

Künstliche Neuronale Netzwerke und Deep Learning

Stefan Selle

Professor für Wirtschaftsinformatik
Fakultät für Wirtschaftswissenschaften
Hochschule für Technik und Wirtschaft des Saarlandes

Saarbrücken, 12.05.2018

Kurzfassung

Die vorliegende Arbeit mit dem Titel *Künstliche Neuronale Netzwerke und Deep Learning* beschäftigt sich mit einem derzeit viel beachteten Teilgebiet der Künstlichen Intelligenz (KI). Künstliche Neuronale Netzwerke (KNN) sind von der Natur inspirierte, lernfähige Systeme, die sich auf sehr vielfältige Art und Weise einsetzen lassen, z.B. um Muster in Daten zu finden (*Data Mining*). Insbesondere in den Bereichen Bild- und Spracherkennung wurden in den letzten Jahren große Fortschritte durch die Verwendung von *Deep Learning* (*DL*) erzielt. Bei diesen Methoden werden sehr große Datensätze (*Big Data*) zunehmend auf grafischen Prozessoren (GPUs) verarbeitet, um spezielle und tiefe KNN zu trainieren. Typische Anwendungen sind das autonome Fahren im Bereich Bilderkennung und digitale Assistenten wie bspw. Apples Siri, Amazons Alexa oder Googles Assistant im Bereich Spracherkennung.

In dieser Arbeit werden zunächst die Grundlagen zu KNN und DL dargestellt. Kurze Erklärungen zu den biologischen Hintergründen liefern zusätzliche Informationen. Drei weit verbreitete Netzwerktypen mit den dahinter liegenden Ideen und Konzepte werden erläutert: *Multilayer Perceptron (MLP)*, *Convolutional Neural Network (CNN)* und *Recurrent Neural Network (RNN)*. Zwei spezielle Netzwerke werden außerdem detailliert vorgestellt: *Long Short-Term Memory (LSTM)* und *Gated Recurrent Unit (GRU)*. Die Ergebnisse einer intensiven Recherche und aktuellen Marktanalyse führen zur Präsentation von 22 verfügbaren *Open Source* DL-Softwarelösungen. Die Python-Bibliotheken TensorFlow und Keras werden schließlich ausgewählt, um exemplarisch drei Anwendungen durchzuführen: zwei Aufgaben stammen aus dem Bereich der Bilderkennung bzw. Objektklassifizierung und eine Aufgabe beschäftigt sich mit dem Thema *Text Mining* bzw. *Sentiment Analysis*. Mit nur wenigen Zeilen Quelltext lassen sich die oben genannten KNN erfolgreich konfigurieren, trainieren und evaluieren. Zu den gegebenen Bild- und Textdaten lassen sich die Modelle auf einer *High-End* Grafikkarte von Nvidia, die CUDA unterstützt, relativ schnell ausführen. Somit ist diese Hardware-Software-Kombination besonders gut geeignet, um in der Lehre und der angewandten Forschung an der Hochschule für Technik und Wirtschaft des Saarlandes (htw saar) zukünftig eingesetzt zu werden.

Inhaltsverzeichnis

Abbildungsverzeichnis	v
Tabellenverzeichnis	vii
Abkürzungsverzeichnis	ix
1 Einleitung	1
2 Theoretische Grundlagen	7
2.1 Künstliches Neuron	7
2.2 Perzepron	8
2.3 Topologie	9
2.4 Lernen	9
2.5 Anwendungskategorien	12
2.6 Modelle	15
2.7 Optimierungstechniken	20
3 Multilayer Perceptron	25
3.1 Aufbau	25
3.2 Backpropagation	26
3.3 Erweiterungen	27
3.4 Aktivierungsfunktionen	28
3.5 Kostenfunktionen	30
3.6 Deep Learning	30
4 Convolutional Neural Network	33
4.1 Visueller Cortex	33
4.2 Aufbau und Schichten	35
4.3 LeNet-5	38
4.4 AlexNet	39
4.5 GoogleLeNet	40
4.6 Sonstige	41
5 Recurrent Neural Network	43
5.1 Bausteine	44
5.2 Sequenzen	45
5.3 Lernverfahren	46
5.4 Long Short-Term Memory	47
5.5 Varianten	49
5.6 Einsatzgebiete	51
6 Software	53
6.1 Caffe	56
6.2 Caffe2	57
6.3 Chainer	58

6.4	CNTK	59
6.5	Darknet	60
6.6	Deep Water	61
6.7	DL4J	62
6.8	Dlib	63
6.9	DSSTNE	64
6.10	Gluon	65
6.11	Keras	66
6.12	Lasagne	67
6.13	MXNet	68
6.14	Neon	69
6.15	OpenNN	70
6.16	Paddle	71
6.17	PyTorch	72
6.18	SINGA	73
6.19	TensorFlow	74
6.20	TFLearn	75
6.21	Theano	76
6.22	Torch	77
7	Anwendungen	79
7.1	MNIST	80
7.2	CIFAR-10	88
7.3	IMDb	93
8	Zusammenfassung und Ausblick	107
	Quellenverzeichnis	115

Abbildungsverzeichnis

1.1	Darstellung miteinander vernetzter Neuronen	2
2.1	Struktur eines künstlichen Neurons	7
2.2	Struktur eines Perzeptrons	8
2.3	Darstellung logischer Schaltungen mittels Perzeptoren	9
2.4	Gradientenabstiegsverfahren im zweidimensionalen Fehlerraum	11
2.5	Typische Probleme des Gradientenabstiegsverfahrens	12
2.6	Black Box Modell	12
2.7	Taylorsche Reihe als Funktionennetz	13
2.8	Data Mining Kategorien	14
2.9	Übertrainieren und Trainingsabbruch	21
2.10	k-fache Kreuz-Validierung	21
3.1	Mehrschichten-Perzeptron	25
3.2	Logistischen Funktion mit Ableitung	28
3.3	Aktivierungsfunktionen im Vergleich	29
3.4	Standardnormalverteilung mit Intervallen	31
4.1	Signalweg von der Netzhaut zur Sehrinde	34
4.2	Funktionale Spezialisierung des visuellen Cortex	35
4.3	Feature Learning durch Deep Learning	35
4.4	Aufbau eines typischen CNN	36
4.5	Max Pooling mit 2x2-Filter und Stride=2	37
4.6	Aufbau des CNN LeNet-5	38
4.7	Aufbau des AlexNet	39
4.8	Schematischer Aufbau eines Inception Moduls	40
4.9	Graphischer Aufbau des GoogleLeNet	40
4.10	Tabellarischer Aufbau des GoogleLeNet	41
4.11	Funktionsweise einer Skip Connection in einer Residual Unit	42
4.12	Squeeze-and-Excitation-Module ersetzen Inception- und Residuen-Module	42
5.1	Beispiel für ein Hidden Markov Model	43
5.2	Drei Feedback-Typen	44
5.3	Basiseinheit im RNN	44
5.4	Ausgerollte RNN-Basiseinheit	45
5.5	Verarbeitung von Sequenzen	45
5.6	Aufbau einer LSTM-Einheit	47
5.7	Ausgerollte LSTM-Einheit	47
5.8	Schritt 1: Forget-Gate	48
5.9	Schritt 2: Input-Gate	48
5.10	Schritt 3: Zelle	48
5.11	Schritt 4: Output-Gate	48
5.12	Peephole-LSTM	49
5.13	Gekoppelte LSTM-Einheit	50
5.14	Gated Recurrent Unit (GRU)	50

5.15 Modell des Spracherkennungsprozesses	51
6.1 Vergleich Caffe und Caffe2	57
6.2 Vergleich Define-and-Run mit Define-by-Run	58
6.3 CNTK-Architektur	59
6.4 Darknet-Nightmare: Der Schrei von Edvard Munch	60
6.5 Deep Water Architektur	61
6.6 Komponenten von Dlib	63
6.7 Architektur mit DSSTNE	64
6.8 MXNet: Technologie-Stack	68
6.9 Neon: Layer Taxonomie	69
6.10 OpenNN: Klassendiagramm	70
6.11 Vergleich Torch und PyTorch	72
6.12 Apache SINGA Software Stack	73
6.13 TensorFlow Technologie-Stapel	74
6.14 Theano: Beispiel-Graph mit Apply-Objekt	76
6.15 Torch 7 Technologie-Stapel	77
7.1 Beispielbilder des MNIST Test-Datensatzes	80
7.2 Fehlerraten für verschiedene Klassifizierer zur MNIST Datenbank	81
7.3 Genauigkeit des MLP für die Klassifikation der MNIST-Datenbank	84
7.4 Fehler bzw. Loss des MLP für die Klassifikation der MNIST-Datenbank . .	85
7.5 Genauigkeit des CNN für die Klassifikation der MNIST-Datenbank	87
7.6 Fehler bzw. Loss des CNN für die Klassifikation der MNIST-Datenbank .	87
7.7 Beispielbilder der CIFER-10 Datenbank	88
7.8 CNN verarbeitet Bilder der CIFER-10 Datenbank	89
7.9 Genauigkeit des CNN für die Klassifikation der CIFAR-10-Datenbank . .	92
7.10 Fehler bzw. Loss des CNN für die Klassifikation der CIFAR-10-Datenbank	93
7.11 Genauigkeit des MLP für die Klassifikation der IMDb-Datenbank	97
7.12 Fehler bzw. Loss des MLP für die Klassifikation der IMDb-Datenbank .	98
7.13 Genauigkeit des CNN für die Klassifikation der IMDb-Datenbank	100
7.14 Fehler bzw. Loss des CNN für die Klassifikation der IMDb-Datenbank . .	100
7.15 Genauigkeit des LSTM für die Klassifikation der IMDb-Datenbank	102
7.16 Fehler bzw. Loss des LSTM für die Klassifikation der IMDb-Datenbank .	102
7.17 Genauigkeit des GRU für die Klassifikation der IMDb-Datenbank	104
7.18 Fehler bzw. Loss des GRU für die Klassifikation der IMDb-Datenbank .	104
8.1 Gartners magische Quadranten zu Data Science und Maschine Learning .	110
8.2 Workflow der KNIME Analytics Platform	111

Tabellenverzeichnis

4.1	Topologie der LeNet-5-Architektur	38
4.2	Topologie der AlexNet-Architektur	39
6.1	Übersicht von DL-Repositories auf der Plattform GitHub	53
6.2	Übersicht von Attributen zu einem Repository auf GitHub	54
6.3	Übersicht ausgewählter DL-Softwarelösungen auf der Plattform GitHub	55
6.4	Steckbrief zur Caffe-Bibliothek	56
6.5	Steckbrief zum Caffe2-Framework	57
6.6	Steckbrief zum Chainer-Framework	58
6.7	Steckbrief zum CNTK-Framework	59
6.8	Steckbrief zum Darknet-Framework	60
6.9	Steckbrief zur Deepwater-Bibliothek	61
6.10	Steckbrief zur DL4J-Bibliothek	62
6.11	Steckbrief zur Dlib-Bibliothek	63
6.12	Steckbrief zur DSSTNE-Bibliothek	64
6.13	Steckbrief zur Gluon-Bibliothek	65
6.14	Übersicht zu Keras-Verwendungsmöglichkeiten	66
6.15	Steckbrief zur Keras-Bibliothek	66
6.16	Programmieren mit der Lasagne-Bibliothek	67
6.17	Steckbrief zur Lasagne-Bibliothek	67
6.18	Programmieren mit dem MXnet-Framework	68
6.19	Steckbrief zur MXNet-Bibliothek	68
6.20	Steckbrief zur Neon-Bibliothek	69
6.21	Steckbrief zur OpenNN-Bibliothek	70
6.22	Steckbrief zur Paddle-Bibliothek	71
6.23	Steckbrief zur PyTorch-Bibliothek	72
6.24	Steckbrief zur SINGA-Bibliothek	73
6.25	Steckbrief zur TensorFlow-Bibliothek	74
6.26	Steckbrief zur TFLearn-Bibliothek	75
6.27	Steckbrief zur Theano-Bibliothek	76
6.28	Steckbrief zur Torch-Bibliothek	77
7.1	Topologie des CNN für die MNIST-Datenbank	85
7.2	Topologie des CNN für die CIFAR-10-Datenbank	89
7.3	Vergleich der Ergebnisse	105

Abkürzungsverzeichnis

Adaline	Adaptive Linear Neuron
Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
API	Application Programming Interface
ART	Adaptive Resonance Theory
ASF	Apache Software Foundation
AWS	Amazon Web Services
BAIR	Berkeley Artificial Intelligence Research
BAM	Bidirectional Associative Memory
BI	Business Intelligence
BLAS	Basic Linear Algebra Subprograms
BN	Batch Normalization
BP	Backpropagation
BPTT	Backpropagation Through Time
BSB	Brain-State-in-a-Box
BSD	Berkeley Software Distribution
BVLC	Berkeley Vision and Learning Center
CIFAR	Canadian Institute For Advanced Research
CGL	Corpus Geniculatum Laterale
CNN	Convolutional Neural Network
CNTK	Cognitive Toolkit
CPU	Central Processing Unit
DAE	Deep Autoencoder
DBN	Deep Belief Network
DL	Deep Learning
DL4J	Deep Learning for Java
DSSTNE	Deep Scalable Sparse Tensor Network Engine

ECS	Elastic Container Service
EMR	Elastic MapReduce
FM	Frequenzmodulation
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HDFS	Hadoop Distributed File System
HMM	Hidden Markov Model
HTML	Hypertext Markup Language
IDS	Intrusion Detection System
IMDb	Internet Movie Database
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoT	Internet of Things
IT	Information Technology
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
kHz	Kilohertz
KDD	Knowledge Discovery in Databases
KI	Künstliche Intelligenz
KNN	Künstliches Neuronales Netzwerk
Lasso	Least Absolute Shrinkage and Selection Operator
LVQ	Linear Vector Quantization
LRN	Local Response Normalisation
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
Madaline	Multiple Adaline
MAPE	Mean Absolute Percentage Error
MILA	Montreal Institute for Learning Algorithms
MKL	Math Kernel Library
MLFF	Multilayer Feedforward Network
MLP	Multilayer Perceptron

MNIST	Modified National Institute of Standards and Technology
ms	Millisekunde
MSE	Mean Squared Error
NER	Named Entity Recognition
NIST	National Institute of Standards and Technology
NLP	Natural Language Processing
NMT	Neural Machine Translation
NP	Nichtdeterministisch polynomiale Komplexität
OCR	Optical Character Recognition
ONNX	Open Neural Network Exchange
OS	Operating System
PyPI	Python Package Index
RBF	Radial Basis Function
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit
RMSprop	Root Mean Square Propagation
RNN	Recurrent Neural Network
Rprop	Resilient Backpropagation
S3	Simple Storage Service
SA	Simulated Annealing
SE	Squeeze-and-Excitation
SDA	Stacked Denoising Autoencoder
SGD	Stochastic Gradient Descent
SOM	Self-Organizing Map
SVM	Support Vector Machine
TCO	Total Cost of Ownership
TSP	Travelling Salesman Problem
UC	University of California
UKW	Ultrakurzwellen
XOR	Exklusives Oder
YOLO	You only look once

1 Einleitung

KI bzw. Künstliche Intelligenz (engl. *Artificial Intelligence (AI)*) ist ein Thema, das aktuell für viel Aufmerksamkeit in den Medien sorgt. Die damit verbundene Unsicherheit bezüglich der möglichen Verdrängung oder Transformation der bestehenden Arbeitsplätze wird insbesondere in Deutschland umstritten diskutiert. Prominente Wissenschaftler und Unternehmer wie bspw. der erst kürzlich verstorbene britische theoretische Astrophysiker Stephen Hawking [Dav14] oder die US-amerikanischen *High Tech* Gründer Elon Musk (PayPal, Tesla, SpaceX) [Mus17] und Sergey Brin (Alphabet bzw. Google) [Bri18] warnen sogar vor den möglichen Risiken der KI für die Menschheit. Wir sind uns zumindest darüber bewusst, dass die Digitalisierung als globales Phänomen das Arbeits- und Privatleben in den nächsten Jahren massiv verändern wird. Zukunftsthemen wie das Internet der Dinge (engl. *Internet of Things (IoT)*) und Industrie 4.0 sind ohne KI als Motor der digitalen Transformation kaum vorstellbar.

Diese Arbeit beschäftigt sich jedoch nicht mit den gesellschaftlichen, politischen oder ökonomischen Fragen zur Künstlichen Intelligenz. Stattdessen werden im Wesentlichen die Technologien beleuchtet, die z.Zt. einen großen Stellenwert in der KI-Forschung und den KI-Anwendungen einnehmen, nämlich Künstliche Neuronale Netzwerke (KNN) und *Deep Learning (DL)*.

