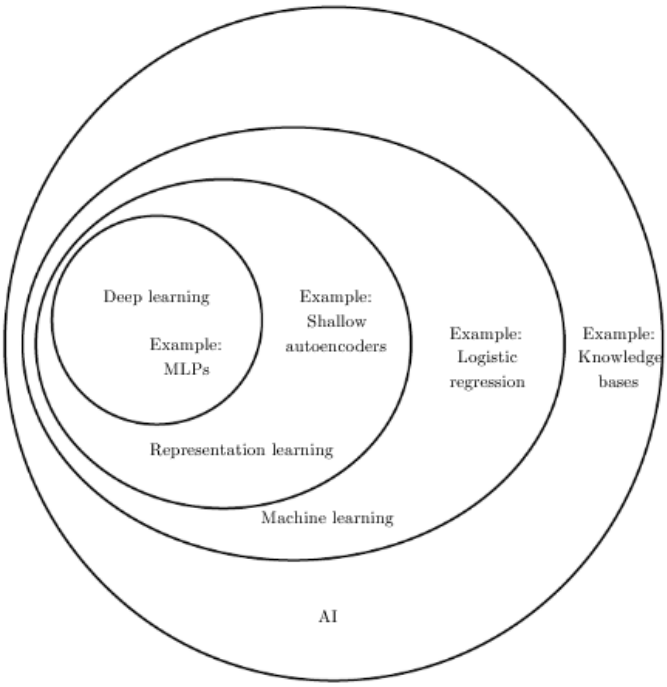


Abbildung 1.1: Zusammenhang von künstlicher Intelligenz und Deep Learning ([32], S. 9)

und der künstlichen Intelligenz und es scheint im Moment der vielversprechendste Ansatz in diesen Gebieten zu sein (vgl. [60], S. 436). In Abbildung 1.1 ist der Zusammenhang der Forschungsbereiche inklusive eines Beispiels einer jeweils darin typischen Technologie dargestellt.

1.2 Künstliche neuronale Netze

Unser Gehirn ist das mit Abstand komplexeste all unserer Organe. Durch dieses sind wir im Stande zu denken, zu fühlen, uns miteinander zu unterhalten sowie jegliche Sinneseindrücke zu verarbeiten. Ebenso steuert es die Motorik. All die durch unsere Wahrnehmung aufgenommenen Reize müssen dabei parallel verarbeitet werden, um derartige Leistungen erbringen zu können. Dadurch ergibt sich folglich auch ein äußerst komplexer Aufbau, wobei jedem Teil des Gehirns eine gewisse Aufgabe zukommt. Bei genauerer Betrachtung lässt sich erkennen, dass das menschliche Gehirn 10^{11} Nervenzellen, die Neuronen, umfasst, die untereinander, teils hochgradig, vernetzt sind. Jede Zelle besteht aus einem Zellkörper inklusive Zellkern, von dem kleine Fortsätze ausgehen. Diese stellen die Verbindung des Neurons zu vorherigen Zellen dar und werden Dendriten genannt. Des Weiteren hängt das Axon an dem Zellkörper, über welches das Neuron mit nachfolgenden Zellen über die sogenannten Synapsen verbunden ist. Kurz nach unserer Geburt sind unsere Neuronen noch relativ schwach vernetzt. Durch das Sammeln von Erfah-

rungen bilden sich zunehmend Verbindungen zwischen den Zellen. Diese Muster können durch Training gestärkt werden. Begriffe, die oft in einem gemeinsamen Kontext auftreten, werden durch eine Verbindung benachbarter Neuronen gespeichert. Entstandene Verbindungen können allerdings auch wieder verschwinden, wenn wir diese Wege lange nicht mehr benützen. Dies äußert sich durch Vergessen.

Wenn wir Reize aus unserer Umwelt wahrnehmen, geschieht dies über die Rezeptoren in den Sinnesorganen, die diese als Signale über unser Nervensystem an das Hirn weiterleiten. Dort gelangen sie zu den Neuronen, die im Grunde die Verarbeitungseinheiten unseres Gehirns bilden. Das Grundprinzip der neuronalen Informationsverarbeitung ist nun das folgende: Die Nervenzelle erhält über ihre Dendriten von vorgeschalteten Neuronen Signale in Form von elektrochemischen Reizen. Erreicht das elektrochemische Potential, das sich aus der Summe all dieser Eingabesignale ergibt, einen gewissen Schwellenwert, so wird das Neuron aktiviert und es sendet ebenfalls einen elektrischen Impuls über das Axon an alle nachkommenden Zellen. Dieses Signal wird dabei auf chemischen Wege über die Synapsen übertragen und von den darauffolgenden Zellen wiederum über deren Dendriten empfangen. Die Synapsen können dabei eine exzitatorische, also verstärkende, oder auch inhibitorische, sprich hemmende, Wirkung haben (vgl. [53], S. 11-15).

Frühere Computerprogramme waren zwar erfolgreich bei der Verarbeitung formaler Sprachen, konnten jedoch für uns als Menschen einfache Probleme, wie das inhaltliche Erfassen eines Satzes oder das Erkennen einer bekannten Person, nicht lösen. Diese bringen nämlich eine gewisse Kontextsensitivität mit sich, die durch intelligentes Verhalten entschlüsselt werden muss. Da unser Gehirn solche Aufgaben durch seine typische Architektur und Arbeitsweise leicht bewältigen kann, wurden nun, ausgehend von diesen Strukturen, Computerprogramme entwickelt, die eine ähnliche Gestalt aufweisen und daher künstliche neuronale Netzwerke genannt werden. Diese sind, analog zum menschlichen Gehirn, aus vielen kleinen Einheiten, den künstlichen Neuronen, aufgebaut, die untereinander vernetzt sind. Ein künstliches Neuron besitzt mehrere Eingangsleitungen $i = 1, \ldots, n$ an denen die Eingangswerte $x_i \in \mathbb{R}$ empfangen werden. Diese werden wie bei hemmenden oder verstärkenden Synapsen jeweils mit einem Skalar $\omega_i \in \mathbb{R}$ gewichtet. Im Anschluss wird deren Summe $\sum_{i=1}^{n} \omega_i \cdot x_i$ gebildet, sowie ein Schwellenwert b hinzugefügt, was zu einem Zwischenergebnis von $\sum_{i=1}^{n} \omega_i \cdot x_i + b$ führt. Dieses wird jetzt als Argument für die Aktivierungsfunktion f verwendet, also $f\left(\sum_{i=1}^{n} \omega_i \cdot x_i + b\right)$. Durch diese Funktion wird festgelegt, ob das Neuron aktiviert wird und „feuert“oder aber inaktiv bleibt. Außerdem wird durch sie der Ausgabewert, der abhängig vom Argument ist, berechnet. Die heute gängigste Funktion für diese Aufgabe ist die ReLU-Funktion (*Rectified Linear Unit*), die entsprechend der Vorschrift $f(x) = \max(0, x)$ arbeitet.

In Abbildung 1.2 sind die vereinfachten Bauweisen menschlicher und künstlicher Neuronen veranschaulicht. Die durch verbundene Neuronen aufgebauten künstlichen neuronalen Netze sind jedoch nur durch die biologischen Strukturen und Prozesse unseres Gehirns inspiriert. Es handelt sich keineswegs um eine detailgetreue Rekonstruktion. In Wahrheit gibt es einige Unterschiede zwischen künstlichen und biologischen neuronalen Netzen. So enthält unser Gehirn in etwa 100 Milliarden Neuronen, die miteinander in alle Raumrichtungen mit unterschiedlicher Stärke vernetzt sind. Die Anzahl an Neuronen in

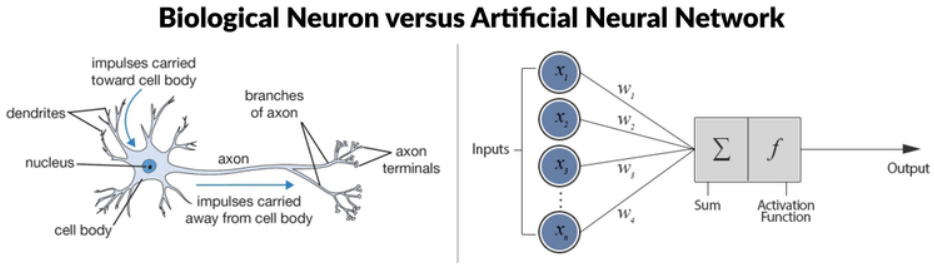


Abbildung 1.2: Vergleich zwischen einem biologischen (links) und einem künstlichen Neuron (rechts) ([103])

künstlichen Netzwerken liegt einige Größenordnungen darunter und die Neuronen sind zudem schichtweise angeordnet, wodurch sie in der Regel ausschließlich zu benachbarten Schichten Verbindungen aufweisen. Zudem arbeiten künstliche Netze synchron, indem zunächst alle Berechnungen innerhalb einer Schicht vollständig durchgeführt werden, ehe die Ergebnisse an die nächste Schicht gegeben werden, die nun mit diesen weiterarbeitet. In unserem Gehirn hingegen verlaufen die meisten Prozesse asynchron.

Ein weiterer und äußerst relevanter Unterschied liegt in der Fehlertoleranz. Biologische Netze weisen eine enorm hohe Vernetzung auf und Informationen werden verteilt gespeichert sowie verarbeitet, teils sogar im Übermaß. Somit können kleine Aussetzer keinen Informationsverlust verursachen. Bei künstlichen Netzen hingegen besteht aufgrund der Bau- und Arbeitsweise keine besonders hohe Fehlertoleranz. Es gibt jedoch besondere Algorithmen, die diese erhöhen können (vgl. [21], S. 43-45). Außerdem wird ein menschliches Neuron aktiviert, indem ein Schwellenpotential erreicht wird, woraufhin stets ein fixer Wert von diesem entsandt wird. Vereinfacht kann man sich diese Funktionsweise wie eine etwas modifizierte Heaviside-Funktion vorstellen. Im Bereich der Computerwissenschaften werden andere Aktivierungsfunktionen verwendet, die teilweise nichtlinear oder auch an manchen Stellen nicht differenzierbar sind. Ein typisches Beispiel hierfür ist die ReLU-Funktion $f(x) = \max(0, x)$. Zu guter Letzt besteht auch im Lernprozess selbst ein großer Unterschied. Wir Menschen lernen durch die altbekannte Hebbsche Lernregel, also entsprechend: „Neurons that fire together, wire together“ ([75]). Sie besagt, dass je öfter zwei Neuronen im selben Moment aktiv sind, desto eher werden sie auch künftig gleichzeitig reagieren. Dies wird durch das Ausbilden zusätzlicher Verbindungen in deren Nachbarschaft ermöglicht. Ein künstliches neuronales Netz hat dem entgegen eine bereits vordefinierte Struktur, in der keine neuen Verbindungen hinzukommen können (vgl. [75]).

1.3 Die historischen Entwicklungen im Bereich künstlicher neuronaler Netze

1.3.1 Kybernetik

Zu Beginn der 1940er Jahre, als die ersten Computer, also programmierbare Datenverarbeitungseinheiten, entwickelt wurden, begann auch das Interesse an künstlicher Intelligenz zuzunehmen (vgl. [55], S. 9). Die Idee, dass man auch Maschinen in die Lage versetzen könnte eigenständig zu lernen, schien äußerst reizvoll. Die ersten Entwicklungen im Bereich des maschinellen Lernens gehen auf die beiden Amerikaner Warren McCulloch und Walter Pitts zurück, die sich das Lernen aus der neurophysiologischen Sichtweise genauer angesehen haben. Sie entwickelten ein Modell von Neuronen, dessen Aufbau und Funktionsweise an jenem von menschlichen Neuronen orientiert war. Die sogenannte McCulloch-Pitts-Zelle, oft auch als McCulloch-Pitts-Neuron bekannt, besteht aus erregenden und hemmenden Eingängen, an denen Signale empfangen werden. Die Zelle funktioniert nach dem Alles-oder-nichts-Prinzip, was bedeutet, dass diese den Wert „1“ ausgibt, wenn die Summe aller erregenden Eingabewerte einen gewissen Schwellenwert erreicht. In allen anderen Fällen wird der Wert „0“ ausgegeben. Ebenso wird das Neuron inaktiviert, sobald eines der Signale, welches bei einem hemmenden Eingang empfangen wird, den Wert „1“ hat. McCulloch und Pitts konnten zeigen, dass ein aus solchen Zellen bestehendes künstliches Netz im Grunde jede logische Funktion darstellen kann (vgl. [67], S. 99-100). Wenige Jahre später gelang es ihnen nachzuweisen, dass Netze dieser Art sogar einfache Formen, wie beispielsweise Dreiecke, erkennen können (vgl. [86], S. 127-128).

Der Ansatz künstliche Neuronen zu konstruieren, um maschinelles Lernen zu ermöglichen, wurde in den folgenden Jahren von einer Vielzahl an Wissenschafterinnen und Wissenschaftern aufgegriffen und weiterentwickelt. So hat beispielsweise der amerikanische Informatiker und Psychologe Frank Rosenblatt im Jahr 1958 die sogenannten *Perzeptronen* eingeführt. Hierbei handelt es sich um einfache künstliche neuronale Netzwerke, welche im ursprünglichsten Fall aus einem einzigen Neuron bestehen, das dann dementsprechend als „einfaches Perzeptron“ bezeichnet wird. Der Unterschied zu McCulloch-Pitts-Zellen liegt darin, dass diese Neuronen nun um Gewichte erweitert wurden, das heißt, hereinkommende Signale $x_1, \ldots, x_n$ werden zuerst mit zugehörigen Werten $\omega_1, \ldots, \omega_n$ gewichtet und erst im darauffolgenden Schritt aufsummiert: $x \cdot \omega$, wobei $x \in \{0,1\}^n$ und $\omega \in \mathbb{R}^n$ Vektoren sind und „$\cdot$“ das Skalarprodukt bezeichnet (vgl. [93], S. 387-388). Die Idee für dieses Netzwerkmodell basiert auf der 1949 postulierten Hebbschen Lernregel, nach welcher sich eine Verbindung zwischen biologischen Neuronen verstärkt, wenn diese während des Lernprozesses beide zeitgleich aktiv sind (vgl. [39], S. 62). Rosenblatt konstruierte das Perzeptron zur Mustererkennung beziehungsweise in weiterer Folge zur Klassifikation. Es war das erste Modell, welches die Gewichte mit Hilfe eines Trainingsdatensatzes von selbst anpassen, sprich selbstständig „lernen“, konnte. Das heißt, das Netzwerk konnte zu einem Input, beispielsweise den Pixelwerten eines Bildes, den richtigen zugehörigen Output, wie etwa eine Objektklasse, liefern. Die Anpassung der Gewichte erfolgte anhand des sogenannten Fehler-Korrektur-Verfahrens, bei dem die Gewichte der Input-

Einheiten nur angepasst werden, wenn der vom Netz gelieferte Output einen falschen Wert für den Klassifizierungsvorgang erzeugt hatte. Diese fehlerkorrigierende Herangehensweise zur Anpassung der Gewichte wurde zu einer Standardmethode im Bereich des maschinellen Lernens und in den letzten Jahrzehnten gab es immer wieder Weiterentwicklungen dieser. Der Mathematiker Henry David Block, der mit Rosenblatt in Cornell zusammenarbeitete, konnte beweisen, dass das Fehler-Korrektur-Verfahren immer eine die Trainingsmenge separierende Hyperebene finden konnte, sofern die Trainingsdaten linear separierbar waren und somit überhaupt eine solche existierte. Da Rosenblatt seine Perzeptronen lieber in Hardware-Form anstatt per Simulation konstruierte, entstand somit der erste Neurocomputer, das *MARK I Perzeptron* (vgl. [79], S. 67-68), der über einen kleinen Bildsensor verfügte und einfache Ziffern erkennen konnte (vgl. [55], S. 11).

Zur selben Zeit wie Rosenblatt arbeiteten auch Bernard Widrow und Marcian E. Hoff in Stanford an künstlichen neuronalen Netzen und entwickelten gemeinsam die adaptive Maschine *ADALINE* (Adaptive Linear Neuron) zur Mustererkennung. Der für einen Input berechnete Wert war nun nicht mehr binär, also „0“ oder „1“, sondern es konnte jede beliebige reelle Zahl herauskommen. Im Gegensatz zu MARK I wurde hier die Adaption der Gewichte etwas anders durchgeführt, nämlich mit dem Ziel, die mittlere quadratische Abweichung zwischen dem berechneten Wert und dem gewünschten Output des Trainingsdatensatzes zu minimieren. In Folge lernt ADALINE somit bei jedem Durchlauf, während bei Rosenblatts Perzeptron nur eine Gewichtsanpassung erfolgte, wenn der ausgegebene Wert fehlerbehaftet war. Um welchen Wert das jeweilige Gewicht eines Inputsignals adaptiert wird, wird durch den Gradienten der mittleren quadratischen Abweichung bestimmt (vgl. [102], S. 97-99). Dieses Verfahren stellt somit eine Weiterentwicklung der beim Perzeptron angewandten Methode dar und ist heutzutage unter dem Namen Widrow-Hoff-Regel (auch Delta-Regel oder Least-Mean-Squares-Algorithmus, kurz LMS-Algorithmus), die ein Spezialfall des Gradientenabstiegsverfahrens ist, bekannt (vgl. [79], S. 69). Solche linearen Modelle, wie sie unter anderem bei ADALINE verwendet wurden, waren allerdings nur bedingt anwendbar. Ein die Grenzen aufzeigendes und dennoch simples Beispiel stellt die XOR-Funktion dar. Diese ist dadurch definiert, dass sie den Output „1“ liefert, wenn die beiden Inputsignale verschieden, also einmal „0“ und einmal „1“, sind und bei denselben Inputwerten den Output „0“ erzeugt. Eine solche binäre Funktion ist nicht linear separierbar, das heißt, die Urbilder von „0“ und „1“ können nicht durch eine Hyperebene getrennt werden. Das hat zur Folge, dass ein Perzeptron mit zwei Inputs Funktionen wie die XOR-Funktion niemals lernen kann. Dieses von Minsky und Paperts 1969 publizierte Resultat sowie weitere zur selben Zeit veröffentlichte Beiträge, die zeigten, dass künstliche Neuronen und deren Netzwerke in der damaligen Form nicht in der Lage seien grenzenlos und fehlerfrei zu lernen, brachten die Forschung in den darauffolgenden fast 15 Jahren ins Stocken. Im Nachhinein stellte sich heraus, dass der Schluss von Minsky und Paperts, dass die Probleme, welche bei einzelnen Perzeptronen bestanden, auch bei Netzwerken dieser auftreten müssten, falsch war. Dieser Irrtum wurde zwar relativ bald erkannt, gelangte jedoch erst Jahre später an die breite Masse. Daher begannen der Staat, sowie viele Forscherinnen und Forscher, nach der Verbreitung des Fehlschlusses an dem Nutzen neuronaler Netze zu zweifeln

und es kam zum sogenannten Künstliche-Intelligenz-Winter, kurz KI-Winter (vgl. [94], S. 22).

Der KI-Winter

Trotz des Einbruchs in der staatlich geförderten Forschung wurden in der Zeit des KI-Winters einige später äußert wichtige Konzepte, Algorithmen und Ideen publiziert. So etwa wurde in diesem Zeitraum das Verfahren der *Backpropagation* (BP), einer heute äußerst wichtigen Technik im Bereich des maschinellen Lernens, insbesondere bei tiefen neuronalen Netzwerken, entwickelt. Backpropagation ist eine Methode zur Berechnung des Gradienten einer Funktion, im Fall neuronaler Netze der Fehlerfunktion. Dieser wird dann für das sogenannte Gradientenabstiegsverfahren, einem Optimierungsverfahren zur Anpassung der Gewichte eines neuronalen Netzwerks, benötigt. Die Backpropagation, auch Rückpropagierung oder Fehlerrückführung genannt, beruht auf der Kettenregel der Differentialrechnung und der Name kommt daher, dass das Zwischenergebnis immer zur früheren Schicht, bei der letzten beginnend, vorgereicht wird. Die Berechnung erfolgt also in „Rückrichtung“, wobei Ergebnisse älterer Schichten für neue Berechnungen wiederverwendet werden. Somit können mögliche redundante Berechnungen vermieden werden. Die mathematische Grundlage des Gradientenabstiegsverfahrens geht auf Cauchy und Hadamard zurück. In den Jahren 1960 beziehungsweise 1961 und 1969 publizierten Kelley sowie Bryson unabhängig voneinander Arbeiten, in denen dieser Algorithmus erstmals zur Adaption der Parameter eines neuronalen-Netzwerk-artigen Systems angewandt wurde. Damals wurden jedoch einige Aspekte, wie direkte Verbindungen zwischen mehreren Schichten sowie eventuell auftretende Vorteile eines *Sparse Networks*, also eines spärlichen Netzwerks, das entsprechend seiner Bezeichnung wenige Vernetzungen aufweist, missachtet. Das Verfahren wurde in den darauffolgenden Jahren von Linnainmaa für diese diskreten Netzwerke mit einem geringen Vernetzungsgrad, die ähnlich zu neuronalen Netzen waren, weiterentwickelt (vgl. [97], S. 90-91). 1982 war Paul Werbos der Erste, der ein neues, effizienteres Verfahren der Backpropagation erfolgreich auf mehrschichtige neuronale Netze anwandte (vgl. [35], S. 59). Seine Publikationen blieben jedoch weitgehend ungehört, weshalb der effiziente Backpropagation-Algorithmus sowie dessen Erfinder und die Anwendung bei neuronalen Netzen relativ unbekannt blieben. Dies war insbesondere dem bereits erwähnten Buch von Minsky und Paperts zu verschulden, da darin der Nutzen neuronaler Netze für Anwendungen im Bereich der künstlichen Intelligenz angezweifelt wurde und sich dementsprechend zunehmend weniger Wissenschafterinnen und Wissenschafter mit neuen Publikationen in diesem Bereich, zum Leidwesen Werbos, auseinandersetzten (vgl. [80], S. 392-393). In Kapitel 2.2.2 wird das Verfahren der Backpropagation noch genauer betrachtet.

Des Weiteren konstruierten Fukushima und sein Team im Jahr 1975 ein Netzwerk, welches Zeichen und Muster erkennen sollte, und nannten es passend zu seinem Anwendungsbereich *Cognitron*. Mit diesem Modell wollte er das beim Perzeptron auftretende Problem, dass zur erfolgreichen Mustererkennung stets ein normiertes, also ein im Erfassungsbereich des Perzeptrons liegendes und größentechnisch adaptiertes, Bild verwendet werden musste, lösen. Inspiriert wurde er hierbei durch die Neurowissenschaften, denn

ein Mensch geht bei der Erkennung von beispielsweise handschriftlichen Buchstaben stets Schritt für Schritt vor. So werden zuallererst einzelne Merkmale wie Linien und Kurven gesucht und im Anschluss wird ausgehend von gemachten Erfahrungen bezüglich deren Auftreten beziehungsweise Fehlen auf den jeweiligen Buchstaben geschlossen ([92], S. 70-71). Fukushima konstruierte nun ausgehend von dem Grundprinzip des schrittweisen Erkennungsprozesses das erste mehrschichtige, schon fast tiefe Netzwerk. Die Schichten waren hierarchisch angeordnet und in jeder Schicht nahm die Anzahl an Neuronen im Vergleich zur vorhergehenden ab ([35], S. 305). Die synaptischen Verbindungen zwischen den Neuronen passten sich, wie auch beim Menschen, ausgehend von gemachten Erfahrungen, ständig an, indem sich deren Gewichte änderten. Die Schichten bestanden aus aktivierenden und inhibitorischen Neuronen und es gab nur Quervernetzungen von einem Neuron zu Neuronen der vorher liegenden Schicht, wenn diese in dessen Nähe liegen (vgl. [35], S. 305). Eine Verbindung zwischen den in hintereinander folgenden Schichten liegenden Neuronen A und B veränderte sich in Fukushimas Modell immer dann, wenn die in der früheren Schicht liegende Zelle A eine aktive war, also ein Signal an die nachkommende Zelle B sandte, und wenn zusätzlich keine der die Zelle B umgebenden Zellen ein stärkeres Signal aussandte (vgl. [25], S. 121-123). Daher wird eine solche Umgebung der Zelle B auch oft als *Connection Competition Region* bezeichnet. Aufgrund der Voraussetzungen zur Anpassung von Verbindungen werden somit immer nur gewisse Neuronen trainiert, nämlich genau jene, welche bereits auch in vorangehenden Trainings relevant waren (vgl. [35], S. 305). Daraus resultiert, dass jede Zelle in der finalen Schicht die Information eines ganzen Bereichs der ersten Schicht enthält. Das Cognitron war ein selbstorganisierendes Netzwerk, das heißt, es hat die Verbindungen zwischen den Neuronen ohne Kenntnis der gewünschten Outputs der Trainingsdaten passend verändert. Diese Methode wird auch unüberwachtes Lernen genannt, da es keine im übertragenen Sinne Lehrperson gibt, die vorgibt, was der zu einem Input richtige Output ist. Des Weiteren hatten diese Netzwerke auch eine Art selbstreparierenden Effekt. Denn fiel ein Neuron irgendwann einmal aus und wurde inaktiv, so sprang automatisch ein benachbartes Neuron ein, und zwar jenes, welches am nächststärksten auf das eingehende Signal reagiert hatte (vgl. [25], S. 121-123). Die Mustererkennung auf Basis dieses Modells gelang jedoch noch nicht unabhängig von der Position sowie Größe des Motivs, was jedoch das ursprüngliche Ziel Fukushimas war.

Über die Jahre hinweg arbeitete Fukushima, durch weitere Publikationen und Erkenntnisse im Bereich der Neurowissenschaften und künstlicher neuronaler Netze beeinflusst, immer wieder an seinem Modell und es entstand im Jahr 1980 das sogenannte *Neocognitron*. Der Aufbau dieses selbstorganisierenden Netzwerks ist inspiriert von dem Modell des visuellen Systems nach Hubel und Wiesel. Diese erforschten neuronale Netze von Säugetieren und erkannten eine hierarchische Struktur. Denn bei solchen Netzen bestehen frühere Schichten aus einfachen Zellen, spätere enthalten komplexe Neuronen und finale Schichten sind aus hyperkomplexen Zellen niedriger beziehungsweise auch höherer Ordnung aufgebaut. Das Neocognitron besitzt nun, angelehnt an Hubel und Wiesel, mehrere Schichten, wobei immer zwei aufeinanderfolgende ein Modul bilden. Innerhalb dessen wird die erste Schicht aus sogenannten S-Zellen (*Simple Cells*) und die zweite Schicht

aus C-Zellen (*Complex Cells*) gebildet. Die Verbindungen zu den S-Zellen sind adaptiv und passen sich nach jedem Trainingsdurchlauf, sprich nach jedem dem Netzwerk vorgesetzten Datensatz, neu an. Während eines Erkennungsvorgangs werden lokale Merkmale zunächst durch die einfacheren Zellen der vorderen Schichten erfasst und anschließend schrittweise zu globalen Merkmalen zusammengefasst, sodass ein ganzheitliches Muster erhalten wird, welches alle Informationen enthält. Durch wiederholtes Stimulieren mit Hilfe der Trainingsdaten aktiviert schlussendlich jedes gewisse Muster genau eine C-Zelle in der finalen Schicht und auch die Umkehrung gilt, das heißt, jedes Neuron in dieser letzten Schicht reagiert selektiv auf genau ein Motiv. Während des Prozesses des Auffindens lokaler Merkmale und des allmählichen Zusammenfügens zu einem Gesamtobjekt sind stets bei jedem Schritt kleine Fehler erlaubt. Durch diese Architektur und Arbeitsweise des neuronalen Netzes ist die Reaktivität der Neuronen auf das jeweils zugehörige Muster (fast) unabhängig von dessen Position, von dessen Größe sowie von leichten Veränderungen der Gestalt (vgl. [26], S. 193-194). Das Neocognitron war die Grundlage der heutigen *Convolutional Neural Networks*, kurz CNNs (vgl. [97], S. 90).

1.3.2 Konnektionismus

Mitte der 1980er Jahre wurden zwei große Werke von Rumelhart und McClelland sowie der PDP-Forschungsgruppe zu *Parallel Distributed Processing* (PDP), später auch oft als Konnektionismus bezeichnet, veröffentlicht. Es kam dadurch zu einem Wiederaufleben im Bereich der neuronalen Netze und der KI-Winter war beendet. Denn in diesen Werken wurde das von Minsky und Papert aufgezeigte Problem der XOR-Funktion durch Anwendung von mehrschichtigen neuronalen Netzen, wie beispielsweise eines mehrschichtigen Perzeptrons (*Multilayer Perceptron*, MLP), gelöst. Der Konnektionismus entstand in Verbindung mit den Kognitionswissenschaften und zwar genauer mit der Idee dort künstliche neuronale Netze einzusetzen (vgl. [23], S. 21). Die Kognitionswissenschaften beschäftigen sich mit der Frage, wie unser Bewusstsein und unser Denken funktioniert, wobei mehrere Disziplinen wie beispielsweise die Philosophie, Psychologie und Neurowissenschaften eine Rolle spielen und stets unterschiedliche Ebenen Teil einer Analyse sind. In den frühen Achtzigerjahren war insbesondere das symbolische logische Schließen im Fokus der Kognitionswissenschafterinnen und -wissenschafter. Allerdings war dieser Vorgang mit Hilfe von damaligen Modellen, wie dem Neocognitron, äußerst schwer umsetzbar, da bei diesen jeder Begriff durch eine hierarchische Struktur mit genau einer lokalen Einheit vernetzt war, was als lokale Repräsentation bezeichnet wird. Im Gegenzug dazu wurde im Konnektionismus versucht, die Informationsverarbeitung mittels künstlicher neuronaler Netze zu realisieren. Dabei war ein neuer Ansatz nötig, der nun auf der Idee beruhte, Verarbeitungsvorgänge mit Hilfe von Teilmustern und -symbolen parallel laufen zu lassen. Das Modell besteht also aus einer Vielzahl gleichzeitig arbeitender Einheiten, die jedoch selbst alle sehr einfach aufgebaut sind (vgl. [68], S. 63). Ein bis heute wichtiges Konzept, das aus dem Konnektionismus hervorging und auf obiger Idee basiert, ist das von Hinton, McClelland und Rumelhart entwickelte Konzept der *verteilten Repräsentation*. Jedes Objekt soll hierbei durch ein konkretes Muster, das aus einer Vielzahl an aktivierten Einheiten besteht, dargestellt werden. Umgekehrt ist wiederum

jede Verarbeitungseinheit an der Darstellung vieler verschiedener Objekte beteiligt. Dies ist zwar eine deutlich kompliziertere Methode, wenn man deren Darstellung sowie Implementierung betrachtet, jedoch ist die Effizienz, mit welcher sie insbesondere bei ihrer Ausführung von der Struktur neuronaler Netze Gebrauch macht, derart herausragend, dass somit die Vorteile gegenüber den Nachteilen überwiegen (vgl. [42], S. 77-78).

Zudem wurde 1986 der bereits im KI-Winter entwickelte Backpropagationsalgorithmus durch die PDP-Werke nun endlich an die Öffentlichkeit gebracht. Ein weiterer großer Fortschritt war die Anwendung der Backpropagation bei tiefen, künstlichen neuronalen Netzwerken zum Trainieren der internen Repräsentationen von gegebenen Inputs durch LeCun im Jahr 1987. Diesen wandte er zwei Jahre später erfolgreich zur Erkennung von handgeschriebenen Postleitzahlen an (vgl. [59]). Heutzutage ist die Backpropagation nach wie vor die am meisten verwendete Methode zur Berechnung des Gradienten beim Gradientenabstiegsverfahren.

Ab dem Ende der 1980er Jahre sowie zu Beginn der 90er Jahre kam es in der Forschung zu neuronalen Netzen zu einem richtigen Boom. Neuronale Netze wurden in den unterschiedlichsten Bereichen angewandt, beispielsweise in der Medizin zur Diagnose bei abdominalen Schmerzen (vgl. [59]) sowie zur Klassifizierung eines Sonar-Signals, welches von zwei unterschiedlichen Objekten im Meer, genauer Metallen oder Steinen, stammen kann (vgl. [33], S. 1135). Bis zu diesem Zeitpunkt waren die meisten Realisierungen mit Hilfe neuronaler Netze, die nur wenige Schichten enthielten. Es wurde jedoch recht bald erkannt, dass die Backpropagation an sich noch kein Wundermittel war und mathematische Probleme enthielt, die insbesondere auftraten, wenn der Algorithmus bei besonders tiefen Netzen angewandt wurde. Hierbei kommt es nämlich zu dem Problem, dass die durch das Verfahren wiederholt rückpropagierten Signale, welche die Informationen des Gradienten der Fehlerfunktion beinhalten, mit der Zahl an Schichten im Netzwerk exponentiell abnehmen oder unbeschränkt wachsen können. Dieses sogenannte Problem der verschwindenden oder explodierenden Gradienten wurde 1991 von Hochreiter in seiner Dissertation erstmals identifiziert und analytisch behandelt (vgl. [45], S. 21). Ebenso tritt es bei rekurrenten neuronalen Netzwerken (RNNs) auf. Solche Netze enthalten zusätzlich Verbindungen, die auch rückgerichtet sind, also zur selben Schicht oder früheren Schichten zurückführen. Netze, bei denen der Informationsfluss hingegen nur in eine Richtung abläuft, die also eine hierarchische Struktur aufweisen, wie beispielsweise das McCulloch-Pitts-Neuron oder Rosenblatts Perzeptron, werden *neuronale Feedforward-Netze (feedforward neural networks, FNNs)*, kurz auch einfach Feedforward-Netze, genannt. Rekurrente Netze können Informationen von vorangegangenen Inputs zwischenspeichern und für künftige Eingaben sowie für zu berechnende Outputs verwenden, wodurch auch ein zeitlicher Aspekt hinzukommt. Daher werden sie meist bei Anwendungen, wo Sequenzen verarbeitet werden müssen, eingesetzt, wie zum Beispiel bei der Spracherkennung. Das Training rekurrenter Netzwerke erfolgt mit Hilfe einer Erweiterung der Backpropagation, der *Backporpagation Through Time*. Auch hier wurde im Jahr 1994 von Bengio et al. das Problem des verschwindenden Gradienten festgestellt, nachdem es insbesondere bei der Verarbeitung langer Sequenzen bei vielen der Netzwerke zu Fehlern gekommen war (vgl. [5], S. 157).