Biologische Neuronale Netzwerke Biologische Nervensysteme arbeiten massiv parallel, sind weitgehend fehlertolerant, verhalten sich adaptiv und als lernende Systeme, die ihre eigenen Parameter bestimmen und anpassen können. Das Wesen der Funktion des Nervensystems besteht in der Kontrolle durch Kommunikation. Das menschliche Gehirn besteht aus etwa 10^{10} bis 10^{12} miteinander vernetzten Nervenzellen, den Neuronen. Etwa 10% der Neuronen dienen der Eingabe und Ausgabe. Die restlichen 90% sind mit anderen Neuronen verknüpft, die Informationen speichern oder bestimmte Umwandlungen des Signals vornehmen, das sich durch das Netzwerk fortpflanzt. Neuronen sind komplexe Zellen, die auf elektrochemische Signale reagieren. Sie setzen sich zusammen aus einem Zellkern, einem Zellkörper, mehreren Dendriten, die über Synapsen Eingabeverknüpfungen zu anderen Neuronen herstellen, sowie einem Axonstrang, der über Endkolben oder Synapsen ein Aktionspotenzial ausgibt (siehe Abb. 1.1). Ein Neuron kann mit hunderten bis tausenden anderen Neuronen verbunden sein. Die Verbindungen erfolgen über zwei allgemeine Synapsentypen: exzitatorische (erregende) und inhibitorische (hemmende). Die neuronale Aktivität wird bestimmt durch die Entstehung eines internen elektrischen Potenzials, dem Membranpotenzial. Dieses Potenzial kann durch die Eingabeaktivitäten seitens anderer Zellen über die Synapsen verstärkt oder abgeschwächt werden. Wenn die kumulativen Eingaben das Potenzial über einen Schwellenwert heben, sendet das Neuron Impulse aus, indem es eine Folge von Aktionspotenzialen über das Axon ausschüttet. Diese Impulse bewirken, dass eine chemische Substanz, der Neurotransmitter, an die Synapsen ausgegeben wird, die wiederum andere Neuronen erregen oder hemmen können. Das Axonsignal ist wegen des Schwellenwerts von Natur aus binär. Die nicht-binären Informationen, die im Nervensystem verarbeitet werden, sind nicht durch die Größe der Spannungen, sondern durch die zeitlichen Abstände des Aktionspotenzials

1 Einleitung

codiert. Das Nervensystem arbeitet demnach mit Frequenzmodulation (FM), also wie bspw. eine UKW-Radiostation. Die Zeit, die ein Reiz zum Durchqueren einer Synapse benötigt, beträgt etwa 1 Millisekunde (ms). Nach dem Feuern entsteht eine unempfindliche Phase, die etwa 10 ms dauert und während derer das Neuron nicht feuern kann. Pro Sekunde können fünfzig bis mehrere hundert Ausschüttungen auftreten. Die Taktfrequenz des biologisch neuronalen Netzwerks liegt damit maximal im unteren kHz-Bereich und ist um mehrere Dimensionen kleiner als die Geschwindigkeit der Prozessoren eines konventionellen Computersystems, welches auf der Von-Neumann-Architektur basiert. Die Leistungen des menschlichen Gehirns beruhen daher in erster Linie auf der hohen Parallelität bei der Informationsverarbeitung. Synapsen können wachsen, verkümmern oder ganz verschwinden. Umgekehrt kann ein Axon neue Zweige mit den zugehörigen Synapsen ausbilden und dadurch mit weiteren Nervenzellen in Kontakt treten. Diese Wachstumsprozesse sind für Gedächtnis und Lernen verantwortlich.

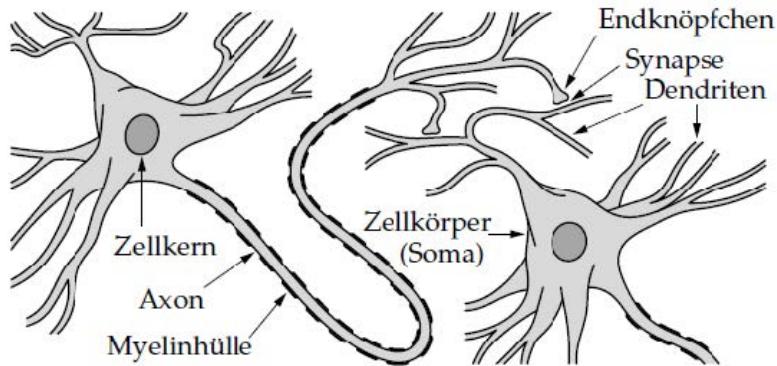


Abbildung 1.1: Darstellung miteinander vernetzter Neuronen [Kru+15]

Geschichtlicher Abriss Die Entwicklung der KNN begann in den 1940er Jahren. Der Neurophysiologe Warren S. McCulloch und der Mathematiker Walter Pitts abstrahierten von den biologischen Vorgängen und schufen 1943 ein Modell des essentiellsten Gehirnbausteins auf Basis logischen Kalküls, das künstliche Neuron [MP43]. Dieses McCulloch-Pitts-Neuron oder Schwellenwert-Neuron genügt fünf Annahmen:

- Ein Neuron ist ein binäres Schaltelement, es ist entweder aktiv oder inaktiv.
- Jedes Neuron besitzt einen festen Schwellenwert.
- Ein Neuron empfängt Eingaben von exzitatorischen Synapsen gleichen Gewichts.
- Ein Neuron empfängt Eingaben von inhibitorischen Synapsen, deren Effekt absolut ist: eine aktive inhibitorische Synapse verhindert die Aktivierung des Neurons.
- Es gibt ein Zeitquantum für die Integration der synaptischen Eingaben. Wenn keine inhibitorische Synapse aktiv ist, werden die exzitatorischen Eingaben addiert und das Neuron wird aktiv, wenn sein Schwellenwert dadurch überschritten wird.

1949 formulierte Donald O. Hebb, ebenfalls Neurophysiologe, ein Modell des menschlichen Lernens, das die Lernvorgänge mit der Anpassung von Neuronenverbindungen bei gleichzeitiger Aktivität miteinander verbundener Neuronen begründet [Heb49]. Das Modell wurde als Hebb'sche Lernregel bekannt.

Die Arbeiten in den 1940er Jahren interessierten Forscher verschiedener Gebiete, darunter den Entdecker des modernen Computermodells, John von Neumann, und den Mitbegründer der Kybernetik, Norbert Wiener. Aufbauend auf der Hebb'schen Lernregel kombinierte der Psychologe Frank Rosenblatt das McCulloch-Pitts-Neuron in einem Netzwerk und

beschrieb 1958 das Perzeptron [Ros58]. Alle grundlegenden Merkmale heutiger KNN sind enthalten: Lernfähigkeit, Selbstorganisation, Generalisierungsfähigkeit und Fehler-toleranz. Durch die richtige Wahl des Schwellenwerts konnten diese Konstrukte logische Operationen realisieren (vgl. Kap. 2.2).

Das erste praktische einsetzbare KNN konstruierten 1960 Bernhard Widrow und Marcian E. Hoff [WH60]. Ihr Adaline (engl. *Adaptive Linear Neuron*) konnte einfache Klassifizierungs-aufgaben ausführen und kam zum Beispiel für die Dämpfung des Echoes in Telefonleitungen zur Anwendung. Wichtigster Schritt für die Entwicklung der KNN war jedoch eine neue Lernregel, die gegenüber dem Perzeptron Vorteile aufwies. Adaline benutzt den kleinsten quadratischen Fehler zwischen gewünschtem und erzeugtem Output als Fehlermaß. Die Widrow-Hoff-Regel ist auch als Delta-Regel bekannt.

1969 erlitt die Erforschung Künstlicher Neuronaler Netzwerke einen Einbruch. Marvin Minsky und Seymour Papert, die zuvor den Begriff Künstliche Intelligenz geprägt haben, veröffentlichten eine vernichtende Kritik an den existierenden Neuronen-Modellen [MP69] mit dem Erfolg, dass zwischen 1970 und 1982 nur noch recht wenige Forscher voneinander getrennt in einzelnen Disziplinen weiterarbeiteten. Dieses Buch war eine elegante mathematische Analyse der Perzeptronen mit ihren Vor- und Nachteilen. Im Wesentlichen wurde dabei gezeigt, welche logischen Funktionen einfache Perzeptronen verarbeiten konnten, welche nicht. Ein Perzeptron ist grundsätzlich nämlich nicht in der Lage, die zur Klasse der nichtlinearen separablen Funktionen gehörende exklusive Oder-Funktion (XOR) zu realisieren.

Dennoch entstanden in den 1970er Jahren richtungsweisende Arbeiten, die aber aufgrund der Kritik Minskys und Paperts zunächst unbeachtet blieben. Der Elektronik-Ingenieur Teuvo Kohonen entwickelte 1972 ein Neuronen-Modell für Steuerungsaufgaben, den Korrelations-Matrix-Speicher [Koh72]. Der Neurophysiologe James A. Anderson entwickelte zeitgleich den linearen Assoziator und das *Brain-State-in-a-Box* Netzwerk [And70; And72] in Anlehnung an natürliche biologische und psychologische Anpassungsvorgänge. Paul J. Werbos legte 1974 in seiner verkannten Dissertation [Wer74] den Grundstein für den heute bekanntesten Lernalgorithmus *Backpropagation*. 1976 entwarfen Stephen Grossberg und Gail Carpenter die *Adaptive Resonance Theory* (ART) auf der Basis, wie das Gehirn Informationen verarbeitet, nämlich als Kombination eines *Top-Down-* und *Bottom-Up*-Ansatzes [Gro76]. Eine Vielzahl von ART-Netzwerken sind seitdem entstanden.

Die Renaissance der KNN begann zwischen 1979 und 1982. Wissenschaftler verschiedener Disziplinen (Biologen, Neurologen, Physiker, Mathematiker) entdeckten ein gemeinsames Interesse an der Forschung nach neuen Neuronen-Modellen. Allerdings entstand erst 1982 ein merklicher Schub für die neuronale Forschung als der renommierte Physiker und Nobelpreisträger John J. Hopfield Neuronen-Modelle für die Erklärung physikalischer Phänomene benutzte [Hop82]. Das Hopfield-Modell führte zur Entwicklung weiterer Netzwerkmodelle, die sich an den physikalischen Energiegesetzen orientierten. Ebenfalls im Jahr 1982 stellte Teuvo Kohonen seinen Ansatz über die selbstorganisierende Bildung topologisch korrekter Merkmalskarten vor [Koh82]. Die Idee Kohonens besteht darin, eine Schicht von Neuronen zu bilden, die auf eine Eingabe mit der Aktivierung einer bestimmten Region reagieren. Ähnliche Eingaben sollen benachbarte Regionen erregen.

Die 1985 von David H. Ackley, Geoffrey E. Hinton und Terence J. Sejnowski vorgestellte Boltzmann-Maschine verwendet Hopfield-Neuronen und hat ein physikalisches Analo-

1 Einleitung

gon: das langsame Abkühlen (*Simulated Annealing (SA)*) einer Kristallschmelze [AES85]. Die Boltzmann-Maschine war das erste KNN, das in der Lage ist, Neuronen innerer Schichten zu trainieren.

Der entscheidende Durchbruch KNN kam 1986 als zeitgleich und unabhängig voneinander David E. Rummelhart, Geoffrey E. Hinton und Ronald J. Williams [RHW86a; RHW86b] sowie David B. Parker [Par85] und Yann LeCun [LeC86] eine neue Lernregel, die generalisierte Delta-Regel, vorstellten, die die Kritik am Perceptron aufhob. Diese heute als *Backpropagation* bekannte Lernregel ist in der Lage, die Verbindungsgewichte zu inneren Einheiten in mehrschichtigen KNN zu bestimmen. Die Idee besteht darin, den Fehler, den das Netz bei der Erzeugung einer Ausgabe macht, rückwärts durch das Netzwerk, also von der Ausgabe- zur Eingabeschicht, weiterzureichen (zu propagieren) und zur Gewichtsveränderung zu verwenden (vgl. Kap. 3.2). Seit Ende der 1980er Jahre werden vollständig vernetzte *Feedforward*-Netzwerke, die sogenannten *Multilayer Perceptron (MLP)*, sehr erfolgreich mit dem *Backpropagation*-Algorithmus trainiert und für zahlreiche Anwendungen eingesetzt.

1990 hat Jeffrey L. Elman dann eine neue Struktur für Rückkopplungen in einem KNN vorgeschlagen, welches als einfaches *Recurrent Neural Network (RNN)* den Namen Elman-Netz bekam [Elm90]. Mit rekurrenten und rekursiven Netzwerken lassen sich besonders gut sequentielle Daten wie Zeitreihen verarbeiten. Sehr ähnliche RNN sind die Jordan-Netze, die nach Michael I. Jordan benannt sind [Jor86; Jor97]. 1997 haben Sepp Hochreiter und Jürgen Schmidhuber den Grundstein für eine bis heute sehr erfolgreich verwendete rekurrente Struktur gelegt und das *Long Short-Term Memory (LSTM)* entwickelt [HS97]. Diese Einheit besitzt eine innere Struktur und besteht aus einer Zelle und mehreren *Gates*, die den Informationsfluss steuern. Ein Jahr später hat erneut Yann LeCun einen sehr wichtigen Beitrag geleistet und das *Convolutional Neural Network (CNN)* für den Anwendungsbereich der Bilderkennung konstruiert [LeC+98].

Seit den 2000er Jahren werden die Netzwerke immer größer bzw. tiefer, um auch komplexe Probleme zu bearbeiten. Beim Training dieser tiefen Netze traten aber unerwartete Probleme auf, bspw. verschwindende Gradienten, durch die das Lernverfahren zum Erliegen kam. Es wurden daher neue Techniken und Netzwerk-Modelle entwickelt, um diesen Problemen entgegenzuwirken, die man unter dem Schlagwort *Deep Learning* zusammenfassen kann. Ein neues, erwähnenswertes KNN wurde 2006 von Geoffrey E. Hinton entwickelt [HOT16]: *Deep Belief Network (DBN)*. In diesem tiefen KNN sind nur die Schichten miteinander verbunden, nicht aber die einzelnen Neuronen. Das DBN kann auch als Konstruktion aus einzelnen RBMs betrachtet werden. Eine *Restricted Boltzmann Machine (RBM)* ist eine Variante der Boltzmann-Maschine (s.o.).

Die 2010er Jahre sind geprägt durch Implementierungen von KNN auf grafischen Prozessoren (GPU). Diese besitzen sehr viele kleine Recheneinheiten, sodass sich das Training gut damit parallelisieren und schnell ausführen lässt. Die bestehenden KNN-Architekturen werden immer weiter verfeinert. Mit der *Gated Recurrent Unit (GRU)* wurde 2014 eine Variante von LSTM vorgestellt, die sehr erfolgreich in der automatischen Textübersetzung eingesetzt wird [Cho+14]. Besondere Aufmerksamkeit erlangte im März 2016 das System AlphaGo von Google DeepMind, welches mit *Deep Learning* trainiert wurde. AlphaGo ist es gelungen, den Südkoreanischen Profispiel Le Sedol unter Turnierbedingungen 4:1 im Brettspiel Go zu schlagen [Wik18a].

Einen sehr guten Überblick über die Geschichte Künstlicher Neuronaler Netzwerke und *Deep Learning* aus verschiedensten Perspektiven bildet die umfassende Arbeit *Deep Learning in Neural Networks: An Overview* von Jürgen Schmidhuber [Sch14].

Ziel und Vorgehen Diese Arbeit hat das Ziel, typische Anwendungen durchzuführen, bei denen Künstliche Neuronale Netzwerke (KNN) bzw. Techniken des *Deep Learning* zum Einsatz kommen, die dann als Basis in der Lehre und der angewandten Forschung an der Hochschule für Technik und Wirtschaft des Saarlandes (htw saar) dienen können. Um dieses Ziel zu erreichen, werden zunächst die theoretischen Grundlagen beschrieben. In Kap. 2 werden die grundlegenden Bausteine des KNN, das künstliche Neuron bzw. das Perzepron sowie einfache Topologien und Modelle vorgestellt. Insbesondere wird dabei auch auf das Lernen und die damit verbundenen Schwierigkeiten sowie erste Lösungsansätze eingegangen. Die unterschiedlichen Kategorien typischer KNN-Anwendungen werden ebenfalls kurz skizziert. Kap. 3 beschäftigt sich dann mit einer speziellen KNN-Architektur, dem sogenannten *Multilayer Perceptron (MLP)*. Dieses Netzwerk ist sehr populär, u.a. wegen des gut verstandenen und gut funktionierenden Lernalgorithmus *Backpropagation*. Wenn allerdings sehr tiefe MLP trainiert werden, so stößt man auf neue Probleme wie bspw. verschwindende Gradienten. Diese Probleme lassen sich mit Hilfe von *Deep Learning* meistern, wobei auch neue KNN-Architekturen entstanden sind, die in den nächsten zwei Kapiteln dargestellt werden. Kap. 4 stellt das *Convolutional Neural Network (CNN)* vor und Kap. 5 beschäftigt sich mit dem *Recurrent Neural Network (RNN)*. Sehr vereinfacht ausgedrückt, können mit CNN Aufgaben der Bilderkennung bearbeitet werden, während RNN vornehmlich in der Spracherkennung verwendet werden. In Kap. 6 werden 22 verschiedene *Open Source* Softwarelösungen zu *Deep Learning* vorgestellt, die aktuell am Markt vorhanden sind. Im nächsten Kapitel werden dann zwei dieser Softwarelösungen (TensorFlow, Keras) verwendet, um drei typische Aufgaben mit Hilfe von KNN zu bearbeiten. Zwei dieser Aufgaben stammen aus dem Bereich Bilderkennung, wobei eine *Data Mining* Klassifikation der auf den Bildern identifizierten Objekte vorgenommen wird. Die dritte Aufgabe fällt in das Gebiet *Text Mining*: Eine *Sentiment Analysis* wird benutzt, um die positiven und negativen Stimmungen in Film-Rezensionen zu erkennen, d.h. letztendlich ist es also ebenfalls eine Klassifikation. Diese Aufgaben lassen sich der Domäne *Big Data* zuordnen, weil digitale Bilder und unstrukturierte Textdaten verarbeitet werden müssen. Im letzten Kapitel wird die Arbeit dann zunächst zusammengefasst und anschließend wird ein Ausblick auf die weitere Entwicklung gegeben.

2 Theoretische Grundlagen

In diesem Kapitel werden zunächst die grundlegenden Bausteine eines Künstlichen Neuronalen Netzwerks (KNN) vorgestellt, zuerst das künstliche Neuron (vgl. Abschnitt 2.1) und dann das Perzepton (vgl. Abschnitt 2.2). Größere Netzwerke bestehen aus vielen Neuronen, die in mehreren Schichten angeordnet sind, und je nach Topologie werden verschiedene Arten unterschieden (vgl. Abschnitt 2.3). Die Lernfähigkeit ist die wichtigste Eigenschaft eines KNN. Anhand der gegebenen Daten wird es trainiert und dabei werden die Netzwerk-Parameter automatisch angepasst (vgl. Abschnitt 2.4). KNN lassen sich für unterschiedliche Aufgabenstellungen verwenden, die nach Kategorien geordnet im Abschnitt 2.5 kurz vorgestellt werden. In Abschnitt 2.6 wird dann ein Bezug zum geschichtlichen Abriss hergestellt (vgl. Kap. 1) und es werden verschiedene Modelle bzw. Architekturen von KNN mit den jeweiligen Lernverfahren und typischen Anwendungsfällen präsentiert. Ein generelles Problem beim Lernen ist die Überanpassung (engl. *Overfitting*). Um dieses Problem zu vermindern, lassen sich spezielle Techniken wie *Early Stopping*, Kreuzvalidierung (*Cross Validation*), Ausdünnung (*Pruning*) und Regularisierung einsetzen, die in Abschnitt 2.7 behandelt werden.

2.1 Künstliches Neuron

Das künstliche Neuron ist die kleinste Struktureinheit eines informationsverarbeitenden Elements des KNN. Es besteht aus mehreren gerichteten Eingabeleitungen (Dendriten), die mit Gewichten (Synapsen) versehen sind, einem Berechnungskörper (Zellkörper) und einer Ausgabeleitung (Axon)¹. In Abb. 2.1 ist die Funktionsweise eines künstlichen Neurons dargestellt.

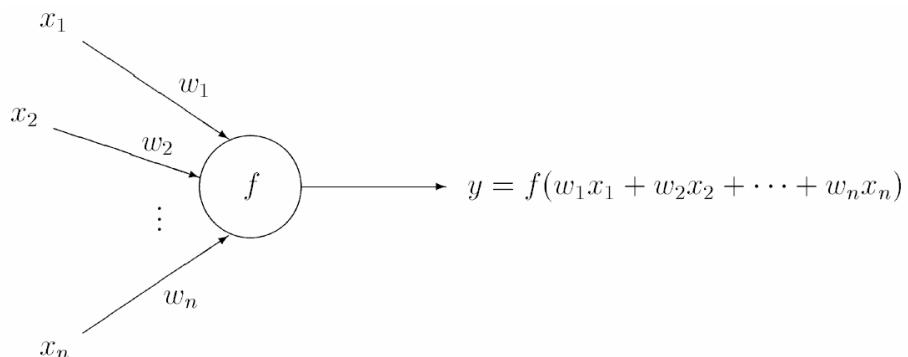


Abbildung 2.1: Struktur eines künstlichen Neurons

Die n Eingabesignale x_i werden mit Gewichtsfaktoren w_i multipliziert und durch eine Propagierungsfunktion $p(x_1, \dots, x_n)$ zu einem Gesamt- bzw. Netto-Eingabesignal zusammengefasst. Eine Aktivierungs- bzw. Transferfunktion $f(p(x_1, \dots, x_n))$ sorgt dann dafür, dass unter Berücksichtigung des Netto-Eingabesignals ein Ausgabesignal y generiert wird.

¹In Klammern sind die entsprechenden Elemente des biologischen Neurons angegeben (vgl. Kap. 1).

2 Theoretische Grundlagen

Das KNN wird dann aus diesen künstlichen Neuronen gebildet, wobei man diese Elemente in Schichten (engl. *Layers*) zusammenfasst. Es gibt eine Eingabeschicht (engl. *Input Layer*) und eine Ausgabeschicht (engl. *Output Layer*). Die Neuronen der Eingabeschicht bekommen als Eingabesignale externen Input und deren Ausgabesignale werden an die Neuronen der nächsten Schicht als Eingaben weitergeleitet. Die Neuronen der Ausgabeschicht bekommen als Eingabesignale die Ausgaben der vorgelagerten Schicht und generieren als Ausgabesignal externen Output. Häufig gibt es in KNN diese inneren, verborgenen Schichten (engl. *Hidden Layers*). Die Neuronen in diesen Schichten haben keine direkte Verbindung mit der Außenwelt, d.h. sie sind nicht mit dem externen Input oder mit dem externen Output verbunden.

Als Propagierungsfunktion wird gewöhnlich für alle Neuronen des KNN dieselbe Funktion benutzt, und zwar die gewichtete Summe aller Eingaben

$$p(x_1, \dots, x_n) = \sum_{i=1}^n w_i x_i . \quad (2.1)$$

Für die Neuronen in einer Schicht wird normalerweise dieselbe Aktivierungsfunktion verwendet. Die Wahl dieser Aktivierungsfunktion ist allerdings sehr stark modellabhängig (vgl. Kap. 3.4).

2.2 Perzeptron

Das Perzeptron (engl. *Perceptron*) ist ein sehr einfaches KNN, das lediglich aus einer einzigen Verarbeitungseinheit besteht. Der Name ist vom englischen Begriff *Perception* abgeleitet und bedeutet Wahrnehmung. Das Perzeptron besteht aus n Eingaben x_i , die jeweils mit den Gewichten w_i behaftet sind. Das Perzeptron kann nur Schwellenwertentscheidungen treffen, d.h. die Ausgabe der Einheit ist 1, falls $w_1 x_1 + w_2 x_2 + \dots + w_n x_n \geq \Theta$ gilt, wobei Θ der sogenannte Schwellenwert (engl. *Bias*) der Einheit ist. Falls $w_1 x_1 + w_2 x_2 + \dots + w_n x_n < \Theta$ gilt, wird null ausgegeben. In Abb. 2.2 ist der Aufbau des Perzeptron schematisch dargestellt.

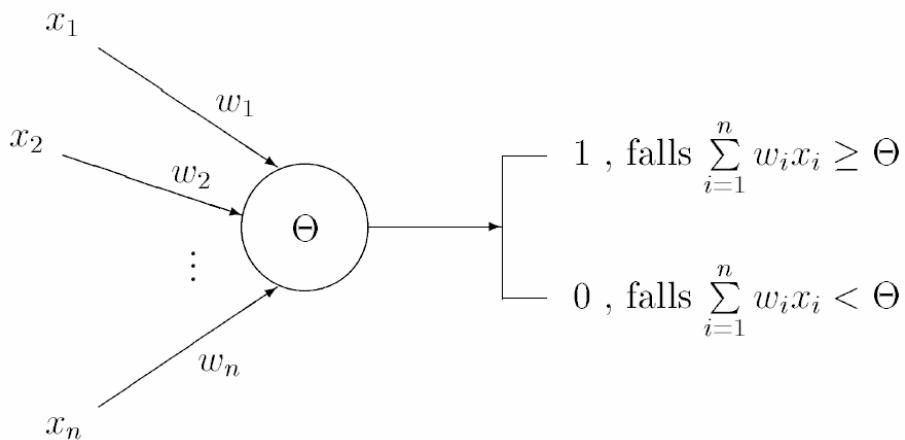


Abbildung 2.2: Struktur eines Perzeptrons

Mit Perzeptronen können logische Funktionen wie UND (engl. *AND*) bzw. ODER (engl. *OR*) bzw. NICHT (engl. *NOT*) einfach realisiert werden (siehe Abb. 2.3).

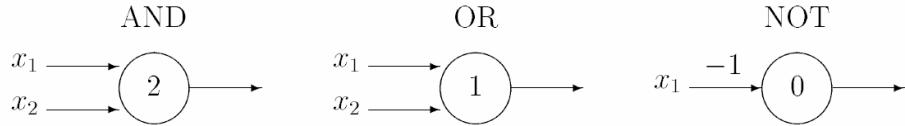


Abbildung 2.3: Darstellung logischer Schaltungen mittels Perzeprontronen

Die AND-Verknüpfung zweier binärer Variablen x_1 und x_2 kann durch ein Perzeptron mit zwei Eingabeleitungen implementiert werden, bei der w_1 und w_2 beide gleich 1 sind und $\Theta = 2$ gilt. Das Perzeptron wird nur dann eine 1 feuern, wenn $x_1 + x_2 \geq 2$ erfüllt ist, d.h. bei binären Eingaben genau dann, wenn $x_1 = x_2 = 1$ gilt. Bei der OR-Verknüpfung wird entsprechend $\Theta = 1$ gesetzt. Es genügt also, wenn $x_1 = 1$ oder $x_2 = 1$ gilt. Zur Darstellung der NOT-Verknüpfung genügt eine Eingabeleitung x_1 , die mit dem Gewicht $w_1 = -1$ versehen wird. Der Schwellenwert ist hier null. Die logische Funktion XOR, also das exklusive ODER, lässt sich jedoch nicht mit einem einzelnen Perzeptron realisieren [MP69].

2.3 Topologie

Künstliche Neuronale Netzwerke lassen sich hinsichtlich ihrer Topologie klassifizieren:

- **Vorwärtsgekoppelte Netzwerke** (engl. *Feedforward Networks*) sind Netzwerke, in denen die Verbindungen nur in eine Richtung gehen, von der Eingabe zur Ausgabe.
- **Rekurrente Netzwerke** (engl. *Recurrent Networks*) sind Netzwerke, in denen es auch Verbindungen in Rückwärtsrichtung gibt, sodass Rückkopplungen entstehen.
- **Zyklische Netzwerke** sind Netzwerke, in denen sich einige Neuronen gegenseitig reizen. Auch hier gibt es also Rückkopplungen.
- **Vollkommen verbundene Netzwerke** sind Netzwerke, in denen jedes Neuron mit allen anderen Neuronen verbunden ist (meistens bis auf sich selbst).
- **Geschichtete Netzwerke** sind Netzwerke, in denen die Neuronen in getrennten, nicht verbundenen Mengen angeordnet sind.
- **Symmetrische Netzwerke** sind Netzwerke, in denen die Verbindung zwischen zwei beliebigen Neuronen in beiden Richtungen gleich ist.
- **Selbstassoziative Netzwerke** sind Netzwerke, in denen Eingabe- und Ausgabe-Neuronen übereinstimmen.
- **Stochastische Netzwerke** sind Netzwerke, in denen eine gewisse Wahrscheinlichkeit besteht, dass ein Neuron nicht aktiviert wird, obwohl es Reize bekommen hat.
- **Asynchrone Netzwerke** sind Netzwerke, in denen die Neuronen nicht alle auf einmal (synchron), sondern zufällig eins nach dem anderen aktiviert werden.

2.4 Lernen

Ein wesentliches Merkmal von Künstlichen Neuronalen Netzwerken ist die allgemeine Lernfähigkeit, d.h. KNN lernen aus Erfahrungen, die sie durch präsentierte Trainingsdaten gewinnen. Lernen heißt Selbstanpassung der Parameter, also der Gewichtungsfaktoren zwischen den Neuronen des KNN, sodass das Netzwerk das gewünschte Verhalten

2 Theoretische Grundlagen

zeigt. Die Schwellenwerte (engl. *Bias*) der Neuronen sind ebenfalls anpassbare Parameter und können als zusätzliche Gewichtungsfaktoren interpretiert werden. Lernmethoden für KNN können in drei grundsätzliche Kategorien eingeteilt werden: überwacht, bestärkt oder nicht-überwacht.

Beim **überwachten Lernen** (engl. *Supervised Learning*) ist während des Lernprozesses ein Lehrer anwesend, und jedes Beispilmuster für das Training des Netzwerks beinhaltet ein Eingabemuster sowie ein Ziel oder ein gewünschtes Ausgabemuster, d.h. die korrekte Antwort. Während des Lernprozesses kann ein Vergleich zwischen der vom Netzwerk berechneten und der korrekten Ausgabe angestellt werden, um den Fehler festzustellen. Der Fehler kann dann verwendet werden, um die Netzparameter (Gewichtungsfaktoren) entsprechend anzupassen, sodass der Netzwerk-Fehler reduziert wird. Nachdem die Gewichtungen für alle Trainingsmuster iterativ angepasst wurden, konvergieren die Gewichtungswerte gegen eine Wertemenge, mit der die erforderlichen Mustervorgaben erzielt werden können. Das Lernen wurde dann erreicht, wenn die Fehler für alle Trainingsmuster auf einen akzeptablen Wert für neue Muster, die sich nicht in der Trainingsmenge befinden, minimiert wurde. Überwachtes Lernen wird auch induktives oder assoziatives Lernen genannt.

Beim **bestärkenden Lernen** (engl. *Reinforcement Learning*) wird ebenfalls die Anwesenheit eines Lehrers vorausgesetzt, aber dem Netzwerk wird die korrekte Antwort nicht präsentiert. Stattdessen erhält es nur einen Hinweis darauf, ob die berechnete Antwort richtig oder falsch ist. Anhand dieser Information muss das Netzwerk seine *Performance* verbessern. Normalerweise erhält es eine Belohnung, indem die Gewichtungen für Einheiten, die die richtige Antwort erzeugt haben, erhöht werden, während für die Einheiten mit der falschen Antwort die Gewichtungswerte reduziert werden.

Beim **nicht-überwachten Lernen** (engl. *Unsupervised Learning*) bzw. entdeckendem Lernen erhält das Netzwerk kein Feedback über die gewünschte oder die korrekte Ausgabe. Es gibt keinen Lehrer, der Zielmuster präsentiert. Deshalb muss das System durch Entdecken und die Übernahme strukturierter Eigenschaften der Eingabemuster lernen, d.h. durch die Anpassung an statistische Gleichmäßigkeiten oder Muster-Cluster aus den Eingabebeispielen der Trainingsdaten. Dieses Lernen kann durch die Verstärkung ausgewählter Gewichtungen bewerkstelligt werden, sodass eine Übereinstimmung mit zentralen, prototypischen Trainingsmustern erzielt wird, welche eine repräsentative Gruppe ähnlicher Muster oder Cluster darstellen.

Im Folgenden werden einige Methoden beschrieben, die beim überwachten und nicht-überwachten Lernen zum Einsatz kommen. Das **Hebbsche Lernen** stellt eine Form der korrelativen Gleichgewichtsanpassung dar. Die dafür grundlegende Theorie wurde von Donald Hebb vorgestellt [Heb49]. Wenn ein Axon der Zelle A nah genug an einer Zelle B liegt, um diese zu erregen, und wiederholt oder andauernd feuert, erfolgt in einer oder beiden Zellen ein Wachstumsprozess oder eine metabolische Veränderung, so dass sich die Einflusseffizienz von A auf B erhöht. Für diese Hebbsche Lernregel wurden zahlreiche Varianten vorgeschlagen, unter anderem eine Gewichtsanpassung basierend auf der Minimierung einer Energie- oder Entropiefunktion.

Beim **konkurrierenden Lernen** werden die Gewichtungen so angepasst, dass Neuronen, die am stärksten auf einen Eingabereiz reagieren, bevorzugt werden. Die Gewichtsanpassung ist normalerweise eine modifizierte Form der Hebbschen Anpassung. Die Neuro-

nen in dieser sogenannten Wettbewerbsschicht konkurrieren. Das Neuron, das die größte Übereinstimmung seiner Gewichtsinformation mit dem Eingabemuster feststellt, gewinnt (Siegerneuron, engl. *Winner-Takes-All*), alle anderen verlieren.