Es gab jedoch ein wenig Trost, nachdem zunehmend mehr empirische Analysen ergaben, dass zusätzliche Schichten in vielen Anwendungsfällen nicht unbedingt zu besseren Ergebnissen führten. Es stellte sich nun die Frage, wie viele Schichten überhaupt nötig seien, um gewisse Funktionen mit neuronalen Netzen darstellen zu können. Die Antwort darauf fanden unter anderem Cybenko im Jahr 1989 sowie 1991 in weiterentwickelter Form Hornik, und sie ist heute als universelles Approximationstheorem bekannt. Dieses besagt, dass ein neuronales Netzwerk mit nur einer Zwischenschicht zwischen der Ein- und Ausgabeschicht, die aus hinreichend vielen Einheiten besteht, bereits jede stetige Funktion in mehreren Variablen sowie auch bestimmte weitere Funktionstypen approximieren kann. Ebenso wurde zu dieser Zeit an einer Lösung des aufgetretenen Problems bei der Backpropagation geforscht. Sowohl Hochreiter als auch Bengio et al. hatten bereits ein paar mögliche Alternativen in ihren Publikationen vorgeschlagen. Einen recht erfolgreichen Ansatz fanden Hochreiter und Schmidhuber, die ein *Long short-Term Memory (LSTM)* Netzwerk konstruierten. Dabei sind im rekurrenten neuronalen Netz zusätzlich spezielle Einheiten, die *Constant Error Caroussels* (CECs), enthalten, welche den zurückpropagierten Fehler konstant halten und somit ein mögliches Verschwinden oder Explodieren des Gradienten bei der Backpropagation verhindern (vgl. [46], S. 1735). Durch diese CEC-Einheiten können LSTM-Netze Ereignisse, die bereits vor hunderten Schritten geschehen sind, speichern und für laufende Berechnungen verwenden. Diesem Phänomen haben die Netze ihren Namen LSTM, auf Deutsch langes Kurzzeitgedächtnis, zu verdanken. Durch den Einsatz von LSTM-Netzen konnten viele bis dahin mit tiefen neuronalen Netzen unlösbare Aufgaben, wie beispielsweise bei der Spracherkennung, wo oft lange Zeitspannen zwischen den Inhalten auftreten, oder bei der optischen Zeichenerkennung, realisiert werden (vgl. [97], S. 95).

Mitte der 1990er Jahre kam es teilweise dazu, dass Unternehmen, die im Bereich der künstlichen Intelligenz arbeiteten, um mehr Investoren zu erhalten, Erfolge versprachen, die jedoch schlussendlich nicht realisierbar waren (vgl. [32], S. 17). Des Weiteren wurde 1995 eine neue Methode des maschinellen Lernens entwickelt, die nicht auf künstlichen neuronalen Netzen basiert. Diese *Support Vector Maschines* (SVMs) wurden zur Klassifizierung in zwei Gruppen konstruiert, wobei stets eine der beiden Klassen ausgegeben wird. Auch neuronale Netze können Klassifizierungsaufgaben bewältigen, jedoch wird hierbei ein ganz anderer Algorithmus eingesetzt, der auf der Berechnung von Wahrscheinlichkeiten beruht. Bei Support Vector Maschines basiert die Zuordnung zu einer Gruppe auf einer anderen Idee, denn hier werden die Inputs zunächst durch eine nichtlineare Abbildung in einen höherdimensionalen Merkmalsraum abgebildet. Dort wird nun eine Hyperebene konstruiert, die diese Einträge bestmöglich separiert. Das mathematische Grundprinzip hinter dieser Methode, bei der eine nichtlineare Klassifizierung durchgeführt wird, ist als *Kernel-Trick* bekannt. Aufgrund der Eigenschaften dieser Hyperebene kann ein sehr geringer Generalisierungsfehler, also der Erwartungswert des Fehlers bei neuen, unbekannten Inputs, erzielt werden (vgl. [15], S. 273). Neben Support Vector Machines und weiteren *Kernel Machines* wurden auch zunehmend sogenannte *Probabilistische Graphische Modelle* (PGMs) im maschinellen Lernen eingesetzt. Aufgrund dieser Vorkommnisse endete das zweite Hoch in der Forschung zu neuronalen

Netzen Mitte der Neunzigerjahre.

Dennoch gab es auch zu dieser Zeit einige große Weiterentwicklungen. Yann LeCun und seine Forschungsgruppe untersuchten mehrschichtige neuronale Netze zur Erkennung handgeschriebener Ziffern. Es wurden gleich mehrere Netzwerkstrukturen betrachtet, die alle mit Hilfe eines gradientenbasierten Algorithmus mit Backpropagation trainiert wurden. Ebenso stellten sie einen besonders vielversprechenden Netzwerktypen, sogenannte *Graph Transformer Networks* (GTNs), deren Architektur auf Convolutional Neuronal Networks basiert, vor. Bei ihren Untersuchungen konnten einige wichtige Einflussfaktoren für ein erfolgreiches Training abgeleitet werden, die auch bei der Lösung möglicher Schwierigkeiten bei anderen Anwendungen von Mustererkennung wichtig sein können. Die Trainingsmenge, die LeCun et al. für ihre Netzwerke verwendeten, war die *Modified National Institute of Standards and Technology Database*, kurz MNIST-Database. Hierbei handelt es sich um eine Datenbank, die handgeschriebene Zeichen enthält. Sie ist eine Zusammensetzung zweier NIST-Datenmengen, welche vom National Institute of Standards and Technology gesammelt und öffentlich zur Verfügung gestellt wurden. Die eine Auswahl handgeschriebener Ziffern stammt von Highschool-Schülerinnen und -Schülern, die andere von Büroangestellten. Durch Kombination dieser beiden erhält man eine Trainingsmenge, die wesentlich vielfältiger ist und somit den Generalisierungfehler vermindern kann (vgl. [57], S. 2278-2318).

1.3.3 Die dritte Welle - Deep Learning

Obwohl mehrschichtige neuronale Netze, wie beispielsweise Fukushimas Perzeptron, bereits seit Jahrzehnten für diverse Anwendungen eingesetzt und beforscht wurden und die Anzahl an Schichten ab den Neunzigerjahren auch deutlich stieg, wurden die Begriffe *Deep Neural Networks*, also neuronale Netze mit etlichen Zwischenschichten, und *Deep Learning* erst ab 2006 populär (vgl. [97], S. 96). Dies ist auf einen Durchbruch in der Forschung zu tiefen Netzen zurückzuführen, der Hinton, Osindero und Teh in diesem Jahr gelungen war. In ihrem Paper zeigten sie, dass eine bestimmte Art von tiefen Feedforward-Netzen, sogenannte *Deep Belief Networks*, erfolgreich trainiert werden konnten. Die Voraussetzung hierfür war, dass die Gewichte eine gewisse Voranpassung erhielten, wodurch die Initialisierung somit nicht mehr auf Zufallswerten beruhte. Dieses Verfahren des unüberwachten Lernens wird als *Greedy Layer-Wise Pretraining* bezeichnet und es wird dabei, wie der Name schon vermuten lässt, immer nur jeweils eine Schicht auf einmal trainiert. Hinton und sein Team testeten den Algorithmus mit Hilfe der MNIST-Datenbank und stellten fest, dass ihr Netzwerk einen Generalisierungsfehler, auch Testfehler genannt, von $1,25\%$ erreichte. Damit übertraf es durch Backpropagation trainierte neuronale Netze mit nur wenigen Schichten, die einen Testfehler von mindestens $1,5\%$ aufwiesen, und sogar Support Vector Maschines, die auf einen Fehler von $1,4\%$ kamen (vgl. [43], S. 1527-1544). Diese Errungenschaft verbreitete sich rasch im Forschungsfeld neuronaler Netze und viele Gruppen wurden durch die Herangehensweise des Pretrainings inspiriert, wodurch sie diese auch für andere Netze adaptierten. Yoshua Bengio et al. erweiterten 2007 den Algorithmus für stetige Inputs, da dieser bisher nur für binäre Inputs konzipiert war. Ein weiteres Resultat ihrer Untersuchungen war, dass

das unüberwachte Pretraining, also die Methode selbst, der Schlüssel zum Erfolg war und dafür nicht zwingend die Architektur von Deep Belief Networks nötig war. Es konnten also ähnliche Erfolge bei herkömmlichen tiefen neuronalen Netzen erzielt werden, wenn die Schichten als *Autoencoder* trainiert wurden (vgl. [6], S. 2-10). Ranzato et al. wandten die neue Methode der unüberwachten Voranpassung der Parameter bei Netzwerken, die mit Hilfe von *Sparse Respresentations* lernen sollten, an. Bei Sparse Representations, also dünnbesetzten Repräsentationen, werden die Inputdaten durch eine Darstellung, deren Einträge fast ausschließlich 0 sind, repräsentiert. Dies erfordert in der Regel eine höher dimensionierte Darstellung, um einen größeren Informationsverlust zu vermeiden (vgl. [32], S. 144). Eine solche Herangehensweise, die *Sparse Coding* genannt wird, hat den Vorteil, dass Merkmale in Bildern leichter separiert werden können, im besten Fall sogar linear. Ranzato et al. konnten beim Testdurchlauf mit ihrem Convolutional Neural Network, dessen erste Schicht mit Hilfe des unüberwachten Lernalgorithmus initialisiert wurde, bei dem MNIST-Datensatz einen Generalisierungsfehler von $0,6\%$ erreichen. Bei zusätzlichen Verzerrungen in der Datenmenge erzielten sie sogar einen Testfehler von $0,39\%$ und stellten somit einen neuen Rekord auf (vgl. [90], S. 1-7). All diese Erfolge basierten auf einer tiefen Architektur, durch die nun auch das Durchführen komplexer Aufgaben möglich war. Denn hierfür ist eine gute, hochdimensionierte Darstellung der Daten nötig, sodass auch übergeordnete, abstraktere Merkmale, sogenannte *High-Level Features*, gelernt werden können. Dies geschieht, indem zunächst niederrangige, einfache Merkmale, die *Low-Level Features*, gesucht und im Anschluss zu höherrangigen Charakteristika kombiniert werden. Architektonisch betrachtet erfordert dies dementsprechend auch viele Zwischenschichten im Netzwerk (vgl. [7], S. 321).

In den darauffolgenden Jahren wurde viel zur Bedeutung der Tiefe bei neuronalen Netzen geforscht. Es konnten einige Limitationen bei seichten Netzen (*Shallow Networks*), also Netze mit nur ein oder in seltenen Fällen zwei Zwischenschichten, sowohl empirisch als auch theoretisch festgestellt werden. So können zwar sowohl seichte als auch tiefe Netze jede stetige Funktion darstellen, jedoch wird bei ersteren ein deutlich höherer Rechenaufwand, aufgrund der höheren Anzahl durchzuführender Operationen, erzeugt. Des Weiteren wird wegen der geringen Tiefe des Netzes eine größere Trainingsdatenmenge benötigt. Zusammenfassend konnte also geschlossen werden, dass tiefe Netze in vielen Anwendungen eine deutlich höhere Effizienz aufweisen (vgl. [7], S. 35-36). Im Jahr 2010 konnte mit einem simplen tiefen Netzwerk, welches die Struktur eines mehrschichtigen Perzeptrons (*Multi-Layer Perceptron*, kurz MLP) aufwies, ein neuer Rekord mit dem MNIST-Datensatz erzielt werden. Dieses wurde mit dem klassischen Backpropagationsalgorithmus trainiert, erfuhr also kein unüberwachtes Pretraining, und erreichte einen Testfehler von $0,35\%$. Die Forschungsgruppe untersuchte dabei gleich unterschiedliche Architekturen, die sich in der Anzahl an Neuronen pro Schicht sowie in der Menge an Schichten unterschieden. Das Netz, das obigen Erfolg erzielt hatte, war jenes mit den meisten Schichten sowie Neuronen innerhalb dieser. Außerdem erkannten die Beteiligten, dass mit Hilfe von bewusst erzeugten Deformationen in den Beispielen des MNIST-Datensatzes, was eine breitere und größere Trainingsmenge zur Folge hatte, bessere Ergebnisse erzielt werden können (vgl. [14], S. 3211-3212).

Die obigen Erfolge waren unter anderem auch deshalb möglich, da sich die Hardware seit den Neunzigerjahren enorm verbessert hatte. CPUs (*Central Processing Units*) wurden immer schneller, jedoch gab es auch hier Grenzen. Verbesserungen in der Verarbeitungsgeschwindigkeit konnten durch den parallelen Einsatz mehrerer CPUs erzielt werden, jedoch waren diese Einheiten meist immer noch zu schwach, um tiefe Netzwerke, die mittlerweile beachtliche Größen erreicht hatten, zu trainieren. Durch den Einsatz von GPUs (*Graphics Processing Units*) konnte die Rechenzeit noch einmal deutlich verkürzt werden. Dies liegt daran, dass ihre Kerne, von denen sie in der Regel hunderte aufweisen, alle zeitgleich arbeiten können und somit eine besonders feine Parallelität in der Verarbeitung ermöglichen. Dadurch können Berechnungsverfahren, die normalerweise schwer zu parallelisieren sind, im großen Maßstab durchgeführt werden (vgl. [88], S. 873-874).

Ein weiterer wichtiger Faktor dafür, dass tiefe Netze heutzutage erfolgreich trainiert werden können, ist das Vorhandensein von ausreichend großen Trainingsdatenmengen. Die Fortschritte, die im Deep Learning in den 2000er Jahren gemacht wurden, waren immens, jedoch fehlten für Netze in dieser Dimension geeignete Trainingssätze, die MNIST-Datenbank war nicht mehr ausreichend. Abhilfe brachte die im Jahr 2009 veröffentlichte *ImageNet*-Datenbank. Diese bestand zu diesem Zeitpunkt aus $3,2$ Millionen Bildern, die jeweils einer von 5247 Kategorien zugeordnet waren, sodass jede im Schnitt 600 hochaufgelöste Bilder enthielt (vgl. [17], S. 248-249). Mittlerweile (Stand Mai 2021) besteht ImageNet aus über 14 Millionen Bildern (vgl. [99]). Neben Datenbanken für visuelle Anwendungen gab es auch welche für andere, wie beispielsweise für die Spracherkennung. Hier wird sehr oft der *TIMIT-Korpus* verwendet, der Sprachaufnahmen sowie alle Transkripte von 630 unterschiedlichen Personen enthält, die jeweils 10 verschiedene Sätze eingesprochen haben (vgl. [64]). Auch im Bereich der Spracherkennung lieferten tiefe Feedforward-Netze neue Rekorde bei Benchmarks mit dem TIMIT-Korpus (vgl. [71], S. 14).

Nun, wo passende Hardware und große Testmengen verfügbar waren, wurden in den darauffolgenden Jahren beeindruckende Errungenschaften mit tiefen Netzwerkarchitekturen erzielt. So schlossen sich 2011 vier Forschungsgruppen, jeweils eine von IBM, Microsoft und Google sowie der University of Toronto, zusammen und das Projekt *Google Brain* war geboren. Das Ziel dieser Kollaboration war herauszufinden, welches Potential in tiefen neuronalen Netzen steckte und wie das Leben der Menschen durch diese erleichtert werden kann (vgl. [44], S. 83). Einer ihrer bekanntesten Erfolge war ein 2012 konstruiertes neuronales Netz, das lernen sollte Gesichter zu erkennen. Dieses hatte eine Milliarde Verbindungen und wurde unüberwacht trainiert, indem es 3 Tage lang 10 Millionen unklassifizierte, also ungelabelte, Bildausschnitte von YouTube-Videos als Inputdaten erhielt. Beim Testen mit Bildern von Gesichtern, unter anderem aus ImageNet, sowie vielen Distraktoren aus anderen Kategorien, konnte das beste Neuron eine Richtigkeit von $81,7\%$ erzielen. Wären alle Gesichter nicht erkannt worden, entspräche dies einem Wert von $64,8\%$. Außerdem war das Ergebnis unabhängig von Transformationen wie Translationen oder Skalierungen. Nachdem die meisten Motive in den Bildern Körperteile und Haustiere waren, schloss die Forschungsgruppe, dass auch diese gelernt wurden. Tatsächlich erkannte das Netzwerk, nachdem zwei Testdurchläufe mit

Bildern von Körpern beziehungsweise Katzengesichtern durchgeführt wurden, diese jeweils mit einer sehr hohen Genauigkeit. Bei der ImageNet-Testmenge schaffte das Netz einen Fehler von $15,8\%$, was eine relative Steigerung von 70% im Vergleich zum damaligen Rekordhalter darstellte. Die erfolgreiche Verwendung ungelabelter Daten zum Training stellte einen Durchbruch im maschinellen Lernen dar. Man stand nun nicht mehr vor der Herausforderung entsprechend große Mengen klassifizierter Daten für Netze dieser Dimension zu bekommen, sondern konnte die deutlich einfacher zugänglichen Datensätze ohne Kennzeichnungen verwenden (vgl. [58], S. 1-7). In den darauffolgenden Jahren konnte das Team in vielen weiteren Gebieten durch den Einsatz seiner Netze große Verbesserungen und neue Errungenschaften bewirken, wie unter anderem eine Adaption des *Google Translate*-Algorithmus, sodass dieser nun deutlich weniger Fehler bei Übersetzungen aufwies. Da die Forschungsgruppe viel Potential für weitere Anwendungsgebiete sah, die sie selbst jedoch nicht alle in Projekten realisieren konnte, konstruierte sie *TensorFlow*. Hierbei handelt es sich um eine Open-Source-Plattform für neuronale Netze, die Bibliotheken, Tools und Beispiele zur Verfügung stellt und mit Hilfe derer nun jede Person ihr eigenes neuronales Netz konstruieren kann. Die von einzelnen Personen erstellten Netzwerke können dann auf der Plattform öffentlich zur Verfügung gestellt werden. Außerdem forschte das Team im Bereich der Medizin, wo es unter anderem ein Programm zur Vorhersage des Grades der Metastasierung bei Brustkrebs konstruierte (vgl. [40], S. 73-75).

Nachdem die Methode des Deep Learnings Einzug in die Industrie gefunden hatte, waren viele Wissenschafterinnen und Wissenschafter erneut an der Frage interessiert, wieso genau die Backpropagation derart schlechte Ergebnisse bei tiefen Netzwerken ohne vorangegangene Initialisierung der Gewichte lieferte. Ebenso stand die Frage im Raum, wie man die eingesetzten Algorithmen noch optimieren konnte. Glorot und Bengio konnten 2010 unter anderem erkennen, dass die gewählte, nichtlineare Aktivierungsfunktion einen großen Einfluss auf die erfolgreiche Gewichtsanpassung hat. Des Weiteren sollten die Gewichte besser schichtweise angepasst werden, da durch das herkömmliche Verfahren der Gradient von Schicht zu Schicht meist immer kleiner wurde. Es stellte sich heraus, dass die ReLU-Funktion mit der Vorschrift $f(x) = \max(0, x)$ die beste von allen bis dahin geläufigen Aktivierungsfunktionen für Netze mit vielen Schichten war (vgl. [29], S. 255-256).

Zusammenfassend lässt sich heute sagen, dass erfolgreiches Lernen drei Grundvoraussetzungen mit sich bringt: genügend Trainingsdaten, leistungsstarke Hardware sowie angepasste, tiefe Netzwerkarchitekturen (vgl. [56]). Die Deep Learning Revolution hält bis jetzt an und es ist zu erwarten, dass auch künftig viele weitere Errungenschaften in der Arbeit mit tiefen Netzen im Bereich der künstlichen Intelligenz entstehen werden.

1.4 Der Lernalgorithmus im Machine Learning

Das Ziel eines Lernalgorithmus besteht darin, eine gewisse Aufgabe zu lösen. Eine solche Aufgabe wird über die Art, wie ein gegebener Input, in diesem Fall auch als Beispiel bezeichnet, verarbeitet werden soll, definiert. Der Input ist in Form eines Vektors $x \in \mathbb{R}^n$

gegeben, wobei die einzelnen Einträge die Merkmale und somit Eigenschaften des gegebenen Inputs repräsentieren. So stellen beispielsweise Pixelwerte die Merkmale eines Bildes dar und wären dementsprechend die Einträge des Inputvektors eines Bildes. Typische Aufgaben können Klassifizierung, Regression oder Transkription sein.

Des Weiteren wird stets ein Maß für die Güte eines Algorithmus benötigt. Im Falle einer Klassifizierung, oder einer Klassifizierung fehlender Daten, wird oft die Genauigkeit bestimmt, das heißt, es wird angesehen, wie viele der Probedaten zum richtigen Output geführt haben. Eine weitere Methode ist das Berechnen der Fehlerrate, bei der die relative Häufigkeit der falschen Outputs betrachtet wird. Bei manchen Aufgaben, wie einer Dichteschätzung, sind diese Methoden jedoch nicht sinnvoll und es benötigt weitere. Um zu beurteilen, wie gut nun ein Lernalgorithmus performt, werden dem Netzwerk typischerweise neue Inputdaten, die bisher noch nicht zum Lernen verwendet wurden, übergegeben. Diese Menge wird auch Testmenge genannt.

1.4.1 Überwachtes und unüberwachtes Lernen

Um eine Aufgabe lösen zu können, müssen während des Lernprozesses Erfahrungen gesammelt werden. Diese werden mit Hilfe eines gegebenen Datensatzes, der der Maschine vorgelegt wird, gemacht. Hierbei handelt es sich um eine Sammlung vieler Beispiele. Je nachdem in welcher Art die Daten vorliegen, unterscheidet man zwischen zwei Typen von Algorithmen. Bei überwachten Lernalgorithmen (*Supervised Learning*) ist zu jedem Input der Datenmenge auch bereits der gewünschte Output vorhanden. Da hier also der Maschine wie durch eine Art Instruktorin beziehungsweise Instruktor gesagt wird, wie die Ausgabe zu einem Input auszusehen hat, wird der Lernalgorithmus als „überwacht“ bezeichnet. Bei einer solchen Form der Daten, bei der sie paarweise vorliegen, spricht man auch von der Trainingsdatenmenge. Die Maschine soll also ausgehend davon eine passende Zuordnung finden.

Beim unüberwachten Lernen (*Unsupervised Learning*) sind hingegen ausschließlich Inputs gegeben und das Ziel ist, dass die Maschine aus diesen gewisse Strukturen ableiten kann. Der Algorithmus soll hierbei ein statistisches Modell finden, das die Inputs möglichst gut repräsentiert und deren Zusammenhänge sowie Muster beinhaltet. Auf Grundlage dessen soll die Maschine Vorhersagen zu neuen, unbekannten Eingaben treffen können. In der Realität besteht jedoch oft keine genaue Grenze zwischen den beiden Algorithmustypen. So existieren beispielsweise auch teilüberwachte Methoden, bei der die Maschine sowohl Datenpaare als auch rohe Inputs erhält. Ebenso können Probleme im unüberwachten Lernen auf kleinere Teilprobleme aufgeteilt werden, die mit Hilfe überwachter Lernprozesse gelöst werden können, et vice versa (vgl. [32], S. 97-104).

Eine weitere Form des maschinellen Lernens stellt das sogenannte bestärkende Lernen (*Reinforcement Learning*) dar. Hierbei sind die Inputs nicht starr, sondern das Netzwerk steht in direkter Interaktion mit seiner Umgebung, indem es in der jeweiligen Situation Entscheidungen trifft. Daraufhin bekommt die Maschine direkte Rückmeldungen auf ihre Erfahrungen. Diese Feedbacks können positiv, im Sinne von Belohnungen, aber auch negativ sein. Infolgedessen soll der Algorithmus die bestmögliche Taktik finden, um sich in der Umgebung zu bewegen.

2 Neuronale Feedforward-Netze

Neuronale Feedforward-Netze, kurz FNNs (*Feedforward Neural Networks*), und insbesondere tiefe Feedforward-Netze sind die heutzutage am meisten verwendeten Architekturen im Bereich des Deep Learnings. Der Name dieser künstlichen neuronalen Netzwerke kommt daher, dass in diesen das Signal stets nur in eine Richtung, nämlich nach vorne, geleitet wird, das heißt, es gibt keine Rückkopplungen. Dies führt dazu, dass das Verhalten des Netzes unabhängig von zurückliegenden Ereignissen ist und es somit ausschließlich auf derzeitige Inputs reagiert. Der Vorteil von Feedforwad-Netzen besteht unter anderem in deren einfachem Aufbau und der Möglichkeit, diese in verschiedensten, teils auch komplexen Anwendungsgebieten einzusetzen (vgl. [95], S. 12).

2.1 Die Funktionsweise eines Feedforward-Netzes

Ein Feedforward-Netz dient der Approximation einer Funktion f. Dies kann beispielsweise eine Klassifikation sein, bei der jedem Input $x \in \mathbb{R}^n$ seine zugehörige Klasse $y = f(x)$ zugeordnet wird. Die Annäherung durch das Netzwerk an die gewünschte Funktion erfolgt über einen Lernprozess. Hierbei bekommt das Netz Trainingsdaten vorgesetzt, mit Hilfe derer es die richtigen Parameter θ für eine adäquate Nachbildung $\hat{f}$ der gesuchten Funktion finden soll. Am Ende sollte das Netz dann in der Lage sein, zu einer gegebenen Eingabe durch die Berechnung von $\hat{y} = \hat{f}(\theta, x)$ eine passende Ausgabe zu liefern.

2.1.1 Aufbau eines Feedforward-Netzes

Ein Feedforward-Netz besteht in der Regel aus mehreren Schichten $s = 0, \ldots, S$ von Neuronen. Die Ausnahme bilden einschichtige Netze, die neben der Inputschicht ausschließlich eine Ausgabeschicht aufweisen. Die erste Schicht eines klassischen Feedforward-Netzes ist die Eingabeschicht (*Input Layer*). Auf diese folgen eine oder mehrere Zwischenschichten (*Hidden Layers*), auch verdeckte Schichten genannt, das Schlusslicht bildet die Ausgabeschicht (*Output Layer*). Die Neuronen einer Schicht sind mit denen der vorherigen sowie jenen der nachfolgenden Schicht verbunden. Innerhalb einer Schicht gibt es jedoch keine Vernetzungen zwischen den Einheiten. Die Zahl der Neuronen in der Ein- und Ausgabeschicht hängt von dem Anwendungsproblem ab. Der Input erfolgt in Form eines Vektors $x \in \mathbb{R}^n$, wobei jedem Neuron der Eingabeschicht genau ein Eintrag, also eine Komponente des Vektors x, übergeben wird, das heißt, bei einem n-dimensionalen Inputvektor besteht die Schicht aus n Neuronen. Ebenso wird der Output $\hat{y} \in \mathbb{R}^m$ als Vektor dargestellt, wobei auch hier die Ausgabe komponentenweise über die einzelnen Neuronen vor sich geht. Wie viele Neuronen in den Zwischenschichten zu

finden sind und wie viele Schichten das Netz überhaupt aufweist, legt die Entwicklerin beziehungsweise der Entwickler selbst fest. Darüber hinaus ist jede Verbindung mit einem Gewicht $W_{kl} \in \mathbb{R}$ ausgestattet, wobei sich der Index l auf das l-te Neuron der vorherigen Schicht und der Index k auf das k-te Neuron der im Moment betrachteten Schicht bezieht. Somit bezeichnet W_{kl} genau das Gewicht der Verbindung dieser beiden Neuronen. Alle Gewichte der Verbindungen können zur Gewichtsmatrix $W^{(s)}$ der jeweils betrachteten Schicht s zusammengefasst werden. Die Dimension dieser Matrix hängt von der Zahl der Neuronen N_{s-1} in der vorherigen sowie von der Neuronenzahl N_s der im Moment betrachteten Schicht s ab, entsprechend $W^{(s)} \in \mathbb{R}^{N_s \times N_{s-1}}$. Des Weiteren ist jedes Neuron mit einer Verzerrung (*Bias*) $b_k \in \mathbb{R}$ ausgestattet, weshalb zu jeder Schicht s auch ein Verzerrungsvektor $b^{(s)}$ gehört. Dieser hat genau so viele Einträge, wie Neuronen in der Schicht vorhanden sind, also $b^{(s)} \in \mathbb{R}^{N_s}$. Zusätzlich besitzt jedes Netzwerk eine (nichtlineare) Aktivierungsfunktion f. Hierfür wird sehr häufig die ReLU-Funktion $f : \mathbb{R} \to \mathbb{R}$ mit $f(x) = \max(0, x)$ verwendet, da sie unter anderem eine nichtlineare, genauer nicht-affine Funktion ist. Der Grund des Einsatzes einer nichtlinearen Funktion besteht darin, dass somit deutlich mehr Funktionen approximiert werden können. Ohne diese wären ausschließlich lineare beziehungsweise affine Abbildungen mit Hilfe von Feedforward-Netzen darstellbar. Die Anwendung der Aktivierungsfunktion auf den Argumentvektor erfolgt in der Regel komponentenweise.

In Abbildung 2.1 ist der Aufbau eines neuronalen Feedforward-Netzwerks für n-dimensionale Eingabe- und eindimensionale Ausgabewerte dargestellt. In diesem Beispiel besteht das Netz aus insgesamt drei Schichten, also $s = 0, 1, 2$, mit den zugehörigen Gewichtsmatrizen $W^{(1)}$ und $W^{(2)}$ sowie Verzerrungsvektoren $b^{(1)}$ und $b^{(2)}$.

2.1.2 Das mathematische Modell eines neuronalen Netzes

Sehen wir uns nun an, welche Schritte innerhalb einer Schicht ablaufen, wenn dem Netzwerk ein Inputvektor übergeben wurde. Der dadurch ausgelöste Verarbeitungsprozess besteht nun aus einer Vielzahl an Teilschritten, die jedoch alle gleich ablaufen. In jedem Teilschritt werden zunächst die Werte der vorherigen Schicht an die derzeit betrachtete Schicht übergeben, wobei jedes Neuron die Summe der gewichteten Outputs der vorherigen Schicht als Eingabe erhält. In der ersten verborgenen Schicht erhält das k-te Neuron somit den Wert $\sum_{l=1}^{n} W_{kl}^{(1)} x_l$. Nun wird noch der Bias des Neurons dazu addiert und es ergibt sich $v_k^{(1)} = \sum_{l=1}^{n} W_{kl}^{(1)} x_l + b_k^{(1)}$. Hier wird nun auch ein wenig klarer, wieso dieser auch als Verzerrung bezeichnet wird. Würde nämlich das Neuron keine Signale empfangen, sprich wenn alle Gewichte auf 0 gesetzt wären, so nimmt es dennoch stets den Wert b_k an. Dieser ist also typisch für genau dieses Neuron und man könnte den Wert schon fast als eine Art Markenzeichen des Neurons bezeichnen. Am Ende wird auf den bisher errechneten Wert noch die Aktivierungsfunkion angewandt: $u_k^{(1)} = f\left(v_k^{(1)}\right) = f\left(\sum_{l=1}^{n} W_{kl}^{(1)} x_l + b_k^{(1)}\right)$. Das Ergebnis $u_k^{(1)}$ ist nun der Output des k-ten Neurons der ersten Zwischenschicht, welcher nun an die Einheiten der nächsten Schicht übergeben wird. Dies erfolgt wieder über eine Gewichtung sowie anschließende Summenbildung und der Berechnungsprozess wird nun in der neuen Schicht gestartet.