Das **Stochastische Lernen** verwendet Wahrscheinlichkeitsfunktionen, um die Anpassung der Gewichtungen vorzunehmen. Das *Simulated Annealing (SA)* gehört zu dieser Klasse von Lernmethoden. Genau genommen ist es kein Lernalgorithmus, sondern ein heuristisches Approximationsverfahren. Aufgrund der hohen Komplexität des Optimierungsproblems können nicht alle Möglichkeiten ausprobiert werden, stattdessen wird eine Näherung benutzt. Die Grundidee stammt vom physikalisch-chemischen Abkühlungsprozess in der Metallurgie. Durch das langsame und kontrollierte Abkühlen werden die Atome einen neuen, geordneten Zustand überführt und bilden stabile Kristalle. Dieser energiearme Zustand liegt nahe am globalen Minimum.

Mehrere Lernparadigmen basieren auf der Reduzierung des Netzwerk-Fehlers mit Hilfe der **Methoden des steilsten Gradienten**. Diese Methoden machen es erforderlich, dass die Aktivierungsfunktion stetig differenzierbar ist. Ausgehend vom aktuellen Fehler wird die Richtung ermittelt, in der sich der Fehler am schnellsten verringert. Damit entspricht der Lernalgorithmus dem Suchen des globalen Minimums in einem nichtlinearen Gleichungssystem. Abb. 2.4 zeigt das Prinzip des steilsten Gradientenabstiegs im zweidimensionalen Fehlerraum.

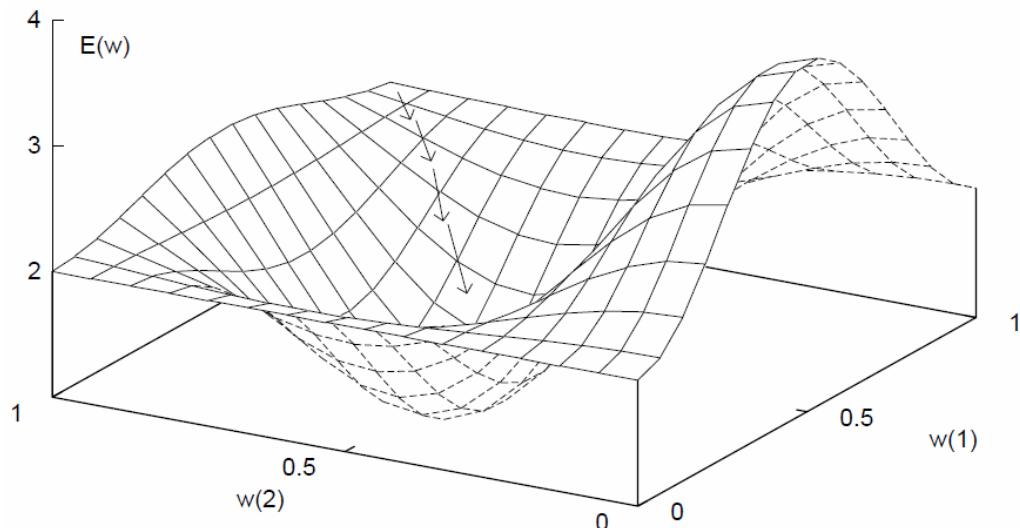


Abbildung 2.4: Gradientenabstiegsverfahren im zweidimensionalen Fehlerraum

Die Probleme, die bei dieser Lernmethode auftreten können, sind Oszillationen der Fehlerfunktion, eine hohe Anzahl von Iterationsschritten des Algorithmus aufgrund von Plateaus innerhalb der Fehlerfunktion, Abbruch des Algorithmus in einem lokalen Minimum der Fehlerfunktion und das Verlassen des globalen Minimums der Fehlerfunktion (siehe Abb.2.5).

In Kap. 3.3 werden Möglichkeiten beschrieben, diese typischen Probleme des Gradientenabstiegverfahrens zu verringern bzw. zu vermeiden.

2 Theoretische Grundlagen

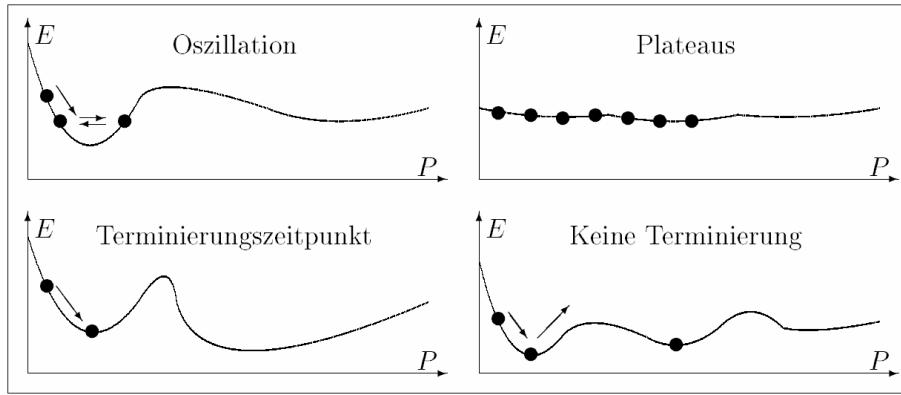


Abbildung 2.5: Typische Probleme des Gradientenabstiegsverfahrens [Fü96]

2.5 Anwendungskategorien

Ein KNN kann als *Black Box* Modell betrachtet werden (siehe Abb. 2.6). Es bekommt Daten als Eingaben (*Input*) und generiert Ausgaben (*Output*). Dabei bleibt die innere Struktur verborgen und die Parameter des KNN werden so angepasst, dass ein Verhalten zwischen Input und Output gelernt und abgebildet wird, ohne die eigentlichen kausalen Zusammenhänge, d.h. Ursache-Wirkungs-Beziehungen, zu kennen. Das KNN ist also ein Modell zur Approximation beliebiger, funktionaler Zusammenhänge.

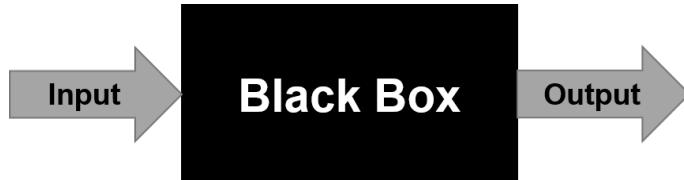


Abbildung 2.6: Black Box Modell

Ein altes Problem der Approximationstheorie besteht darin, eine vorgegebene Funktion durch die Komposition primitiver Funktionen exakt oder annähernd auszudrücken. Ein klassisches Beispiel ist die Approximation von eindimensionalen Funktionen mittels Polynomen. Die Taylorsche Reihe für eine Funktion $F(x)$, die am Punkt x_0 approximiert wird, hat die Form

$$F(x) = \sum_{i=1}^n w_i (x - x_0)^i , \quad (2.2)$$

wobei die Konstanten w_i vom Wert der Funktion F und ihrer Ableitung am Punkt x_0 abhängen. Für $n \rightarrow \infty$ wird der Funktionswert exakt durch die Taylorsche Reihe ausgedrückt.

Abb. 2.7 zeigt, wie die polynomiale Approximation einer Funktion $F(x)$ als Funktionsnetz dargestellt werden kann. An den Knoten des Netzes werden die Funktionen $z \rightarrow z^0, z \rightarrow z^1, \dots, z \rightarrow z^n$, also die Summanden der Taylorschen Reihe, berechnet. Der Ausgabeknoten sammelt additiv die ankommenden Informationen und gibt den Wert des ausgewerteten Polynoms aus. Die Netzgewichte w_i können mit Hilfe eines Lernalgorithmus bestimmt werden.

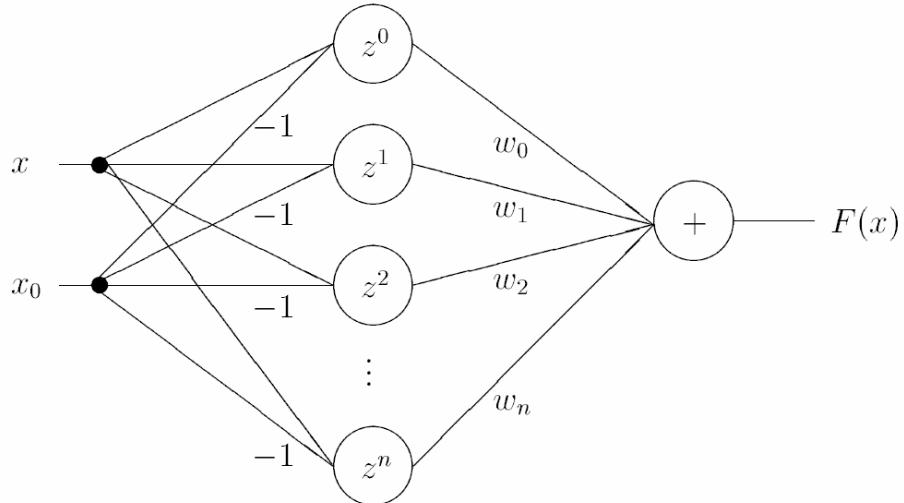


Abbildung 2.7: Taylorsche Reihe als Funktionennetz

Nach dem Theorem von Andrej N. Kolmogorov [Kol57; HN87b] ist es möglich, mehrdimensionale stetige Funktionen als ein endliches Netz primitiver eindimensionaler Funktionen darzustellen. Die Bestimmung der Netzparameter in einem KNN ist normalerweise ein NP-vollständiges Problem². Mit Hilfe des Erfüllbarkeitsproblems kann jedoch gezeigt werden, dass sich ein NP-vollständiges Problem in polynomieller Zeit auf ein Lernproblem für KNN reduzieren lässt [GJ79].

Künstliche Neuronale Netzwerke werden häufig im Bereich *Data Mining* eingesetzt. Nach William J. Frawley versteht man unter *Data Mining* [FPS91]:

« die Extraktion und Entdeckung von implizitem, bisher nicht bekanntem und potenziell nützlichem Wissen aus Daten. »

In diesem Zusammenhang wird auch oft der Begriff Wissensentdeckung in Datenbanken (engl. *Knowledge Discovery in Databases (KDD)*) benutzt, den Usama M. Fayyad folgendermaßen definiert [Fay+96]:

« KDD ist der nicht-triviale Prozess der Identifizierung valider, neuer, potenziell nützlicher und schließlich verständlicher Muster in Daten. »

Daraus lässt sich die folgende Aussage ableiten [Fay+96]:

« Data Mining ist ein Schritt im KDD-Prozess; es ist die Anwendung spezifischer Algorithmen zur Extraktion von Mustern aus Daten. »

Allgemein lassen sich die Aufgaben bzw. Anwendungen im Bereich *Data Mining* in vier Kategorien einteilen (siehe Abb. 2.8). Bei **Prognosen** besteht die Aufgabe darin, einen Ausschnitt aus der Wirklichkeit zu modellieren, um Vorhersagen, d.h. Prognosen über zukünftige Entwicklungen in diesem Ausschnitt treffen zu können. Prognoseanwendungen von KNN basieren typischerweise auf Zeitreihenanalysen, d.h. einer Zahlenfolge liegt ein funktionaler Zusammenhang zugrunde, der in einem mathematischen Modell abgebildet wird. Es müssen dabei keine expliziten Annahmen über die Zusammenhänge in den Daten gemacht werden (*Black Box Modell*). Ein KNN zur Vorhersage zukünftiger Entwicklungen wird mit Gegenwarts- und Vergangenheitsdaten trainiert (Ex-Post-Analyse).

²NP steht für nichtdeterministisch polynomiale Komplexität. Das bedeutet, dass kein Algorithmus bekannt ist, der alle Instanzen des Problems in polynomieller Komplexität lösen kann.

2 Theoretische Grundlagen

Es lernt also, wie sich Zustände der Vergangenheit entwickelt haben und überträgt diese Entwicklung auf die Gegenwart oder Zukunft. Prinzipiell sind alle KNN, die mittels Assoziationen lernen, für diese Aufgaben anwendbar.

	Prognosen	Assoziation	Segmentierung	Klassifikation
Visualisierung				
Beschreibung	Identifikation von Trends im Datenzeitbezug	Suche nach Abhängigkeiten zwischen den Objekten	Erschaffen einheitlicher, homogener Objektteilmengen	Aufteilung der Objekte in vordefinierte Klassen
Beispiel	Vorhersage in Verkauf und Umsatz (→ Absatz-/Produktionsplanung)	Analyse von Shopping-Warenkörben (→ Produkt-Empfehlungen)	Erstellen eines Kunden-Portfolios (→ Differenziertes Marketing)	Churn-Analyse (→ Kundenbindungsmaßnahmen)

Abbildung 2.8: Data Mining Kategorien

Prognose und Klassifikation sind eng miteinander verknüpft. Das Ziel einer **Klassifikation** ist letztendlich ebenfalls eine Prognose. Der wesentliche Unterschied zwischen Prognose- und Klassifikationsanwendungen liegt in der Art der Eingabe- und Ausgabedaten. Im Gegensatz zu Prognosen bestehen die Eingaben bei Klassifikationen aus statischen, zeitpunktbezogenen Daten (z.B. abgeschlossenen Kreditfällen). Der Prognosewert einer Klassifikation entspricht daher auch nicht einer Zeitreihenfortsetzung. Prognosen durch Klassifikation beruhen auf der Annahme, dass die vergangenheitsbezogenen Daten typische Muster enthalten, die erstens eine Klassifizierung mittels einer Trennfunktion erlauben und zweitens, dass gefundene Trennfunktionen auch für künftige, unbekannte Muster gültig sind. Die Ausgabedaten bei der Prognose sind normalerweise kontinuierliche Werte, die kardinalskaliert sind, während bei der Klassifikation diskrete Werte, die nominalskaliert sind, betrachtet werden. Im Beispiel Kreditvergabe wird das KNN mit den Daten, der bereits abgeschlossenen Kreditfälle, trainiert. Die Klassen könnten also sein: Kredit wurde zurückgezahlt oder nicht, d.h. im ersten Fall gab es keine Ausfälle und im zweiten Fall hat der Kunde die vereinbarten Raten nicht pünktlich oder sogar gar nicht zurückzahlen können. Man spricht hier auch von einer binären Klassifikation, weil es nur zwei unterscheidbare Klassen gibt. Das KNN wird nun mit den verfügbaren Daten der Kunden trainiert und hierzu können Merkmale wie bspw. Alter, monatliches Einkommen, Geschlecht, Familienstatus usw. verwendet werden. Möchte nun ein neuer Kunde einen Kreditvertrag abschließen, dann fließen seine Daten in das trainierte Modell ein und eine der beiden Klassen wird bestimmt. Im ersten Fall gilt der Kunde als kreditwürdig und bekommt diesen, während im zweiten Fall kein Kredit gewährt wird. Die Anwendung des Klassifizierers ist also eigentlich eine Prognose in die Zukunft. Die Methoden der Prognose und Klassifikation werden also auch im Bereich *Predictive Analytics* eingesetzt. Dieses einfache Beispiel zeigt auch bereits die ethischen Probleme, die mit dem Einsatz der maschinellen Lernens verbunden sind. Ggf. werden einzelne Personen benachteiligt, weil sie die falschen Merkmale aufweisen.

Klassifikation und **Segmentierung** sind auf dem ersten Blick ähnliche Anwendungen, auf dem zweiten Blick gibt es aber deutliche Unterschiede. Die Klassifikation gehört zum überwachten Lernen und die Segmentation zum nicht-überwachten Lernen (vgl. Kap. 2.4). Während bei der Klassifikation alle Trainingsdaten bereits in Klassen eingeteilt sind, muss das Segmentierungsverfahren diese Trennung selbst vornehmen. Dabei wird nach Ähnlichkeiten als Muster in den Daten gesucht. Ähnliche Datensätze bzw. Objekte werden in möglichst homogene Teilmengen (Segmente, Gruppen, Cluster) eingesortiert. Diese Teilmengen sollten aber wiederum sehr verschieden voneinander sein, d.h. zwischen den Teilmengen sollte eine möglichst große Heterogenität vorliegen. Spezielle Typen von KNN, wie bspw. das Kohonen-Netz, sind geeignet, um mit dem nicht-überwachten Lernen diese Segmentierung automatisch vorzunehmen.

In die vierte Kategorie fallen Anwendungen, in denen nach **Assoziationen** zwischen den betrachteten Objekten gesucht werden. Damit sind Zusammenhänge, meistens (lineare) Korrelationen, zwischen dem gemeinsamen häufigen Auftreten von zwei oder mehreren Ereignissen gemeint. Ein Beispiel ist die Warenkorbanalyse, in der untersucht wird, welche Produkte häufig zusammen gekauft werden, um daraus Regeln abzuleiten, die dann für Produktempfehlungen verwendet werden können: z.B. "Kunden, die dieses Produkt gekauft haben, kauften auch ..." Einige KNN lassen sich als sogenannte Assoziativspeicher einsetzen. Assoziationen sind gelernte Verbindungen zwischen den Eingaben und Ausgaben. Das menschliche Gedächtnis funktioniert sehr ähnlich und arbeitet ebenfalls mit Assoziationen, die als Erinnerungen an bestimmte Erlebnisse gespeichert werden. Durch ähnliche Eingaben werden Assoziationen geweckt und das gespeicherte Muster mit den größten Ähnlichkeiten als Ausgabe präsentiert.

2.6 Modelle

Ein vollständiger Überblick zu allen existierenden Modellen KNN ist angesichts der unüberschaubaren Anzahl bekannter, modifizierter und vollkommen neuer Typen kaum mehr möglich. Im Folgenden werden daher einige bekannte Modelle kurz vorgestellt. Dabei wird auch auf die Lernalgorithmen und Einsatzgebiete eingegangen.

Perzepron Das Perzepron wurde bereits in Kap. 2.2 behandelt. Die Lernregel eines Perzeprons lässt sich folgendermaßen beschreiben:

- Wenn die Ausgabe eins (aktiv) ist und eins sein soll oder wenn sie null (inaktiv) ist und null sein soll, dann werden die Gewichtungen nicht verändert.
- Wenn die Ausgabe null ist, aber eins sein sollte, werden die Gewichtungen für alle aktiven Eingabeverknüpfungen erhöht.
- Wenn die Ausgabe eins ist, aber null sein sollte, werden die Gewichtungen für alle aktiven Eingabeverknüpfungen verringert.

Mathematisch lässt sich der Perzepron-Lernalgorithmus als Änderung der Gewichte

$$\Delta w_i = \eta(t - y)x_i \quad (2.3)$$

schreiben, wobei x_i die i -te Eingabe, y die tatsächlich realisierte Ausgabe und t die gewünschte bzw. erwartete Ausgabe (engl. *Teaching Input*) sind. Der Parameter η ist der Lerngeschwindigkeitskoeffizient bzw. die Lernrate. Wenn η sehr klein ist, erfolgt das Lernen zwar langsam, aber stabil.

2 Theoretische Grundlagen

Adaline Das Adaline (engl. *Adaptive Linear Neuron*) ist ein einzelnes Neuron mit Schwellenwertlogik und einer bipolaren Ausgabe mit den Werten $\{+1, -1\}$ [WH60]. Eingaben für diese Einheit erfolgen normalerweise auch bipolar. Die Gewichtungen werden nach der Lernregel von Widrow-Hoff, die auf der Minimierung des Netzwerkfehlers beruht, angepasst. Der Gesamtfehler des Netzwerks ist gerade die Summe der Einzelfehler

$$E_{\text{tot}} = \sum_{p=1}^P E^p \quad , \quad (2.4)$$

wobei P die Anzahl der Trainingsmuster (engl. *Pattern*) darstellt. Der Einzelfehler ergibt sich aus dem Fehlerquadrat

$$E^p = \frac{1}{2}(t^p - y^p)^2 \quad . \quad (2.5)$$

Die Ausgabe ist gerade die gewichtete Summe der n Eingaben abzüglich des Schwellenwertes Θ , also

$$y^p = \left(\sum_{i=1}^n w_i x_i^p \right) - \Theta \quad . \quad (2.6)$$

Die Gewichtsänderung ist proportional zum Gradienten des Gesamtfehlers, also

$$\Delta w_i = -\eta \frac{\partial E_{\text{tot}}}{\partial w_i} = \eta \sum_{p=1}^P (t^p - y^p) x_i^p \quad . \quad (2.7)$$

Die Proportionalitätskonstante η kann hier ebenfalls als Lerngeschwindigkeitskoeffizient interpretiert werden. Für η wird oft der Wert $1/n$ gewählt. Der Ausdruck $(t^p - y^p)$ wird häufig mit δ^p bezeichnet und deshalb trägt der Lernalgorithmus auch den Namen Delta-Regel. Die Delta-Regel konvergiert, der Fehler behält allerdings einen Wert grösser null bei. Adaline-Netzwerke werden zur Mustererkennung und als Assoziativspeicher benutzt. Ein gegebenes KNN mit n Eingabeeinheiten kann unter Verwendung der Delta-Regel maximal n linear unabhängige Muster fehlerfrei speichern.

Madaline Durch die Kombination mehrerer Adaline in einem Netzwerk erhält man ein Madaline-Netzwerk (engl. *Multiple Adaline*) [WL90]. Die Lernregel basiert auf dem Prinzip der minimalen Störung. Dabei werden die Eingaben jedes Elements gestört, indem die Eingabe um einen kleinen Betrag Δs verändert wird und die Änderung in der quadratischen Fehlersumme der Ausgabe beobachtet wird. Man erhält so

$$\Delta w_i = \eta \sum_{p=1}^P (t^p - y^p) x_i^p \frac{\Delta f}{\Delta s_i^p} \quad , \quad (2.8)$$

wobei Δf die Änderung der Ausgabe (Aktivierungsfunktion) ist. Madaline-Netzwerke erlauben die Realisierung von Abbildungen mit nichtlinearer Teilbarkeit, wie z.B. die XOR-Funktion.

Hopfield-Netz Hopfield-Netzwerke [Hop82] sind einschichtige, rekursive Netzwerke mit symmetrischen Gewichtungsmatrizen, d.h. es gilt $w_{ij} = w_{ji}$ für alle $i, j = 1, 2, \dots, n$. Hopfield-Netze speichern eine Anzahl P von Prototypenmustern, die sogenannten Fixpunkt-Attraktoren. Die gespeicherten Muster können durch eine direkte Berechnung spezifiziert werden, wie etwa mit der Hebb'schen Lernregel

$$\Delta w_{ij} = \eta x_i y_j \quad , \quad (2.9)$$

oder sie werden anhand eines Aktualisierungsschemas wie der Delta-Regel erlernt. Nachdem ein Netzwerk P Prototyp-Muster gelernt hat, können diese zum assoziativen Wiederfinden herangezogen werden. Um ein spezielles Muster zu finden, arbeitet das Netzwerk rekursiv, indem die Ausgabesignale des Netzwerks wiederholt in die Eingaben einfließen, und zwar zu jedem Aktualisierungszeitpunkt t , bis das Netzwerk sich schließlich stabilisiert. Die Lösung kann zyklisch sein mit fester Periodendauer T . Damit das Netzwerk als Assoziativspeicher dient, muss die Lösung jedoch in endlicher Zeitdauer gegen einen Fixpunkt $\bar{x}(t+1) = \bar{x}(t)$ konvergieren. Der Status des Netzwerks lässt sich durch die Energiefunktion

$$E = -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_j x_j \quad (2.10)$$

beschreiben. Hopfield konnte zeigen, dass die Energie irgendwann einen stabilen Zustand erreichen muss, wenn sich das KNN gemäß seiner Dynamik entwickelt, weil die definierte Energiefunktion E nicht nach jeder Aktualisierung anwachsen kann. Sie muss kleiner werden oder zumindest gleichbleiben. Weil es eine endliche Anzahl Systemzustände gibt, muss das Netzwerk irgendwann gegen ein lokales Minimum konvergieren. Den Energeminima entsprechen Fixpunkt-Attraktoren, den gespeicherten Mustern. Der Status, den das System bei Konvergenz annimmt, bestimmt das Ausgabemuster. Hopfield-Netze wurden hauptsächlich in Optimierungsanwendungen eingesetzt, wie etwa beim NP-Problem des Handlungsreisenden (engl. *Travelling Salesman Problem (TSP)*), bei der Erstellung von Zeitplänen oder bei der Funktionsoptimierung.

Bidirektonaler assoziativer Speicher Der bidirektionale assoziative Speicher (engl. *Bi-directional Associative Memory (BAM)*) ist einerseits ein generalisiertes, heteroassoziatives arbeitendes Hopfield-Netzwerk, das andererseits auch Ähnlichkeiten mit dem ART-Modell hat [Kos87]. Es besteht aus zwei Schichten, wobei die Neuronen nur binäre Zustände annehmen können. Die beiden Schichten sind in beide Richtungen vollständig miteinander verbunden, wobei die Gewichte symmetrisch sind. Das Lernziel eines BAM ist es, die Eingabevektoren der Dimension n mit m -dimensionalen Ausgabevektoren innerhalb der Gewichtungsmatrix adäquat abzubilden. Der Lernschritt innerhalb eines BAM-Netzwerkes ähnelt dem des Hopfield-Netzes, d.h. es kann nach der Regel von Hebb oder der Delta-Regel trainiert werden. Das BAM lässt sich gut zur Musterergänzung einsetzen, wobei in jede der beiden Schichten ein Teilmuster eingegeben wird.

Brain-State-in-a-Box Das *Brain-State-in-a-Box (BSB)* ist ein rekursives, autoassoziatives Netzwerk, wobei der Systemzustand innerhalb des quadratischen, schachtelähnlichen Hyper-Bereichs, der von den Kanten $[+1, -1]$ begrenzt ist, gefangen ist [And72]. Das Netzwerk besteht aus einer einzelnen Schicht von n Einheiten. Es ist in der Struktur und Arbeitsweise ähnlich den Hopfield-Netzen. Die Einheiten dürfen Eigen-Feedback-Verbindungen $w_{ii} \neq 0$ haben, und einige Gewichtungsverbindungen dürfen weggelassen werden, d.h. für einige (i, j) darf gelten $w_{ij} = 0$. Für die Anpassung der Gewichtungen in einem BSB-Netzwerk kann das Hebb'sche oder das Delta-Regel-Lernen angewendet werden. Wie auch bei den Hopfield-Netzen werden BSB-Netzwerke zur Mustererkennung und Optimierung eingesetzt.

Die Boltzmann-Maschine Die Boltzmann-Maschine [AES85] ist ein rekursives, stochastisches Netzwerk. Die Statuszustände, die das Netzwerk annehmen kann, werden durch

2 Theoretische Grundlagen

die Boltzmann-Verteilung bestimmt, einer exponentiellen Form der Wahrscheinlichkeitsverteilung, die für die Modellierung der Statuszustände eines physikalischen Systems bei thermischem Gleichgewicht genutzt wird. Wie das Hopfield-Netzwerk hat eine Boltzmann-Maschine eine symmetrische Gewichtungsmatrix, die Konvergenz gegen einen stabilen Status garantiert. Anders als das Hopfield-Netzwerk kann die Boltzmann-Maschine jedoch verborgene Einheiten haben. Während des Arbeitens und des Trainings der Boltzmann-Maschine wird eine Lernregel benutzt, die man als kontrolliertes Abkühlen (engl. *Simulated Annealing*) bezeichnet. Diese Methode ermöglicht dem Netzwerk lokale Minima zu verlassen und gegen einen globalen Gleichgewichtszustand zu konvergieren. Die Boltzmann-Maschine kann als Assoziativspeicher oder zur Lösung von Optimierungsproblemen verwendet werden.

Selbstorganisierende Karten Selbstorganisierenden (sensorischen) Karten (engl. *Self-Organizing Map (SOM)*) werden auch nach ihrem Entwickler Kohonen-Netze genannt [Koh82]. Es sind einschichtige, rückkopplungsfreie Netzwerke, die i.a. bis zu zwei Dimensionen aufweisen, wobei jeder Eingang mit allen Elementen verbunden ist. Sie arbeiten auf der Basis des konkurrierenden, nicht-überwachten Lernens (vgl. Kap. 2.4). Eine topologische Struktur wird auf der Wettbewerbsschicht (Kohonen-Schicht) definiert, d.h. Neuronen der Wettbewerbsschicht werden in Form eines Hyperquaders angeordnet. Das Ziel ist es, selbstorganisierende Karten so zu trainieren, dass benachbarte Cluster durch benachbarte Neuronen der Wettbewerbsschicht repräsentiert werden. Während des Lernprozesses wird dem Netzwerk eine Folge von Eingabemustern präsentiert, die in der Regel durch eine Wahrscheinlichkeitsverteilung erzeugt wurden. Für die Änderung der Gewichtsfaktoren des aktvierten j -ten Neurons gilt

$$\Delta w_{ij} = \eta(x_i - w_{ij}) \quad (2.11)$$

unter der Nebenbedingung (Normierung)

$$\sum_{i=1}^n w_{ij} = 1 \quad . \quad (2.12)$$

SOM-Netzwerke werden zur Datenkomprimierung, kombinatorische Optimierung, Rotorsteuerung sowie Sprach- und Mustererkennung eingesetzt.

Lineare Vektor Quantisierung Die Vektorquantisierung ist ein Prozess, stetig, reellwertige Vektoren \vec{x} aus einer Menge $A \subseteq \mathbb{R}^n$ auf den nächsten Referenz-Gewichtungswert \vec{w}_i aus der Menge $B \subseteq \mathbb{R}^m$ abzubilden. Die Eingabevektoren \vec{x} der Dimension n werden in einer endlichen Anzahl von Klassen transformiert, wobei jede Klasse durch einen Prototyp-Vektor \vec{w}_i dargestellt ist. Normalerweise versteht man unter nächstliegend den euklidischen Abstand

$$d(\vec{x}, \vec{w}_i) = \sqrt{\sum_{j=1}^n (w_{ij} - x_j)^2} \quad . \quad (2.13)$$

Vektorquantisierungs-Netzwerke arbeiten als konkurrierende Netzwerke. Wenn das Lernen überwacht erfolgt, nennt man sie auch lineare Vektorquantisierung (engl. *Linear Vector Quantization (LVQ)*). Wenn die Klasse korrekt erkannt wurde, wird der Gewichtungsvektor der gewinnenden Einheit in Richtung des Eingabevektors verschoben. Wenn ein falscher Prototyp ausgewählt wurde, wird der Gewichtungsvektor vom Eingabevektor weg verschoben. Diese Verschiebungen sind abhängig von dem euklidischen Abstand (Gl. 2.13). Die LVQ wird in den Aufgabenbereichen Mustererkennung und Datenkomprimierung eingesetzt.

Das Kognitron Das Kognitron-Netzwerk [Fuk80] stellt eine hierarchische Struktur dar, die aus mehreren kaskadenförmig angeordneten, modularen Einheiten besteht, die Signale in Vorwärtsrichtung verarbeiten. Diese Netzwerkarchitektur wurde ursprünglich für Aufgaben der Bildverarbeitung entworfen und orientiert sich daher stark an das System des menschlichen Sehens. Die Eingabeschicht (Sensorschicht) ist ein rechteckiges Feld aus Rezeptorneuronen mit lichtsensiblen Zellen und ist für die Eigenschaften auf Zell- oder Pixelebene verantwortlich. Zellen in höheren Schichten lernen, *Low-Level-Eigenschaften* (z.B. das Dach des Buchstaben A in der Zeichenerkennung) zu integrieren, und sind deshalb für globalere Eigenschaften zuständig. Die Integration setzt sich bis zur Ausgabeschicht fort, die vollständige Objekte im Bild identifiziert, wobei für jedes identifizierte Objekt eine Zelle erforderlich ist. Das Lernen wird schichtweise und konkurrierend ausgeführt, wobei die Zelle, die am meisten auf das Trainingsmuster reagiert, repräsentativ für diese Zellebene wird und deren Gewichtung so angepasst wird, dass sie noch mehr auf das Muster reagiert. Diese Netzwerke werden hauptsächlich in Anwendungen für die invariante Zeichen- und Objekterkennung eingesetzt.

Counterpropagation Counterpropagation-Netzwerke [HN87a] sind dreischichtige, vorwärtsgekoppelte Netzwerke, die in nahezu perfekter Kombination zwei Lernstrategien vereinigen und die durch diese Kopplung wesentlich leistungsfähiger geworden sind, als Netzwerke mit den isolierten Einzelstrategien. Die Eingabeschicht dient dem Counterpropagation-Netz lediglich der Verteilung der Eingangsaktivitäten, sie führt keine Berechnungen aus. Jedes Eingabeneuron ist mit jedem Neuron der Kohonen-Schicht über ein Gewicht w_{ij} und letztere sind wiederum über ein Gewicht w_{jk} mit den Neuronen der Ausgabeschicht (Grossberg-Schicht) verbunden. Die Kohonenschicht wird nach dem Ansatz des konkurrierenden Lernens trainiert (*Winner-Takes-All*). Die Grossberg-Schicht dagegen wird durch überwachtes Lernen trainiert. Counterpropagation-Netzwerke werden im Bereich der Mustererkennung, Mustervervollständigung und Signalverbesserung genutzt.