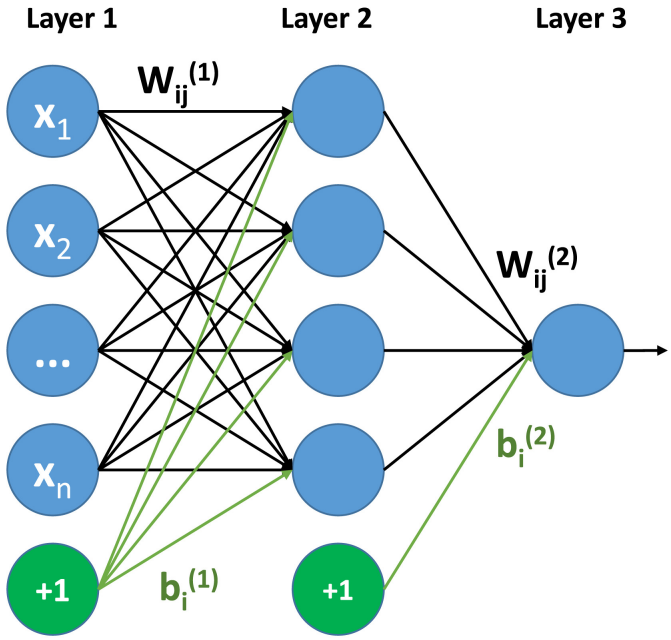


Abbildung 2.1: Typischer Aufbau eines einfachen Feedforward-Netzes mit der Inputschicht (Layer 1), einer Zwischenschicht (Layer 2) und der Outputschicht (Layer 3) ([4], Fig. 1)

Da jede Schicht s, abgesehen von der Eingabeschicht ($s = 0$) selbst, eine eigene Gewichtsmatrix $W^{(s)}$ und eigenen Biasvektor $b^{(s)}$ besitzt, ist der jeweilige Outputvektor dieser Schicht durch

$$u^{(s)} = f\left(v^{(s)}\right) = f\left(W^{(s)}u^{(s-1)} + b^{(s)}\right) \tag{2.1}$$

festgelegt. Es wird demnach also zu Beginn eines jeden Teilschrittes eine affine Abbildung, die durch $W^{(s)}$ und $b^{(s)}$ festgelegt ist, auf die Inputs, genauer auf die Outputs der vorherigen Schicht, angewandt, um den Vektor $v^{(s)}$ zu erhalten. Dieser ist das Argument der nicht-affinen Aktivierungsfunktion f, die jetzt komponentenweise angewandt wird und zum Ergebnisvektor $u^{(s)}$ führt. Der so erhaltene Vektor ist nun der Output der betrachteten Schicht. Zusammengefasst wird also der Eingabevektor $x = u^{(0)} \in \mathbb{R}^n$ über mehrmaliges, abwechselndes Anwenden affiner Abbildungen und der Aktivierungsfunk-

tion am Ende in den Ausgabevektor $\hat{y} \in \mathbb{R}^m$ überführt, also

$$\begin{aligned}
\hat{y} = u^{(S)} &= f\left(v^{(S)}\right) = f\left(W^{(S)} u^{(S-1)} + b^{(S)}\right) \\
&= f\left(W^{(S)} f\left(v^{(S-1)}\right) + b^{(S)}\right) \\
&= f\left(W^{(S)} f\left(W^{(S-1)} u^{(S-2)} + b^{(S-1)}\right) + b^{(S)}\right) \\
&= f\left(W^{(S)} f\left(W^{(S-1)} f\left(v^{(S-2)}\right) + b^{(S-1)}\right) + b^{(S)}\right) \\
&\vdots \\
&= f\left(W^{(S)} f\left(W^{(S-1)} f\left(\dots f\left(W^{(1)} u^{(0)} + b^{(1)}\right)\dots\right) + b^{(S-1)}\right) + b^{(S)}\right) \\
&= f\left(W^{(S)} f\left(W^{(S-1)} f\left(\dots f\left(W^{(1)} x + b^{(1)}\right)\dots\right) + b^{(S-1)}\right) + b^{(S)}\right).
\end{aligned} \tag{2.2}$$

Die Gesamtabbildung $x \mapsto \hat{y}$ ergibt sich de facto aus einer Verkettung mehrerer Funktionen. Bei einigen Netzen, beispielsweise bei solchen mit der ReLU-Funktion, entfällt meist die Anwendung der Aktivierungsfunktion in der Ausgabeschicht, das heißt, in diesem Fall wäre $\hat{y} = v^{(S)}$. Die Gewichtsmatrizen W und Biasvektoren b aller Schichten werden auch als *Parameter* dieser Zuordnung bezeichnet und gemeinsam durch das Symbol θ repräsentiert. Der Ausgabevektor ist also bei fixierten Parameterwerten ausschließlich vom Input x abhängig: $\hat{y}(x)$. Will man auch die Abhängigkeit von den Parametern kennzeichnen, so schreibt man $\hat{y}(x, \theta)$ (vgl. [100], S. 59-64).

2.1.3 Typische Aktivierungsfunktionen in neuronalen Netzen

Die Aktivierungsfunktion spielt eine wichtige Rolle in einem neuronalen Netz, denn durch sie wird der Output des jeweiligen Neurons durch ein nichtlineares Element mitbestimmt. Durch eine geeignete Wahl dieser Funktion wird die Approximation unterschiedlichster Funktionen durch Feedforward-Netze möglich. Häufig besitzt jedes Neuron innerhalb eines Netzes dieselbe Aktivierungsfunktion. Es gibt jedoch auch Netze, bei denen die Neuronen von Schicht zu Schicht verschiedene Aktivierungsfunktionen aufweisen. Im Laufe der Jahrzehnte kamen hierfür die unterschiedlichsten Funktionen zum Einsatz. Im Folgenden werden die wichtigsten Aktivierungsfunktionen in neuronalen Netzen sowie deren Eigenschaften vorgestellt (vgl. [18]).

- Heaviside-Funktion

$$H(x) = \begin{cases} 0 & \text{wenn } x < 0 \\ 1 & \text{wenn } x \geq 0 \end{cases}$$

  Die Heaviside-Funktion ist die einfachste unter allen Aktivierungsfunktionen und findet bei binären Klassifikationsproblemen Anwendung. Sie gehört zu den Treppenfunktionen und ihr Wertebereich ist $\{0, 1\}$. Für $x = 0$ ist sie nicht differenzierbar, im restlichen Definitionsbereich verschwindet ihre Ableitung. Daher ist sie für

gradientenbasierte Lernalgorithmen wie das Gradientenabstiegsverfahren ungeeignet und wird heutzutage kaum mehr verwendet (vgl. [101], S. 205).

- Sigmoidfunktion
  Eine sigmoidale Funktion ist eine beschränkte, differenzierbare Funktion, die einen Wendepunkt aufweist und deren erste Ableitung nicht-negativ ist (vgl. [31], S. 14213). Der Graph einer Sigmoidfunktion hat dementsprechend die Form einer S-Kurve und ist in Abbildung 2.2a dargestellt. Der bekannteste Vertreter dieses Funktionstyps ist die logistische sigmoidale Funktion,

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

  die daher oft auch einfach direkt Sigmoidfunktion genannt wird. Ihr Wertebereich ist gleich dem Intervall $(0, 1)$, weshalb sie bei Problemen, bei denen Wahrscheinlichkeiten berechnet werden sollen, zum Einsatz kommt. Sie ist überall differenzierbar und die zugehörige, stetige Ableitungsfunktion lautet

$$\sigma'(x) = \frac{e^x}{(e^x+1)^2} = \sigma(x)\,(1-\sigma(x)).$$

  Für sehr kleine Argumente ist der Wert der Ableitung besonders groß, da die Funktion hier eine große Steigung besitzt. Somit rufen bereits geringe Änderungen des Arguments starke Änderungen der Funktionswerte hervor. Genau das Gegenteil passiert bei betragsmäßig besonders großen Argumenten, bei denen sich die Funktionswerte kaum mehr ändern. Dementsprechend kann das Problem des verschwindenden Gradienten auftreten, wodurch der Lernprozess stark verlangsamt wird und im schlimmsten Fall stecken bleibt. Des Weiteren kann die Tatsache, dass sie ausschließliche positive Werte annimmt, ebenfalls zu Schwierigkeiten im Lernprozess führen.

- Tangens hyperbolicus
  Auch der Graph des Tangens hyperbolicus nimmt die typische S-Form an, was in Abbildung 2.2b zu sehen ist. Die Funktionsgleichung lautet

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{e^{2x}+1} = 2\sigma(2x) - 1,$$

  wodurch sich im Unterschied zur Sigmoidfunktion Funktionswerte im Intervall $(-1, 1)$ ergeben. Daher wird sie unter anderem zur binären Klassifikation verwendet. Die Funktion ist glatt und die Ableitung ist durch

$$\tanh'(x) = 1 - \tanh^2(x)$$

  gegeben. Die tanh-Funktion ist punktsymmetrisch zum Ursprung, das heißt, negative Argumente liefern negative Funktionswerte und analog führen positive Argumente zu positiven Outputs. Auch hier kommt es im Allgemeinen aus demselben

Grund wie auch bei der logistischen Sigmoidfunktion zum Problem des verschwindenden Gradienten (vgl. [101], S. 207-210). Diese Aktivierungsfunktion kommt häufig bei rekurrenten Netzen zum Einsatz, bei denen lineare oder stückweise lineare Aktivierungsfunktionen aufgrund der Rückkopplungen in der Netzwerkstruktur meist ungeeignet sind (vgl. [32], S. 192).

- Rectifier-Funktion („Rectified Linear Unit"-Funktion, kurz ReLU-Funktion)
  Die ReLU-Funktion ist die heutzutage am häufigsten in, insbesondere tiefen, Feedforward-Netzen eingesetzte Aktivierungsfunktion. Einer ihrer Vorteile liegt in der Einfachheit ihrer Berechnung, da sie wie folgt definiert ist:

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

  Sie enthält also im Gegenzug zu den beiden vorherigen Aktivierungsfunktionen keine Exponentialfunktion. Ihr Graph ist in Abbildung 2.2c dargestellt. Der Wertebereich der ReLU-Funktion umfasst alle nicht-negativen reellen Zahlen. Im Kontext von Feedforward-Netzen bedeutet dies, dass Neuronen mit ReLU-Aktivierungsfunktionen nur feuern, wenn der errechnete Wert nicht-negativ ist. Dies führt zu einer gewissen Spärlichkeit des Netzes, die sich als äußerst vorteilhaft für Lernprozesse, die unter anderem der Objekterkennung dienen, erwiesen haben (vgl. [30], S. 317-318). Aufgrund der Eigenschaft, dass $f(x) \geq 0$ gilt, ist sie in vielen Fällen ungünstig für Ausgabeschichten, da auf diese Weise nie negative Werte als Outputs generiert werden können. Die ReLU-Funktion ist stückweise linear, weshalb sie beispielsweise für rekurrente Netze ungeeignet ist. Denn bei diesen werden häufig betragsmäßig große Inputs erwartet, was zu einer enormen Aktivierung des Neurons führen würde. Die Ableitungsfunktion lautet

$$f'(x) = \begin{cases} 0 & \text{wenn } x < 0 \\ 1 & \text{wenn } x > 0. \end{cases} \tag{2.5}$$

  Da die Ableitung stets einen konstanten Wert annimmt, ist das Problem des verschwindenden Gradienten kein Thema. Für $x = 0$ ist die ReLU-Funktion nicht differenzierbar, was jedoch in der Praxis bei der Anwendung des Gradientenabstiegsverfahrens (für Details siehe Kapitel 2.2.1) normalerweise unproblematisch ist, unter anderem deshalb, da man sich bei dieser Methode sowieso stets nur sehr nahe an ein lokales Minimum der betrachteten Funktion begibt und dementsprechend akzeptiert, dass das Beinahe-Minimum an wenigen Stellen undefinierte Gradienteneinträge aufweist. Oft wird in solchen Fällen auch einfach ein Wert festgelegt, den die Funktion bei $x = 0$ annehmen soll. Dabei wird häufig der links- oder rechtsseitige Grenzwert gewählt. Setzt man zum Beispiel $f'(0) = 0$, so ergibt sich die Heaviside-Funktion. Auf diese Weise kann das Problem der nicht vollständigen Differenzierbarkeit für Anwendungen umgangen beziehungsweise auch einfach ignoriert werden (vgl. [32], S. 188-189). Manchmal kann es jedoch auch zum Verhängnis

werden, dass sowohl Funktion als auch Ableitung für negative Argumente den Wert 0 annehmen. Denn somit können bereits geringe Änderungen der Parameter zu einer Inaktivierung eines Neurons führen. Dieses Phänomen wird daher oft als *Dying ReLU* bezeichnet und ist ein Spezialfall des Problems des verschwindenden Gradienten.

- *Parametric ReLU*-Funktion (kurz PReLU-Funktion)
  Die PReLU- Funktion wurde entwickelt, um das soeben erwähnte Problem zu umgehen. Sie ist wie folgt definiert:

$$f(x) = \max(0, x) + \alpha \min(0, x) = \begin{cases} \alpha x & \text{wenn } x < 0 \\ x & \text{wenn } x \geq 0. \end{cases} \quad \alpha \neq 0$$

  Der Wertebereich ist dementsprechend $(-\infty, +\infty)$. Ein Spezialfall der PReLU-Funktion ist die sogenannte *Leaky ReLU*-Funktion, kurz LReLU-Funktion, bei der der Parameter auf $\alpha = 0,01$ fixiert ist. Die Ableitung der PReLU-Funktion ist durch

$$f'(x) = \begin{cases} \alpha & \text{wenn } x < 0 \\ 1 & \text{wenn } x > 0 \end{cases}$$

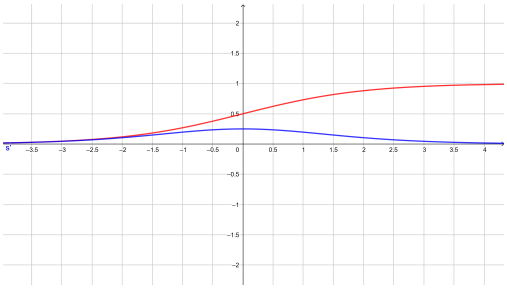
  gegeben, also auch für negative Werte ungleich null. Der Parameter α muss nicht wie bei der LReLU-Funktion fix sein, sondern kann auch innerhalb des Netzes variieren, das heißt, $f_i(x_i) = \max(0, x_i) + \alpha_i \min(0, x_i)$. In diesem Fall kann neben den Netzwerkparametern θ auch der Parameter α_i gelernt werden (vgl. [38], S. 1026-1027).

- Softplus-Funktion
  Die Softplus-Funktion ist eine glatte Version der ReLU-Funktion, und dementsprechend innerhalb des gesamten Definitionsbereichs (unendlich oft) stetig differenzierbar. Ihre Funktionsgleichung lautet

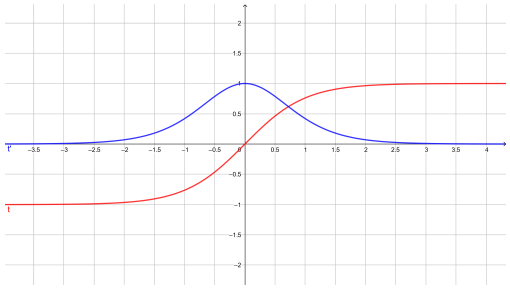
$$f(x) = \ln(1 + e^x)$$

  und ihre Ableitungsfunktion ist die bereits oben erwähnte Sigmoidfunktion (2.3). Sie findet allerdings kaum Anwendung, da mehrere Untersuchungen gezeigt haben, dass die ReLU-Funktion trotz des Dying ReLU-Problems bessere Trainingsresultate liefert als ihre glatte Variante. Dieses überraschende Ergebnis zeigt auch einmal mehr, dass die Zwischenschichten eines neuronalen Netzes zurecht auch verdeckte Schichten genannt werden, denn bis heute ist oftmals nicht ganz klar, welche Ursachen genau hinter dem einen oder anderen Phänomen bei einem Lernprozesses in tiefen Netzen stecken (vgl. [30], S. 318).

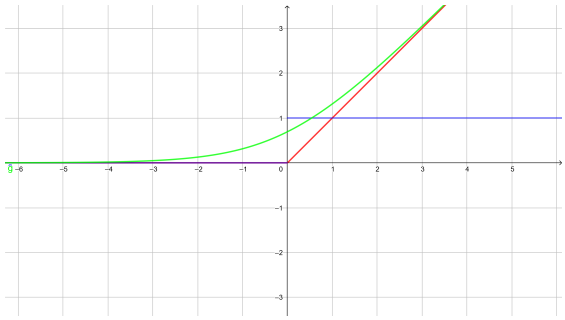
Neben den soeben vorgestellten Aktivierungsfunktionen gibt es noch eine Vielzahl weiterer, die teils sehr spezifisch bei gewissen Anwendungskontexten eingesetzt werden.



(a) Sigmoidfunktion (rot) und ihre Ableitung (blau)



(b) Tanh-Funktion (rot) und ihre Ableitung (blau)



(c) ReLU-Funktion (rot), ihre Ableitung (blau) und die Softplus-Funktion (grün)

Abbildung 2.2: Drei häufig verwendete Aktivierungsfunktionen in FNNs (eigene Darstellung)

2.2 Lernen in Feedforward-Netzen

Das Ziel des Lernprozesses in einem neuronalen Netz ist die besten Parameter zu finden, um schlussendlich zu einem gegebenen Input den gewünschten Output beziehungsweise einen äußerst guten Näherungswert für diesen zu bekommen. Dafür werden im Fall des überwachten Lernens dem Netz Trainingsdaten, die aus lauter Paaren von Vektoren $(x, y)_i$, $i = 1, \ldots, k$, bestehen, vorgesetzt. Innerhalb des Trainingsprozesses erhält das Netz nun den Input x_i und berechnet mit Hilfe seiner momentanen Parameter einen Output $\hat{y}_i$. Dieser wird mit der zur Eingabe gewünschten Ausgabe y_i verglichen und der Fehler evaluiert. Dafür eignet sich beispielsweise die Berechnung der quadratischen Abweichungen. Für ein gegebenes Trainingspaar (x, y) kann diese durch

$$(\hat{y} - y)^T (\hat{y} - y) = \|\hat{y} - y\|^2 \tag{2.6}$$

berechnet werden. Oftmals wird noch der Faktor $\frac{1}{2}$ hinzugefügt, der die Berechnungen während des späteren Lernprozesses, bei dem die Ableitung benötigt wird, ein wenig vereinfacht. Man erhält somit eine Funktion K, die durch

$$K(\hat{y}, y) = \frac{1}{2}\|\hat{y} - y\|^2 \tag{2.7}$$

gegeben ist. Häufig, insbesondere im Bereich der Klassifikation, wird auch die mittlere quadratische Abweichung (*Mean Square Error*, kurz MSE) über eine Menge von Trainingspaaren verwendet. Dabei werden die quadratischen Abweichungen aufsummiert und im Anschluss noch der Faktor $\frac{1}{k}$, wobei k für die Anzahl an Paaren steht, hinzugefügt. Solche Funktionen werden *Kostenfunktion*, oft auch Verlustfunktion (*Loss Function*), genannt (vgl. [32], S. 106-107). Da die Trainingsdaten (x, y) vorgegeben sind, kann man die Kostenfunktion auch als Funktion der Parameter auffassen:

$$L : \theta \mapsto K\left(\hat{y}\left(\theta, x\right), y\right). \tag{2.8}$$

Das Ziel ist nun, die Parameter θ des Netzes so zu verändern, dass die Kostenfunktion L ihren minimalen Wert annimmt. Die zugehörigen Parameterwerte sind dann genau jene, die den minimalen Fehler für die gesamte Trainingsdatenmenge liefern. Ein Minimum $\theta_{min} = \arg\min L\left(\theta\right)$ kann mit Hilfe des Gradienten der Funktion ∇L gesucht werden. Das Verfahren zum Auffinden des Minimums wird *Gradientenabstiegsverfahren* genannt, wobei es hierbei neben der klassischen Methode noch einige weitere Varianten gibt. Um den Gradienten zu berechnen, wird in den meisten Fällen die Methode der *Backpropagation* angewandt. Hierbei wird der Gradient, basierend auf der Kettenregel für Funktionen in mehreren Variablen, Schicht für Schicht, also schrittweise, ermittelt und zwar genau entgegen der Richtung des Informationsflusses. Im Anschluss werden die momentanen Parameterwerte mit Hilfe des soeben berechneten Gradienten angepasst und das Prozedere beginnt erneut. Nach einigen Durchläufen landet man schlussendlich im Idealfall bei einem lokalen Minimum mit einem geringen zugehörigen Kostenfunktionswert, oder sehr nahe bei einem solchen. Die entsprechenden Parameterwerte θ_{min} sind dann genau jene, die die gewünschten Funktionswerte $y = f(x)$ für die Inputs x bestmöglich durch $\hat{y} = \hat{f}(x, \theta_{min})$ approximieren (vgl. [78], Kap. 1).

2.2.1 Das Gradientenabstiegsverfahren

Das Gradientenverfahren ist ein weitverbreitetes, iteratives Verfahren zur Optimierung, bei der in der Regel ein lokales Minimum einer differenzierbaren Funktion gesucht wird. Angewandt auf ein Problem im Deep Learning ist das Prozedere das folgende: Man möchte die idealen Parameter des Netzes finden, bei denen die Kostenfunktion, also der bei den Trainingsdaten gemachte Fehler, einen minimalen Wert annimmt. Dazu werden die Parameterwerte schrittweise verändert, sodass man sich langsam an die bestmöglichen Werte annähert. Die Änderung geschieht mit Hilfe des Gradienten der Funktion, der die Information darüber liefert, wie genau die Parameterwerte verändert werden sollen. Da die meisten im Deep Learning auftretenden Funktionen viele lokale Minima besitzen, wovon nur die wenigsten auch globale sind, und weiters einige kritische Regionen, wie Sattelpunkte und flache Bereiche um diese herum, aufweisen, reicht es in der Regel aus, sehr nahe bei einem lokalen Minimum zu landen. Die dadurch erhaltene Funktion $\hat{f}$ ist dann normalerweise bereits eine äußerst gute Approximation der (unbekannten) Funktion f.

Mathematisch betrachtet läuft das Gradientenverfahren über mehrere Iterationen ab. Nehmen wir an, wir befinden uns an der Stelle θ_0. Wir wollen nun unsere Parameter

um $\Delta\theta_1$ verändern, sodass wir einen neuen Wert θ_1 erhalten, für den die Kostenfunktion einen kleineren Wert als zuvor annimmt. Es soll also $L(\theta_1) < L(\theta_0)$ gelten. Klarerweise ist es wünschenswert, genau das θ zu finden, das den kleinstmöglichen Wert für $L(\theta)$ in einem gewissen Bereich liefert. Dieses kann mit Hilfe des Gradienten der Kostenfunktion $\nabla_\theta L$ gefunden werden. Denn dieser gibt für ein gegebenes Argument genau die Richtung des steilsten Anstiegs an. Dementsprechend erhält man durch $-\nabla_\theta L$ genau die Gegenrichtung, sprich die Richtung des steilsten Abstiegs, also jene, durch die die Kostenfunktionswerte am meisten verkleinert werden. Wir erhalten daher unsere neuen Parameter θ_1 durch $\theta_1 = \theta_0 - \eta\nabla_\theta L(\theta_0)$. Wir verändern also unsere Parameterwerte im ersten Schritt um $\Delta\theta_1 = -\eta\nabla_\theta L(\theta_0)$. Die Konstante η wird in diesem Kontext als *Lernrate* bezeichnet und fixiert die Schrittgröße. Im einfachsten Fall wird diese einfach auf einen relativ kleinen Wert gesetzt. Es gibt jedoch auch Methoden, wie zum Beispiel das Liniensuchverfahren, bei dem zunächst unterschiedliche Werte getestet werden und schlussendlich der im Sinne der Minimierung beste gewählt wird. Die Parameteranpassung wird jetzt immer weiter fortgeführt, das heißt, im n-ten Schritt werden die neuen Werte gemäß

$$\theta_n = \theta_{n-1} - \eta\nabla_\theta L(\theta_{n-1}) \tag{2.9}$$

ermittelt. Dieses Verfahren konvergiert, wenn es konvergiert, schlussendlich gegen eine Stelle $\lim_{n\to\infty}\theta_n = \theta_\infty$, für die $\nabla_\theta L(\theta_\infty) = 0$ gilt, da dies eine notwendige Bedingung für ein lokales Extremum ist. Da man in der Praxis jedoch nicht unendlich viele Schritte durchführen kann, reicht es meist aus, bereits dann aufzuhören, wenn der Gradient der Kostenfunktion fast null ist. In seltenen Fällen kann die Gleichung $\nabla_\theta L(\nabla) = 0$ auch direkt gelöst werden, um einen kritischen Punkt zu finden. Hier erspart man sich die Iterationen (vgl. [32], S. 80-84).

Nun ist jedoch noch die Frage offen, wie der Gradient der Kostenfunktion berechnet werden kann. Dieser ist von den Parameterwerten θ abhängig, die aus den Gewichten $W_{kl}^{(s)}$ und den Verzerrungen $b_k^{(s)}$ bestehen. Außerdem beruhen die Parameterwerte hinterer Schichten auf jenen der vorderen, denn diese hängen über teils mehrere Verkettungen von Funktionen zusammen. Da ein tiefes Feedforward-Netz meist tausende und oft auch mehr Parameter aufweist, ergibt sich hierbei ein immenser Rechenaufwand. Zur Lösung dieses Problems wurde das Verfahren der *Backpropagation*, siehe Kapitel 2.2.2, entwickelt. Bevor wir uns der Methode widmen, folgen jedoch noch ein paar Informationen zu einer heute sehr häufig eingesetzten Form des Gradientenabstiegsverfahrens, dem stochastischen Gradientenabstiegsverfahren.

Das stochastische Gradientenabstiegsverfahren

Eine Grundvoraussetzung für einen erfolgreichen Trainingsprozess ist das Vorhandensein von Trainingsdaten. Im Falle eines Klassifikationsproblems wären diese beispielsweise Bilder mit ihrer zugehörigen Klasse. Als Inputs erhält das Netz dementsprechend die Pixelwerte, wodurch ein Eingabevektor schnell über 1000 Einträge enthalten kann. Hat man nun viele Trainingsdaten, so bringt eine Berechnung der zugehörigen Outputs, sowie ein darauffolgender Optimierungsschritt, einen hohen Rechen- und in Folge auch zeit-

lichen Aufwand mit sich. Techniken, die den gesamten Trainingsdatensatz verwenden, werden als *Batch-*, manchmal auch *deterministische*, *Gradientenverfahren* bezeichnet. In der Praxis ist es jedoch oft einfacher nicht alle zur Verfügung stehenden Daten in den Trainingsprozess einzubeziehen. Beim *stochastischen Gradientenverfahren (SDG)* wird ein einziger zufällig ausgewählter Trainingsdatenpunkt verwendet. Das heißt, zunächst wird dessen Output berechnet, dann die Kostenfunktion ausgewertet und anhand des Ergebnisses erfolgt die Parameteranpassung mittels Gradientenabstiegsverfahren. Danach werden dieselben Schritte mit einem nächsten, ebenfalls per Zufall gewählten Beispiel durchgeführt und immer so weiter. Da stets nur ein Trainingspaar verwendet wird, kommt es klarerweise zu einer Fluktuation der Kostenfunktionswerte und der Fehler nimmt dementsprechend auch manchmal zu. Nach ausreichend vielen Schritten erreicht man bei dieser Methode jedoch ebenfalls ein lokales Minimum der Kostenfunktion, beziehungsweise eine diesem sehr nahe Stelle (vgl. [12], S. 422). Häufig wird auch ein Mittelweg gewählt, bei dem gleich ein paar zufällige Trainingsbeispiele, das heißt mehr als ein Datenpunkt aber wiederum nicht alle Trainingsdaten, pro Optimierungsschritt verwendet werden. Diese Methode wird *Mini-Batch-Gradientenverfahren* genannt und ist heutzutage in den meisten Fällen insbesondere bei besonders großen Trainingsdatensätzen das bevorzugte Verfahren für den Lernprozess in tiefen Feedforward-Netzen (vgl. [9], S. 442).

2.2.2 Das Verfahren der Backpropagation

Der Informationsfluss eines Feedforward-Netzes geht stets nur in eine Richtung vor sich. Das Netz bekommt Inputs x und berechnet über die verdeckten Schichten die zugehörigen Ausgaben $\hat{y} = \hat{f}(x, \theta)$. Da die gewünschten Outputs y während des Trainingsprozesses bekannt sind, soll nun der gemachte Fehler über die Kostenfunktion $L(\theta)$ eruiert werden. Das Verfahren der Backpropagation hilft nun, die erhaltenen Information über die Kostenfunktion *rückwärts*, also entgegen der Arbeitsrichtung, durch das gesamte Netz zu transportieren, um dabei den Gradienten der Funktion zu berechnen. Er wird also rekursiv ermittelt und bildet, wie wir bereits wissen, dann die Grundlage des Gradientenabstiegsverfahrens zur Anpassung der Gewichte und in Folge der Fehlerminimierung. Der Vorteil des Backpropagationsverfahrens liegt in dessen Einfachheit sowie im geringen Rechenaufwand, den es mit sich bringt. Dies ist insbesondere bei Netzen mit sehr vielen Neuronen von großem Nutzen. Oftmals wird mit Backpropagation fälschlicherweise der gesamte Lernalgorithmus, also inklusive des Gradientenabstiegsverfahrens, bezeichnet. In Wahrheit ist die Backpropagation aber ausschließlich eine Methode zur Berechnung des Gradienten einer gegebenen differenzierbaren Funktion, die auch nicht zwingend etwas mit künstlichen Netzen zu tun haben muss.

Mathematisch betrachtet ist die Grundlage des Backpropagation-Algorithmus die Kettenregel der Differentialrechnung für Funktionen in mehreren Variablen. Diese kommt zum Einsatz, wenn die Ableitung einer Verkettung von Funktionen gefunden werden soll. Wie wir bereits gesehen haben, kommen Berechnungen in tiefen neuronalen Netzen ebenfalls durch eine Verkettung mehrerer Funktionen zustande, wobei auch die beim Gradientenverfahren anzupassenden Parameter θ involviert sind. Dementsprechend muss auch

der Gradient der Kostenfunktion $\nabla_\theta L(\theta)$ mit Hilfe der Kettenregel ermittelt werden. Da θ alle Parameter des Feedforward-Netzes enthält, kann man den Gradienten demnach über alle partiellen Ableitungen nach den Gewichten und Verzerrungen berechnen. Das bedeutet, wir müssen

$$\frac{\partial L}{\partial W_{pq}^{(1)}}, \frac{\partial L}{\partial b_p^{(1)}}, \frac{\partial L}{\partial W_{pq}^{(2)}}, \frac{\partial L}{\partial b_p^{(2)}}, \ldots, \frac{\partial L}{\partial W_{pq}^{(S)}}, \frac{\partial L}{\partial b_p^{(S)}} \tag{2.10}$$

für eine konkrete Stelle $\theta = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \ldots, W^{(S)}, b^{(S)}\right)$ finden. Hier wird nun deutlich klarer, was für ein Aufwand nötig wäre, den Gradienten entsprechend obiger Reihenfolge mittels Kettenregel zu berechnen. Denn einerseits gibt es unzählige Parameter und dementsprechend auch viele Ableitungen, andererseits werden für die Ableitungen nach den Parametern der früheren Schichten Rechnungen benötigt, die wiederum in späteren auch nochmals vorkommen. Mit Hilfe der Backpropagation werden genau diese Mehrfachberechnungen vermieden.

Der Gradient unserer Kostenfunktion lässt sich bei vorgegeben Inputs x über die Kettenregel

$$\nabla_\theta L(\theta) = \sum_k \frac{\partial}{\partial \hat{y}_k} L\left(\hat{y}\left(\theta, x\right), y\right) \nabla_\theta \hat{y}_k\left(\theta, x\right) = \sum_k \frac{\partial L}{\partial \hat{y}_k} \nabla_\theta \hat{y}_k \tag{2.11}$$

berechnen, wobei bei der letzten Formel, aufgrund der besseren Übersichtlichkeit, die Abhängigkeiten in der Notation weggelassen wurden. Es ergibt sich somit für die Gewichte sowie die Verzerrungen für $s = 1, \ldots, S$:

$$\frac{\partial L}{\partial W_{pq}^{(s)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial W_{pq}^{(s)}} \tag{2.12}$$

$$\frac{\partial L}{\partial b_p^{(s)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial b_p^{(s)}}. \tag{2.13}$$

Der erste Faktor lässt sich aufgrund der relativ einfachen Gestalt der Kostenfunktion $K\left(\hat{y}, y\right) = \frac{1}{2}\|\hat{y} - y\|^2$ leicht unter Ausnützung der Linearität und Verwendung der Potenzregel berechnen. Man erhält somit:

$$\frac{\partial L}{\partial \hat{y}_k} = \hat{y}_k - y_k. \tag{2.14}$$

Schwieriger wird es nun bei den partiellen Ableitungen nach den einzelnen Parametern. In Gleichung (2.2) konnte man sehen, dass die Parameter teils äußerst verschachtelt sind. Somit beeinflusst das Gewicht eines frühen Neurons auch die Berechnungen aller nachfolgenden Neuronen und infolge dessen auch den Fehler, den diese machen.

Gehen wir dazu einmal von der letzten Schicht S aus. Hier ist, aufgrund der Ketten-

regel,

$$\begin{aligned}\frac{\partial \hat{y}_k}{\partial b_p^{(S)}} &= \frac{\partial}{\partial b_p^{(S)}} f\left(v_k^{(S)}\right) = \frac{\partial}{\partial b_p^{(S)}} f\left(\left(W^{(S)}u^{(S-1)}\right)_k + b_k^{(S)}\right) \\ &= f'\left(\left(W^{(S)}u^{(S-1)}\right)_k + b_k^{(S)}\right) \frac{\partial}{\partial b_p^{(S)}} \left(\left(W^{(S)}u^{(S-1)}\right)_k + b_k^{(S)}\right) \\ &= f'\left(v_k^{(S)}\right) \frac{\partial}{\partial b_p^{(S)}} b_k^{(S)} = \begin{cases} 0 & \text{wenn } p \neq k \\ f'\left(v_k^{(S)}\right) & \text{wenn } p = k. \end{cases}\end{aligned} \tag{2.15}$$

Mit Hilfe des Kronecker-Deltas lässt sich obiges Ergebnis auch kürzer schreiben:

$$\frac{\partial \hat{y}_k}{\partial b_p^{(S)}} = \delta_{kp} f'\left(v_k^{(S)}\right). \tag{2.16}$$

Somit können wir nun in Gleichung (2.13) einsetzen und erhalten für die partielle Ableitung der Kostenfunktion nach der Verzerrung

$$\frac{\partial L}{\partial b_p^{(S)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial b_p^{(S)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \delta_{kp} f'\left(v_k^{(S)}\right) = \frac{\partial L}{\partial \hat{y}_p} f'\left(v_p^{(S)}\right). \tag{2.17}$$

Nun fehlt uns noch die Ableitung der Aktivierungsfunktion. Diese ist in der Regel leicht berechnet, für Details siehe Kapitel 2.1.3, und muss nun also an der Stelle $v_p^{(S)}$ ausgewertet werden. Der Wert der Stelle ist bereits bekannt, da dieser im Zuge der sogenannten Forward-Propagation zur Berechnung des Outputs $\hat{y}$ zu einem gegebenen Input x bereits ermittelt wurde. Somit steht $v_p^{(S)}$ zur Verfügung und die Auswertung $f'\left(v_p^{(S)}\right)$ ist leicht durchführbar.

Betrachten wir nun die partiellen Ableitungen nach den Gewichten der letzten Schicht. Aufgrund der Kettenregel gilt

$$\begin{aligned}\frac{\partial \hat{y}_k}{\partial W_{pq}^{(S)}} &= \frac{\partial}{\partial W_{pq}^{(S)}} f\left(v_k^{(S)}\right) = \frac{\partial}{\partial W_{pq}^{(S)}} f\left(\sum_l W_{kl}^{(S)} u_l^{(S-1)} + b_k^{(S)}\right) \\ &= f'\left(v_k^{(S)}\right) \frac{\partial}{\partial W_{pq}^{(S)}} \sum_l W_{kl}^{(S)} u_l^{(S-1)} + 0 \\ &= \delta_{kp} f'\left(v_k^{(S)}\right) u_q^{(S-1)}.\end{aligned} \tag{2.18}$$

Setzt man dies in Gleichung (2.12) ein, so folgt

$$\frac{\partial L}{\partial W_{pq}^{(S)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial W_{pq}^{(S)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} \delta_{kp} f'\left(v_k^{(S)}\right) u_q^{(S-1)} = \frac{\partial L}{\partial \hat{y}_p} f'\left(v_p^{(S)}\right) u_q^{(S-1)}. \tag{2.19}$$

Und auch hier ist der erste Teil unseres Ergebnisses bereits berechnet, nämlich durch Gleichung (2.17). Der Wert des letzten Faktors ist ebenso schon vorhanden, da dieser

im Zuge der Forward-Propagation zur Ausgabenberechnung ermittelt wurde. Folglich können wir die partielle Ableitung der Kostenfunktion nach den Gewichten der letzten Schicht einfach durch

$$\frac{\partial L}{\partial W_{pq}^{(S)}} = \frac{\partial L}{\partial b_p^{(S)}} u_q^{(S-1)} \tag{2.20}$$

berechnen. Dieselben Überlegungen lassen sich auch auf die frühere Schicht $S-1$ anwenden. Es ist

$$\begin{aligned}
\frac{\partial \hat{y}_k}{\partial b_p^{(S-1)}} &= \frac{\partial}{\partial b_p^{(S-1)}} f\left(v_k^{(S)}\right) \\
&= \frac{\partial}{\partial b_p^{(S-1)}} f\left(\sum_l W_{kl}^{(S)} f\left(\left(W^{(S-1)} u^{(S-2)}\right)_l + b_l^{(S-1)}\right) + b_k^{(S)}\right) \\
&= f'\left(v_k^{(S)}\right) \frac{\partial}{\partial b_p^{(S-1)}} \left(\sum_l W_{kl}^{(S)} f\left(\left(W^{(S-1)} u^{(S-2)}\right)_l + b_l^{(S-1)}\right) + b_k^{(S)}\right) \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} \frac{\partial}{\partial b_p^{(S-1)}} f\left(\left(W^{(S-1)} u^{(S-2)}\right)_l + b_l^{(S-1)}\right) \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} \frac{\partial}{\partial b_p^{(S-1)}} f\left(v_l^{(S-1)}\right) \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) \frac{\partial}{\partial b_p^{(S-1)}} b_l^{(S-1)} \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) \delta_{pl} \\
&= f'\left(v_k^{(S)}\right) W_{kp}^{(S)} f'\left(v_p^{(S-1)}\right).
\end{aligned} \tag{2.21}$$

Damit können wir nun die zugehörige Komponente des Gradienten berechnen:

$$\begin{aligned}
\frac{\partial L}{\partial b_p^{(S-1)}} &\overset{(2.13)}{=} \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial b_p^{(S-1)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} f'\left(v_k^{(S)}\right) W_{kp}^{(S)} f'\left(v_p^{(S-1)}\right) \\
&\overset{(2.17)}{=} \sum_k \frac{\partial L}{\partial b_k^{(S)}} W_{kp}^{(S)} f'\left(v_p^{(S-1)}\right).
\end{aligned} \tag{2.22}$$

Nun gilt es, die partiellen Ableitungen nach den Gewichten der vorletzten Schicht zu

finden. Diese lassen sich wie folgt berechnen:

$$\begin{aligned}
\frac{\partial \hat{y}_k}{\partial W_{pq}^{(S-1)}} &= \frac{\partial}{\partial W_{pq}^{(S-1)}} f\left(v_k^{(S)}\right) \\
&= \frac{\partial}{\partial W_{pq}^{(S-1)}} f\left(\sum_l W_{kl}^{(S)} f\left(\left(W^{(S-1)} u^{(S-2)}\right)_l + b_l^{(S-1)}\right) + b_k^{(S)}\right) \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} \frac{\partial}{\partial W_{pq}^{(S-1)}} f\left(\sum_r W_{lr}^{(S-1)} u_r^{(S-2)} + b_l^{(S-1)}\right) \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) \frac{\partial}{\partial W_{pq}^{(S-1)}} \sum_r W_{lr}^{(S-1)} u_r^{(S-2)} \\
&= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) \delta_{pl} u_q^{(S-2)} \\
&= f'\left(v_k^{(S)}\right) W_{kp}^{(S)} f'\left(v_p^{(S-1)}\right) u_q^{(S-2)}.
\end{aligned} \tag{2.23}$$

Anhand dessen erhalten wir nun für den Eintrag des Gradienten der Kostenfunktion:

$$\begin{aligned}
\frac{\partial L}{\partial W_{pq}^{(S-1)}} &\overset{(2.12)}{=} \sum_k \frac{\partial L}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial W_{pq}^{(S-1)}} = \sum_k \frac{\partial L}{\partial \hat{y}_k} f'\left(v_k^{(S)}\right) W_{kp}^{(S)} f'\left(v_p^{(S-1)}\right) u_q^{(S-2)} \\
&\overset{(2.22)}{=} \frac{\partial L}{\partial b_p^{(S-1)}} u_q^{(S-2)}.
\end{aligned} \tag{2.24}$$

Auch hier kommt es wieder nach ein paar Umformungen zu einem sehr simplen Ausdruck, für den ausschließlich bereits berechnete Werte benötigt werden. Und genau darin steckt die Kraft der Backpropagation: Wenn man die Berechnung des Gradienten in „Rückrichtung“ durch schrittweises Arbeiten von hinten nach vorne angeht, so sind alle benötigten Werte aufgrund vorangegangener Rechenschritte bereits im Netz vorhanden.

Die Formeln zur Berechnung der Gradientenkomponenten nach den Parametern der früheren Schichten ergeben sich in analoger Weise zu den bisherigen. Somit können die partiellen Ableitungen der nächstfolgenden Schicht durch

$$\begin{aligned}
\frac{\partial \hat{y}_k}{\partial b_p^{(S-2)}} &= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) W_{lp}^{(S-1)} f'\left(v_p^{(S-2)}\right) \\
\frac{\partial L}{\partial b_p^{(S-2)}} &= \sum_k \frac{\partial L}{\partial b_k^{(S-1)}} W_{kp}^{(S-1)} f'\left(v_p^{(S-2)}\right) \\
\frac{\partial \hat{y}_k}{\partial W_{pq}^{(S-2)}} &= f'\left(v_k^{(S)}\right) \sum_l W_{kl}^{(S)} f'\left(v_l^{(S-1)}\right) W_{lp}^{(S-1)} f'\left(v_p^{(S-2)}\right) u_q^{(S-3)} \\
\frac{\partial L}{\partial W_{pq}^{(S-2)}} &= \frac{\partial L}{\partial b_p^{(S-2)}} u_q^{(S-3)}
\end{aligned}$$

berechnet werden (vgl. [22], S. 10-16). Bei genauerer Betrachtung lässt sich ein gewisses Muster erkennen, wie die einzelnen Ableitungen aufgebaut sind. Es handelt sich um

lauter Rekursionen, die sich dementsprechend ein wenig kompakter notieren lassen. Wir können also zusammenfassend eine allgemeine Formel für die Gradientenberechnung aufstellen. Dabei lassen sich die Komponenten der letzten Schicht, also der Ausgabeschicht, durch

$$\frac{\partial L}{\partial b_p^{(S)}} = \frac{\partial L}{\partial \hat{y}_p} f'\left(v_p^{(S)}\right) \tag{2.25}$$

$$\frac{\partial L}{\partial W_{pq}^{(S)}} = \frac{\partial L}{\partial b_p^{(S)}} u_q^{(S-1)} \tag{2.26}$$

berechnen. Die partiellen Ableitungen nach den Parametern der darauffolgenden verdeckten Schichten, $s = S-1, \ldots, 1$, erhalten wir durch:

$$\frac{\partial L}{\partial b_p^{(s)}} = \sum_k \frac{\partial L}{\partial b_k^{(s+1)}} W_{kp}^{(s+1)} f'\left(v_p^{(s)}\right) \tag{2.27}$$

$$\frac{\partial L}{\partial W_{pq}^{(s)}} = \frac{\partial L}{\partial b_p^{(s)}} u_q^{(s-1)}. \tag{2.28}$$

Mit Hilfe obiger Formeln lassen sich also alle Komponenten des Gradienten der Kostenfunktion $\nabla_\theta L$ berechnen. Dieser wird nun, wie bereits in Kapitel 2.2.1 beschrieben, in weiterer Folge zur Anpassung der Gewichte mittels (stochastischen) Gradientenabstiegsverfahrens genützt (vgl. [78], Kap. 2). Dieses wird nämlich direkt nach der Berechnung der jeweiligen partiellen Ableitungen innerhalb einer Schicht durchgeführt. Die Anpassung der Gewichte und Verzerrungen einer Schicht s läuft entsprechend der Formel (2.9) ab, das heißt, diese erfolgt für den t-ten Schritt des Gradientenverfahrens gemäß (vgl. [36], S. 142)

$$\begin{aligned} b_p^{(s)}(t) &= b_p^{(s)}(t-1) - \eta \frac{\partial L}{\partial b_p^{(s)}(t-1)} \\ &= b_p^{(s)}(t-1) - \eta \sum_k \frac{\partial L}{\partial b_k^{(s+1)}(t-1)} W_{kp}^{(s+1)}(t-1) f'\left(v_p^{(s)}(t-1)\right) \end{aligned} \tag{2.29}$$

$$\begin{aligned} W_{pq}^{(s)}(t) &= W_{pq}^{(s)}(t-1) - \eta \frac{\partial L}{\partial W_{pq}^{(s)}(t-1)} \\ &= W_{pq}^{(s)}(t-1) - \eta \frac{\partial L}{\partial b_p^{(s)}(t-1)} u_q^{(s-1)}(t-1). \end{aligned} \tag{2.30}$$

Wird das Batch-Gradientenabstiegsverfahren zur Parameteranpassung verwendet, so wird zunächst der vollständige Gradient jedes Trainingsbeispiels mittels Backpropagation berechnet und im Anschluss das arithmetische Mittel gebildet. Erst dann kann die Änderung der Parameterwerte durch das Gradientenverfahren mit Hilfe des erhaltenen arithmetischen Mittels der Gradienten erfolgen (vgl. [32], S.209).

3 Das universelle Approximationstheorem für Feedforward-Netze

Feedforward-Netze werden in den verschiedensten Bereichen, wie beispielsweise der Klassifikation oder auch der Spracherkennung, eingesetzt. Für eine erfolgreiche Anwendung ist stets ein zuvor stattfindendes Training nötig, das dazu verhilft die Parameter des Netzwerks geeignet anzupassen. Nach erfolgreicher Adaption ist das Netz im Stande eine problemspezifische Zielfunktion bestmöglich zu approximieren. Dabei stellt sich allerdings die grundlegende Frage, welche Funktionen durch neuronale Feedforward-Netze in der Theorie überhaupt dargestellt werden können. Das *universelle Approximationstheorem* liefert die Antwort auf genau diese Frage. Es gibt mehrere Versionen dieses Theorems, wobei manche für beliebig weite Netze mit einer fixen Tiefe, also einer bestimmten Anzahl an Schichten, und andere für Netze beliebiger Tiefe, jedoch mit festgelegter Weite, genauer gesagt Neuronenzahl pro Schicht, formuliert wurden. Die berühmteste Version ist jene für Netzwerke mit einer Zwischenschicht, also einer fixen Tiefe, die jedoch keine Neuronenzahl vorgegeben haben, also eine beliebige Weite aufweisen. Diese Version besagt, dass ein solches Feedforward-Netz mit einer nicht-polynomialen Aktivierungsfunktion zumindest jede stetige Funktion beliebig gut approximieren kann.

3.1 Das universelle Approximationstheorem für Feedforward-Netze mit einer Zwischenschicht

Historisch betrachtet geht die Entdeckung der universellen Approximationseigenschaft unter anderem auf Funahashi ([27]), Cybenko ([16]) sowie Hornik, Stinchcomb und White ([48]) zurück, die alle unabhängig voneinander im Jahr 1989 Beweise veröffentlichten, dass Feedforward-Netze mit einer verdeckten Schicht und der sigmoidalen Aktivierungsfunktion jede stetige Funktion annähern können. Dieses Ergebnis wurde in den darauffolgenden Jahren immer wieder erweitert. Hornik zeigte, dass diese Eigenschaft nicht nur auf Sigmoidfunktionen beschränkt ist, sondern auch eine Vielzahl anderer Aktivierungsfunktionen zu denselben Ergebnissen führen. Hierbei sind die Bedingungen an die Funktion lediglich, dass diese stetig und beschränkt sein muss, sowie dass sie nichtkonstant sein muss. Daraus ergibt sich die Schlussfolgerung, dass der entscheidende Punkt für eine erfolgreiche Funktionsapproximation die Struktur der Feedforward-Netze zu sein scheint (vgl. [51], S. 253-254). Leshno et al. konnten eine noch allgemeinere Bedingung für die Aktivierungsfunktion für eine erfolgreiche Approximation finden: Sie muss nichtpolynomial sowie lokal beschränkt sein (vgl. [62], S. 863).

Im Folgenden wird die Version des universellen Approximationstheorems von Allan

Pinkus ([84]), die durch Leshno et al. inspiriert wurde, präsentiert, da diese eine große Klasse an in Feedforward-Netzen eingesetzten Aktivierungsfunktionen einschließt. Dabei gehen wir von einem Netz mit einer Zwischenschicht, bestehend aus k Neuronen, mit der Aktivierungsfunktion σ aus. Zur Vereinfachung besitzt die Outputschicht weder eine Aktivierungsfunktion noch Verzerrungen. Mit Hilfe dieses Netzes soll eine stetige multivariate Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ approximiert werden. Dementsprechend gilt für den Input $x \in \mathbb{R}^n$. Da der Outputvektor komponentenweise berechnet wird, kann man diese Zuordnung auch auf m Funktionen mit eindimensionalen Funktionswerten, entsprechend $f_l : \mathbb{R}^n \to \mathbb{R}, l = 1, \ldots, m$, aufsplitten. Daher wird im kommenden Theorem und dessen Beweis zur Vereinfachung ein Netz mit einem eindimensionalen Outputvektor $y \in \mathbb{R}$ betrachtet. Dieser ergibt sich durch

$$y = y(x) = \sum_{i=1}^{k} c_i \sigma \left(\sum_{j=1}^{n} w_{ij} x_j + b_i \right), \tag{3.1}$$

wobei $w_{ij} \in \mathbb{R}$ das Gewicht zwischen des j-ten Eingabeneurons und des i-ten Neurons der Zwischenschicht ist, $b_i \in \mathbb{R}$ die Verzerrung dieses Neurons darstellt und c_i für das Gewicht zwischen dem Neuron i und der Ausgabeeinheit steht. In Vektorschreibweise lässt sich obige Berechnung kompakter, mit dem Eingabevektor $x \in \mathbb{R}^n$ und dem Gewichtsvektor $w_i \in \mathbb{R}^n$ des i-ten Zwischenschichtneurons, notieren:

$$y = \sum_{i=1}^{k} c_i \sigma \left(w_i \cdot x + b_i \right), \tag{3.2}$$

wobei $w_i \cdot x$ das Standardskalarprodukt bezeichnet und σ komponentenweise auf Vektoren wirkt. Sei F die Menge aller durch obiges Feedforward-Netz darstellbaren Funktionen, also $\mathcal{F} = \{f_\theta \mid \theta \in \Theta\}$. Dabei steht Θ für den Raum aller möglichen Netzwerkparameter, sprich in diesem Fall die Anzahl k an Zwischenschichtneuronen, die Gewichte w, die Verzerrungen b und alle erlaubten Aktivierungsfunktionen σ, und $\theta = (k, w, b, \sigma)$ repräsentiert ein fixes Quadrupel aus diesem Raum. Die Idee ist nun zu zeigen, dass die Menge $\mathcal{F}$ der durch das Netz darstellbaren Funktionen jede stetige reelle Funktion $f \in C(\mathbb{R}^n)$ approximieren kann. Dafür muss für den betrachteten Raum eine Topologie oder Metrik vorgegeben sein, durch die definiert werden kann, wann eine Funktion „nahe“ bei einer anderen liegt. Ausgehend davon kann dann festgelegt werden, wann eine Menge von Funktionen dicht in einer anderen Funktionenmenge liegt. Die detaillierten Definitionen sind im Anhang 5.1 nachlesbar. Für unseren Fall wollen wir nun zeigen, dass die Menge $\mathcal{F}$ bezüglich der Topologie der gleichmäßigen Konvergenz dicht in der Menge aller stetigen reellen Funktionen $C(\mathbb{R}^n)$ liegt. Denn dies bedeutet, dass es zu jeder stetigen Funktion $f \in C(\mathbb{R}^n)$ und jeder kompakten, also beschränkten und abgeschlossenen, Teilmenge $K \subset \mathbb{R}^n$ eine Funktion $g \in \mathcal{F}$ gibt, die f beliebig genau approximiert (vgl. [62], S. 861-863). Es gilt dann also für jede kompakte Teilmenge $K \subset \mathbb{R}^n$, dass

$$\forall f \in C(\mathbb{R}^n) \ \ \forall \varepsilon > 0 \ \exists g \in \mathcal{F} : \max_{x \in K} |f(x) - g(x)| < \varepsilon. \tag{3.3}$$

Da die Gewichte, Verzerrungen und Neuronenzahl in der Zwischenschicht passend aus den reellen beziehungsweise letztere aus den natürlichen Zahlen gewählt werden können, bleibt die Frage, welche Bedingungen für die Aktivierungsfunktion nötig sind, sodass die Dichteeigenschaft in $C(\mathbb{R}^n)$ erfüllt ist (vgl. [84], S. 151). Diese wird nun durch das folgende universelle Approximationstheorem nach Allan Pinkus beantwortet. Darin steht die Menge $\mathcal{M}(\sigma, n)$ für die Menge aller Funktionen, die durch ein Feedforward-Netz mit der Aktivierungsfunktion σ, einer n-dimensionalen Eingabeschicht sowie einer Zwischenschicht dargestellt werden können. Sie enthält also alle endlichen Linearkombinationen $\sum_{i=1}^{k} c_i \sigma(w_i \cdot x + b_i)$, $k \in \mathbb{N}$. Die Menge aller endlichen Linearkombinationen einer Teilmenge $A = \{a_1, \dots a_n\}$ eines $\mathbb{K}$-Vektorraums V wird in der Mathematik als *lineare Hülle*, manchmal auch *Span* oder *Erzeugnis*, bezeichnet und durch $\operatorname{span}(A) = \langle A \rangle = \left\{ \sum_{i=1}^{k} c_i a_i \mid c_i \in \mathbb{K}, a_i \in A, k \in \mathbb{N} \right\}$ notiert. Die im Satz betrachtete Menge $\mathcal{M}(\sigma, n)$ der durch das Netz darstellbaren Funktionen ist also nichts anderes als

$$\begin{aligned}\mathcal{M}(\sigma, n) &= \left\{ \sum_{i=1}^{k} c_i \sigma(w_i \cdot x + b_i) \mid c_i \in \mathbb{R}, w \in \mathbb{R}^n, b \in \mathbb{R}, k \in \mathbb{N} \right\} \\ &= \operatorname{span}\{\sigma(w \cdot x + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R}\}.\end{aligned}$$

Sehen wir uns nun an, welche Bedingungen die Aktivierungsfunktion σ erfüllen muss, sodass eine beliebige stetige reelle Funktion $f : \mathbb{R}^n \to \mathbb{R}$ mit Hilfe einer Funktion aus $\mathcal{M}(\sigma, n)$ approximiert werden kann.

Satz 3.1 (**Das universelle Approximationstheorem nach Pinkus**). *Sei $n \in \mathbb{N}$ und $\sigma \in C(\mathbb{R})$. Betrachte nun die Menge*

$$\mathcal{M}(\sigma, n) := \operatorname{span}\{\sigma(w \cdot x + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R}\}. \tag{3.4}$$

Dann liegt $\mathcal{M}(\sigma, n)$ genau dann dicht in $C(\mathbb{R}^n)$, in der Topologie der gleichmäßigen Konvergenz auf kompakten Teilmengen, wenn σ keine Polynomfunktion ist.

Der folgende Beweis des universellen Approximationssatzes orientiert sich an jenem von Pinkus ([84], S. 153-162). In diesem werden sogenannte Ridge-Funktionen verwendet, die wie folgt definiert sind:

Definition 3.2 (Ridge-Funktion). Eine *Ridge-Funktion* ist eine Funktion der Form $G : \mathbb{R}^n \to \mathbb{R}$,

$$G(x) = g(a \cdot x) = g(a_1 x_1 + a_2 x_2 + \cdots + a_n x_n), \tag{3.5}$$

wobei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion und $a \in \mathbb{R}^n$ ist (vgl. [84], S. 154).

Ridge-Funktionen spielen beispielsweise bei hyperbolischen partiellen Differentialgleichungen sowie in der Statistik eine wichtige Rolle, ebenso bei neuronalen Netzen. Denn in jedem Feedforward-Netzwerk mit der Aktivierungsfunktion σ treten Funktionen entsprechend der Gestalt $\sigma(w \cdot x + b)$ auf. Diese sind bei genauerer Betrachtung für beliebige σ, w und b Ridge-Funktionen. Ridge-Funktionen haben einige interessante Eigenschaften, unter anderem, dass sie dicht in $C(\mathbb{R})$ liegen (für Details siehe Satz 5.7). Diese Tatsache wird auch in dem nun folgenden Beweis des Theorems verwendet.

Beweis *von Satz 3.1.* Der Beweis des universellen Approximationstheorems erfolgt in mehreren Schritten, die jeweils aus einem Lemma und dessen Beweis bestehen.

1. Schritt: Ist σ eine Polynomfunktion, dann liegt $\mathcal{M}(\sigma, n)$ nicht dicht in $C(\mathbb{R}^n)$.

   Beweis. Nehmen wir an, dass die Funktion σ ein Polynom r-ten Grades ist. Dann muss auch $\sigma(w \cdot x + b)$, mit $w \in \mathbb{R}^n, b \in \mathbb{R}$, ein Polynom mit einem Grad von maximal r sein. Daraus folgt wiederum, dass auch jede Linearkombination dieser Funktionen erneut eine Polynomfunktion maximal r-ten Grades ist. Das heißt, $\mathcal{M}(\sigma, n)$ enthält ausschließlich Funktionen diesen Typs. Dementsprechend liegt die Menge sicher nicht dicht in $C(\mathbb{R}^n)$ und die Möglichkeit zur Approximation aller stetiger Funktionen ist folglich nicht gegeben. □

   Somit ist bewiesen, dass die Bedingung, dass σ keine Polynomfunktion ist, notwendig dafür ist, dass $\mathcal{M}(\sigma, n)$ dicht in $C(K)$ liegt. Es bleibt zu zeigen, dass sie auch hinreichend ist. Dieser Beweis wird in mehreren Schritten geführt und dabei ein wenig „von vorne nach hinten“ aufgerollt.

2. Schritt: Wenn $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$ liegt, dann folgt daraus, dass $\mathcal{M}(\sigma, n)$ dicht in $C(\mathbb{R}^n)$ ist.

   Beweis. Sei $K \subset \mathbb{R}^n$ eine kompakte Teilmenge und $f \in C(K)$. Weiters sei $\mathcal{R} = \operatorname{span}\{g(a \cdot x) \mid g \in C(\mathbb{R}), a \in \mathbb{R}^n\}$. Nach Satz 5.7 ist diese Menge dicht in $C(\mathbb{R}^n)$ und dementsprechend auch in $C(K)$. Somit gibt es für jedes $\varepsilon > 0$ Funktionen $g_i : \mathbb{R} \to \mathbb{R}$ und Vektoren $a_i \in \mathbb{R}^n$, $i = 1, \dots k$, sodass

   $$\left| f(x) - \sum_{i=1}^{k} g_i(a_i \cdot x) \right| < \frac{\varepsilon}{2} \qquad \forall x \in K \tag{3.6}$$

   gilt. Betrachten wir nun für jedes i die Menge $\{a_i \cdot x \mid x \in K\} \subset \mathbb{R}$. Da die Menge K kompakt ist, muss es ein abgeschlossenes Intervall $[\alpha_i, \beta_i]$ geben, sodass $\left\{a^i \cdot x \mid x \in K\right\} \subseteq [\alpha_i, \beta_i]$, für $i = 1, \dots, k$, ist. Nun liegt aufgrund unserer Annahme $\mathcal{M}(\sigma, 1)$ für alle $i = 1, \dots, k$ dicht in $C([\alpha_i, \beta_i])$. Es muss also Konstanten $c_{ij}, w_{ij}, b_{ij} \in \mathbb{R}$, $j = 1, \dots, m_i$, geben, sodass für alle $i = 1, \dots, k$

   $$\left| g_i(x) - \sum_{j=1}^{m_i} c_{ij} \sigma(w_{ij} x + b_{ij}) \right| < \frac{\varepsilon}{2k} \qquad \forall x \in [\alpha_i, \beta_i] \tag{3.7}$$

   ist. Und daher gilt nun aufgrund der beiden Ungleichungen sowie der Dreiecksun-

gleichung, dass

$$\begin{aligned}
&\left| f(x) - \sum_{i=1}^{k} g_i(a_i \cdot x) + \sum_{i=1}^{k} \left[g_i(a_i \cdot x) - \sum_{j=1}^{m_i} c_{ij} \sigma(w_{ij} a_i \cdot x + b_{ij}) \right] \right| \\
&= \left| f(x) - \sum_{i=1}^{k} g_i(a_i \cdot x) + \sum_{i=1}^{k} g_i(a_i \cdot x) - \sum_{i=1}^{k} \sum_{j=1}^{m_i} c_{ij} \sigma(w_{ij} a_i \cdot x + b_{ij}) \right| \\
&= \left| f(x) - \sum_{i=1}^{k} \sum_{j=1}^{m_i} c_{ij} \sigma(w_{ij} a_i \cdot x + b_{ij}) \right| < \frac{\varepsilon}{2} + k \frac{\varepsilon}{2k} = \varepsilon \qquad \forall x \in K.
\end{aligned}$$

Also liegt $\mathcal{M}(\sigma, n)$ dicht in $C(\mathbb{R}^n)$. □

Da wir soeben gesehen haben, dass aus der Dichtheit für $n = 1$ die Dichtheit für ein beliebiges $n \in \mathbb{N}$ folgt, können wir uns in den kommenden Schritten von nun an auf den eindimensionalen Fall beschränken. Es bleibt also zu zeigen, dass $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$ liegt. Dafür betrachten wir zunächst den Fall, dass die Aktivierungsfunktion σ glatt ist.

3. Schritt: Sei $\sigma \in C^\infty(\mathbb{R})$ mit der zusätzlichen Eigenschaft, dass σ keine Polynomfunktion ist. Dann ist $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$.

Beweis. Aufgrund der Tatsache, dass σ unendlich oft stetig differenzierbar und keine Polynomfunktion ist, muss es eine Stelle $b_0 \in \mathbb{R}$ geben (siehe Satz 5.8) an der weder die Funktion selbst noch eine ihrer Ableitungen verschwindet. Für diese Stelle gilt also $\sigma^{(k)}(b_0) \neq 0$, $\forall k \in \mathbb{N}_0$. Nun ist $\sigma \in C^\infty(\mathbb{R})$ und die Linearkombination

$$\frac{\sigma((w+h)x + b_0) - \sigma(wx + b_0)}{h} \in \mathcal{M}(\sigma, 1), \ \forall w \in \mathbb{R} \text{ und } h \neq 0.$$

Aufgrund dessen muss

$$\begin{aligned}
\frac{d}{dw} \sigma(wx + b_0) \Big|_{w=0} &= \lim_{h \to 0} \frac{\sigma((w+h)x + b_0) - \sigma(wx + b_0)}{h} \Big|_{w=0} \\
&= x \cdot \sigma'(b_0)
\end{aligned}$$

ein Element der abgeschlossenen Hülle von $\mathcal{M}(\sigma, 1)$, also von $\overline{\mathcal{M}(\sigma, 1)}$, sein. Ebenso muss wegen derselben Folgerungen dementsprechend für jedes $k \in \mathbb{N}$

$$\frac{d^k}{dw^k} \sigma(wx + b_0) \Big|_{w=0} = x^k \cdot \sigma^{(k)}(b_0) \in \overline{\mathcal{M}(\sigma, 1)}$$

gelten. Nun ist laut Annahme $\sigma^{(k)}(b_0) \neq 0$ für alle $k \in \mathbb{N}_0$ und so enthält die Menge $\overline{\mathcal{M}(\sigma, 1)}$ alle Monome und auch alle Linearkombinationen dieser. Dies bedeutet,

dass auch alle Polynome Elemente der Menge $\overline{\mathcal{M}(\sigma, 1)}$ sein müssen. Nach dem Approximationssatz von Weierstraß (siehe Satz 5.9) folgt, dass für jede kompakte Teilmenge $K \subset \mathbb{R}$ gilt:

$$C(K) \subseteq \overline{\mathcal{M}(\sigma, 1)}.$$

Und somit ist $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$. □

Für glatte Aktivierungsfunktionen gilt also die Dichteeigenschaft. Im Folgenden lockern wir unsere Bedingungen für σ und verlangen nurmehr, dass die Funktion stetig sein muss.

4. Schritt: Sei $\sigma \in C(\mathbb{R})$, unter der Voraussetzung, dass σ keine Polynomfunktion ist. Dann liegt $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$.