Adaptive Resonanztheorie Die *Adaptive Resonance Theory (ART)* wurde als Erweiterung konkurrierender-kooperierender Lernsysteme entwickelt [Gro76]. Dabei wurde versucht, das Stabilitäts-Elastizitäts-Problem sowie andere instabile Lerneigenschaften von konkurrierenden Netzwerken zu umgehen. Damit KNN vergleichbar mit den biologischen Vorbildern arbeiten, müssen sie in der Lage sein, sinnvolle Informationen im Gedächtnis zu behalten, während gleichzeitig neue wichtige Informationen erlernt, irrelevante ignoriert und veraltete oder unwichtige vergessen werden. Mit andern Worten sollen diese Netzwerke einen hohen Stabilitätsgrad aufweisen, wenn sie adaptiv neue Kategorien oder Konzepte lernen, und gleichzeitig anpassbar sein, um diese neuen Kategorien oder Konzepte zu erkennen und zu erlernen, also auch ein hohes Maß an Elastizität aufweisen. Diese beiden Ziele konkurrieren miteinander. ART-Netzwerke bilden n -dimensionale Eingabemuster auf Ausgabekategorien oder Klassen ab, die auf den Eigenschaften des Eingabemusters basieren. Ähnliche Eingabemuster (nächster Nachbar) werden in derselben Klasse gruppiert, nicht-ähnliche Muster in separaten, verschiedenen Klassen. Das Lernen in ART-Netzwerken erfolgt während der normalen Arbeitsweise des Netzwerkes in Echtzeit. Dabei handelt es sich um eine Form stetigen, nicht-überwachten adaptiven Lernens, wobei automatisch eine neue Kategorie gebildet wird, wenn dem Netz ein neues Eingabemuster präsentiert wird. Es werden solange neue Kategorien für neue Eingabemuster erzeugt, bis das Netzwerk seinen Pool noch nicht verwendeter Ausgabekategorie-Neuronen erschöpft hat. Alle weiteren neuen Eingabemuster werden zurückgewiesen. Eingabemuster, die ähnlich bereits eingerichteten Kategorien sind, werden sofort erkannt, indem an dem Neuron der selektierten Kategorie eine hohe Ausgabe erzeugt wird. Einga-

2 Theoretische Grundlagen

ben, die mit existierenden Kategorien übereinstimmen, initiieren außerdem ein gewisses Lernen für diese Kategorie, ohne dass dabei die Stabilität der erlernten Kategorien erhöht wird. Einige Einschränkungen von ART-Netzwerken betreffen ihre allgemeine Komplexität, Schwierigkeiten bei der Festlegung der Fehlerkriterium-Parameter für bestimmte Anwendungen sowie die relativ ineffiziente Verwendung von Ausgabeneuronen (für jede erlernte Kategorie ist ein Neuron erforderlich). ART-Netzwerke werden bspw. in Anwendungen zur Sensordatenfusion, Diagnose und Steuerung eingesetzt.

Sonstige In den Lehrbüchern *Computational Intelligence* von Rudolf Kruse *et al.* [Kru+15] und *Grundkurs Künstliche Intelligenz* von Wolfgang Ertel [Ert16] sind einige dieser Modelle ausführlich dargestellt. Weitere KNN, die in dieser Arbeit zwar genannt, aber nicht weiter beschrieben werden, sind: *Deep Belief Network (DBN)* [HOT16], *Generative Adversarial Network (GAN)* [Goo+14], *Deep Autoencoder (DAE)* [CYL+14] und *Stacked Denoising Autoencoder (SDA)* [Vin+08]. Entsprechende Quellen zu diesen Netzwerken sind angegeben, sodass sich interessierte Leser sich über den Aufbau, das Training und die Verwendung informieren können.

2.7 Optimierungstechniken

Die Entwicklung eines KNN kann man in folgenden Phasen einteilen [Fü96]:

1. Definition des Problems
2. Datenakquisition
3. Datenpräsentation
4. Wahl des Netzwerkmodells
5. Wahl der Netzwerkstruktur
6. Wahl der Netzparameter
7. Training des KNN
8. Testen des trainierten KNN
9. Anwendung

In Phase 1 muss zunächst untersucht werden, ob das Problem geeignet ist, um mit einem KNN bearbeitet zu werden. Die Beschaffenheit der Daten (Phase 2) muss quantitativen (ausreichende Anzahl) und qualitativen Anforderungen (weitgehende Fehlerfreiheit) genügen. In Phase 3 findet eine Codierung (binäre, ganze oder reelle Werte), eine Skalierung (symmetrische Häufigkeitsverteilung um den Mittelwert) und eine Normierung (lineare Abbildung auf das Intervall $[0, 1]$) der Daten statt [Ste93]. Die nächsten drei Phasen befassen sich mit der konkreten Ausarbeitung des KNN. Die Fragen, die man hier beantworten muss, sind u.a. welches Netzwerk-Modell benutzt werden soll, wie viele Schichten mit wie vielen Neuronen in der jeweiligen Schicht das KNN haben soll, wie die Verbindungen zwischen den Neuronen zu gestalten sind, oder welche Aktivierungsfunktionen verwendet werden können. In Phase 7 steht man möglicherweise vor der Auswahl mehrerer alternativer Lernregeln. Außerdem werden hier die Daten mindestens in zwei disjunkte Mengen aufgeteilt: Trainings- und Testmenge. Die Reihenfolge der Trainingsdaten, die das Netz zum Lernen benutzt, erfolgt nach Möglichkeit zufallsgesteuert. Beim Testen des Netzes (Phase 8) wird die Leistung des trainierten Modells anhand der Testmenge gemessen. Nach dem Training und Testen gelangt das KNN schließlich zur Anwendung bzw. Generalisierung, vorausgesetzt die Tests waren erfolgreich.

Der effektive Einsatz von neuronalen Systemen erfordert eine Feinabstimmung von Parameterwerten. Hierzu sind jedoch nur wenige heuristische Regeln bekannt. Daher ist man weitgehend auf eine experimentelle Vorgehensweise (*Trial-and-Error-Strategie*) angewiesen. Einige Optimierungstechniken werden im Folgenden kurz vorgestellt.

Early Stopping Präsentiert man dem Netzwerk zu häufig die gleichen Trainingsmuster, dann kann es passieren, dass das Netzwerk diese Muster auswendig lernt. Muster, die nicht zu dieser Trainingsmenge gehören, werden dann schlechter verarbeitet. Um dieses Überlernen (engl. *Overlearning*) bzw. diese Überanpassung (engl. *Overfitting*) zu verhindern, wird während des Lernens der Netzfehler auf einer Trainings- und einer Validierungsmenge gemessen. Wenn der Netzfehler der Validierungsmenge grösser wird, ist das Training abzubrechen (*Early Stopping*).

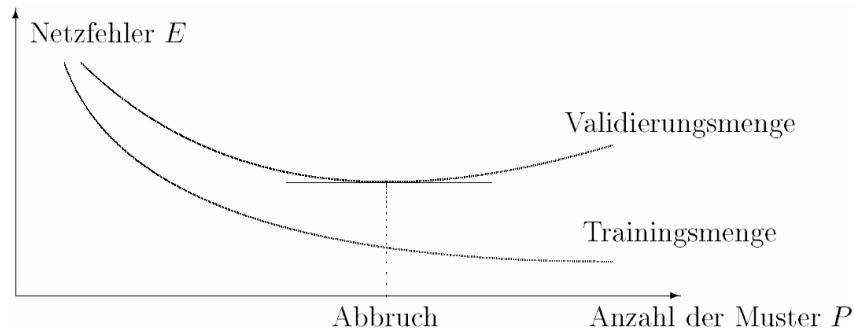


Abbildung 2.9: Übertrainieren und Trainingsabbruch

Kreuz-Validierung Die Methode Kreuz-Validierung (engl. *Cross Validation*) kann eine Überanpassung des Netzwerks vermindern. Die aus N Elementen bestehende Datenmenge wird dabei in $k \leq N$ etwa gleich große disjunkte Teilmengen T_1 bis T_k aufgeteilt, d.h. partitioniert. Dann werden k Trainings- bzw. Testläufe durchgeführt. Im ersten Durchlauf wird bspw. mit den Mengen T_2 bis T_k trainiert und T_1 zum Validieren verwendet. Im zweiten Durchlauf wird dann mit den Mengen T_1 und T_3 bis T_k trainiert und T_2 zum Validieren verwendet, usw. Im k -ten und letzten Durchlauf wird mit den Mengen T_1 bis T_{k-1} trainiert und T_k zum Validieren verwendet (siehe Abb. 2.10).

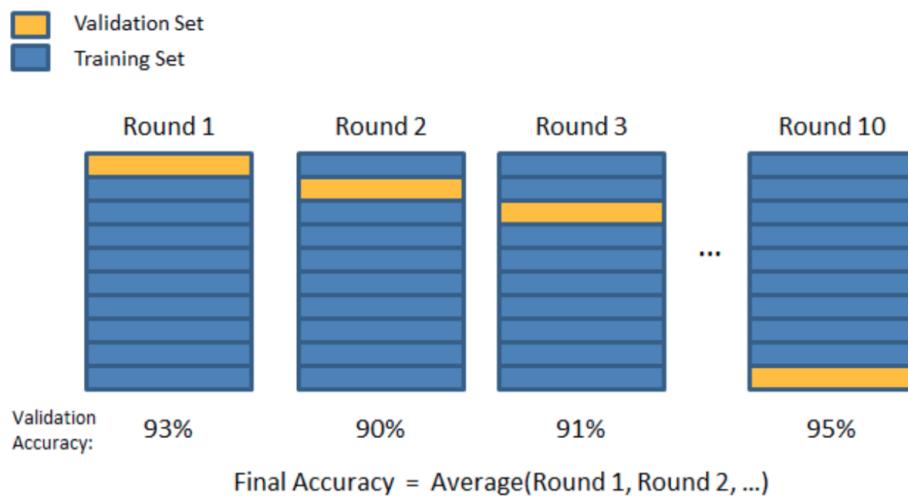


Abbildung 2.10: k -fache Kreuz-Validierung für $k = 10$ [Bro17]

2 Theoretische Grundlagen

Dieses Verfahren nennt man auch k -fache Kreuzvalidierung. Der Vorteil dieser Methode ist, dass zwar nicht gleichzeitig aber zumindest nacheinander, alle Daten sowohl zum Trainieren als auch zum Validieren verwendet werden. Allerdings ist noch offen, wie die Elemente bzw. Datensätze auf die k disjunkten Mengen verteilt werden (*Sampling*). Dies könnte bspw. nach dem Zufallsprinzip erfolgen. Eine Weiterentwicklung stellt die k -fache stratifizierte Kreuzvalidierung dar. Bei der Aufteilung der k Teilmengen wird dafür gesorgt, dass jede dieser Mengen annähernd die gleiche Verteilung bezüglich eines Merkmals besitzt. Ein Spezialfall stellt die *Leave-One-Out*-Kreuzvalidierung dar. Hier gilt $k = N$, d.h. die Anzahl der Durchläufe beträgt N und die jeweilige Test- bzw. Validierungsmenge besteht nur aus einem Datensatz. Der Vorteil dieser Methode ist, dass sich dadurch verschiedene Modelle sehr gut miteinander vergleichen lassen, da Zufallseffekte in der Aufteilung nicht mehr relevant sind. Ein Nachteil ist allerdings, dass bei einer großen Datenmenge sehr viele Durchläufe notwendig sind und dementsprechend das Training sehr viel Zeit in Anspruch nimmt.

Ausdünnungstechniken Sehr komplexe Modelle können Übertrainieren begünstigen. Es gibt verschiedene Möglichkeiten, Komplexität im Netzwerk durch topologieverändernde Eingriffe zu reduzieren. Die sogenannten Ausdünnungstechniken (engl. *Pruning*) werden benutzt, um die Anzahl der Neuronen bzw. Parameter zu reduzieren. Ansatzpunkt ist dabei das Gewicht oder die Ausgabe der Neuronen. Aufgrund einer Sensitivitätsanalyse werden unwichtige Eingabeneuronen, die den Fehler des Netzwerks nur marginal beeinflussen, ausfindig gemacht und entfernt (*Input-Pruning*). Unter *Weight-Pruning* sollen hier allgemein Verfahren verstanden werden, die geeignet sind, nicht benötigte Gewichte aus einem Netzwerk zu entfernen oder auf null zu setzen. Die Standardmethode (Kleinste Gewichte) benutzt als Testgröße den Betrag des Gewichts. Bei dem Pruning nach statistischer Signifikanz [HFZ92] werden Informationen über die Verteilung (z.B. Standardabweichung) der Gewichtsveränderungen für jede Verbindung im Netzwerk betrachtet. Das *Optimal-Brain-Damage* [LDS89] ist eine Methode, die die zweite Ableitung der Fehlerfunktion bei der Gewichtsänderung berücksichtigt. Eine andere Variante besteht darin, paarweise Korrelationsanalysen der Neuronenaktivitäten in verborgenen Schichten durchzuführen [ZHF92]. Zwei Neuronen mit hohen Korrelationskoeffizienten können dabei zusammengelegt werden (*Hidden Merging*). Die Betrachtung der Varianz der Neuronenaktivitäten erlaubt ferner, Neuronen zu identifizieren, die unabhängig vom anliegenden Eingabemuster stets dieselbe oder annähernd gleiche Ausgabe produzieren. Sie wirken faktisch als Schwellenwert und können komplett entfernt werden.

Regularisierung Eine einfache Möglichkeit, die Komplexität des Netzwerks während des Trainings zu reduzieren, ist die *Dropout*-Technik. Hierbei werden per Zufall eine vorher definierte Menge an Neuronen in einer Schicht deaktiviert, d.h. ausgeschaltet, sodass diese Einheiten während einer Trainingsepochen ausfallen (engl. *drop out*). Im nächsten Berechnungsschritt können dann per Zufall auch andere Einheiten ausgewählt werden, die dann kurzfristig ausgeschaltet bleiben. Diese Technik kann auch als temporäre Ausdünnungstechnik interpretiert werden.

Eine andere Möglichkeit der Regularisierung setzt auf der zu minimierenden Fehlerfunktion des Netzwerkes an und wird damit direkter Bestandteil des Lernverfahrens. Hierzu

wird ein Strafterm bzw. Komplexitätsterm eingeführt. Es gilt:

$$E_{\text{tot}} = \sum_{p=1}^P E^p + \lambda C(w_1, \dots, w_n) \quad , \quad (2.14)$$

wobei C der noch unspezifizierte, von den Gewichtungen w_i abhängige Komplexitätsterm und $\lambda > 0$ ein Parameter ist, der den Ausgleich zwischen der Anpassung an die Trainingsdaten und dem dabei beobachteten Zuwachs an Komplexität beschreibt. Die älteste Form der Modellierung, bekannt unter dem Namen Standard-Gewichtszerfall-Verfahren (engl. *Weight Decay*), bestraft größere Gewichte. Der Strafterm wächst quadratisch bei einer trainingsbedingten linearen Gewichtsvergrößerung:

$$C(w_1, \dots, w_n) = \sum_{i=1}^n w_i^2 \quad . \quad (2.15)$$

Die Verwendung solcher quadratischen Strafterme wird auch als L2-Regularisierung bezeichnet. In der Regressionsanalyse ist diese Regularisierungstechnik auch als *Ridge Regression* bekannt.

Demgegenüber bestraft der *Weigend*-Strafterm zwar ebenso größere Gewichte, die Bestrafung wächst jedoch nicht (quadratisch) unendlich bei einer trainingsbedingten linearen Gewichtsvergrößerung, sondern besitzt eine natürliche Obergrenze [WRH91]:

$$C(w_1, \dots, w_n) = \sum_{i=1}^n \left(\frac{\left(\frac{w_i}{w_0}\right)^2}{1 + \left(\frac{w_i}{w_0}\right)^2} \right) \quad . \quad (2.16)$$

Im Fall eines linearen Verhaltens des Strafterms spricht man von L1-Regularisierung. In der Regressionsanalyse wird auch die Regularisierungstechnik *Lasso* für *Least Absolute Shrinkage and Selection Operator* verwendet. Im Unterschied zu Gl. 2.16 wird jedoch der Absolutbetrag der Gewichte verwendet:

$$C(w_1, \dots, w_n) = \sum_{i=1}^n |w_i| \quad . \quad (2.17)$$

3 Multilayer Perceptron

Das Mehrschichten-Perzepton (engl. *Multilayer Perceptron (MLP)*) ist eines der bekanntesten und vielfältig einsetzbaren KNN, insbesondere weil es einen einfachen Aufbau hat (vgl. Abschnitt 3.1) und mit dem *Backpropagation*-Algorithmus (vgl. Abschnitt 3.2) eine Lernregel zur Verfügung gestellt wird, die häufig zu sehr guten Ergebnissen führt. Zu *Backpropagation* gibt es mittlerweile einige Erweiterungen (vgl. Abschnitt 3.3). Eine besondere Rolle beim Training spielt die Aktivierungsfunktion der Neuronen, für die verschiedene Alternativen verwendet werden können (vgl. Abschnitt 3.4). Beim überwachten Lernen muss außerdem eine Kostenfunktion benutzt werden, um die Fehler zu berechnen und zu bewerten (vgl. Abschnitt 3.5). Wenn ein MLP aus sehr vielen verborgenen Schichten besteht, d.h. das Netzwerk sehr tief ist, dann stößt man auf Schwierigkeiten im Training, die mit Hilfe von speziellen Techniken gelöst werden können, die unter dem Begriff *Deep Learning (DL)* zusammenfasst werden (vgl. Abschnitt 3.6). Aufgrund der Vielseitigkeit und Abbildungsfähigkeit von MLPs werden diese Netzwerke in zahlreichen Anwendungen in den Bereichen Mustererkennung, Funktionenapproximation, Klassifizierung, Prognose, Diagnose, Steuerung und Optimierung eingesetzt.