Beweis. Es sei ϕ eine auf ganz $\mathbb{R}$ glatte Funktion mit kompaktem Träger, also $\phi \in C_c^\infty(\mathbb{R})$. (Für Details siehe Defintion 5.10.) Wir betrachten nun für jedes $\phi \in C_c^\infty(\mathbb{R})$ dessen Faltung mit der Aktivierungsfunktion σ, also

$$\sigma_\phi(x) = (\sigma * \phi)(x) = \int_{-\infty}^{\infty} \sigma(x - t)\, \phi(t)\, dt.$$

Diese konvergiert für alle $t \in \mathbb{R}$, aufgrund der Tatsache, dass σ und ϕ stetig sind und ϕ einen kompakten Träger hat, weshalb das Integral wohldefiniert ist. Des Weiteren ist $\sigma_\phi \in \overline{\mathcal{M}(\sigma, 1)}$, was sich leicht mit Hilfe einer Grenzwertbildung der Riemann-Summen herleiten lässt. Außerdem gilt aufgrund obiger Voraussetzungen, dass $\sigma_\phi \in C^\infty(\mathbb{R})$ ist. Da für alle $w, b \in \mathbb{R}$

$$\sigma_\phi(wx + b) = \int_{-\infty}^{\infty} \sigma(wx + b - t)\, \phi(t)\, dt \tag{3.8}$$

gilt, ist also $\overline{\mathcal{M}(\sigma_\phi, 1)} \subseteq \overline{\mathcal{M}(\sigma, 1)}$. Aufgrund der Tatsache, dass $\sigma_\phi \in C^\infty(\mathbb{R})$ gilt, können wir hier ebenfalls die Idee von Schritt 3 anwenden. Es ist also

$$\frac{d^k}{dw^k} \sigma_\phi(wx + b) \bigg|_{w=0} = x^k \sigma_\phi^{(k)}(b) \ \in \overline{\mathcal{M}(\sigma_\phi, 1)}$$

für alle $b \in \mathbb{R}$ und $k \in \mathbb{N}_0$.
An dieser Stelle nehmen wir nun an, dass $\mathcal{M}(\sigma, 1)$ nicht dicht in $C(\mathbb{R})$ liegt, und zeigen, dass dies auf einen Widerspruch führen muss. Wenn jetzt also $\mathcal{M}(\sigma, 1)$ die Dichteeigenschaft nicht erfüllt, dann muss es mindestens ein $k \in \mathbb{N}_0$ geben, für das $x^k \notin \overline{\mathcal{M}(\sigma, 1)}$ gilt. Folglich muss auch $x^k \notin \overline{\mathcal{M}(\sigma_\phi, 1)}$ für jede Funktion $\phi \in C_c^\infty(\mathbb{R})$ gelten, da $\overline{\mathcal{M}(\sigma_\phi, 1)} \subseteq \overline{\mathcal{M}(\sigma, 1)}$ ist. Dies impliziert, dass für dieses $k \in \mathbb{N}_0$

$$\sigma_\phi^{(k)}(b) = 0 \quad \forall b \in \mathbb{R}, \forall \phi \in C_c^\infty(\mathbb{R}) \tag{3.9}$$

gilt. Das bedeutet, dass σ_ϕ für alle $\phi \in C_c^\infty(\mathbb{R})$ ein Polynom maximal $(k-1)$-ten Grades sein kann. Nun gibt es jedoch eine Folge von Funktionen $(\phi_n)_{n \in \mathbb{N}}$ mit

$\phi_n \in C_c^\infty(\mathbb{R})$, zum Beispiel Mollifier, für die die Folge (σ_{ϕ_n}) kompakt gegen σ konvergiert (siehe Satz 5.13). Da die Funktionen σ_{ϕ_n} für alle $n \in \mathbb{N}$ Polynome vom Grad maximal $k-1$ sind, muss auch σ diesem Funktionstypus angehören. Dies ist jedoch ein Widerspruch zu unserer Annahme, dass σ kein Polynom ist. Es muss also $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$ liegen. □

□

Somit sind Feedforward-Netze also universelle Approximatoren. Es sei hier noch zu erwähnen, dass die alleinige Bedingung, dass σ keine Polynomfunktion sein darf, nicht ausreichend ist. Es benötigt stets noch eine weitere Eigenschaft, durch die die Funktion eine gewisse Glattheit bekommt (vgl. [84], S. 158). An dieser Stelle möchte ich auch noch kurz auf die Wichtigkeit der Verzerrung eingehen, denn ohne diese wäre eine so breite Approxmationsmöglichkeit nicht gegeben. Dies wird schnell klar, wenn wir uns dazu ein kurzes Beispiel ansehen. Sei hierfür die Aktivierungsfunktion σ eine klassische Sinusfunktion und wir nehmen an, wir arbeiten ohne eine Verzerrung b. Da die Sinusfunktion eine ungerade Funktion ist, würde die Menge

$$\operatorname{span}\{\sin(w \cdot x) \mid w \in \mathbb{R}\}$$

folglich ausschließlich ungerade Funktionen beinhalten. Aufgrund dieser Tatsache wäre es unmöglich die gerade Cosinusfunktion $\cos(x)$ darzustellen. Infolgedessen kann diese Menge niemals dicht in $C(\mathbb{R})$ liegen. Lässt man hingegen Verzerrungen zu, so ist dies kein Problem, denn $\cos(x) = \sin\left(x + \frac{\pi}{2}\right)$ (vgl. [62], S. 863).

Des Weiteren darf man aufgrund obiger Tatsache nicht einfach schlussfolgern, dass es nur ein Feedforward-Netz braucht und schon lässt sich jedes Problem durch ein wenig Training lösen. Denn das Theorem besagt ausschließlich, dass ein hinreichend großes Feedforward-Netz jede stetige Funktion mit beliebiger Genauigkeit *darstellen* kann, wenn die Parameterwerte passend gesetzt sind. In der Realität gibt es jedoch einige Probleme, die während des Trainings auftreten können. Zum einen kann es passieren, dass der Optimierungsalgorithmus keine geeigneten Parameterwerte liefert, durch die die Zielfunktion gut genug approximiert werden könnte. Zum anderen kann eine falsche Funktion erhalten werden, da beim Training das Problem der *Überanpassung* (*Overfitting*) auftritt (vgl. [32], S. 195). Dabei ist zwar der Fehler zwischen der genäherten Funktion und der Trainingsdaten besonders gering, jedoch kann es zu großen Fehlern kommen, wenn dem Netz unbekannte, neue Daten vorgelegt werden. Folglich ist der Fehler zwischen der Funktion und möglichen Testdaten besonders groß, da das Netzwerk „zu viele “ Eigenschaften aus den Trainingsdaten gelernt hat, die teils gar nicht relevant sind (vgl. [32], S. 109-110). Die beim Training auftretenden Probleme sowie mögliche Lösungsansätze werden in Kapitel 3.2.5 noch einmal genauer betrachtet.

Wie bereits oben erwähnt, reicht die Bedingung, dass σ keine Polynomfunktion sein darf, für eine erfolgreiche Approximation nicht aus, weshalb zusätzlich die Stetigkeit der Aktivierungsfunktion verlangt wurde. Die folgende Erweiterung von Satz 3.1 besagt nun, dass auch Netze mit einer unstetigen nicht-polynomialen Aktivierungsfunktion σ die

Approximationseigenschaft erfüllen, wenn σ dafür Riemann-integrierbar und beschränkt ist (vgl. [84], S. 158). Diese Eigenschaft ist gleichbedeutend damit, dass die Funktion σ beschränkt sowie fast überall stetig ist. Die zweite Eigenschaft bedeutet, dass die Menge der Unstetigkeitsstellen von σ eine Nullmenge, also eine Menge mit Lebesgue-Maß null, ist (vgl. [98], S. 312).

Satz 3.3 (**Das universelle Approximationstheorem nach Pinkus 2**). *Sei $n \in \mathbb{N}$ und $\sigma : \mathbb{R} \to \mathbb{R}$ beschränkt und auf jedem endlichen Intervall $I \subset \mathbb{R}$ Riemann-integrierbar. Betrachte nun die Menge*

$$\mathcal{M}(\sigma, n) := \operatorname{span}\{\sigma(w \cdot x + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R}\}. \tag{3.10}$$

Dann liegt $\mathcal{M}(\sigma, n)$ genau dann dicht in $C(\mathbb{R}^n)$, in der Topologie der gleichmäßigen Konvergenz auf kompakten Teilmengen, wenn σ keine Polynomfunktion ist.

Beweis von Satz 3.3. Die Schritte 1 bis 4 des Beweises von Satz 3.1 gelten nach wie vor und bilden hier den Anfang des Beweises. Es benötigt jedoch noch einen weiteren, finalen Schritt, in dem die Bedingungen für σ ein letztes Mal gelockert werden.

5. Schritt: Sei $\sigma : \mathbb{R} \to \mathbb{R}$ beschränkt, auf jedem endlichen Intervall Riemann-integrierbar sowie (fast überall) keine Polynomfunktion. Dann liegt $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$.

   Beweis. Es sei, wie in Schritt 4, $\phi \in C_c^\infty(\mathbb{R})$ und wir nehmen o.B.d.A. an, dass der kompakte Träger von ϕ im symmetrischen Intervall $[-\alpha, \alpha]$ liegt. Dieses zerlegen wir nun in m Teilintervalle. Dazu legen wir fest, dass

   $$y_i = -\alpha + \frac{2i\alpha}{m} \quad \text{für } i = 0, \ldots, m$$

   sowie

   $$\begin{aligned} \Delta_i &= [y_{i-1}, y_i] \quad \text{und} \\ \Delta y_i &= y_i - y_{i-1} \quad \text{für } i = 1, \ldots, m, \end{aligned}$$

   ist. Für jedes $m \in \mathbb{N}$ gilt nun, dass

   $$\sum_{i=1}^{m} \sigma(x - y_i)\, \phi(y_i)\, \Delta y_i \ \in \mathcal{M}(\sigma, 1).$$

   Diese Summe konvergiert auf jeder kompakten Teilmenge $K \subset \mathbb{R}$ für $m \to \infty$ gleichmäßig gegen σ_ϕ. Um diese Behauptung zu zeigen, beginnen wir mit ein paar

Umformungen:

$$\begin{aligned}
&\left|\sigma_\phi(x) - \sum_{i=1}^{m} \sigma(x-y_i)\,\phi(y_i)\,\Delta y_i\right| \\
&\quad= \left|\int_{-\alpha}^{\alpha} \sigma(x-y)\,\phi(y)\,dy - \sum_{i=1}^{m} \sigma(x-y_i)\,\phi(y_i)\,\Delta y_i\right| \\
&\quad= \left|\sum_{i=1}^{m} \int_{\Delta_i} \left[\sigma(x-y)\,\phi(y) - \sigma(x-y_i)\,\phi(y_i)\right] dy\right| \\
&\quad= \left|\sum_{i=1}^{m} \int_{\Delta_i} [\sigma(x-y)\,\phi(y) \underbrace{-\sigma(x-y_i)\,\phi(y) + \sigma(x-y_i)\,\phi(y)}_{=0} - \sigma(x-y_i)\,\phi(y_i)]dy\right| \\
&\quad= \left|\sum_{i=1}^{m} \int_{\Delta_i} \left[\sigma(x-y) - \sigma(x-y_i)\right]\phi(y)\,dy\right. \\
&\qquad\left. + \sum_{i=1}^{m} \int_{\Delta_i} \sigma(x-y_i)\left[\phi(y) - \phi(y_i)\right] dy\right|.
\end{aligned} \tag{3.11}$$

Da σ auf $K\setminus[-\alpha,\alpha]$ beschränkt ist und ϕ auf $[-\alpha,\alpha]$ stetig ist sowie einen kompakten Träger in $[-\alpha,\alpha]$ besitzt, was zur gleichmäßigen Stetigkeit führt, folgt daraus, dass

$$\lim_{m\to\infty} \sum_{i=1}^{m} \int_{\Delta_i} \sigma(x-y_i)\left[\phi(y) - \phi(y_i)\right] dy = 0 \tag{3.12}$$

ist. Des Weiteren gilt

$$\begin{aligned}
&\left|\sum_{i=1}^{m} \int_{\Delta_i} \left[\sigma(x-y) - \sigma(x-y_i)\right]\phi(y)\,dy\right| \\
&\leq \|\phi\|_{L^\infty([-\alpha,\alpha])} \sum_{i=1}^{m} \left[\sup_{y\in\Delta_i} \sigma(x-y) - \inf_{y\in\Delta_i} \sigma(x-y)\right] \frac{2\alpha}{m}.
\end{aligned} \tag{3.13}$$

Nun ist σ auf $K\setminus[-\alpha,\alpha]$ Riemann-integrierbar, weshalb sich bei Grenzwertbildung

$$\lim_{m\to\infty} \sum_{i=1}^{m} \left[\sup_{y\in\Delta_i} \sigma(x-y) - \inf_{y\in\Delta_i} \sigma(x-y)\right] \frac{2\alpha}{m} = 0 \tag{3.14}$$

ergibt. Aufgrund der Gleichungen (3.11) bis (3.14) gilt also

$$\lim_{m\to\infty} \left|\sigma_\phi(x) - \sum_{i=1}^{m} \sigma(x-y_i)\,\phi(y_i)\,\Delta y_i\right| = 0.$$

Da $\sum_{i=1}^{m} \sigma(x-y_i)\,\phi(y_i)\,\Delta y_i \in \mathcal{M}(\sigma,1)$ ist, folgt dementsprechend, dass $\sigma_\phi \in \overline{\mathcal{M}(\sigma,1)}$ ist. Außerdem gilt nach wie vor, dass

$$\sigma_\phi(x) = \int_{-\infty}^{\infty} \sigma(x-t)\,\phi(t)\,dt \;\in C^\infty(\mathbb{R})$$

denn σ ist nach den Voraussetzungen auf jedem endlichen Intervall Riemann-integrierbar sowie beschränkt und es ist $\phi \in C_c^\infty(\mathbb{R})$. Des Weiteren gilt für die Folge $(\sigma_{\phi_n})_{n\in\mathbb{N}}$, wobei ϕ_n zum Beispiel wieder Mollifier sind, und jede kompakte Teilmenge $K \subset \mathbb{R}$ sowie $p \in [1, \infty)$, dass

$$\lim_{n\to\infty} \|\sigma - \sigma_{\phi_n}\|_{L^p(K)} = 0.$$

Jetzt können wir dieselbe Argumentation wie in Schritt 4 anwenden: Wäre also $\mathcal{M}(\sigma, 1)$ nicht dicht in $C(\mathbb{R})$, dann würde dadurch folgen, dass σ_{ϕ_n} für jedes $n \in \mathbb{N}$ eine Polynomfunktion maximal $(k-1)$-ten Grades wäre. Dies würde bedeuten, dass auch σ (fast überall) ein Polynom höchstens $(k-1)$-ten Grades wäre, was wiederum einen Widerspruch zu unserer Annahme, dass σ kein Polynom sei, darstellt. Somit muss also $\mathcal{M}(\sigma, 1)$ dicht in $C(\mathbb{R})$ liegen. □

□

Betrachtet man obige Beweise genauer, so fällt auf, dass man theoretisch gar nicht die gesamten reellen Zahlen benötigt um passende Parameter zu finden. Tatsächlich reicht es für die Gewichte w aus, dass die Menge, aus denen diese stammen, eine Nullfolge enthält. Wenn nun weiters I ein beliebiges offenes Intervall ist und σ auf genau diesem Riemann-integrierbar ist, dann genügt es, dass die Verzerrungen b aus diesem Intervall stammen, um insgesamt die Dichtheit der Menge $\mathcal{M}(\sigma, 1)$ zu garantieren. Das folgende Korollar fasst diese Überlegungen zusammen (vgl. [84], S. 156-160).

Korollar 3.4. *Es sei $\mathcal{W} \subseteq \mathbb{R}$ eine Menge, die eine Nullfolge enthält und $I \subseteq \mathbb{R}$ ein beliebiges offenes Intervall. Weiters sei $\sigma : \mathbb{R} \to \mathbb{R}$ eine beschränkte Funktion, die auf I Riemann-integrierbar und (fast überall) keine Polynomfunktion ist. Dann ist die Menge*

$$\mathcal{M}(\sigma, 1) = \operatorname{span}\{\sigma(w \cdot x + b) \mid w \in \mathcal{W}, b \in I\}$$

dicht in $C(\mathbb{R})$.

3.2 Folgerungen und weiterführende Betrachtungen

3.2.1 Graphische Veranschaulichung der Approximation durch ein ReLU-Netzwerk

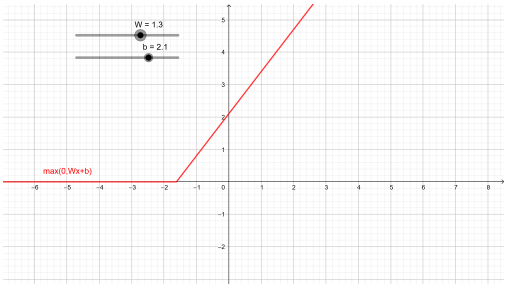
Da das universelle Approximationstheorem bisher nur in der Theorie betrachtet wurde, wird nun eine sehr reduzierte graphische Darstellung einer Funktionsapproximation mittels eines ReLU-Netzes mit einer einzigen Zwischenschicht vorgestellt. Zur Vereinfachung der Veranschaulichung wurde eine univariate Zielfunktion f gewählt. Konkret handelt es sich in unserem Beispiel bei dieser um eine Polynomfunktion dritten Grades mit der zugehörigen Gleichung $f(x) = x^3 + x^2 - x + 1$. Da der Beweis des universellen Approximationstheorems darauf beruht, dass die Menge aller Polynome im Abschluss der Menge

aller durch ein seichtes Feedforward-Netz darstellbaren Funktionen liegt, schien eine Polynomfunktion als Zielfunktion am geeignetsten. Denn durch anschließende Anwendung des Satzes von Weierstraß, Satz 5.9, folgt daraus die Approximation einer beliebigen stetigen Funktion. Das kompakte Intervall, auf dem die Approximation stattfinden soll, ist $[-2, 2]$.

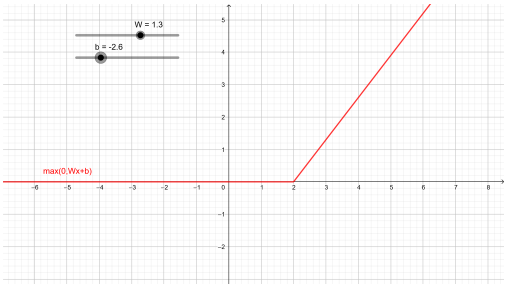
Um nun eine händische Annäherung mittels eines ReLU-Netzes durchzuführen, muss zunächst überlegt werden, wie sich eine Änderung der Parameter auf den Graphen der durch ein solches Netz dargestellten Funktion auswirkt. Die Ausgabe des i-ten Neurons der Zwischenschicht ist durch $\max(0, w_i x + b_i)$, $w_i, b_i \in \mathbb{R}$, $i = 1, \dots N$ gegeben. Eine Änderung der Verzerrung b_i bringt nun eine Verschiebung des Graphen entlang der x-Achse mit sich. Wenn $w_i > 0$ ist, erfolgt die Verschiebung für $\Delta b_i > 0$ nach links und für $\Delta b_i < 0$ nach rechts. Ist $w_i < 0$, so sind die Richtungen der Verschiebung genau umgekehrt. Der „Knick“ des Graphen wandert ebenfalls und befindet sich danach bei $x(b_i) = -\frac{b_i}{w_i}$, für $w_i \neq 0$ konstant. Das Gewicht w_i bestimmt die Steigung der Geraden und auf welcher Seite des Knicks der inhomogen-lineare Abschnitt des Graphen liegt. Ist $w_i > 0$, so ist der Graph monoton steigend und der affine Abschnitt befindet sich auf der rechten Seite des Knicks. Ist hingegen $w_i < 0$, so ist die Funktion monoton fallend und der streng monoton fallende Abschnitt befindet sich links vom Knick. Auch die Stelle des Knicks selbst wandert und zwar entsprechend $x(w_i) = \frac{-b_i}{w_i}$, für b_i konstant und $w_i \neq 0$. In Abbildung 3.1 sind ein paar Beispiele für fixe Parameterwerte gegeben. Im Falle eines ReLU-Netzes mit einer Zwischenschicht werden die Outputs der Neuronen der verdeckten Schicht noch gewichtet bevor diese ausgegeben werden. Das Hinzufügen einer Gewichtung c_i, $i = 1, \dots N$, zum Output wäre graphisch betrachtet eine Streckung ($|c_i| > 1$) oder Stauchung ($0 < |c_i| < 1$) in y-Richtung. Für $c_i < 0$ findet zudem eine Spiegelung entlang der x-Achse statt.

Da nun der Einfluss der Parameter auf die Funktion des ReLU-Netzwerks klar ist, haben wir alle Informationen, die wir benötigen um die Zielfunktion $f(x) = x^3 + x^2 - x + 1$ anzunähern. Gehen wir dazu davon aus, dass wir sechs Neuronen in der Zwischenschicht haben. Der Output unseres Netzes ist dementsprechend durch $\hat{f}(x) = \sum_{i=1}^{6} c_i \sigma(w_i x + b_i)$ gegeben. Das Ziel ist nun passende Werte für c_i, w_i und b_i zu finden, um eine möglichst gute Approximation zu erreichen. Da wir uns bei sechs Neuronen in einem äußerst kleinen Netzwerk befinden, kann man bereits durch Probieren relativ schnell einigermaßen geeignete Gewichte und Verzerrungen finden. Brendan Fortuner versuchte dies und berichtete darüber in seinem Eintrag auf der Internetplattform *Towards Data Science*. Eine erfolgreiche Annäherung von ihm ist in Abbildung 3.2 zu sehen.

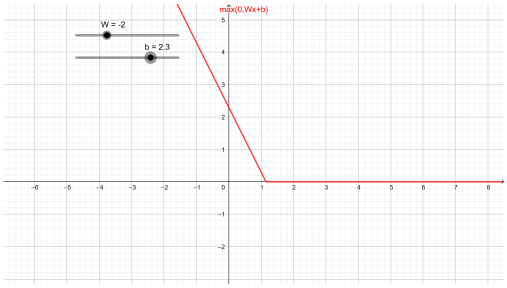
In der Realität, wo hochdimensionale Netze zum Einsatz kommen, erfolgt die Anpassung, wie bereits in früheren Kapitel erwähnt, selbstverständlich nicht manuell sondern mittels Gradientenverfahren. Dabei werden die Parameter zunächst auf Zufallswerte gesetzt. In manchen Fällen, wenn schon ein wenig Vorwissen vorhanden ist, werden die Startparameterwerte auch oft schon zum Problem passend gewählt. Nach der Initialisierung werden dem Netzwerk nun Trainingsdaten vorgesetzt und durch eine Berechnung des Fehlers sowie einem anschließenden Gradientenverfahren werden die Parameter durch konkret berechnete Werte angepasst. Fortuner hat auch dies getestet, indem er einem



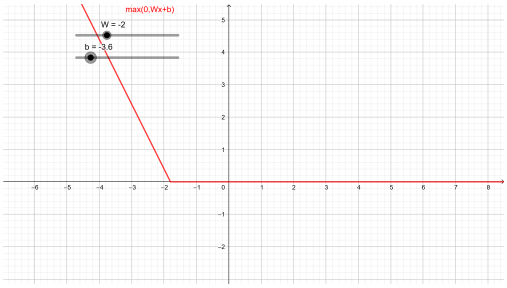
(a) Graph von $\sigma(x) = \max(0, 1, 3x + 2, 1)$



(b) Graph von $\sigma(x) = \max(0, 1, 3x - 2, 6)$



(c) Graph von $\sigma(x) = \max(0, -2x + 2, 3)$



(d) Graph von $\sigma(x) = \max(0, -2x - 3, 6)$

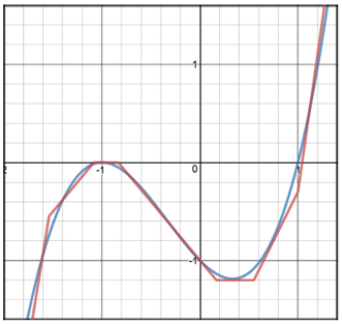
Abbildung 3.1: Mögliche Outputs eines ReLU-Neurons in der Zwischenschicht

ReLU-Netzwerk mit sechs Neuronen in der Zwischenschicht und beliebigen Startparametern die Trainingsdaten $\{(-2, f(-2)), (-1, f(-1)), \ldots, (2, f(2))\}$ vorsetzte. Zudem legte er fest, dass das Training mittels stochastischem Gradientenverfahren erfolgen und eine quadratische Fehlerfunktion minimiert werden sollte. Als Lernrate η wählte er $0,001$. Und tatsächlich konnte das Netzwerk nach einigen Schritten eine relativ gute Approximation erzielen. Das Ergebnis ist in Abbildung 3.3 dargestellt. Das Netzwerk hat dabei die folgenden Parameterwerte berechnet:

$$W = (-1.5272, -1.0567, -0.2828, 1.0399, 0.1243, 2.2446) \text{ und}$$
$$c = (-2.2466, 1.0707, -1.0643, 1.8229, -0.4581, 2.9386).$$

Den Verzerrungsvektor b hat Fortuner in seinem Eintrag leider nicht angegeben.

Es sei hier noch erwähnt, dass die Güte des Resultats von der kompakten Teilmenge, auf die man sich anfangs beschränkt, abhängt. Wird ein Netz, wie oben, auf dem Intervall $[-2, 2]$ trainiert, so ist die erhaltene Funktion nur auf diesem Bereich eine gute Approximation. Folglich kann für Inputs, die außerhalb des Intervalls liegen, ein völlig unpassender Output erhalten werden. Will man, dass das Netz auf einem größeren oder ganz anderen Bereich gut performt, so muss das neuronale Netzwerk noch einmal komplett neu trainiert werden (vgl. [24]).



$$
\begin{aligned}
n_1(x) &= Relu(-5x - 7.7)\\
n_2(x) &= Relu(-1.2x - 1.3)\\
n_3(x) &= Relu(1.2x + 1)\\
n_4(x) &= Relu(1.2x - .2)\\
n_5(x) &= Relu(2x - 1.1)\\
n_6(x) &= Relu(5x - 5)
\end{aligned}
$$

$$
\begin{aligned}
Z(x) = &-n_1(x) - n_2(x) - n_3(x)\\
&+ n_4(x) + n_5(x) + n_6(x)
\end{aligned}
$$

Abbildung 3.2: Manuelle Approximation der Zielfunktion (blau) mit Hilfe eines ReLU-Netzwerks mit 6 Neuronen in der Zwischenschicht (rot) ([24])

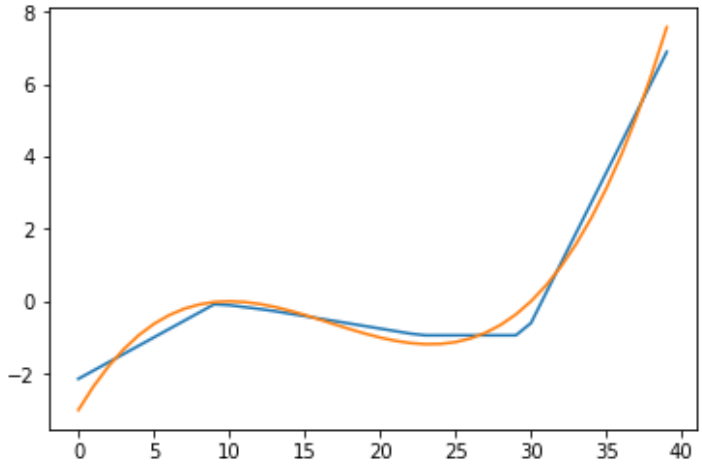


Abbildung 3.3: Die Graphen der Zielfunktion (blau) und der durch das Training erhaltenen Funktion (orange) ([24])

3.2.2 Weitere Versionen des universellen Approximationstheorems

Das universelle Approximationstheorem für Ableitungen

Nachdem jede stetige Funktion mit Hilfe eines Feedforward-Netzes mit einer Zwischenschicht beliebig genau dargestellt werden kann, kommt unter anderem die Frage auf, ob auch die Approximation von Ableitungen möglich ist. Und tatsächlich lautet die Antwort, wie so oft unter gewissen Voraussetzungen, ja. Hornik und sein Team konnten 1990 beziehungsweise 1991 zeigen, dass für ein solches Netz mit einer m-mal stetig differenzierbaren Aktivierungsfunktion, $\sigma \in C^m(\mathbb{R})$, die nichtkonstant sowie beschränkt ist, dicht in $C^m(\mathbb{R}^n)$ liegt (vgl. [51], S. 253). Das Besondere daran ist, dass auch Ableitungen von Funktionen, die im klassischen Sinne nicht differenzierbar sind, jedoch eine schwache Ableitung besitzen, angenähert werden können. Hierunter fällt beispielsweise die Betragsfunktion (vgl. [49], S. 551).

Das in dieser Arbeit vorgestellte Theorem zur gleichzeitigen Approximation einer Funktion und deren Ableitungen wurde 1996 von Xin Li formuliert und bewiesen ([63]) sowie in weiterer Folge 1999 von Allan Pinkus ([84]) etwas abgeändert. Es besagt, dass jede multivariate Funktion und all ihre existierenden stetigen m-ten partiellen Ableitungen durch ein Feedforward-Netz mit einer Zwischenschicht beliebig genau approximiert werden können, wenn die Aktivierungsfunktion gewisse Bedingungen erfüllt. Bevor die konkrete Formulierung präsentiert wird, müssen jedoch noch ein paar Notationen festgelegt werden.

Gegeben sei der Multiindex $\mathbf{m} = (m_1, m_2, \ldots, m_n) \in \mathbb{N}_0^n$. Für diesen wird definiert, dass

$$|\mathbf{m}| := \sum_{i=1}^{n} m_i$$

und

$$\mathbf{l} \leq \mathbf{m} :\Leftrightarrow l_i \leq m_i,\ i = 1, \ldots, n,$$

wobei $\mathbf{l} = (l_1, l_2, \ldots, l_n) \in \mathbb{N}_0^n$ ein weiterer Multiindex ist. Des Weiteren sei

$$\mathbf{x}^{\mathbf{m}} = x_1^{m_1} \cdots x_n^{m_n}$$

und somit definieren wir den Differentialoperator $D^{\mathbf{m}}$ wie folgt:

$$D^{\mathbf{m}} = \frac{\partial^{|\mathbf{m}|}}{\partial \mathbf{x}^{\mathbf{m}}} = \frac{\partial^{|\mathbf{m}|}}{\partial x_1^{m_1} \cdots \partial x_n^{m_n}}. \tag{3.15}$$

Angewandt auf eine differenzierbare Funktion f, ergibt $D^{\mathbf{m}} f$ die partiellen Ableitungen $|\mathbf{m}|$-ter Ordnung von f. Für eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$ gilt $f \in C^{\mathbf{m}}(\mathbb{R}^n)$, genau dann, wenn $D^{\mathbf{k}} f \in C(\mathbb{R})$ für alle $\mathbf{k} \leq \mathbf{m}$, $\mathbf{k} \in \mathbb{N}_0^n$, ist, also wenn all ihre $\mathbf{k}$-ten partiellen Ableitungen existieren und diese stetig sind. Weiteres legen wir fest, dass

$$C^{\mathbf{m}^1, \ldots, \mathbf{m}^r}(\mathbb{R}^n) = \bigcap_{j=1}^{r} C^{\mathbf{m}^j}(\mathbb{R}^n) \tag{3.16}$$

ist, und für eine Zahl $m \in \mathbb{N}_0$ ist

$$C^m(\mathbb{R}^n) = \bigcap_{|\mathbf{m}|=m} C^{\mathbf{m}}(\mathbb{R}^n) = \left\{ f \mid D^{\mathbf{k}} f \in C(\mathbb{R}^n)\ \forall |\mathbf{k}| \leq m \right\}. \tag{3.17}$$

Das folgende Theorem legt nun fest, welche Bedingungen erfüllt sein müssen, um eine Funktion und ihre partiellen Ableitungen beliebig genau durch ein Feedforward-Netz gleichzeitig zu approximieren (vgl. [84], S. 162-163).

Satz 3.5 (Das universelle Approximationstheorem für Ableitungen). *Es seien $\mathbf{m}^j \in \mathbb{N}_0^n$, $j = 1, \ldots, r$, und $m = \max\left\{ \left| \mathbf{m}^j \right| \mid j = 1, \ldots, r \right\}$. Weiters sei $\sigma \in C^m(\mathbb{R}^n)$, wobei σ keine Polynomfunktion ist. Betrachte nun die Menge*

$$\mathcal{M}(\sigma) = \operatorname{span}\left\{ \sigma(w \cdot x + b) \mid w \in \mathbb{R}^n, b \in \mathbb{R} \right\}.$$

Dann liegt die Menge $\mathcal{M}(\sigma)$ dicht in $C^{\mathbf{m}^1, \ldots, \mathbf{m}^r}(\mathbb{R}^n)$.

Die Dichteeigenschaft der obigen Menge ist gleichbedeutend damit, dass es für jede Funktion $f \in C^{\mathbf{m}^1,\ldots,\mathbf{m}^r}(\mathbb{R}^n)$ und jede kompakte Teilmenge $K \subset \mathbb{R}^n$ eine Funktion $g \in \mathcal{M}(\sigma)$ gibt, sodass für jedes $\varepsilon > 0$

$$\max_{x \in K} \left| D^{\mathbf{k}} f(\mathbf{x}) - D^{\mathbf{k}} g(\mathbf{x}) \right| < \varepsilon, \tag{3.18}$$

für alle $\mathbf{k} \in \mathbb{N}_0^n$ mit $\mathbf{k} \leq \mathbf{m}^j$ für ein j, $j = 1, \ldots, r$, gilt. Im Folgenden wird eine Beweisskizze des obigen Theorems gegeben, da der gesamte Beweis den Rahmen dieser Arbeit sprengen würde.