3.1 Aufbau

Das mehrschichtige vorwärtsgekoppelte Netzwerk (engl. *Multilayer Feedforward Network (MLFF)*) mit *Backpropagation*-Lernen (BP) wird auch als Mehrschichten-Perzepton bezeichnet, weil es Perzepton-Netzwerken mit mehreren Schichten sehr ähnlich ist. Es handelt sich um ein vollverknüpftes Netzwerk, das aus einer Eingabeschicht, einer oder mehrerer innerer Schichten und einer Ausgabeschicht besteht (siehe Abb. 3.1).

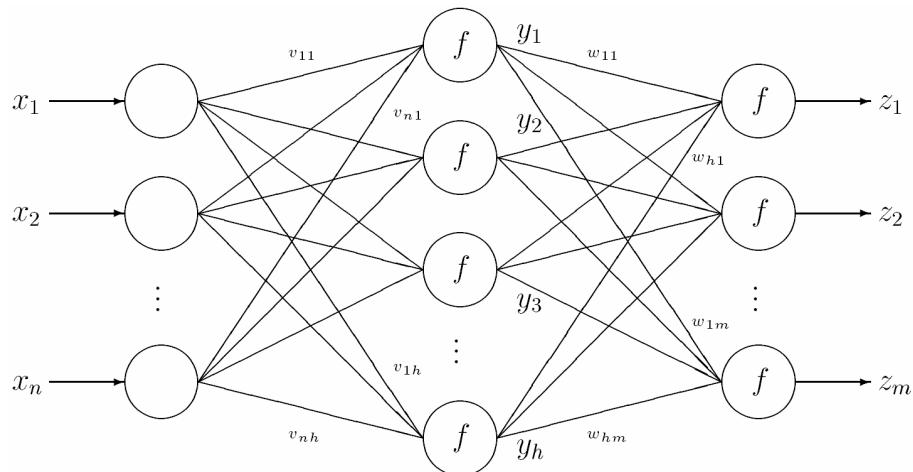


Abbildung 3.1: Schematische Darstellung eines MLP mit einer inneren Schicht. Aus Darstellungsgründen sind nicht alle Gewichtungen eingezeichnet.

Die n Neuronen der Eingabeschicht nehmen den Input x_1 bis x_n entgegen. Über die Gewichte v_{11} bis v_{nh} werden die Eingaben an die h Neuronen der verborgenen Schicht (engl. *Hidden Layer*) weitergeleitet. Diese Neuronen benutzen eine Aktivierungsfunktion f und

3 Multilayer Perceptron

besitzen einen veränderbaren Schwellenwert (engl. *Bias*). Dadurch wird die Aktivierung als Ausgaben y_1 bis y_h bestimmt. Diese Werte werden über die Gewichte w_{11} bis w_{hm} an die m Neuronen der Ausgabeschicht weitergeleitet. Auch diese m Neuronen besitzen eine Aktivierungsfunktion f und ein Bias. Die Aktivierungsfunktion dieser Ausgabeneuronen kann sich von denen der verborgenen Neuronen unterscheiden. Die berechneten Aktivierungen werden schließlich als Ergebnis bzw. Output z_1 bis z_m zurückgeliefert. Dieser Prozess wird auch als *Feedforward* bezeichnet. Er wiederholt sich für jeden Datensatz. Beim Training bzw. Lernvorgang werden die Gewichte der Verbindungen und die Schwellenwerte der Neuronen angepasst. Hierzu werden die Fehler, die das Netzwerk gemacht hat, zurück durch das Netzwerk propagierte, also von der Ausgabeschicht in Richtung Eingabeschicht. Deshalb heißt dieser Lernalgorithmus auch *Backpropagation*.

3.2 Backpropagation

Die Bestimmung der Parameter (Gewichte, Schwellenwerte) im MLP erfolgt mit der *Backpropagation*-Lernmethode [RHW86a; RHW86b][Par85][LeC86]. Es handelt sich um eine Optimierungsprozedur zur Minimierung des Netzfehlers, die auf dem steilsten Gradientenabstieg basiert (vgl. Kap. 2.4). Die berechneten Fehler werden als Eingaben für *Feedback*-Verknüpfungen verwendet, welche dann rückwärts Schicht für Schicht, also iterativ, die Gewichtungen anpassen. Betrachtet man ein Netz mit n Eingabeneuronen x_i , h Neuronen der verborgenen Schicht y_j und m Neuronen der Ausgabeschicht z_k . Die Gewichte zwischen Eingabe- und verborgener Schicht sind v_{ij} , die zwischen verborgener und Ausgabeschicht w_{jk} . Folglich gilt für die Ausgabesignale

$$z_k = f \left(\sum_{j=1}^h w_{jk} y_j \right) = f \left(\sum_{j=1}^h w_{jk} f \left(\sum_{i=1}^n v_{ij} x_i \right) \right) . \quad (3.1)$$

Die zu minimierende Energie- bzw. Kostenfunktion lautet

$$E_{\text{tot}} = \frac{1}{P} \sum_{p=1}^P E^p \quad \text{mit} \quad E^p = \frac{1}{2} \sum_{k=1}^m (t_k^p - z_k^p)^2 . \quad (3.2)$$

Die Gewichtsänderungen sind proportional zum Gradienten der Energiefunktion:

$$\Delta w_{jk} = -\eta \frac{\partial E_{\text{tot}}}{\partial w_{jk}} \quad \text{und} \quad \Delta v_{ij} = -\eta \frac{\partial E_{\text{tot}}}{\partial v_{ij}} . \quad (3.3)$$

Mit den Definitionen

$$H_j^p = \sum_{i=1}^n v_{ij} x_i^p , \quad I_k^p = \sum_{j=1}^h w_{jk} y_j^p \quad \text{und} \quad (3.4)$$

$$\delta_k^p = (t_k^p - z_k^p) f'(I_k^p) , \quad \delta_j^p = f'(H_j^p) \sum_{k=1}^m \delta_k^p w_{jk} \quad (3.5)$$

folgt aus Gl. 3.3

$$\Delta w_{jk} = \frac{\eta}{P} \sum_{p=1}^P \sum_{k=1}^m \delta_k^p y_j^p \quad \text{und} \quad \Delta v_{ij} = \frac{\eta}{P} \sum_{p=1}^P \sum_{j=1}^h \delta_j^p x_j^p . \quad (3.6)$$

Der *Backpropagation*-Lernalgorithmus heißt auch generalisierte Delta-Regel.

3.3 Erweiterungen

Zu *Backpropagation* existieren sehr viele Erweiterungen [Rud16]. Im Kern ist es ein Gradientenabstiegsverfahren (engl. *Gradient Descent Method*), also ein Optimierungsproblem. Je nachdem, wie viele Daten zum Trainieren benutzt werden, unterscheidet man drei verschiedene Varianten.

Die Methode ***Batch Gradient Descent***, auch *Vanilla Gradient Descent* genannt, berechnet die Gradienten der Energie- bzw. Kostenfunktion für die gesamten Trainingsdaten und danach wird dann die Aktualisierung (engl. *Update*) der Netzwerk-Parameter vorgenommen. Man spricht auch von *Offline Learning*. Dieses Verfahren ist aber ungeeignet, wenn sehr große Datenmengen verarbeitet werden müssen. Ggf. passt die Menge an Daten gar nicht in den Speicherbereich. Außerdem kann das Trainings sehr langsam sein.

Die Methode ***Stochastic Gradient Descent (SGD)*** dagegen aktualisiert die Parameter in jedem Trainingsschritt nach der Berechnung der Gradienten der Fehlerfunktion bzw. *Loss*-Funktion. Als Abgrenzung zur ersten Variante wird dieses Verfahren als *Online Learning* bezeichnet. Neue Trainingsdaten können relativ leicht hinzugefügt werden. Des Weiteren ist das Training meistens viel schneller als bei *Batch Gradient Descent*. Allerdings kann es aufgrund der vielen Aktualisierungen zu großen Fluktuationen kommen, welche die Konvergenz des Verfahrens erheblich erschweren.

Eine Lösung hierfür stellt die dritte Alternative dar, die das Beste aus beiden Welten verbindet: Die Methode ***Mini-Batch Gradient Descent*** teilt die Trainingsdaten in mehrere Mengen auf und führt dann für jede dieser Mengen das *Batch Gradient Descent* durch. Der Nachteil dieser Methode ist, dass ein weiterer Parameter als sogenannter *Hyperparameter* hinzukommt, nämlich die Größe der *Mini-Batch* Teilmengen. Je nach Anwendung verwendet man häufig die Größen 32, 64, 128 oder 256, also ein Vielfaches der Zahl 2, in Anlehnung an die *Low Level* Datenverarbeitung in Prozessoren (CPU bzw. GPU).

Probleme mit dem Gradientenabstiegsverfahren können sich u.a. durch flache Plateaus der Fehlerfunktion ergeben (vgl. Kap. 2.4). Mit dem **variablen Trägheitsterms (Momentum Term)** verhindert man, dass das Lernen in einem flachen Bereich des Fehlergebirges nur noch langsam vorangeht. Man gibt dem Verfahren einen Schwung mit, indem man einen Bruchteil der letzten Gewichtsänderung nochmals addiert [RHW86b]:

$$\Delta w_{kj}^p = \eta \sum_{k=1}^m \delta_k^p y_j^p + \alpha \Delta w_{kj}^{p-1} \quad \text{und} \quad \Delta v_{ij}^p = \eta \sum_{j=1}^h \delta_j^p x_j^p + \alpha \Delta v_{ij}^{p-1} . \quad (3.7)$$

1992 haben Martin Riedmiller und Heinrich Braun das Lernverfahren ***Resilient Backpropagation (Rprop)*** als Erweiterung von *Backpropagation* vorgestellt [RB92; RB93]. Übersetzt bedeutet der Name des Algorithmus *Elastische Fortpflanzung* und ist vergleichbar mit dem Verhalten einer Springfeder, bei der eine Rückstellkraft wirkt. Im Gegensatz zu *Backpropagation* wird für jedes Gewicht eine individuelle Schrittweite bestimmt und die letzte Gewichtsänderung wird im nächsten Iterationsschritt miteinbezogen. Des Weiteren wird nur das Vorzeichen des Gradienten verwendet und nicht der Wert des Gradienten selbst. Die Idee hinter dem adaptiven Vorgehen zur Gewichtsanpassung kann man wie folgt beschreiben: Falls sich das Vorzeichen des Gradienten in einem Gewicht ändert, d.h. man also "über das Ziel hinausgeschossen" ist, dann wird die neue Schrittweite reduziert und man springt wieder etwas zurück. Falls dagegen die Vorzeichen beider Gradienten gleich sind, dann erhöht man die Schrittweite, um schneller voranzukommen.

In seiner Vorlesung *Neural Networks for Machine Learning* an der Universität Toronto hat Geoffrey E. Hinton am 06.02.2014 eine Erweiterung zu Rprop vorgestellt [HSS14]: **Root Mean Square Propagation (RMSprop)**. Rprop funktioniert sehr gut zusammen mit *Batch Gradient Descent*, wenn die Parameteranpassung *offline* erfolgt. In Kombination mit *Mini-Batch* kann es aber Schwierigkeiten geben, wenn nur das Vorzeichen und nicht auch der Wert des Gradienten betrachtet wird. Die Parameter (Gewichte) können evtl. viel zu große Werte annehmen. Die Idee von RMSprop ist, dass die jeweilige Lernrate noch durch einen Korrekturterm dividiert wird. Dieser Korrekturterm hängt nun von der Größe des Gradienten ab. Es wird ein exponentiell geglätteter Gradient verwendet, d.h. die Wurzel (engl. *Root*) des Mittelwerts (engl. *Mean*) der quadrierten Gradienten (engl. *Squared Gradients*). RMSprop kann als Verallgemeinerung von Rprop betrachtet werden und ist sowohl für *Mini-Batch* als auch *Full-Batch* geeignet.

Der Optimierer **Adam** für *Adaptive Moment Estimation* ist wiederum eine Erweiterung von RMSprop [KB14]. Hier werden nicht nur die Gradienten zur Korrektur der Lernrate verwendet, sondern auch die zweiten partiellen Ableitungen, d.h. die Momente. Adam wird inzwischen in vielen Fällen als Standard-Algorithmus für Optimierungsprobleme wie dem Gradientenabstieg eingesetzt [Kar18].

3.4 Aktivierungsfunktionen

Im *Backpropagation*-Algorithmus werden die ersten Ableitungen der Aktivierungsfunktion berechnet. Dies setzt voraus, dass diese Funktion stetig differenzierbar ist. Als Aktivierungsfunktion wird häufig die sigmoide, S-förmige, monoton steigende und differenzierbare logistische Funktion

$$f(x) = \frac{1}{1 + \exp(-b \cdot x)} \quad (3.8)$$

benutzt, wobei b eine Konstante ist, die die Steilheit der Kurve bestimmt (siehe Abb. 3.2). Der Wertebereich der Funktion liegt zwischen 0 und 1. Die Ableitung hat die Form

$$f'(x) = b \cdot f(x)(1 - f(x)) \quad . \quad (3.9)$$

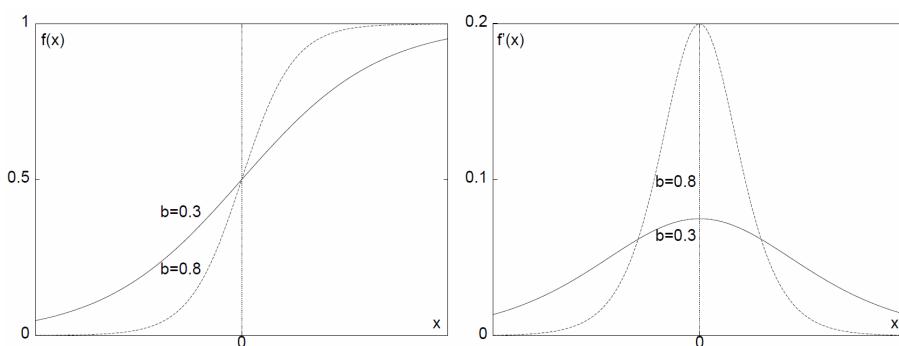


Abbildung 3.2: Logistischen Funktion mit Ableitung für zwei Parameter b

Einerseits ist die sigmoide Aktivierungsfunktion also differenzierbar, andererseits lässt sich die erste Ableitung sehr einfach analytisch bestimmen, weil sich diese aus der Original-Funktion berechnen lässt.

Eine andere, sehr beliebte Aktivierungsfunktion, ist der *Tangens Hyperbolicus*, also:

$$f(x) = \tanh(x) = \frac{2}{1 + \exp(-2x)} - 1 \quad (3.10)$$

mit der ersten Ableitung

$$f'(x) = 1 - \tanh^2(x) \quad . \quad (3.11)$$

Im Jahr 2000 haben Richard H. R. Hahnloser und Kollegen eine neuartige Aktivierungsfunktion eingeführt, welche biologisch inspiriert ist [Hah+00]:

$$f(x) = x^+ = \max(0, x) = \begin{cases} 0 & \text{falls } x < 0 \\ x & \text{falls } x \geq 0 \end{cases} \quad . \quad (3.12)$$

Diese Funktion liefert nur den positiven Teil des Arguments x zurück. In der Elektrotechnik gibt es Gleichrichter (engl. *Rectifier*) zur Umwandlung von Wechsel- in Gleichspannung, die eine ganz ähnliche Funktionsweise haben. Deshalb nennt man diese Funktion auch *Rectifier*. Ein Neuron, welches dieses Aktivierungsfunktion verwendet, wird auch als *Rectified Linear Unit (ReLU)* bezeichnet. Ein Problem ist allerdings, dass diese Funktion für $x = 0$ nicht stetig differenzierbar ist. Aus diesem Grund wird die Funktion meistens angenähert und stattdessen die Gleichung

$$f(x) = \log(1 + \exp(x)) \quad (3.13)$$

benutzt. Im Bereich *Deep Learning* werden ReLU-Aktivierungen sehr erfolgreich eingesetzt. Abb. 3.3 zeigt den Verlauf der drei bisher vorgestellten Aktivierungsfunktionen.