Beweisskizze für Satz 3.5. Die Menge aller Polynome liegt dicht in der Menge $C^{\mathbf{m}^1,\ldots,\mathbf{m}^r}(\mathbb{R}^n)$. Ein Beweis dazu ist unter anderem im zugehörigen Paper von Li ([63], Proposition 4.2 und Korollar 4.3) sowie bei Ito ([50], Lemma 1) zu finden. Somit reicht es zu zeigen, dass alle Polynomfunktionen durch das neuronale Netz, also durch die Elemente der Menge $\mathcal{M}(\sigma, n)$, approximiert werden können.
Betrachten wir nun eine beliebige Polynomfunktion h auf $\mathbb{R}^n$. Dann kann diese für ein gewisses $s \in \mathbb{N}$ entsprechend

$$h(\mathbf{x}) = \sum_{i=1}^{s} p_i(\mathbf{a}_i \cdot \mathbf{x}) \tag{3.19}$$

geschrieben werden, wobei $\mathbf{a}_i \in \mathbb{R}^n$ und p_i, mit $i = 1, \ldots, s$, Polynome in einer Variablen sind. Der Satz sowie der zugehörige Beweis sind im Anhang unter Satz 5.14 zu finden. Aufgrund dieser Tatsache lässt sich der Beweis auf den eindimensionalen Fall reduzieren. Daher reicht es aus zu zeigen, dass jedes Polynom in einer Variablen auf jedem endlichen Intervall $[\alpha, \beta]$ durch

$$\mathcal{M}(\sigma, 1) = \operatorname{span}\{\sigma(w \cdot x + b) \mid w \in \mathbb{R}, b \in \mathbb{R}\} \tag{3.20}$$

bezüglich der Norm

$$\|f\|_{C^m([\alpha,\beta])} = \max_{k=0,1,\ldots,m} \max_{x \in [\alpha,\beta]} \left| f^{(k)}(x) \right|$$

approximiert werden kann.
Da $\sigma \in C^m(\mathbb{R})$ und keine Polynomfunktion ist, liegt die Menge $\mathcal{M}\left(\sigma^{(m)}, 1\right)$ dicht in $C(\mathbb{R})$. Dies beruht auf denselben Gedanken, die in Schritt 3 beziehungsweise Schritt 4 des Beweises von Satz 3.1 ausgeführt sind. Betrachten wir nun eine beliebige Funktion $f \in C(\mathbb{R})$. Aufgrund der Dichteeigenschaft gibt es zu jedem $\varepsilon > 0$ ein $g \in \mathcal{M}(\sigma, 1)$, sodass gilt:

$$\left\| f^{(m)} - g^{(m)} \right\|_{C^m([\alpha,\beta])} < \varepsilon.$$

Angenommen, es läge nun jedes Polynom maximal $(m-1)$-ten Grades in der abgeschlossenen Hülle von $\mathcal{M}(\sigma, 1)$ bezüglich der Norm $\|\cdot\|_{C^m([\alpha,\beta])}$. Sei p ein Polynom, für das

$$p^{(k)}(\alpha) = (f - g)^{(k)}(\alpha), \quad k = 0, 1, \ldots m-1, \tag{3.21}$$

gilt. Nun gilt, aufgrund des Hauptsatzes der Differential- und Integralrechnung sowie der Stetigkeit, dass

$$\begin{aligned}\left|f^{(k-1)}(x) - (g+p)^{(k-1)}(x)\right| &= \left|\int_\alpha^x f^{(k)}(t) - (g+p)^{(k)}(t)\,dt\right| \\ &\leq (\beta-\alpha)\max_{t\in[\alpha,\beta]}\left|f^{(k)}(t) - (g+p)^{(k)}(t)\right|\end{aligned}$$

ist. Durch m-malige Integration ergibt sich, dass $g+p$ die Funktion f bezüglich der Norm $\|\cdot\|_{C^m([\alpha,\beta])}$ beliebig genau approximiert. Es fehlt also noch zu zeigen, dass alle Polynome maximal $(m-1)$-ten Grades im Abschluss von $\mathcal{M}(\sigma,1)$ bezüglich der Norm $\|\cdot\|_{C^m([\alpha,\beta])}$ enthalten sind. Wir müssen also beweisen, dass die Monome $1, x, \ldots, x^{m-1}$ in $\overline{\mathcal{M}(\sigma,1)}$ liegen.
Da σ auf ganz $\mathbb{R}$ m-mal stetig differenzierbar und keine Polynomfunktion ist, können wir die Argumentation von Schritt 2 aus dem Beweis von Satz 3.1 anwenden. Dementsprechend gibt es eine Stelle b_0 für die gilt, dass $\sigma^{(k)}(b_0) \neq 0$, $k = 0, 1, \ldots, m-1$. Somit liegen die Funktionen $x^k\sigma^{(k)}(b_0)$, $k \leq m-1$, für jedes beliebige Intervall $[\alpha,\beta]$ bezüglich der Maximumsnorm $\|\cdot\|_{C([\alpha,\beta])}$ in $\overline{\mathcal{M}(\sigma,1)}$. Durch ein paar weitere Argumentationsschritte, die hier in der Skizze ausgelassen werden, folgt dann, dass die Monome auch bezüglich der strengeren Norm $\|\cdot\|_{C^m([\alpha,\beta])}$ im Abschluss von $\mathcal{M}(\sigma,1)$ liegen. □

Das universelle Approximationstheorem für Netzwerke mit beliebiger Tiefe

Bisher wurden nur Approximationstheoreme für neuronale Netze mit einer Zwischenschicht, in der hinreichend viele Neuronen vorhanden sind, betrachtet. Nachdem jedoch immer mehr Versuche zeigten, dass tiefe Netze zu besseren Ergebnissen führen (vgl. [14], S. 3210), stieg das Interesse bezüglich der Approximationsmöglichkeiten für Netze mit hinreichender Tiefe, aber beschränkter Weite innerhalb der Zwischenschichten. Die ersten Arbeiten dazu waren von Zhou Lu et al., die 2017 zeigten, dass jede Lebesgue-integrierbare Funktion $f : \mathbb{R}^n \to \mathbb{R}$ mittels eines ReLU-Netzwerks, dessen Zwischenschichten maximal $n+4$ Neuronen pro Schicht aufweisen, bezüglich der L^1-Norm beliebig genau approximiert werden kann (vgl. [66], S. 4). Im selben Jahr erkannten Hanin und Sellke, dass ein ReLU-Netzwerk mit n Inputs, m Outputs und einer maximalen Neuronenzahl von $m+n$ innerhalb der Zwischenschichten jede stetige Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ auf jeder kompakten Teilmenge $K \subset \mathbb{R}^n$ mit beliebiger Genauigkeit annähern kann (vgl. [37], S. 2). Diese beiden Erkenntnisse beziehen sich jedoch ausschließlich auf neuronale Netze mit der ReLU-Aktivierungsfunktion. Das folgende Theorem von Kidger und Lions erweitert die bisherigen Erkenntnisse, indem hierbei eine deutlich größere Klasse Aktivierungsfunktionen betrachtet wird (vgl. [52], S. 2).

Satz 3.6 (**Das universelle Approximationstheorem für Netze mit beliebiger Tiefe**). *Es sei $\sigma : \mathbb{R} \to \mathbb{R}$ eine stetige, nicht-affine Funktion, die zumindest an einer Stelle stetig differenzierbar mit einer dort nicht verschwindenden Ableitung ist. Die Menge $\mathcal{M}(\sigma, n, m)$ sei die Menge aller Funktionen, die durch ein neuronales Feedforward-Netz*

mit n Inputs und m Outputs sowie beliebig vielen Zwischenschichten mit je $m+n+2$ Neuronen und der Aktivierungsfunktion σ darstellbar sind. Dann liegt die Menge $\mathcal{M}(\sigma, n, m)$ für jede beliebige kompakte Teilmenge $K \subset \mathbb{R}^n$ dicht in $C(K)$ bezüglich der Supremumsnorm.

Es gibt also für jede Funktion $f \in C(K)$ ein Netzwerk, repräsentiert durch $\hat{f} \in \mathcal{M}(\sigma, m, n)$, sodass für jedes $\varepsilon > 0$ und jede kompakte Teilmenge $K \subset \mathbb{R}^n$

$$\sup_{x \in K} \left\| f(x) - \hat{f}(x) \right\| < \varepsilon \tag{3.22}$$

gilt. Geeignete Aktivierungsfunktionen wären somit beispielsweise alle stückweise stetig differenzierbaren Funktionen, deren abschnittsweise Ableitung nicht überall verschwindet. Dies inkludiert neben der ReLU-Funktion auch viele weitere, derzeit im Deep Learning eingesetzten Funktionen, wie die PReLU- oder die Softplus-Funktion.

3.2.3 Grenzen der Tiefe beziehungsweise Weite für eine erfolgreiche Approximation

Nachdem Anfang der 1990er Jahre die ersten Approximationstheoreme für beliebig weite Netze aufkamen, stellte sich bald die Frage, wie groß solche Netze dann sein müssten. Die ersten Ergebnisse hierzu wurden 1993 von Barron gefunden, der Feedforward-Netze mit einer Zwischenschicht und einer sigmoidalen Aktivierungsfunktion untersuchte. Sein Ziel war es, eine obere Grenze für die benötigte Zahl an Neuronen in der Zwischenschicht zu finden, um jede Funktion, deren Fouriertransformation gewisse Bedingungen erfüllt, mit einem beliebig vorgegebenen Fehler $\varepsilon > 0$ zu approximieren. Nach einer Analyse kam er zu dem Ergebnis, dass dafür maximal $\mathcal{O}\left(\varepsilon^{-2}\right)$ Neuronen gebraucht werden. Ein interessantes Nebenergebnis ist, dass diese Zahl für die betrachtete Funktionsklasse unabhängig von der Anzahl an unabhängigen Variablen der Funktion ist, was man so zunächst nicht vermuten würde. In den darauffolgenden Jahren wurden die unterschiedlichsten zu approximierenden Funktionsklassen und auch weitere Aktivierungsfunktionen genauer untersucht, um dabei mögliche obere und untere Grenzen für Neuronenzahlen zu finden. Mhaskar betrachtete beispielsweise Funktionen mit dem Definitionsbereich $[-1, 1]^n$, die absolut stetig sind. Um diese mit einem maximalen Fehler von ε zu approximieren, werden $\mathcal{O}(\varepsilon^{-n})$ Neuronen in der Zwischenschicht eines Netzwerks mit einer glatten nicht-polynomialen Aktivierungsfunktion, wie der logistischen Aktivierungsfunktion $\frac{1}{1+e^{-x}}$, benötigt. Auch Funktionen im Sobolev-Raum

$$W^{k,p}(Q) = \{ f \in L^p(K) \mid \forall |\alpha| \leq k : D^\alpha f \in L^p(Q) \},$$

mit $k \in \mathbb{N}$, $1 \leq p \leq \infty$, $\alpha \in \mathbb{N}^n$, $K \subseteq \mathbb{R}^n$ offen, wurden genauer untersucht. Dieser Raum enthält alle Funktionen, deren gemischte partielle Ableitungen bis zur k-ten Ordnung an fast allen Stellen existieren, also deren schwache partielle gemischte Ableitungen bis zur Ordnung k existieren und im Lebesgue-Raum $L^p(Q)$ liegen. Die zugehörige Norm

ist durch

$$\|f\|_{W^{k,p}(Q)} = \begin{cases} \left(\sum_{|\alpha|\leq k} \|D^\alpha f\|^p_{L^p(Q)}\right)^{\frac{1}{p}} & \text{wenn } p < \infty \\ \max_{|\alpha|\leq k} \|D^\alpha f\|_{L^p(Q)} & \text{wenn } p = \infty \end{cases}$$

gegeben. Um eine Funktion $f \in W^{k,p}$ zu approximieren, benötigt es $\mathcal{O}\left(\varepsilon^{-\frac{n}{k}}\right)$ Neuronen in der Zwischenschicht (vgl. [70], S. 164-169). Die Zahl an Neuronen nimmt also exponentiell mit der Zahl an unabhängigen Variablen zu. Dieses Problem wird manchmal auch als *Fluch der Dimensionalität* bezeichnet und tritt ebenfalls in anderen Bereichen der Mathematik, wie beispielsweise der numerischen Integration, auf. Der exponentielle Zusammenhang wird schnell klar, wenn man boolesche Funktionen betrachtet. Die Anzahl an möglichen Kombinationen für Inputs einer booleschen Funktion in n binären Variablen beträgt 2^n. Nun gibt es zu jeder dieser Kombinationen je zwei Outputmöglichkeiten, nämlich 0 und 1. Somit erhält man insgesamt 2^{2^n} boolesche Funktionen in n Variablen. Um eine davon zu wählen werden 2^n Bits benötigt, was de facto $\mathcal{O}(2^n)$ Freiheitsgraden entspricht (vgl. [32], S. 195).

Mit der Zeit erkannte man, dass das Hinzufügen weiterer Schichten eine große Reduktion des Rechenaufwandes mit sich bringt. So benötigt man im schlimmsten Fall für die Approximation von Funktionen, die mittels eines k-schichtigen Netzwerks mit der Heaviside-Aktivierungsfunktion mit nur wenigen Neuronen repräsentiert werden, eine exponentielle Anzahl an Neuronen, wenn ein nurmehr $(k-1)$-schichtiges Netzwerk eingesetzt wird. Somit steigt auch die Zahl an zu lernenden Parametern wesentlich an. Folglich werden viel mehr Trainingsdaten benötigt, um auch nur ansatzweise an denselben Generalisierungsfehler des ursprünglichen Netzes heranzukommen (vgl. [8], S. 14-20).

Da sich die ReLU-Funktion in den darauffolgenden Jahren als besonders vielversprechend erwies und die aktuellsten Arbeiten Netze mit genau dieser Aktivierungsfunktion verwenden, werden im folgenden Kapitel deren Approximationseigenschaften und mögliche Grenzen gesondert untersucht.

3.2.4 Die Approximationsmöglichkeiten von Feedforward-Netzen mit der ReLU-Aktivierungsfunktion

Die ReLU-Funktion ist die am häufigsten eingesetzte Aktivierungsfunktion in Feedforward-Netzen. Da sie eine stetige stückweise lineare Funktion ist, folgt daraus, dass jede durch ein solches ReLU-Netwerk dargestellte Funktion ebenfalls diese Gestalt besitzt. Ebenso gilt die Rückrichtung, nämlich, dass jede stetige stückweise lineare Funktion durch ein tiefes Feedforward-Netz, welches die ReLU-Funktion als Aktivierungsfunktion verwendet, exakt dargestellt werden kann (vgl. [3], S. 3).

Ein ReLU-Netz stellt also eine stückweise lineare Funktion dar. Ein linearer Bereich einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^m$ ist nun eine zusammenhängende offene Teilmenge $R \subseteq \mathbb{R}^n$, wenn die Einschränkung $f|_R$ affin ist und für jede offene Obermenge $\tilde{R} \supset R$ die Einschränkung $f|_{\tilde{R}}$ nicht-affin ist. Es liegt relativ leicht auf der Hand, dass ein Netzwerk mit einer höheren Anzahl an linearen Bereichen eine bessere Approximation für eine

nichtlineare Funktion bietet, als eines, welches eine deutlich geringere Anzahl an solchen aufweist. Somit kann die Zahl an möglichen linearen Bereichen als eine Art obere Approximationsgrenze eines ReLU-Netzes angesehen werden. Nun stellt sich jedoch die Frage, wie viele solcher Bereiche ein ReLU-Netz darstellen kann und wie dies im Zusammenhang mit der Anzahl an Schichten sowie Neuronen steht.

ReLU-Netzwerke mit einer Zwischenschicht

Betrachten wir zunächst ein seichtes Netzwerk, also eines mit nur einer Zwischenschicht. Bei einem solchen Netz der Tiefe 2 teilt jedes ReLU-Neuron der Zwischenschicht den Inputraum in zwei Teilräume. Die Grenze ist dabei durch eine Hyperebene gegeben, wobei die Inputs der einen Seite all jene sind, die den Output 0 erzeugen, und auf der anderen liegen jene, die zu einem positiven Ausgabewert führen. Betrachten wir nun ein Netzwerk mit einem n-dimensionalen Input, $x \in \mathbb{R}^n$, einem m-dimensionalen Output, $y \in \mathbb{R}^m$, sowie N Neuronen in der Zwischenschicht. Sowohl die Zwischen- als auch die Outputschicht verfügen über eine Gewichtsmatrix W und einen Verzerrungsvektor b mit entsprechenden Dimensionen. Mathematisch betrachtet ist das Netz also durch

$$f(x) = W^{(2)}\sigma\left(W^{(1)}x + b^{(1)}\right) + b^{(2)} \tag{3.23}$$

gegeben, wobei $\sigma(x) = \max(0, x)$ die ReLU-Funktion ist. Zu jedem Neuron i der Zwischenschicht, $i = 1, \dots N$ gehört nun eine Hyperebene H_i, die durch die Gleichung $W^{(1)}_{i*}x + b^{(1)}_i = 0$ festgelegt ist, wobei $W^{(1)}_{i*} = (w_{i,1}, \dots, w_{i,n})$ für die i-te Zeile der Gewichtsmatrix der Zwischenschicht steht. Unterhalb der Hyperebene H_i liegen nun alle Inputs x, die den Wert 0 als Output erzeugen. Alle Werte darüber liefern eine lineare Aktivierung, deren Gradient durch $W^{(1)}_{i*}$ gegeben ist.

An dieser Stelle folgen nun ein paar Definitionen, um die kommenden Aussagen ein wenig zu vereinfachen. Eine endliche Menge von Hyperebenen $\{H_i\}_{i=1}^N$ bildet ein sogenanntes n-dimensionales *Arrangement* $\mathcal{A} = \{H_i\}_{i=1}^N$ von Hyperebenen in $\mathbb{R}^n$ (vgl. [82], S. 2-4). Eine *Region*, manchmal auch *Zelle* oder *Kammer*, eines Arrangements $\mathcal{A}$ ist dann eine zusammenhängende Komponente, also maximale zusammenhängende Teilmenge (vgl. [54], S. 103), des Komplements der Vereinigung aller Hyperebenen, also von $\mathbb{R}^n \setminus \left(\cup_{i=1}^N H_i\right)$. Um die Bedeutung einer Region ein wenig zu veranschaulichen, nehmen wir an, dass unser Arrangement passend zu unserem Netzwerk durch die Menge $\mathcal{A} = \{H_i\}_{i=1}^N$, mit $H_i = \left\{x \in \mathbb{R}^n \mid W^{(1)}_{i*}x + b^{(1)} = 0\right\}$ für ein gewisses $W^{(1)} \in \mathbb{R}^{N\times n}$ und $b^{(1)} \in \mathbb{R}^N$, gegeben ist. Sei nun $s \in \{-1, +1\}^N$ ein bestimmter Vorzeichenvektor. Dann bilden alle Punkte der Menge $\mathcal{R} = \left\{x \in \mathbb{R}^n \mid \operatorname{sgn}\left(W^{(1)}x + b^{(1)}\right) = s\right\}$ eine Region des Arrangements $\mathcal{A}$.

Angewandt auf unsere Problemstellung ergibt sich nun für ein einschichtiges Netz, dass die Anzahl der möglichen linearen Bereiche der Funktion f gleich der Zahl aller Regionen, die durch die Menge aller Hyperebenen $\{H_i\}$ gebildet werden, ist. Soll nun die Approximationsgüte eines solchen Netzes herausgefunden werden, so müssen wir eine Antwort auf die folgende Frage finden: Wie viele Regionen können durch ein Arrangement von N Hyperebenen im Raum $\mathbb{R}^n$ erzeugt werden?

Um dieses Problem zu lösen gehen wir davon aus, dass die Hyperebenen sich in allgemeiner Lage zueinander befinden, also keinerlei Parallelität aufweisen (vgl. [47], S. 284). Formal bedeutet dies, dass für jede Teilmenge von $\mathcal{A}$ folgendes gilt:

$$\{H_1, \ldots, H_l\} \subseteq \mathcal{A} \Rightarrow \begin{cases} \dim\left(H_1 \cap \cdots \cap H_l\right) = n - l & \text{für } l \leq n \\ H_1 \cap \cdots \cap H_l = \emptyset & \text{für } l > n. \end{cases}$$

In diesem Fall erreichen wir die maximale Anzahl an Regionen. Sind die Gewichte und Verzerrungen generisch, so befindet sich das Arrangement in allgemeiner Lage. Generisch bedeutet in diesem Fall, dass jedes Arrangement durch eine beliebig kleine Änderung von W und b in ein sich in allgemeiner Lage befindendes Arrangement geändert werden kann. Das folgende Theorem, dessen Urform von Zaslavsky stammt und unter anderem von Pascanu et al. etwas umformuliert wurde, besagt, was die maximale Anzahl an Regionen im betrachteten Setting ist.

Satz 3.7. *Es sei $\mathcal{A}$ ein Arrangement, bestehend aus N Hyperebenen des Raums $\mathbb{R}^n$, die sich in allgemeiner Lage befinden. Weiters bezeichne $r(\mathcal{A})$ die Zahl aller Regionen von $\mathcal{A}$. Dann ist*

$$r(\mathcal{A}) = \sum_{j=0}^{n} \binom{N}{j}. \tag{3.24}$$

Beweisskizze. Für eine Beweisskizze von Satz 3.7 betrachten wir den Raum $\mathbb{R}^2$, also die euklidische Ebene. Wir müssen nun zeigen, dass die Zahl der Regionen eines zweidimensionalen Arrangements von N Hyperebenen, hier in unserem Fall also Geraden, in diesem Raum gleich

$$r(\mathcal{A}_N) = \sum_{j=0}^{2} \binom{N}{j} = \binom{N}{2} + \binom{N}{1} + \binom{N}{0} = \binom{N}{2} + N + 1 \tag{3.25}$$

ist. Dies machen wir im Folgenden durch eine vollständige Induktion nach N.

Induktionsanfang.

Sei $N = 0$. In diesem Fall existiert nur eine Region, nämlich der gesamte Raum $\mathbb{R}^n$. Es gilt also $r(\mathcal{A}_0) = 1$. Die rechte Seite von Gleichung (3.25) ergibt

$$\binom{0}{2} + 0 + 1 = 1,$$

da $\binom{0}{0} = 1$ und $\binom{0}{j} = 0$, wenn $j > 0$, ist.

Induktionsvoraussetzung.

Für ein Arrangement aus N Hyperebenen in allgemeiner Lage gelte

$$r(\mathcal{A}_N) = \binom{N}{2} + N + 1. \tag{I.V.}$$

Induktionsschritt.

Nehmen wir an, wir haben N Geraden $\{H_1, \ldots, H_N\}$ in allgemeiner Lage und fügen nun eine weitere Gerade H_{N+1} hinzu, die ebenfalls in allgemeiner Lage zu allen bereits vorhandenen steht. Diese muss folglich jede der N Geraden an einer anderen Stelle schneiden, wodurch sich N Schnittpunkte, die die Gerade H_{N+1} in $N+1$ Segmente teilen, ergeben. Jedes dieser Segmente teilt eine bereits vorhandene Region in zwei und somit kommen nun $N+1$ neue Regionen dazu. Rechnerisch bedeutet dies:

$$\begin{aligned} r\left(\mathcal{A}_{N+1}\right) &= r\left(\mathcal{A}_N\right) + (N+1) \\ &\overset{\text{(I.V.)}}{=} \binom{N}{2} + N + 1 + N + 1 \\ &= \frac{N!}{2!(N-2)!} + N + 1 + N + 1 \\ &= \frac{N(N-1)}{2} + \frac{2N}{2} + (N+1) + 1 \\ &= \frac{N(N+1)}{2} + (N+1) + 1 \\ &= \frac{N(N+1)}{2} \frac{((N+1)-2)!}{((N+1)-2)!} + (N+1) + 1 \\ &= \binom{N+1}{2} + (N+1) + 1. \end{aligned}$$

Die im Induktionsschritt verwendete Methode ist auch als Sweep-Verfahren bekannt, wobei in unserem Fall H_{N+1} die sogenannte Sweep-Gerade war. Durch eine analoge Überlegung lässt sich die Formel für den $\mathbb{R}^3$ herleiten, wobei hier die Sweep-Ebene benutzt wird. Der Beweis für $\mathbb{R}^n$ ergibt sich dann ebenfalls mittels vollständiger Induktion, in diesem Fall nach n, wobei Gleichung (3.25) als Induktionsanfang dient. □

Dieses Ergebnis können wir nun auf unser seichtes neuronales Netzwerk anwenden, wodurch sich das folgende Lemma ergibt (vgl. [82], S. 4-5).

Korollar 3.8. *Es sei ein neuronales ReLU-Feedforward-Netz mit n Inputs, m Outputs und N Neuronen in der Zwischenschicht gegeben. Dann ist die maximale Anzahl der linearen Bereichen $\mathcal{R}(n, N, m)$ der Funktion, die durch dieses Netzwerk dargestellt wird, gleich $\sum_{j=0}^{n} \binom{N}{j}$.*

Ein interessanter Aspekt ist, dass die maximale Zahl an darstellbaren linearen Bereichen unabhängig von der Dimension der Outputschicht ist. Ein Beweis dafür kann im Paper, in dem auch obiges Korollar zu finden ist, nachgelesen werden (vgl. [82], S. 4, Lemma 2). Es ist also $\mathcal{R}(n, N, m) = \mathcal{R}(n, N, 1)$. Um das asymptotische Verhalten anzusehen, sei $n \in \mathcal{O}(1)$, also konstant. Dann gilt für die Zahl an maximal möglichen

linearen Bereichen bei einem Netz mit einem eindimensionalen Output, dass

$$\mathcal{R}(n, N, 1) \in \mathcal{O}(N^n) \tag{3.26}$$

$$\mathcal{R}(n, kN, 1) \in \mathcal{O}(k^n N^n) \tag{3.27}$$

ist. Die zweite Formel (3.27) betrachtet ein Netz mit einer Zwischenschicht, die kN Neuronen aufweist. Sie wird für uns im kommenden Abschnitt noch einmal wichtig sein, da wir dort ReLU-Netzwerke mit k Zwischenschichten zu je N Neuronen betrachten und somit ein Vergleich möglich wird.

Die Zahl der Neuronen in der verdeckten Schicht eines ReLU-Feedforward-Netzes, die für eine Approximation einer absolut stetigen Funktion auf einer kompakten Teilmenge $K \subset \mathbb{R}^n$ bezüglich der L^2-Norm bei einer vorgegebenen Genauigkeit ε nötig ist, ist $\mathcal{O}(\varepsilon^{-n})$. Dies deckt sich mit obigem Resultat für andere Aktivierungsfunktionen. Ebenso gilt für die Approximation einer Funktion $f \in W^{k,2}$, $k \in \mathbb{N}$, dass $\mathcal{O}\left(\varepsilon^{-\frac{n}{k}}\right)$ Zwischenschichtneuronen erforderlich sind (vgl. [87], S. 509).

ReLU-Netzwerke mit mehreren Zwischenschichten

Betrachten wir nun tiefe ReLU-Netze. Hier war es ebenfalls Pascanu, der für diese eine minimale, also nicht wie oben maximale, Anzahl an linearen Bereichen einer durch das Netz dargestellten Funktion ableiten konnte. Ein Beweis dazu ist im zugehörigen Paper ([82], S. 10) zu finden.

Satz 3.9. *Ein ReLU-Feedforward-Netz mit n Inputs, k Zwischenschichten mit zugehörigen Neuronenanzahlen $N_1, \ldots, N_k$ und m Outputs kann eine Funktion mit mindestens $\left(\prod_{i=0}^{k-1} \left\lfloor \frac{N_i}{n} \right\rfloor\right) \sum_{j=0}^{n} \binom{N$ linearen Bereichen darstellen.*

Um dieses Ergebnis noch ein wenig besser mit dem für einschichtige Netze vergleichen zu können, betrachten wir noch das asymptotische Verhalten. Sei auch hierfür wieder $n \in \mathcal{O}(1)$, und wir nehmen an, dass jede der k Zwischenschichten dieselbe Anzahl an Einheiten, nämlich N Stück, aufweist. Ebenso ist die Zahl an möglichen linearen Bereichen wieder unabhängig von der Neuronenzahl der Ausgabeschicht. Dann gilt für die Anzahl $\mathcal{R}$ an linearen Bereichen, dass

$$\mathcal{R}\left(n, \underbrace{N, \ldots, N}_{k}, m\right) = \mathcal{R}\left(n, \underbrace{N, \ldots, N}_{k}, 1\right) \in \Omega\left(\left\lfloor \frac{N}{n} \right\rfloor^{k-1} N^n\right) \tag{3.28}$$

ist. Vergleicht man nun dieses Resultat mit jenem von Gleichung (3.27), so sieht man, dass ein Netz mit mehreren verdeckten Schichten bei gleicher Neuronenzahl, nämlich kN, deutlich mehr lineare Abschnitte erzeugen kann, als ein Netz mit nur einer Zwischenschicht. Genauer steigt diese Anzahl sogar exponentiell mit der Zahl k an Zwischenschichten, sowie $(k-1)$-polynomial mit N (vgl. [82], S. 10-14). Montúfar, Pascanu et al. konnten 2014 diese untere Grenze für eindimensionale Outputs noch etwas verbessern: $\left(\prod_{i=0}^{k-1} \left\lfloor \frac{N_i}{n} \right\rfloor^n\right) \sum_{j=0}^{n} \binom{N_k}{j}$. Für das asymptotische Verhalten eines Netzes, in dem jede Zwischenschicht gleich viel Neuronen N aufweist, folgt daraus, dass

$\mathcal{R} \in \Omega\left(\left(\frac{N}{n}\right)^{(k-1)n} N^n\right)$, falls $N > n$ ist (vgl. [72], S. 8). 2017 konnte Montúfar auch eine obere Grenze für die Anzahl an möglichen linearen Bereichen einer durch ein Netz mit k Zwischenschichten mit zugehöriger Neuronenzahl $N_1, \ldots, N_k$ dargestellten Funktion finden. Diese ist durch

$$\prod_{i=1}^{k} \sum_{j=0}^{m_{k-1}} \binom{N_i}{j}, \quad m_{k-1} = \min\{N_1, \ldots, N_{k-1}\} \tag{3.29}$$

gegeben (vgl. [73], S. 2).

Für die Praxis, wenn ReLU-Netze implementiert werden sollen, ist es natürlich auch wichtig zu wissen, wie viele Zwischenschichten und Neuronen ein Netz aufweisen sollte, um eine gute Approximation zu liefern. Yarotsky konnte 2017 zeigen, dass jede $C^k([0,1]^n)$-Funktion durch ein ReLU-Netz mit einer Tiefe von $\mathcal{O}\left(\ln\left(\frac{1}{\varepsilon}\right)\right)$ und insgesamt $\mathcal{O}\left(\varepsilon^{-\frac{n}{k}} \ln\left(\frac{1}{\varepsilon}\right)\right)$ Neuronen mit einer Fehlerschranke von ε approximiert werden kann (vgl. [105], S. 9).

Das folgende Theorem aus dem von Sunghwan Moon 2021 publizierten Paper liefert eine maximale Zahl an verdeckten Schichten sowie eine Zahl nötiger Neuronen pro Schicht um eine stetige Funktion auf einer kompakten Teilmenge beliebig genau zu approximieren (vgl. [74], S. 2).

Satz 3.10. *Für $\alpha > 0$ und $n \in \mathbb{N}$ sei $f : [-\alpha, \alpha]^n \to \mathbb{R}$ eine stetige Funktion. Weiters sei $\mathcal{M}(n,k)$ die Menge aller Funktionen, die durch ein ReLU-Feedforward-Netz mit nk^n+1, $k \in \mathbb{N}$, Zwischenschichten zu je maximal $n+5$ Neuronen darstellbar sind. Dann gibt es zu jedem $\varepsilon > 0$ ein $k \in \mathbb{N}$, sodass eine Funktion $\hat{f} \in \mathcal{M}(n,k)$ existiert, die*

$$\sup_{x \in [-\alpha,\alpha]^n} \left| f(x) - \hat{f}(x) \right| < \varepsilon \tag{3.30}$$

erfüllt.

Wie die im Theorem angegebene Zahl an Zwischenschichten sowie die Zahl der Neuronen in diesen vom vorgegebenen Fehler ε abhängt, wird laut Moon noch genauer untersucht werden (vgl. [74], S. 11). Neben obiger Ergebnisse konnten auch einige Negativresultate festgestellt werden. So etwa ist ein ReLU-Netz mit einer beliebigen Tiefe, jedoch einer beschränkten Weite von n Neuronen pro Zwischenschicht nicht imstande eine stetige Funktion $f : K \to \mathbb{R}^m$, wobei $K \subset \mathbb{R}^n$ eine kompakte Teilmenge ist, beliebig genau zu approximieren. Die Menge aller durch ein solches Netz darstellbaren Funktionen liegt demnach nicht dicht in $C(K)$ (vgl. [37], S. 10). Gleiches gilt für den Raum $L^1(\mathbb{R}^n)$, das heißt, Netze mit einer beschränkten Weite von n Neuronen pro verdeckter Schicht können auch nicht alle Lebesgue-integrierbaren Funktionen bezüglich eines beliebig kleinen Fehlers approximieren (vgl. [66], S. 5).

Ein tiefes ReLU-Netzwerk benötigt also bei einer vorgegebenen Fehlerschranke ε deutlich weniger Neuronen zur Approximation als eines mit nur einer Zwischenschicht. Daher ist es in vielen Fällen durchaus sinnvoll, diese Netzwerkstruktur vorzuziehen. Möchte

man ein vorhandenes seichtes ReLU-Netzwerk in ein tiefes überführen, so ist dies relativ einfach realisierbar. Das folgende Lemma sowie der zugehörige Beweis stammen von E und Wang. Es enthält die Information darüber, wie man eine solche Umwandlung vornehmen kann (vgl. [20], S. 1739).

Lemma 3.11. *Sei $f_N : \mathbb{R}^n \to \mathbb{R}$ ein ReLU-Netzwerk mit einer Zwischenschicht, die aus N Neuronen besteht. Dann kann f_N für jede Zerlegung $N = N_1 + \cdots + N_L$, $N_l \in \mathbb{N}$, $l = 1, \ldots, L$, durch ein ReLU-Netzwerk mit L verdeckten Schichten mit der jeweiligen Anzahl $N_l + n + 1$ an Neuronen dargestellt werden.*

Beweis. Sei $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ der Inputvektor. Die Funktion f_N lässt sich aufgrund der Voraussetzungen wie folgt darstellen:

$$f_N(x) = \sum_{h=1}^{N} c_h \sigma(W_h x + b_h) + c_0, \tag{3.31}$$

wobei $\sigma(x) = \max(0, x)$ ist und $W_h \in \mathbb{R}^n$ sowie $b_h \in \mathbb{R}$ die Parameter sind. Wir konstruieren nun ein ReLU-Netzwerk mit L verdeckten Schichten, in dem die l-te Schicht $N_l + n + 1$ Einheiten aufweist. Dafür sei $u_{l,k}$ die Ausgabe des k-ten Neurons der l-ten Zwischenschicht. Weiters führen wir zur Vereinfachung die Notation

$$\{u_{l,r:s}\} = \{u_{l,r}, u_{l,r+1}, u_{l,r+2}, \ldots, u_{l,s}\}, \quad \text{mit } r < s, r, s \in \mathbb{N},$$

ein. Nun legen wir für $j = 1, \ldots, N_l$ und $l = 1, \ldots, L$ die folgenden Werte für die Outputs der Zwischenschichten fest:

$$\begin{aligned} &u_{L,1:n} = u_{L-1,1:n} = \cdots = u_{1,1:n} = x_{1:n}, \\ &u_{l,n+j} = \sigma(W_{l,j} x + b_{l,j}), \\ &u_{1,n+N_1+1} = c_0, \end{aligned} \tag{3.32}$$

mit $W_{l,j} = W_{N_1+\cdots+N_{l-1}+j}$ und $b_{l,j} = b_{N_1+\cdots+N_{l-1}+j}$. Für $l = 1, \ldots, L-1$ sei

$$u_{l+1,n+N_{l+1}+1} = u_{l,n+N_l+1} + \sum_{j=1}^{N_l} c_{l,j} u_{l,n+j}, \tag{3.33}$$

wobei $c_{l,j} = c_{N_1+\cdots+N_{l-1}+j}$ ist. Durch diese Formeln folgt, dass

$$u_{l,n+N_l+1} = c_0 + \sum_{j=1}^{N_1+\cdots+N_{l-1}} c_j \sigma(W_j x + b_j)$$

ist. Und unser konstruiertes Netzwerk repräsentiert die Funktion f_N durch:

$$f_N = u_{L,n+N_L+1} + \sum_{j=1}^{N_L} c_{L,j} u_{L,n+j}. \tag{3.34}$$

□

Um die Beweisstruktur ein wenig besser nachvollziehen zu können, wird diese auf ein einfaches Beispiel angewandt. Nehmen wir hierzu an, wir haben ein ReLU-Netzwerk mit einer dreidimensionalen Inputschicht, also $n = 3$, und $N = 8$ Neuronen in der Zwischenschicht. Unser Netzwerk ist also durch

$$f_N(x) = \sum_{h=1}^{8} c_h \sigma(W_h x + b_h) + c_0, \tag{3.35}$$

mit $W_h \in \mathbb{R}^3$, $c_0, c_h, b_h \in \mathbb{R}$ und $\sigma(x) = \max(0, x)$, gegeben. Der Inputvektor ist $x = (x_1, x_2, x_3)^T \in \mathbb{R}^3$. Wir wollen aus diesem Netzwerk nun ein ReLU-Netzwerk mit drei verdeckten Schichten konstruieren, das dieselbe Funktion f_N darstellt. Wir wählen hierfür die Zerlegung $N = 8 = 3 + 2 + 3$, und somit ist $N_1 = 3$, $N_2 = 2$ und $N_3 = 3$. Entsprechend Lemma 3.11 haben unsere verdeckten Schichten $3 + N_l + 1$, $l = 1, 2, 3$, Neuronen. Wie auch im Beweis bezeichne $u_{l,k}$ die Ausgabe des k-ten Neurons der l-ten Zwischenschicht. Nun wenden wir die Festlegungen aus (3.32) an. Somit erhalten wir für die Outputs der ersten $n = 3$ Neuronen der verdeckten Schichten, wobei die erste Spalte jene der ersten Zwischenschicht enthält usw., die folgende Werte:

$$\begin{aligned} u_{1,1} &= x_1 & u_{2,1} &= x_1 & u_{3,1} &= x_1 \\ u_{1,2} &= x_2 & u_{2,2} &= x_2 & u_{3,2} &= x_2 \\ u_{1,3} &= x_3 & u_{2,3} &= x_3 & u_{3,3} &= x_3. \end{aligned}$$

Die nachfolgenden N_l Neuronen liefern die Ausgabewerte

$$\begin{aligned} u_{1,4} &= \sigma(W_1 x + b_1) & u_{2,4} &= \sigma(W_4 x + b_4) & u_{3,4} &= \sigma(W_6 x + b_6) \\ u_{1,5} &= \sigma(W_2 x + b_2) & u_{2,5} &= \sigma(W_5 x + b_5) & u_{3,5} &= \sigma(W_7 x + b_7) \\ u_{1,6} &= \sigma(W_3 x + b_3) & & & u_{3,6} &= \sigma(W_8 x + b_8). \end{aligned}$$

Hierbei wurde verwendet, dass $W_{l,j}$ durch $W_{l,j} = W_{N_1 + \cdots + N_{l-1} + j}$ gegeben ist. Dasselbe gilt analog für $b_{l,j}$ sowie $c_{l,j}$. Somit ergab sich beispielsweise $u_{2,5}$ bei unserem Netz entsprechend

$$u_{2,5} = \sigma(W_{2,2} x + b_{2,2}) = \sigma(W_{3+2} x + b_{3+2}) = \sigma(W_5 x + b_5).$$

Die Outputs des jeweiligen $(3 + N_l + 1)$-ten Neurons der Zwischenschichten sind

$$\begin{aligned} u_{1,7} &= c_0 & u_{2,6} &= u_{1,7} + \sum_{j=1}^{3} c_{1,j} u_{1,3+j} & u_{3,7} &= u_{2,6} + \sum_{j=1}^{2} c_{2,j} u_{2,3+j} \\ & & &= u_{1,7} + \sum_{j=1}^{3} c_j \sigma(W_j x + b_j) & &= u_{2,6} + \sum_{j=4}^{5} c_j \sigma(W_j x + b_j). \end{aligned}$$

Die ersten $n = 3$ Neuronen der verdeckten Schichten nehmen also genau die Werte des Inputvektors an. Diese Setzung ist nötig, damit die darauffolgenden N_l Neuronen der

kommenden Zwischenschicht ihre Ausgabewerte berechnen können. Denn in den Neuronen $(n+1)$ bis $(n+N_l)$ einer Zwischenschicht werden die Inputs gewichtet, der entsprechende Bias addiert und dann die ReLU-Funktion darauf angewandt. Sie übernehmen also den Part, der im ursprünglichen Netzwerk von den Neuronen der einzigen Zwischenschicht durchgeführt wurde. In der jeweiligen Zusatzeinheit, dem $(n+N_l+1)$-ten Neuron, der verdeckten Schichten werden dann die N_{l-1} Outputwerte der Neuronen $(n+1)$ bis $(n+N_{l-1})$ der vorherigen Schicht mit den c_j, $j=1,\ldots,N_{l-1}$, gewichtet und aufsummiert. Im Anschluss wird noch der Wert des Extraneurons der vorherigen Einheit hinzuaddiert.

Zu guter Letzt werden im Neuron der Ausgabeschicht alle neu berechneten Werte der letzten verdeckten Schicht zur derzeitigen Summe hinzugefügt. Es ist also

$$\begin{aligned} f_N &= u_{3,7} + \sum_{j=1}^{3} c_{3,j} u_{3,3+j} = u_{3,7} + \sum_{j=6}^{8} c_j \sigma\left(W_j x + b_j\right) \\ &= u_{2,6} + \sum_{j=4}^{5} c_j \sigma\left(W_j x + b_j\right) + \sum_{j=6}^{8} c_j \sigma\left(W_j x + b_j\right) = u_{2,6} + \sum_{j=4}^{8} c_j \sigma\left(W_j x + b_j\right) \\ &= u_{1,7} + \sum_{j=1}^{3} c_j \sigma\left(W_j x + b_j\right) + \sum_{j=4}^{8} c_j \sigma\left(W_j x + b_j\right) = u_{1,7} + \sum_{j=1}^{8} c_j \sigma\left(W_j x + b_j\right) \\ &= c_0 + \sum_{j=1}^{8} c_j \sigma\left(W_j x + b_j\right) \end{aligned}$$

und dies ist genau dieselbe Funktion, die auch unser einschichtiges Netzwerk darstellt:

$$f_N(x) = \sum_{j=1}^{8} c_h \sigma\left(W_h x + b_h\right) + c_0. \tag{3.36}$$

3.2.5 Mögliche Probleme und wie sie behoben werden können

Das No Free Lunch Theorem

Aufgrund des universellen Approximationstheorems gibt es zu jeder stetigen Zielfunktion ein Feedforward-Netzwerk, das diese beliebig genau approximiert. Die theoretische Existenz eines solchen Netzes ist zwar äußerst erfreulich, jedoch muss in der Realität erst einmal so ein Netz gefunden werden. Dazu benötigt es unter anderem passende Netzwerkarchitekturen sowie ein erfolgreiches Training. Für Letzteres muss also ein Algorithmus gefunden werden, durch den die gesuchten Parameterwerte zur Approximation erhalten werden. 1996 veröffentlichte Wolpert das *No Free Lunch Theorem*, welches besagt, dass zwei beliebige Algorithmen A und B im Schnitt denselben Generalisierungsfehler erzeugen, wenn man alle möglichen Probleme betrachtet. Es gibt also gleich viele Probleme, bei denen A ein besseres Ergebnis liefert, wie solche, bei denen B einen kleineren Testfehler erzeugt. Dies gilt selbst dann, wenn einer der Algorithmen völlig

zufällig arbeitet (vgl. [104], S. 1343). Im Kontext des überwachten Lernens wäre das Problem die Approximation der unbekannten Zielfunktion f. Es gibt also nicht „den einen“ Lernalgorithmus, der auf jedes Problem angewandt eine hervorragende Leistung erbringt.

All die Überlegungen gehen jedoch davon aus, dass wir absolut kein Vorwissen über unser Problem haben. Glücklicherweise ist dies in der Realität eher selten und es zeigt sich, dass Algorithmen, die für konkrete Probleme konstruiert sind, auch eine gute Performanz liefern. Wir müssen also das Problem genauer verstehen, um anschließend einen mit Hilfe von gewissen Festlegungen auf das Ziel abgestimmten Lernalgorithmus konstruieren zu können. Das Hinzufügen von Vorgaben, die zur Verringerung des Generalisierungs- aber nicht des Trainingsfehlers führen sollen, wird als *Regularisierung* bezeichnet (vgl. [32], S. 116). Typische Regularisierungsstrategien sind beispielsweise die Annahme einer stetigen Zielfunktion sowie einer hierarchischen Struktur von Faktoren, durch die der Trainingswert y gegeben ist (vgl. [10], S. 1800). Allgemein gilt es jedoch zu beachten, dass zu viele Vorgaben, insbesondere wenn diese nur leichte Vermutungen sind, eine negative Auswirkung auf den Lernerfolg mit sich bringen können und in Wahrheit ein anderer Algorithmus, der durch die vielen Annahmen ausgeschlossen wurde, in diesem Problemkontext eine bessere Leistung erzielen könnte.

Overfitting

Während des Trainingsprozesses kann es passieren, dass es zu einer Überanpassung (*Overfitting*) kommt, bei der das Netz teils Zusammenhänge aus der Trainingsdatenmenge ableitet, die, wenn man alle möglichen Daten betrachten würde, gar nicht existieren. Dies hat zur Folge, dass sich das Netz sozusagen auf diese Trainingsdaten spezialisiert, das heißt, der Trainingsfehler wird äußerst klein, jedoch ist der Generalisierungsfehler bei unbekannten Daten umso größer. Um eine Überanpassung zu vermeiden, wurden mehrere Regularisierungsstrategien für tiefe Netze entwickelt.

Eine Möglichkeit besteht darin, die Parameterwerte durch gewisse Bedingungen einzuschränken. Die *L_2-Parameter-Regularisierung*, auch als *Weight Decay* bezeichnet, führt beispielsweise dazu, dass sich die Parameterwerte an den Ursprung annähern. Dies wird herbeigeführt, indem die Kostenfunktion um den Regularisierungsterm $\alpha\frac{1}{2}\|\theta\|_2^2$, auch Parameter-Norm-Strafterm genannt, ergänzt wird. Hierbei ist α ein im Vorhinein festgelegter Hyperparameter, durch den man steuern kann, wie sehr die Präferenz immer kleinerer Parameter in den Algorithmus eingehen soll. Durch den Strafterm wird verhindert, dass die Komplexität des Modells zunimmt und manche Merkmale eine zu große Wichtigkeit erhalten. Eine Alternative zur L_2-Regularisierung ist die *L_1-Parameter-Regularisierung*, bei der der Strafterm durch $\alpha\|\theta\|_1$ gegeben ist. Dies führt dazu, dass ein paar Parameter während des Trainingsdurchlaufs auf 0 gesetzt werden, wodurch das Netzwerk eine gewisse Spärlichkeit erhält. Es werden somit weniger wichtige Faktoren ausgeblendet (vgl. [32], S. 227-233).

Eine weitere Methode, um Overfitting zu vermeiden, ist das *Early Stopping*, also ein frühzeitiger Abbruch. Im Trainingsverlauf passiert es häufig, dass der Trainingsfehler nur mehr sehr langsam kleiner wird, jedoch der Fehler der Validierungsdaten zunimmt. Bei

den Validierungsdaten handelt es sich um eine von den Trainingsdaten separierte Menge, die während des Trainingsprozesses zur Überprüfung des momentanen (vermutlichen) Generalisierungsfehlers dient. Je nachdem wie dieser ausfällt, können dann gewisse Hyperparameter, wie beispielsweise der Parameter α der L_2-Regularisierung, entsprechend angepasst werden. Sobald der Validierungsfehler zu steigen beginnt, wird nun bei der Methode des frühen Abbruchs eine Kopie der momentanen Parameter gespeichert. Am Ende des Trainingsprozesses werden dann nicht die zuletzt berechneten Parameterwerte ausgegebenen, sondern jene, die den geringsten Fehler für die Validierungsdaten ergeben haben (vgl. [32], S. 241-244).

Problematische Eigenschaften der Kostenfunktion

Netzwerke im Deep Learning bestehen aus einer Vielzahl an Neuronen. Folglich gibt es eine große Zahl an Parametern, was eine hochdimensionale Kostenfunktion mit sich bringt. Diese ist normalerweise weder linear oder quadratisch noch konvex und weist zudem eine Vielzahl an lokalen Extrema sowie Sattelpunkte auf. Es gibt folglich keine Garantie, dass beim Training des Netzwerks mittels Gradientenabstiegsverfahren ein gutes lokales Minimum gefunden wird. Denn ist beispielsweise ein lokales Minimum gefunden, kann es passieren, dass dieses keinen ausreichend kleinen Kostenfunktionswert aufweist. Eine mögliche Lösung besteht darin, einen erneuten Trainingsdurchlauf mit einem anderen Startpunkt, also anderen Parameterwerten als zuvor, durchzuführen. Außerdem hat sich gezeigt, dass sich die Kostenfunktionswerte lokaler Minima bei zunehmender Dimensionalität der Funktion immer mehr an jenen des globalen Minimums annähern. Dementsprechend stellt das Auftreten vieler lokaler Minima in der Praxis meist gar kein Problem dar (vgl. [13], S. 193).

Neben lokaler Extremstellen treten bei Funktionen in mehreren Variablen zusätzlich viele Sattelpunkte auf, die ebenfalls als kritische Punkte einen Gradienten von null aufweisen. Lee et al. konnten zeigen, dass das Gradientenabstiegsverfahren dennoch fast immer gegen ein lokales Minimum konvergiert, wenn die Sattelpunkte strikt sind und eine hinreichend kleine Lernrate gewählt wird. Ein Sattelpunkt θ heißt strikt, wenn die Hessematrix $H_L(\theta)$, in unserem Fall jene der Kostenfunktion, zumindest einen negativen Eigenwert aufweist (vgl. [61], S. 1246). Zudem besteht auch immer die Möglichkeit, dass die Konvergenz des Algorithmus äußerst langsam vor sich geht. Dies passiert insbesondere in Bereichen, bei denen die Kostenfunktion besonders flach ist. Die Algorithmen brauchen hier oftmals exponentiell viele Schritte, um Sattelpunkten, die sehr oft auch noch von besonders flachen Bereichen mit entsprechend kleinen Gradienten umgeben sind, zu entkommen (vgl. [19], S. 2). Empirische Analysen haben ergeben, dass das stochastische Gradientenverfahren deutlich schneller als das deterministische konvergiert. Ebenso wurde dieses teils noch verändert, um die Konvergenz noch ein wenig zu beschleunigen. So wurde beispielsweise das *gestörte stochastische Gradientenabstiegsverfahren* (*Noisy* oder *Perturbed Gradient Descent*) entwickelt, bei dem die Parameteranpassung durch das Hinzufügen einer kleinen Störung n entsprechend $\theta_n = \theta_{n-1} - \eta\left(\nabla_\theta L\left(\theta_{n-1}\right) + n\right)$ erfolgt. Ein solcher Algorithmus erreicht das lokale Minimum nach bereits polynomiell vielen Schritten (vgl. [28], S. 6).

Neben besonders flachen Bereichen in der Oberfläche der Kostenfunktion gibt es das andere Extrem, nämlich sogenannte Klippen, also besonders steile Abschnitte. Hier ist der zugehörige Gradient sehr groß, was eine starke Änderung der Parameter zur Folge hat. Es kann dadurch passieren, dass man am lokalen Minimum „vorbeispringt“. Um dies zu umgehen, kann die Methode des *Gradienten-Clipping* angewandt werden. Hierbei wird der Gradient, sobald die Norm des Gradienten einen gewissen Schwellenwert s überschreitet, neu skaliert: $\nabla_\theta L^{(\text{neu})} = \frac{s}{\|\nabla_\theta L\|} \nabla_\theta L$. Im Anschluss erfolgt die Parameteranpassung dann mit dem umskalierten Gradienten (vgl. [81], S. 6).

Das Problem der instabilen Gradienten

Das folgende Problem ist typisch für tiefe neuronale Netze und tritt beim Training mittels Gradientenabstiegsverfahren auf. Um die Ursache dieser Schwierigkeit zu verstehen, betrachten wir noch einmal die Formeln (2.27) und (2.28) zur Berechnung des Gradienten der Zwischenschichten $s = S-1, \ldots, 1$ mittels Backpropagation:

$$\frac{\partial L}{\partial b_p^{(s)}} = \sum_k \frac{\partial L}{\partial b_k^{(s+1)}} W_{kp}^{(s+1)} f'\left(v_p^{(s)}\right)$$
$$\frac{\partial L}{\partial W_{pq}^{(s)}} = \frac{\partial L}{\partial b_p^{(s)}} u_q^{(s-1)}.$$

Die Werte partieller Ableitungen früherer Schichten hängen also von jenen der hinteren Schichten sowie auch von deren Parameterwerten und der Ableitung der Aktivierungsfunktion ab. Durch diese Zusammenhänge ergeben sich teils instabile Gradienten. Wenn nun hintere Schichten besonders kleine (große) Parameter oder partielle Ableitungen besitzen und auch die Ableitung der Aktivierungsfunktion entsprechend klein (groß) ist, ergeben sich dadurch noch kleinere (größere) Ergebnisse für die partiellen Ableitungen der früheren Schichten. Es kommt somit also zum *Problem des verschwindenden oder explodierenden Gradienten* (vgl. [78], Kap. 5). Sehr kleine Gradienten bringen den Lernprozess ins Stocken, da die Parameterwerte kaum mehr geändert werden. Explodierende Gradienten haben zu Folge, dass die Parameter sehr hohe Werte annehmen und man das Minimum der Kostenfunktion überspringt. Um Letzteres zu vermeiden, kann das bereits erwähnte Gradienten-Clipping-Verfahren angewandt werden. Ebenso bieten sich Parameter-Regularisierungsstrategien an. Besonders kleine Gradienten werden oft durch bestimmte Aktivierungsfunktionen hervorgerufen. So nimmt die Ableitung der sehr lang eingesetzten Sigmoidfunktion $\sigma'(x) = \frac{e^x}{(e^x+1)^2}$ (siehe Abbildung 2.2a) stets nur Werte im Intervall $\left(0, \frac{1}{4}\right]$ an, wodurch die Ergebnisse in (2.12) und (2.13) bei jedem Schritt verringert werden (vgl. [29], S. 256). Hier bietet die ReLU-Funktion Abhilfe, da die Ableitung den konstanten Wert 1 für positive und 0 für negative Argumente annimmt (vgl. [30], S. 6).

Ein Spezialfall des verschwindenden Gradientenproblems, der ausschließlich bei der ReLU-Funktion auftritt, ist das *Dying-ReLU*-Phänomen. Da die ReLU-Funktion für alle nichtpositiven Argumente den Wert 0 ergibt, kann es bei stark negativen Verzerrungen oder unpassenden Gewichten passieren, dass ein ReLU-Neuron inaktiviert wird. Eine

weitere mögliche Ursache ist beispielsweise eine schlecht gewählte Lernrate, durch die die Parameter bei der Anpassung mit dem berechneten Gradienten auf einen negativen Wert gesetzt werden. Es hat sich jedoch herausgestellt, dass das Problem in der Praxis nicht so schlimm ist, solange es noch aktive Verbindungen gibt, durch die der Gradient zurückpropagiert werden kann ([30], S. 4-5). Da es in seltenen Fällen jedoch dazu kommen kann, dass während des Trainingsprozesses alle ReLU-Neuronen inaktiv werden, wurden ein paar Methoden entwickelt, um dieses Problem zu vermeiden. So kann beispielsweise die zufällige asymmetrische Initialisierung der Parameter Abhilfe bieten. Diese geht von der sogenannten *He-Initialisierung* für ReLU-Netze aus, bei der die Verzerrungen alle auf 0 gesetzt werden und die Gewichte für eine Schicht s, $s = 1, \dots, S$, aus einer Normalverteilung $\mathcal{N}\left(0, \frac{2}{N_{s-1}}\right)$, wobei N_{s-1} die Dimension des Inputs für die Schicht s ist, zufällig gezogen werden (vgl. [38], S. 1029). Bei der abgeänderten Form werden nun alle Zwischenschichten, bis auf die erste, teils mit positiven Gewichten und Verzerrungen, die aus einer asymmetrischen Verteilung gezogen werden, ausgestattet. Durch diese Ungleichmäßigkeit soll das Dying-ReLU-Problem verhindert werden (vgl. [65], S. 8).

Um das Training mittels stochastischem Gradientenabstiegsverfahren auch bei besonders kleinen Gradienten und Parametern zu beschleunigen, wurde zudem die *Momentum-Methode* entwickelt. Hierbei wird auch der vorherige Gradient zur Parameteranpassung verwendet, indem diese entsprechend

$$\begin{aligned} v_n &= \alpha v_{n-1} - \eta \nabla_\theta L\left(\theta_{n-1}\right) \\ \theta_n &= \theta_{n-1} + v_n, \end{aligned} \tag{3.37}$$

mit dem Momentum-Parameter $\alpha \in [0, 1)$ und $n \in \mathbb{N}$, upgedatet werden. Hierbei enthält v die Information über die Geschwindigkeit, also sowohl die Richtung als auch das Tempo, des Gradientenabstiegs. Je größer α im Vergleich zu η ist, desto mehr spielt die vorherige Geschwindigkeit, also auch der zuvor berechnete Gradient, eine Rolle. Dies kann dabei helfen, dass starke Oszillationen während des Iterationsprozesses vermieden werden. Befindet sich beispielsweise das lokale Minimum der Kostenfunktion in einem schmalen Tal, so zeigt der Gradient kaum in die eigentlich benötigte Richtung und es wären viele Iterationen nötig, um bei der Minimumsstelle zu landen. Durch die Backpropagation mit Momentum wird der momentane Gradient nicht so stark gewichtet und vorherige Richtungen werden einbezogen, sodass die gewünschte Richtung besser eingehalten werden kann. Dies ermöglicht eine raschere Konvergenz mit deutlich weniger Schritten (vgl. [92], S. 187). Typische Werte für den Momentum-Parameter α sind $0,5$, $0,9$ oder $0,99$, jedoch können diese auch während des Lernprozesses angepasst werden. Wichtiger ist es jedoch die Lernrate η immer wieder anzugleichen. Dabei startet man meist bei einer relativ kleinen Zahl, die während des Trainings oft nochmals verringert wird (vgl. [32], S. 294).

4 Fazit

Aufgrund ihrer breiten Approximationseigenschaft steckt ein großes Potential in künstlichen neuronalen Netzwerken. Sie können viele insbesondere im Alltag auftretende Funktionen beliebig genau annähern. Unter gewissen Voraussetzungen sogar deren Ableitungen. Dazu ist bereits eine Zwischenschicht ausreichend, was das universelle Approximationstheorem zeigt. Dadurch können die Netzwerke in den unterschiedlichsten Problemkontexten Anwendung finden, wie beispielsweise in der medizinischen Diagnostik oder der Zeitreihenanalyse. Je komplexer ein Problems ist, umso mehr Zwischenschichten sind nötig, um eine effiziente Approximation durch ein neuronales Netzwerk zu bewirken. Und auch für diese Netze gibt es ein zugehöriges universelles Approximationstheorem, das sicherstellt, dass theoretisch jede stetige Funktion in mehreren Variablen beliebig genau approximiert werden kann. Somit wurden in den letzten Jahren tiefe neuronale Netzwerke zum neuen Standard im maschinellen Lernen. Solche Netze können selbst Texte erstellen, Handschrift nachstellen oder Schwarz-Weiß-Bilder durch Farbe zum Leben erwecken (vgl. [76], S. 408).

Um einen erfolgreichen Einsatz künstlicher neuronaler Netzwerke zu ermöglichen, bedarf es jedoch an viel Wissen und Können sowie manchmal auch an ein wenig Glück. So muss die Netzwerkarchitektur festgelegt, eine dem Kontext angemessene Aktivierungsfunktion gewählt sowie passende Startparameter gefunden werden. Lange Zeit war die ReLU-Funktion die gängigste Wahl bei der Aktivierungsfunktion und sie wird auch heute noch gern verwendet. Allerdings wurden in den letzten paar Jahren auch neue Funktionen kreiert, die bessere Leistungen als die ReLU-Funktion erbringen konnten. Ein Beispiel hierfür ist die 2017 von Google Brain entwickelte *Swish*-Funktion $f(x) = x \cdot \sigma(x)$, wobei σ die logistische Sigmoidfunktion (2.3) ist. Sie erzielte beim Einsatz in tiefen Netzwerken beeindruckende Ergebnisse und überbot somit alle bisherig eingesetzten Funktionen (vgl. [89], S. 1). Zwei Jahre später wurde die *Mish*-Funktion $f(x) = x \cdot \tanh(\text{softplus}(x))$ in einem anderen Paper vorgestellt, die wiederum die bisherigen Ergebnisse der Swish- und ReLU-Funktion beim ImageNet-Test übertraf.

Das obige Beispiel ist nur eines von vielen, die zeigen, wie sehr das Feld des Deep Learnings noch erforscht werden muss. Es werden ständig neue Papers publiziert, die bisherige Ergebnisse erweitern und verbessern sowie auch teils völlig neue Erkenntnisse liefern. Wir können also damit rechnen, dass uns neuronale Netzwerke auch in den kommenden Jahren noch in immer mehr Kontexten bereichern werden.

Literaturverzeichnis

[1] Adams, R. A. (1975). *Sobolev Spaces*. Academic Press.

[2] Amann, H., & Escher, J. (2008). *Analysis III* (2. Aufl.). Birkhäuser Basel.

[3] Arora, R., Basu, A., Mianjy, P., & Mukherjee, A. (2018). Understanding Deep Neural Networks with Rectified Linear Units. *International Conference on Learning Representations (ICLR) 2018*. https://arxiv.org/abs/1611.01491

[4] Bahra, G. & Wiese, L. (2020). Parameterizing neural networks for disease classification. *Expert Systems 37(1)*, e12465. https://doi.org/10.1111/exsy.12465

[5] Bengio, Y., Simard, P. & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5(2)*, 157-166. https://doi.org/10.1109/72.279181

[6] Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2007). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems 19 (NIPS'2006)*. https://papers.nips.cc/paper/2006/file/5da713a690c067105aeb2fae32403405-Paper.pdf

[7] Bengio, Y. & LeCun, Y. (2007). Scaling learning algorithms towards AI. In L. Bottou, O. Chapelle, D. DeCoste & J. Weston (Hrsg.), *Large-Scale Kernel Machines* (S. 321-359), The MIT Press.

[8] Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning 2(1)*, 1–127. https://doi.org/10.1561/2200000006

[9] Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures. In G. Montavon, G.B. Orr & Müller KR. (Hrsg.), *Neural Networks: Tricks of the Trade* (S. 437-478). Springer. *https://doi.org/10.1007/978-3-642-35289-8_26*

[10] Bengio, Y., Courville, A. & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8)*, 1798-1828. https://doi.org/10.1109/TPAMI.2013.50

[11] Boghossian, A. & Johnson, P. D. (1989). Pointwise conditions for analyticity and polynomiality of infinitely differentiable functions. *Journal of Mathematical Analysis and Applications, 140(2)*, 301-309. https://doi.org/10.1016/0022-247X(89)90065-6

[12] Bottou L. (2012) Stochastic Gradient Descent Tricks. In G. Montavon, G.B. Orr & Müller KR. (Hrsg.), *Neural Networks: Tricks of the Trade* (S. 421-436). Springer. *https://doi.org/10.1007/978-3-642-35289-8_25*

[13] Choromanska, A., Henaff, M., Mathieu, M., Ben Arous, G. & LeCun, Y.. (2015). The Loss Surfaces of Multilayer Networks. *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, 38*, 192-204. http://proceedings.mlr.press/v38/choromanska15.html

[14] Ciresan, D. C., Meier, U., Gambardella, L. M. & Schmidhuber, J. (2010). Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *Neural Computation, 22 (12)*, 3207–3220. https://doi.org/10.1162/NECO_a_00052

[15] Cortes, C. & Vapnik, V. (1995). Support vector networks. *Machine Learning, 20*, 273–297. https://doi.org/10.1023/A:1022627411411

[16] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems, 2(4)*, 303–314. https://doi.org/10.1007/BF02551274

[17] Deng, J., Dong, W., Socher, R., Li, L., Li, K. & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, 248-255. https://doi.org/10.1109/CVPR.2009.5206848

[18] Ding, B., Qian, H. & Zhou, J. (2018). Activation functions and their characteristics in deep neural networks. *Chinese Control And Decision Conference (CCDC)*, 1836-1841. https://doi.org/10.1109/CCDC.2018.8407425

[19] Du, Simon S., Jin, Chi, Lee, Jason D., Jordan, M. I., Póczos, B. & Singh, A. (2017). Gradient Descent Can Take Exponential Time to Escape Saddle Points. *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17), 11*, 1067–1077. https://dl.acm.org/doi/pdf/10.5555/3294771.3294873

[20] E, W. & Wang, Q. (2018). Exponential convergence of the deep neural network approximation for analytic functions. *Science China Mathematics, 61(10)*, 1733–1740. https://doi.org/10.1007/s11425-018-9387-x

[21] Eluyode1, O.S. & Akomolafe D. T. (2013). Comparative study of biological and artificial neural networks. *European Journal of Applied Engineering and Scientific Research, 2(1)*, 36-46.

[22] Embacher, Franz (2020). *Funktionen, die im Deep Learning auftreten, und deren Gradienten.* mathe online. https://www.mathe-online.at/skripten/techn_Funktionen_im_Deep_Learning_Gradienten/techn_Funktionen_im_Deep_Learning_Gradienten.pdf

[23] Franklin, S. (2014). History, motivations, and core themes. In K. Frankish & W. Ramsey (Hrsg.), *The Cambridge Handbook of Artificial Intelligence* (S. 15-33). Cambridge University Press. https://doi.org/10.1017/CBO9781139046855.003

[24] Fortuner, B. (2017, 7. März). *Can neural networks solve any problem?*. https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6)

[25] Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics, 20*, 121-136.

[26] Fukushima, K. (1980) Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics 36*, 193–202. https://doi.org/10.1007

[27] Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks, 2(3)*, 183–192. https://doi.org/10.1016/0893-6080(89)90003-8

[28] Ge, R., Huang, F., Jin, C. & Yuan, Y.(2015). Escaping From Saddle Points - Online Stochastic Gradient for Tensor Decomposition. *Proceedings of The 28th Conference on Learning Theory, PMLR 40*, 797-842. http://proceedings.mlr.press/v40/Ge15.pdf

[29] Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, 9*, 249-256. http://proceedings.mlr.press/v9/glorot10a.html

[30] Glorot, X., Bordes, A. & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. *Proceedings of Machine Learning Research, 15*, 315-323. http://proceedings.mlr.press/v15/glorot11a.html

[31] Godeau, U., Bouget, C., Piffady, J., Pozzi, T., Gosselin, F. (2020). Lack of definition of mathematical terms in ecology: The case of the sigmoid class of functions in macro-ecology. *Ecology and Evolution 10(24)*, 14209–14220. https://doi.org/10.1002/ece3.7016

[32] Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. The MIT Press. http://www.deeplearningbook.org

[33] Gorman, R. P. & Sejnowski, T. (1988). Learned Classifications of Sonar Targets Using a Massively Parallel Network. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 36(7)*, 1135-1140. https://doi.org/10.1109/29.1640

[34] Goswami, S. (2020). Deep Learning – A State-of-the-Art Approach to Artificial Intelligence. In S. Bhattacharyya, V. Snasel, A. Ella Hassanien, S. Saha, B. Tripathy & A. Hassanien (Hrsg.), *Deep Learning: Research and Applications* (S. 1-20). De Gruyter. https://doi.org/10.1515/9783110670905-001

[35] Graupe, D. (2013). *Principles Of Artificial Neural Networks* (3. Aufl.). World Scientific.

[36] Gupta, M., Jin, L., & Homma, N. (2003). *Static and dynamic neural networks: From fundamentals to advanced theory.* Wiley. https://doi.org/10.1002/0471427950

[37] Hanin, B. & Sellke, M. (2017). Approximating Continuous Functions by ReLU Nets of Minimal Width. *arXiv preprint.* https://arxiv.org/abs/1710.11278

[38] He, K., Zhang, X., Ren, S. & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE International Conference on Computer Vision (ICCV) 2015*, 1026-1034. https://doi.org/10.1109/ICCV.2015.123

[39] Hebb, D. (2002). *The organization of behavior: A neuropsychological theory.* Psychology Press. (Originales Werk 1949 veröffentlicht)

[40] Helms, M., Ault, S. V., Mao, G., Wang, J. (2018). An Overview of Google Brain and Its Applications *Proceedings of the 2018 International Conference on Big Data and Education (ICBDE '18)*, 72–75. https://doi.org/10.1145/3206157.3206175

[41] Heuser, H. (2004). *Lehrbuch der Analysis. Teil 2.* Teubner Verlag.

[42] Hinton, G. E., McClelland, J. & Rumelhart, D. (1986). Distributed representations. In D. E. Rumelhart & J. L. McClelland (Hrsg.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (Volume 1, S. 77–109). MIT Press.

[43] Hinton, G. E., Osindero, S. & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation, 18(7)*, 1527-1554. https://doi.org/10.1162/neco.2006.18.7.1527

[44] Hinton, G. E., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N. & Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine, 29(6)*, 82–97. https://doi.org/10.1109/MSP.2012.2205597

[45] Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen* (Dissertation). TU München.

[46] Hochreiter, S. & Schmidhuber, J. (1997). Long Short-term Memory. *Neural Computation, 9(8)*, 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[47] Hochstättler, W. (2017). *Lineare Optimierung.* Springer Spektrum. https://doi.org/10.1007/978-3-662-54425-9

[48] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks, 2(5)*, 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[49] Hornik, K., Stinchcombe, M., & White, H. (1990). Universal Approximation of an Unknown Mapping and Its Derivatives Using Multilayer Feedforward Networks. *Neural Networks, 3(5)*, 551-560. https://doi.org/10.1016/0893-6080(90)90005-6

[50] Ito, Y. (1993). Approximations of differentiable functions and their derivatives on compact sets by natural networks. *The Mathematical Scientist 18(1)*, 11-19. http://www.appliedprobability.org/content.aspx?Group=tms&Page=TMS181

[51] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks, 4(2)*, 251-257. https://doi.org/10.1016/0893-6080(91)90009-T

[52] Kidger, P. & Lions, T. (2020). Universal approximation with deep narrow networks. *Proceedings of Machine Learning Research, 125*, 1-22.

[53] Kinnebrock, W. (2018). *Neuronale Netze* (2.Aufl). Oldenbourg Wissenschaftsverlag. https://doi-org/10.1515/9783486786361

[54] Kowalsky, H.J. (1961). *Topologische Räume*. Springer Basel AG.

[55] Kriesel, D. (2007). *Ein kleiner Überblick über Neuronale Netze*. http://www.dkriesel.com

[56] Kurenkov, A. (2020). A Brief History of Neural Nets and Deep Learning, *Skynet Today*. https://skynettoday.com/overviews/neural-net-history

[57] LeCun, Y, Bottou, L., Bengio, Y. & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86(11)*, 2278-2324. https://doi.org/10.1109/5.726791

[58] Le, Q., Ranzato, M., Monga, R., Devin, M., Corrado, G., Chen, K., Dean, J. & Ng, A. (2012). Building high-level features using large scale unsupervised learning. *International Conference on Machine Learning 2012*. https://icml.cc/2012/papers/73.pdf

[59] LeCun, Y. (2004). *Research and Projects*. Yann LeCun. http://yann.lecun.com/ex/research/index.html

[60] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature, 521(7553)*, 436-444. https://doi.org/10.1038/nature14539

[61] Lee, J.D., Simchowitz, M., Jordan, M.I. & Recht, B.. (2016). Gradient Descent Only Converges to Minimizers. *29th Annual Conference on Learning Theory, PMLR 49*, 1246-1257.

[62] Leshno, M., Lin, V. Ya., Pinkus, A. & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks, 6(6)*, 861–867. https://doi.org/10.1016/S0893-6080(05)80131-5

[63] Li, X. (1996). Simultaneous approximations of multivariate functions and their derivatives by neural networks with one hidden layer. *Neurocomputing, 12(4)*, 327-343. https://doi.org/10.1016/0925-2312(95)00070-4

[64] Linguistic Data Consortium, The Trustees of the University of Pennsylvania (1993). *TIMIT Acoustic-Phonetic Continuous Speech Corpus*. https://doi.org/10.35111/17gk-bn40

[65] Lu, L., Shin, Y., Su, Y. & Karniadakis, G. E. (2020). Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics, 28(5)*, 1671-1706. https://doi.org/10.4208/cicp.OA-2020-0165

[66] Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. (2017). The Expressive Power of Neural Networks: A View from the Width. *31st Conference on Neural Information Processing Systems (NIPS 2017)*. https://proceedings.neurips.cc/paper/2017/hash/32cbf687880eb1674a07bf717761dd3a-Abstract.html

[67] McCulloch, W. S. & Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology, 52(1-2)*, 99-115. https://doi.org/10.1007/BF02459570 (Originales Werk 1943 veröffentlicht)

[68] Medler , D. A. (1998). A Brief History of Connectionism. *Neural Computing Surveys 1*, 61–101.

[69] Mey, S. (2021, 4. April). Künstliche Intelligenz komponiert ‚neuen' Nirvana-Song. *Der Standard*. https://www.derstandard.de/story/2000125583234/kuenstliche-intelligenz-komponiert-neuen-nirvana-song

[70] Mhaskar, H. N. (1996). Neural Networks for Optimal Approximation of Smooth and Analytic Functions. *Neural Computation, 8(1)*, 164–177. https://doi.org/10.1162/neco.1996.8.1.164

[71] Mohamed, A., Dahl, G. E. & Hinton, G. (2012). Acoustic Modeling Using Deep Belief Networks. *IEEE Transactions on Audio, Speech, and Language Processing, 20(1)*, 14-22. https://doi.org/10.1109/TASL.2011.2109382

[72] Montúfar, G. F., Pascanu, R., Cho, K. & Bengio, Y. (2014). On the number of linear regions of deep neural networks. *Advances in Neural Information Processing Systems 27 (NIPS 2014)*. https://papers.nips.cc/paper/2014/file/109d2dd3608f669ca17920c511c2a41e-Paper.pdf

[73] Montúfar, G. (2017). Notes on the number of linear regions of deepneural networks. *SampTA 2017*. https://www.researchgate.net/publication/322539221_Notes_on_the_number_of_linear_regions_of_deep_neural_networks

[74] Moon S. (2021). ReLU Network with Bounded Width Is a Universal Approximator in View of an Approximate Identity. *Applied Sciences 11(1)*, 427. https://doi.org/10.3390/app11010427

[75] Nagyfi, R. (2018, 4. September). *The differences between Artificial and Biological Neural Networks*. Towards Data Science. https://towardsdatascience.com/the-differences-between-artificial-and-biological-neural-networks-a8b46db828b7

[76] Neapolitan, R. E. & Jiang, X. (2018) *Artificial Intelligence: With an Introduction to Machine Learning* (2. Aufl.). Chapman and Hall/CRC Press. https://doi.org/10.1201/b22400

[77] Neunhäuserer, J. (2017). *Mathematische Begriffe in Beispielen und Bildern*. Springer Spektrum. https://doi.org/10.1007/978-3-662-53710-7

[78] Nielsen, M. (2019). *Neural Networks and Deep Learning*. http://neuralnetworksanddeeplearning.com/

[79] Nilsson, N. (2009). *The Quest for Artificial Intelligence*. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511819346

[80] Olazaran, M. (1993). A Sociological History of the Neural Network Controversy. *Advances in Computers, 37*, 335-425. https://doi.org/10.1016/S0065-2458(08)60408-8

[81] Pascanu, R., Mikolov, T. & Bengio, Yoshua (2013). On the difficulty of training recurrent neural networks. *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML'13), 28(3)*, 1310-1318. http://proceedings.mlr.press/v28/pascanu13.pdf

[82] Pascanu, R., Montufar, G., & Bengio, Y. (2014). On the number of response regions of deep feed forward networks with piece-wise linear activations. *International Conference on Learning Representations (ICLR) 2014*. https://arxiv.org/abs/1312.6098

[83] Petersen, B. E. (1983). *Introduction to the Fourier Transform & Pseudo-differential Operators*. Pitman.

[84] Pinkus, A. (1999). Approximation theory of the MLP model in neural networks. *Acta Numerica, 8*, 143-195. https://doi.org/10.1017/S0962492900002919

[85] Pinkus, A. (2015). *Ridge Functions*. Cambridge University Press. https://doi.org/10.1017/CBO9781316408124

[86] Pitts, W. & McCulloch, W. S. (1947). How we know universals the perception of auditory and visual forms. *The Bulletin of Mathematical Biophysics, 9(3)*, 127-147. https://doi.org/10.1007/BF02478291

[87] Poggio, T., Mhaskar, H., Rosasco, L., Miranda, B. & Liao, Q. (2017). Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing, 14(5)*, 503–519. https://doi.org/10.1007/s11633-017-1054-2

[88] Raina, R., Madhavan, A. & Ng, A. (2009). Large-scale deep unsupervised learning using graphics processors. *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*, 873–880. https://doi.org/10.1145/1553374.1553486

[89] Ramachandran, P., Zoph, B., & Le, Q.V. (2018). Searching for Activation Functions. *ICLR 2018 Workshop Submission.* https://openreview.net/forum?id=SkBYYyZRZ

[90] Ranzato, M., Poultney, C., Chopra, S., and LeCun, Y. (2006). Efficient learning of sparse representations with an energy-based model. *Advances in Neural Information Processing Systems 19 (NIPS'2006).* https://papers.nips.cc/paper/2006/file/87f4d79e36d68c3031ccf6c55e9bbd39-Paper.pdf

[91] Ranzato, M., Boureau, Y., & LeCun, Y. (2007). Sparse feature learning for deep belief networks. *Advances in Neural Information Processing Systems 20 (NIPS'2007).* https://papers.nips.cc/paper/2007/file/c60d060b946d6dd6145dcbad5c4ccf6f-Paper.pdf

[92] Rojas, R. (1996). *Neural networks: A systematic introduction.* Springer.

[93] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review, 65(6)*, 386-408. https://doi.org/10.1037/h0042519

[94] Russell, S. & Norvig, P. (2016). *Artificial intelligence: A modern approach.* Pearson Education.

[95] Sharkawy, Abdel-Nasser. (2020). Principle of Neural Network and Its Main Types: Review. *Journal of Advances in Applied & Computational Mathematics, 7(1)*, 8-19. 10.15377/2409-5761.2020.07.2

[96] Sharma, S., Sharma S., Athaiya, A. (2020). Activation Functions in Neural Networks. *International Journal of Engineering Applied Sciences and Technology 4(12)*, 310-316. https://doi.org/10.33564/IJEAST.2020.v04i12.054

[97] Schmidhuber, J. (2014). Deep Learning in Neural Networks: An overview. *Neural Networks, 61(1)*, 85-117. https://doi.org/10.1016/j.neunet.2014.09.003

[98] Sohrab, H. H. (2014). *Basic Real Analysis*. Birkhäuser Basel. https://doi.org/10.1007/978-1-4939-1841-6

[99] Stanford Vision Lab (2020). *About ImageNet*. https://www.image-net.org/about.php

[100] Sugomori, Y., Kaluza, B., Soares, F. M., & Souza, A. M. F. (2017). *Deep Learning: Practical Neural Networks with Java*. Packt Publishing.

[101] Szandala, T. (2021). Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. In: Bhoi A., Mallick P., Liu CM., Balas V. (Hrsg.) *Bio-inspired Neurocomputing* (203-224). Springer. https://doi.org/10.1007/978-981-15-5495-7_11

[102] Widrow, B. & Hoff, M. E. (1960). Adaptive Switching Circuits. In *1960 IRE WESCON Convention Record*, volume 4, 96–104. IRE, New York.

[103] Willems, K. (2019, 10. September). Keras Tutorial: Deep Learning in Python. DataCamp. https://www.datacamp.com/community/tutorials/deep-learning-python

[104] Wolpert, D. H. (1996). The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation, 8(7)*, 1341-1390. https://doi.org/10.1162/neco.1996.8.7.1341

[105] Yarotsky, D. (2017). Error bounds for approximations with deep ReLU networks. *Neural Networks, 94*, 103-114. https://doi.org/10.1016/j.neunet.2017.07.002

Abbildungsverzeichnis

1.1 Zusammenhang von künstlicher Intelligenz und Deep Learning ([32], S. 9) 4
1.2 Vergleich zwischen einem biologischen (links) und einem künstlichen Neuron (rechts) ([103]) 6

2.1 Typischer Aufbau eines einfachen Feedforward-Netzes mit der Inputschicht (Layer 1), einer Zwischenschicht (Layer 2) und der Outputschicht (Layer 3) ([4], Fig. 1) 21
2.2 Drei häufig verwendete Aktivierungsfunktionen in FNNs (eigene Darstellung) 26

3.1 Mögliche Outputs eines ReLU-Neurons in der Zwischenschicht 46
3.2 Manuelle Approximation der Zielfunktion (blau) mit Hilfe eines ReLU-Netzwerks mit 6 Neuronen in der Zwischenschicht (rot) ([24]) 47
3.3 Die Graphen der Zielfunktion (blau) und der durch das Training erhaltenen Funktion (orange) ([24]) 47

5 Anhang

5.1 Mathematische Definitionen und Sätze für die Beweise aus Kapitel 3

Definition 5.1 (Metrik)**.** Eine *Metrik* auf einer Menge X ist eine Abbildung $d : X \times X \to \mathbb{R}$, die folgende Eigenschaften für beliebige $x, y, z \in X$ erfüllt:

1. $d(x,y) \geq 0$ und $d(x,y) = 0 \Leftrightarrow x = y$
2. $d(x,y) = d(y,x)$
3. $d(x,z) \leq d(x,y) + d(y,z)$.

(X,d) wird dann metrischer Raum genannt.

Definition 5.2 (Abgeschlossene Hülle)**.** Die *abgeschlossene Hülle*, auch *Abschluss*, einer Teilmenge Ω eines metrischen Raums (X,d) ist

$$\overline{\Omega} = \{x \in X \mid \forall \epsilon > 0 \, \exists y \in \Omega : d(x,y) < \epsilon\}. \tag{5.1}$$

Definition 5.3 (Lebesgue-Raum)**.** Es sei X ein metrischer Raum mit dem Borel'schen Maß μ und $0 < p < \infty$. Wir definieren nun den Vektorraum

$$\mathcal{L}^p(X,\mu) = \left\{ f : X \to \mathbb{R} \mid f \text{ ist messbar und } \int_X |f(x)|^p \, d\mu(x) < \infty \right\},$$

also den Raum aller auf X definierten Funktionen f, für die $|f|^p$ Lebesgue-integrierbar ist. Nun gelten zwei Funktionen dieses Raumes als äquivalent (oder fast überall gleich), wenn sie sich nur auf einer Nullmenge unterscheiden. Dies drücken wir durch die Äquivalenzrelation $\sim$ aus. Der *Lebesgue-Raum*, auch *L^p-Raum*, ist nun durch den Quotientenraum

$$L^p(X,\mu) = \mathcal{L}^p / \sim$$

gegeben. Die zugehörige L^p-Norm ist

$$\|f\|_p := \left(\int_X |f|^p \, d\mu(x) \right)^{\frac{1}{p}}.$$

Für $p = \infty$ ist der Raum L^∞ durch $L^\infty(X,\mu) := \mathcal{L}^\infty(X,\mu) / \sim$ mit

$$\mathcal{L}^\infty(X,\mu) := \{ f : X \to \mathbb{R} \mid f \text{ ist messbar und } \|f\|_\infty < \infty \},$$

und der Norm

$$\|f\|_\infty = \operatorname{ess\,sup}_{x \in X} |f(x)|$$

gegeben, wobei „ess sup“ das wesentliche Supremum bezeichnet, das weiter unten in (5.3) definiert wird (vgl. [77], S. 241).

Definition 5.4 (wesentlich beschränkt)**.** Eine bezüglich des Lebesgue-Maßes λ fast überall definierte Funktion $f : \Omega \to \mathbb{R}$, wobei $\Omega \subseteq \mathbb{R}^n$ eine messbare Menge ist, heißt *wesentlich beschränkt* auf Ω, wenn es eine Konstante $C \in \mathbb{R}_0^+$ gibt, sodass

$$|f(x)| \leq C \quad \text{für fast alle } x \in \Omega \tag{5.2}$$

gilt, sprich wenn $f(x)$ fast überall beschränkt ist. In diesem Fall schreiben wir $f \in L^\infty(\Omega)$, wobei die Norm durch

$$\|f(x)\|_{L^\infty(\Omega)} = \inf\{C | \lambda(\{x : |f(x)| > C\} = 0)\} =: \operatorname{ess\,sup}_{x \in \Omega} |f(x)|, \tag{5.3}$$

gegeben ist. Ist $\Omega \subset \mathbb{R}^n$ ein Gebiet, also eine nichtleere zusammenhängende offene Teilmenge des Raums $\mathbb{R}^n$, so heißt f *lokal wesentlich beschränkt* auf Ω, wenn für jede kompakte Teilmenge $K \subset \Omega$ gilt, dass $f \in L^\infty(K)$ ist. In diesem Fall schreiben wir $f \in L^\infty_{\text{loc}}(\Omega)$.

Definition 5.5 (dicht)**.** Die Menge $F \subset L^\infty_{\text{loc}}(\mathbb{R}^n)$ heißt, bezüglich der Topologie der gleichmäßigen Konvergenz, dicht in $C(\mathbb{R}^n)$, wenn es zu jeder Funktion $f \in C(\mathbb{R}^n)$ und kompakten Teilmenge $K \subset \mathbb{R}^n$ eine Folge von Funktionen $g_j \in F$ gibt, sodass (vgl. [62], S. 863)

$$\lim_{j \to \infty} \|f - g_j(x)\|_{L^\infty(K)} = 0. \tag{5.4}$$

Definition 5.6 (Ridge-Funktion)**.** Eine *Ridge-Funktion* ist eine Funktion der Form $G : \mathbb{R}^n \to \mathbb{R}$,

$$G(x) = g(a \cdot x) = g(a_1 x_1 + a_2 x_2 + \cdots + a_n x_n), \tag{5.5}$$

wobei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion und $a \in \mathbb{R}^n$ ist (vgl. [84], S. 154).

Satz 5.7 (Satz von Vostrecov und Kreines)**.** *Sei $A \subset \mathbb{R}^n$ und*

$$\mathcal{R}(A) = \operatorname{span}\{g(a \cdot x) \mid g \in C(\mathbb{R}), a \in A\}.$$

Dann liegt $\mathcal{R}(A)$ genau dann dicht in $C(\mathbb{R}^n)$, wenn es keine nichttriviale homogene Polynomfunktion gibt, die auf A verschwindet (vgl. [84], S. 155).

Ein detaillierter Beweis von Satz 5.7 ist unter anderem im Werk von Pinkus ([85], S. 61-63) zu finden.

Beweisidee. $\mathcal{R}(A)$ beinhaltet unter anderem die Funktionen $cos(a \cdot x)$ und $sin(a \cdot x)$, $a \in A$. Diese sind für jede kompakte Teilmenge $K \subset \mathbb{R}^n$ dicht in $C(K)$. Eine weitere Teilmenge wäre $\operatorname{span}\left\{(a \cdot x)^k \mid a \in \mathbb{R}^n, k \in \mathbb{N}_0\right\}$, die bekanntlich ebenfalls dicht in $C(\mathbb{R}^n)$ liegt (vgl. [84], S. 154). □

Satz 5.8. *Es sei I ein beliebiges reelles Intervall und $f \in C^\infty(I)$. Wenn es zu jedem $x \in I$ ein $n_x \in N$ gibt, sodass $f^{(n_x)}(x) = 0$ ist, dann muss f ein Polynom sein (vgl. [11], S. 301).*

Satz 5.9 (Approximationssatz von Weierstraß)**.** *Zu jeder stetigen Funktion $f : [a,b] \to \mathbb{R}$ und jedem $\varepsilon > 0$ gibt es ein Polynom p, sodass gilt (vgl. [41], S. 63):*

$$|f(x) - p(x)| < \varepsilon \quad \forall x \in [a,b]\,. \tag{5.6}$$

Definition 5.10 (Träger)**.** Es sei X ein topologischer Raum und $f : X \to \mathbb{R}$ eine stetige Funktion. Der *Träger* von f ist

$$\operatorname{supp}(f) := \overline{\{x \in X \mid f(x) \neq 0\}}, \tag{5.7}$$

also der Abschluss der Menge aller Nichtnullstellen von f (vgl. [62], S. 864).

Die Menge der auf $[a,b]$ definierten Polynome $P([a,b]) \subset C([a,b])$ liegt laut obigem Satz also dicht in $C([a,b])$, das heißt, $\overline{P([a,b])} = C([a,b])$.

Definition 5.11 (Faltung)**.** Es sei $f \in \mathcal{L}^p(\mathbb{R}^n)$, $p \in [1,\infty)$, und $g \in \mathcal{L}^1(\mathbb{R}^n)$. Dann ist die *Faltung* von f und g definiert durch (vgl. [2], S. 170)

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)\, g(x-t)\, dt. \tag{5.8}$$

Definition 5.12 (Mollifier)**.** Es sei $\phi : \mathbb{R}^n \to \mathbb{R}$ eine glatte Funktion mit kompaktem Träger, also $\phi \in C_c^\infty(\mathbb{R}^n)$, die nichtnegativ ist und folgende Eigenschaften aufweist:

(i) $\phi(x) = 0$, wenn $|x| \geq 1$ und

(ii) $\int_{\mathbb{R}^n} \phi(x) = 1$.

Ein bekanntes Beispiel hierfür ist die Funktion

$$\phi(x) = \begin{cases} k e^{-\frac{1}{1-x^2}} & \text{wenn } |x| < 1 \\ 0 & \text{wenn } |x| \geq 1, \end{cases}$$

wobei $k > 0$ so zu wählen ist, dass die Gültigkeit von Eigenschaft (ii) erhalten bleibt. Sei nun $\varepsilon > 0$, dann ist $\phi_\varepsilon(x) = \frac{1}{\varepsilon^n}\phi\left(\frac{x}{\varepsilon}\right)$ ebenfalls eine glatte nichtnegative Funktion mit kompaktem Träger und es gilt:

(i) $\phi_\varepsilon(x) = 0$, wenn $|x| \geq \varepsilon$ und

(ii) $\int_{\mathbb{R}^n} \phi_\varepsilon(x) = 1$.

Eine solche Funktion ϕ_ε wird *Mollifier* genannt (vgl. [1], S. 29).

Mollifier spielen eine wichtige Rolle bei der Funktionsapproximation, denn durch sie können nicht-glatte Funktionen durch Folgen glatter Funktionen angenähert werden. Dies geschieht über eine Faltung, was durch das folgende Theorem ausgedrückt wird. Dieses, sowie ein zugehöriger Beweis, ist im Buch von Petersen ([83], S. 7) zu finden.

Satz 5.13. *Es sei $f \in C(\mathbb{R}^n)$, $K \subset \mathbb{R}^n$ eine beliebige kompakte Teilmenge und ϕ_ε ein Mollifier. Dann konvergiert die Faltung $(f * \phi_\varepsilon)(x)$ für $\varepsilon \to 0$ für alle $x \in K$ gleichmäßig gegen $f(x)$.*

Satz 5.14. *Es sei $h : \mathbb{R}^n \to \mathbb{R}$ eine Polynomfunktion. Dann lässt sich h für ein gewisses $s \in \mathbb{N}$ wie folgt schreiben:*

$$h(\mathbf{x}) = \sum_{i=1}^{s} p_i(\mathbf{a}_i \cdot \mathbf{x}), \tag{5.9}$$

wobei $\mathbf{a}_i \in \mathbb{R}^n$ und p_i Polynomfunktionen in einer Variablen, für $i = 1, \ldots, s$, sind (vgl. [84], S. 163).

Der folgende Beweis stammt aus dem Werk von Pinkus ([84], S. 163-164).

Beweis. Betrachte den Vektorraum H_k aller auf $\mathbb{R}^n$ definierten homogenen Polynome vom Grad k, also

$$H_k = \left\{ \sum_{|\mathbf{l}|=k} b_{\mathbf{l}} \mathbf{x}^{\mathbf{l}} \mid b_{\mathbf{l}} \in \mathbb{R} \right\} = \operatorname{span}\left\{ \mathbf{x}^{\mathbf{l}} \mid |\mathbf{l}| = k \right\}. \tag{5.10}$$

Die Dimension des Raumes ist durch

$$s = \dim H_k = \binom{n-1+k}{k}$$

gegeben. Weiters sei P_k der Vektorraum aller Polynome maximal k-ten Grades:

$$P_k = \left\{ \sum_{|\mathbf{l}|\leq k} b_{\mathbf{l}} \mathbf{x}^{\mathbf{l}} \mid b_{\mathbf{l}} \in \mathbb{R} \right\} = \operatorname{span}\left\{ \mathbf{x}^{\mathbf{l}} \mid |\mathbf{l}| \leq k \right\} = \cup_{t=0}^{k} H_k. \tag{5.11}$$

Nun seien $\mathbf{m}^1, \mathbf{m}^2 \in \mathbb{N}_0^n$ zwei Multiindizes mit $\left|\mathbf{m}^1\right| = \left|\mathbf{m}^2\right| = k$. Dann gilt

$$\begin{aligned} D^{\mathbf{m}^1} \mathbf{x}^{\mathbf{m}^2} &= \frac{\partial^k}{\partial x_1^{m_1^1} \partial x_2^{m_2^1} \cdots \partial x_n^{m_n^1}} x_1^{m_1^2} x_2^{m_2^2} \cdots x_n^{m_n^2} \\ &= \delta_{\mathbf{m}^1, \mathbf{m}^2} m_1^1! \cdots m_n^1! = \delta_{\mathbf{m}^1, \mathbf{m}^2} \mathbf{m}^1!, \end{aligned} \tag{5.12}$$

wobei δ das Kronecker-Delta bezeichnet. Für ein Polynom $q \in H_k$, das also durch $q(\mathbf{x}) = \sum_{|\mathbf{l}|=k} b_{\mathbf{l}} \mathbf{x}^{\mathbf{l}}$, mit $b_{\mathbf{l}} \in \mathbb{R}$ gegeben ist, sei der Differentialoperator $q(D)$ wie folgt definiert:

$$q(D) = q\left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n} \right) = \sum_{|\mathbf{l}|=k} b_{\mathbf{l}} D^{\mathbf{l}}.$$

Somit kann jedes nichttriviale lineare Funktional L aus dem Raum H_k mit Hilfe eines Polynoms $q \in H_k$ durch

$$L(p) = q(D)\, p \tag{5.13}$$

für jedes $p \in H_k$ ausgedrückt werden. Betrachten wir nun die Funktionen $(\mathbf{a} \cdot \mathbf{x})^k$, die in H_k liegen. Für diese gilt, für $\mathbf{m} \in \mathbb{N}_0^n$, $|\mathbf{m}| = m$, dass

$$D^{\mathbf{m}} (\mathbf{a} \cdot \mathbf{x})^k = \begin{cases} \frac{k!}{(k-m)!} \mathbf{a}^{\mathbf{m}} (\mathbf{a} \cdot \mathbf{x})^{k-m} & m \leq k \\ 0 & m > k. \end{cases} \tag{5.14}$$

Folglich ist $D^{\mathbf{m}} (\mathbf{a} \cdot \mathbf{x})^k = k! \mathbf{a}^{\mathbf{m}}$ für $|\mathbf{m}| = k$. Außerdem gilt für $(\mathbf{a} \cdot \mathbf{x})^k$ und $q \in H_m$, dass

$$q(D) (\mathbf{a} \cdot \mathbf{x})^k = \begin{cases} \frac{k!}{(k-m)!} q(\mathbf{a}) (\mathbf{a} \cdot \mathbf{x})^{k-m} & m \leq k \\ 0 & m > k. \end{cases} \tag{5.15}$$

Und dementsprechend ist $q(D) (\mathbf{a} \cdot \mathbf{x})^k = k! q(\mathbf{a})$, wenn $q \in H_k$, also $m = k$, ist.
Nun wissen wir, da $s = \dim H_k$ ist, dass es s Punkte $\mathbf{a}_1, \ldots, \mathbf{a}_s$ gibt, sodass $s = dim H_k|_A$, mit der Menge $A = \{\mathbf{a}_1, \ldots, \mathbf{a}_s\}$, gilt. Die Menge $\left\{ (\mathbf{a}_i \cdot \mathbf{x})^k \mid i = 1, \ldots, s \right\}$ ist nun eine Basis für H_k. Um dies zu zeigen, nehmen wir an, dies wäre nicht der Fall. Dann gäbe es ein nichttriviales lineares Funktional, das für jedes $(\mathbf{a}_i \cdot \mathbf{x})^k$ verschwindet. Es gilt also für ein nichttriviales $q \in H_k$, dass

$$0 = q(D) (\mathbf{a}_i \cdot \mathbf{x})^k = k! q(\mathbf{a}_i), \quad i = 1, \ldots, s, \tag{5.16}$$

ist. Dies stellt jedoch einen Widerspruch zu unserer Wahl der Menge A dar, da diese $s = dim H_k|_A$ erfüllt. Dementsprechend muss also $\operatorname{span} \left\{ (\mathbf{a}_i \cdot \mathbf{x})^k \mid i = 1, \ldots, s \right\} = H_k$ gelten. Folglich ist dann auch $H_t = \operatorname{span} \left\{ (\mathbf{a}_i \cdot \mathbf{x})^t \mid i = 1, \ldots, s \right\}$, für $t = 0, 1, \ldots, k$. Denn wäre dies nicht so, gäbe es für ein $t \in \{0, 1, \ldots, s\}$ ein nichttriviales $q \in H_t$, das auf A verschwindet. Sei weiters $p \in H_{k-t}$ ein beliebiges, von Null verschiedenes Polynom. Dann gilt, dass die Funktion $pq \in H_k$ auf ganz A verschwindet, was jedoch einen Widerspruch zu obiger Feststellung darstellt. Aufgrund dieser Tatsache ergibt sich eine Basis für den Raum aller Polynomfunktionen vom Grad $\leq k$:

$$P_k = \operatorname{span} \left\{ (\mathbf{a}_i \cdot \mathbf{x})^t \mid i = 1, \ldots, s, t = 0, 1, \ldots, k \right\}. \tag{5.17}$$

Sei nun π_k der Raum aller Polynomfunktionen in einer Variablen vom Grad k. Dann folgt, dass

$$P_k = \left\{ \sum_{i=0}^{s} p_i (\mathbf{a}_i \cdot \mathbf{x}) \mid p_i \in \pi_k, i = 1, \ldots, s \right\} \tag{5.18}$$

ist. Und somit ergibt sich, dass jedes $h \in P_k$ entsprechend Gleichung (5.9) geschrieben werden kann. □

Zusammenfassung

Diese Arbeit beschäftigt sich mit künstlichen neuronalen Netzwerken mit dem Fokus darauf, welche Funktionen durch diese dargestellt werden können. Der hierfür zentrale Satz ist das universelle Approximationstheorem für Feedforward-Netzwerke mit einer verdeckten Schicht. Dieses besagt, dass solche Netze jede stetige Funktion beliebig genau approximieren können.

Des Weiteren wird eine kurze Einführung in die Welt neuronaler Netze und des Deep Learnings gegeben sowie weitere Versionen des Theorems vorgestellt. Ebenso werden mögliche Probleme beim Training des Netzwerks, also dem Approximieren der Zielfunktion, betrachtet, und Lösungsansätze gegeben.

Abstract

This master thesis deals with artificial neural networks with the focus on the question, which functions could be represented by them. The main theorem for this problem is the universal approximation theorem for feedforward networks with one hidden layer. It states, that these networks can approximate any continous function arbitrarily well.

In addition, there will be a short introduction into the world of neural networks and deep learning as well as some more versions of the theorem. Furthermore some potential problems that can occour while training such networks, so that they can approximate the objective function, will be examined and some approaches to solving these will be given.