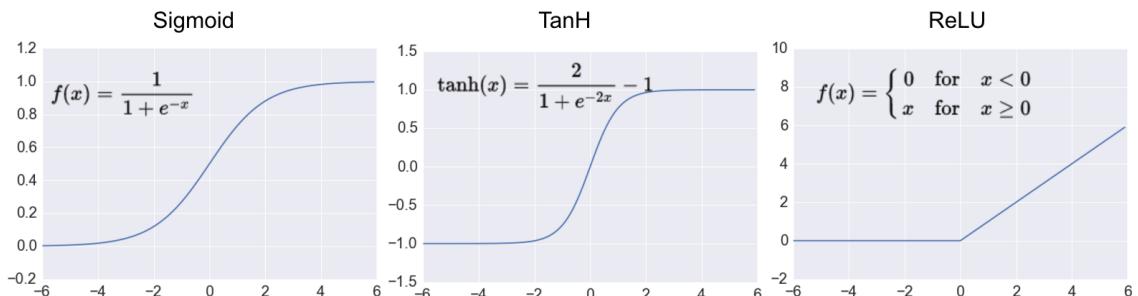


Abbildung 3.3: Aktivierungsfunktionen im Vergleich [Mou16]

Eine sehr spezielle Aktivierungsfunktion stellt die *Softmax*-Funktion dar:

$$f(x)_i = \frac{\exp(x_i)}{\sum_{k=1}^K \exp(x_k)} \quad \text{für } i = 1, \dots, K \quad . \quad (3.14)$$

Die *Softmax*-Funktion kann auch als normierte Exponentialfunktion bezeichnet werden. In Klassifizierungsaufgaben kommt diese Funktion häufig für die Einheiten der Ausgabeschicht zum Einsatz, weil die Ausgabe dann als die jeweilige Klassenwahrscheinlichkeit p_i der K Klassen interpretiert werden kann.

3.5 Kostenfunktionen

Im *Backpropagation*-Algorithmus wird eine Energie- bzw. Kostenfunktion als Zielfunktion definiert, die es zu minimieren gilt. Hierbei werden zunächst die Differenzen zwischen der Netzausgabe und den tatsächlichen Werten (*Teaching Input*) aller Ausgabeneuronen quadriert und summiert. Die Fehlerfunktion zu einem Trainingsdatensatz nennt man auch *Loss*-Funktion. Beim *Offline*-Lernen werden die einzelnen Fehler zu allen Trainingsdaten dann noch summiert und anschließend normiert, d.h. durch die Anzahl der Trainingsdatensätze dividiert. Diese Gesamtfehlerfunktion ist sehr ähnlich zum *Mean Squared Error (MSE)*, allerdings wird beim MSE noch eine Quadratwurzel aus der Summe gezogen. In der Regressionsanalyse benutzt man bspw. die Methode der kleinsten Quadrate als Standardverfahren, um die Parameter eines Modells zu schätzen, das den Zusammenhang zwischen einer abhängigen Variable zu unabhängigen Variablen möglichst gut funktional abbildet. Der Vorteil des Quadrierens liegt darin, dass sich positive und negative Abweichungen bzw. Fehler nicht gegenseitig aufheben können. Eine alternative Fehlerfunktion ist der *Mean Absolute Error (MAE)*, also der mittlere absolute Fehler. Hier werden die Differenzen nicht quadriert, sondern der Absolutbetrag benutzt. Statt der absoluten Fehler können auch die relativen Fehler betrachtet werden, also die prozentualen Fehler. Das entsprechende Fehlermaß heißt *Mean Absolute Percentage Error (MAPE)*.

In Klassifikationsaufgaben sind die Datensätze der Trainingsmenge bereits festen Klassen zugeteilt. Man spricht auch von nominalskalierten Merkmalen oder von kategorischen Variablen. In diesen Anwendungen kommen noch andere Kosten- bzw. *Loss*-Funktionen zum Einsatz. Als Beispiele können die Funktionen *Square Loss*, *Hinge Loss*, *Logistic Loss* und *Cross Entropy Loss* genannt werden [Wik18g]. Letztere Funktion ist besonders interessant, weil sie stetig differenzierbar ist und somit in Gradientenabstiegsverfahren benutzt werden kann. Die *Cross Entropy Loss* Funktion ist die erste Wahl in tiefen neuronalen Netzwerken, die zur Klassifikation eingesetzt werden.

3.6 Deep Learning

Wenn KNN sehr tief werden, d.h. aus sehr vielen verborgenen Schichten bestehen, dann kommen spezielle Techniken für das Training zum Einsatz, die man unter dem Begriff *Deep Learning (DL)* zusammenfassen kann. Zunächst stellt sich aber die Frage: Was ist besonders am Training tiefer KNN?

Zur Beantwortung der Frage sehen wir uns ein Beispiel an. Betrachten wir ein MLP mit sehr vielen Schichten und wenden wir den *Backpropagation*-Algorithmus an, dann werden die berechneten Fehler Schicht für Schicht von den Ausgabeneuronen über die verborgenen Neuronen zu den Eingabeneuronen zurückgegeben und dabei die Parameter angepasst. Normalerweise werden vor dem Training die Parameter des Netzwerks, also die Gewichte der Verbindungen und die Schwellenwerte der Neuronen, zufällig initialisiert, wobei hier meistens die Standardnormalverteilung mit Mittelwert 0 und Standardabweichung 1 verwendet wird (siehe Abb. 3.4). Im Mittel sind die Werte der Parameter also null und 68,2 % der Parameterwerte liegen im Intervall $[-1, +1]$. Die Ausgaben der KNN sind aufgrund der verwendeten Aktivierungsfunktionen meistens auf das Intervall $[0, 1]$ (sigmoide Funktion) oder auf das Intervall $[-1, +1]$ (*Tangens Hyperbolicus*) normiert. Die Aktualisierung der Parameter ist proportional zum Gradienten der Fehlerfunktion. Die Ableitungen dieser Aktivierungsfunktionen liegen aber ebenfalls im Intervall $[0, 1]$. Wenn zwei Faktoren, die im Intervall $[0, 1]$ liegen, miteinander multipliziert werden, dann ist

das Produkt kleiner als der kleinere der beiden Faktoren. Das Produkt nähert sich also der Zahl Null an, je mehr solcher Multiplikationen durchgeführt werden. Genau solche Operationen werden als Matrix-Multiplikationen beim *Backpropagation* bzw. Gradientenabstiegsverfahren zur Bestimmung der Parameteränderungen durchgeführt.

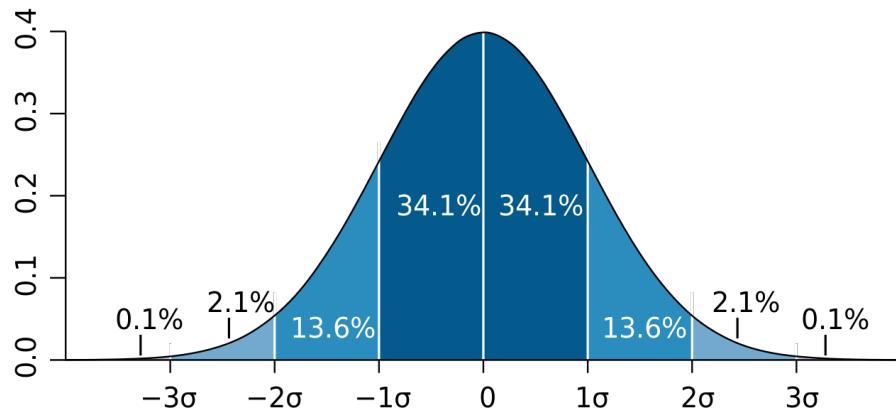


Abbildung 3.4: Standardnormalverteilung mit Intervallen [Toe07]

Besteht das KNN nun aus sehr vielen verborgenen Schichten, dann werden auch viele solcher Multiplikationen durchgeführt. Die berechneten Gradienten des Gradientenabstiegsverfahrens werden somit immer kleiner und verschwinden schließlich. Das Training des Netzwerks kommt damit praktisch zum Stillstand. Dieses Phänomen wurde bereits 1991 von Josef Hochreiter im Rahmen seiner Diplomarbeit [Hoc91] zum ersten Mal erwähnt. 2010 haben Xavier Glorot und Yoshua Bengio es dann genauer untersucht [GB10].

Um das Problem der verschwindenden Gradienten zu behandeln, werden verschiedene Lösungsmöglichkeiten vorgeschlagen und eingesetzt:

1	Initialisierung	<i>Xavier Initialization</i>	[GB10]
2	Aktivierungsfunktion	<i>Rectified Linear Unit (ReLU)</i>	Kap. 3.4
3	Normierung	<i>Batch Normalization (BN)</i>	[IS15]
4	Training	<i>Gradient Clipping</i>	[PMB12]
5	Wiederverwendung	<i>Pre-Trained Layers</i>	[Gé17]
6	Spezielle Netzwerke	<i>Deep Belief Network (DBN)</i> <i>Convolutional Neural Network (CNN)</i> <i>Long Short-Term Memory (LSTM)</i>	[HOT16] Kap. 4 Kap. 5.4

Einige dieser Lösungsvorschläge werden im Rahmen dieser Arbeit behandelt, somit sind Verweise zu den entsprechenden Kapiteln genannt. Zu den anderen Lösungsvorschlägen sind Quellen angegeben, mit deren Hilfe sich die Lösung erschließen lässt.

4 Convolutional Neural Network

Das *Convolutional Neural Network* (CNN) hat seinen Namen aufgrund der mathematischen Operation Faltung (engl. *Convolution*) bekommen. Die Faltung oder Konvolution zweier Funktionen f und g wird durch die Gleichung

$$(f * g)(x) = \int_{\mathbb{R}^n} f(\tau)g(x - \tau)d\tau \quad (4.1)$$

beschrieben. Die Faltung kann bspw. benutzt werden, um eine gegebene Funktion zu glätten. Hierzu wird diese mit Hilfe eines speziellen Glättungskerns (engl. *Kernels*) gefaltet. Der Kernel wirkt dabei auch wie ein Filter. Bezogen auf die Anwendung der digitalen Bildverarbeitung kann ein solcher Kernel bzw. ein solches Filter verwendet werden, um bspw. einen Bereich eines Bildes abzutasten, um dadurch Kanten oder Merkmale (engl. *Features*) auf dem Bildausschnitt zu entdecken. Nimmt man bspw. ein relativ kleines Bild mit einer Größe von 100×100 Pixeln, so besteht dieses bereits aus 10.000 einzelnen Bildpunkten. Jeder Pixel wird dann noch über einen Wert beschrieben, der bspw. die Farbe oder Intensität (Graustufe) angibt. Mit Hilfe des klassischen *Multilayer Perceptron* (vgl. Kap. 3) ist es schwierig, solche Bilder zu verarbeiten, um bspw. Objekte darauf zu erkennen und zu klassifizieren. Denn ein solches Netzwerk müsste ja bereits über 10.000 Eingabe-Neuronen verfügen. Nun kommt die Idee der Faltung ins Spiel. Das CNN funktioniert im Prinzip wie ein MLP. Allerdings wird jeweils nur ein Bildausschnitt mit Hilfe eines speziellen Filters bzw. Kernel abgetastet und das Ergebnis auf einer *Feature Map* gespeichert. Die Eingabeschicht dieses KNN besteht dann aus einem Stapel von *Feature Maps*. Deshalb können CNN auch sehr gut im Bereich der Bildverarbeitung, insbesondere in der Objekterkennung und Objektklassifizierung eingesetzt werden. In den folgenden Abschnitten wird diese grundlegende Idee, der Aufbau und die Funktionsweise weiter ausgeführt und vertieft, sowie einige prominente CNN als Beispiele vorgestellt.

4.1 Visueller Cortex

1981 wurde der Nobelpreis für Physiologie und Medizin u.a. dem gebürtigen Kanadier David H. Hubel und dem schwedischen Wissenschaftler Torsten N. Wiesel für ihre Entdeckungen über die Informationsbearbeitung im Sehwahrnehmungssystem überreicht. Ab 1958 beschäftigten sich die beiden Wissenschaftler mit dem Aufbau und der Informationsverarbeitung des visuellen Cortex und führten hierzu Untersuchungen mit Katzen und Affen durch. Dabei bekamen sie eine erste Ahnung davon, wie das Gehirn die sensorischen Informationen analysiert. Ausgehend von den Rezeptorenzellen der Netzhaut (Retina) des Auges, den ca. 6 Millionen Zapfen für Farbinformationen und den ca. 120 Millionen Stäbchen für Hell-Dunkel und Bewegungen, werden die auftreffenden Photonen des Lichts in elektrische Signale umgewandelt und über den Sehnerv und den seitlichen Kniehöcker des Thalamus (*Corpus Geniculatum Laterale (CGL)*) zur Sehrinde, dem primären visuellen Cortex (V1), geleitet, wo dann die eigentliche Verarbeitung beginnt (siehe Abb. 4.1).

4 Convolutional Neural Network

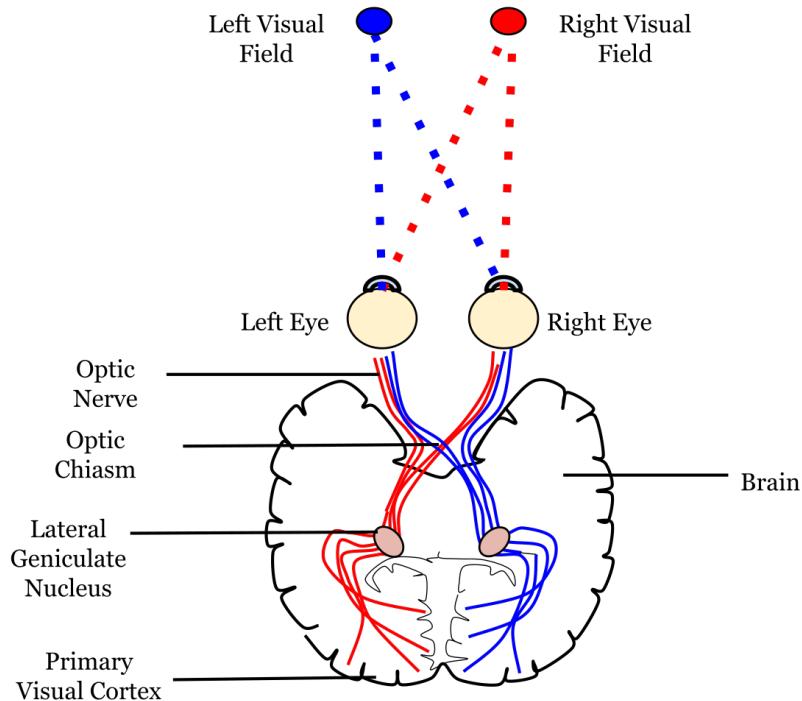


Abbildung 4.1: Signalweg von der Netzhaut zur Sehrinde [Mad16]

Hubel studierte zunächst Physik und Mathematik an der McGill Universität in Montréal, bevor er sich dann schließlich für ein Medizin-Studium entschied. Seine interdisziplinären Kenntnisse nutzte Hubel, um eine Mikroelektrode zu entwickeln, mit der die Aktivität einzelner Zellen der primären visuellen Cortex von Katzen erforscht werden konnten. 1958 trafen sich dann Hubel und Wiesel im Labor von Stephen W. Kuffler an der Johns Hopkins Universität, bevor sie dann ein Jahr später zur Harvard Medical School wechselten. Mit ihren Untersuchungen haben Hubel und Wiesel schließlich eine komplexe, mehrstufige Architektur des primären visuellen Cortex V1 identifiziert [Ley12].

Die Nervenzellen im primären visuellen Cortex sind in sechs Schichten zu Hypersäulen angeordnet. Eine Hypersäule repräsentiert einen Ort auf der Netzhaut, also nur einen kleinen Bereich im Gesichtsfeld, nimmt eine Fläche von 1 mm x 1 mm ein, enthält zwischen 50.000 und 100.000 Nervenzellen und stellt das kleinste vollständige Analysemodul der Sehrinde dar [Wis18]. Die Zellen in diesen Säulen reagieren nur auf bestimmte Reize:

- | | |
|------------------------|---|
| 1 Orientierungssäulen | Gleiche Reizrichtung |
| 2 Positionssäulen | Gleicher Ort auf der Netzhaut |
| 3 Augendominanzsäulen | Rechtes oder linkes Auge |
| 4 Merkmalsdetektoren | Lichtstreifen einer bestimmten Ausrichtung |
| 5 Komplexe Zellen | Lichtstreifen, die sich in einer best. Richtung bewegen |
| 6 Hyperkomplexe Zellen | Streifen bestimmter Länge oder Ecken und Winkel |

Es gibt also spezialisierte Zellen in V1, die für spezifische Aufgaben des Sehens benötigt werden. Diese Spezialisierung setzt sich auch in der sekundären und tertiären visuellen Rinde (V2 bis V5) fort, wobei hier bspw. die Wahrnehmungsaspekte Farben und Stereosehen relevant sind (siehe Abb. 4.2).

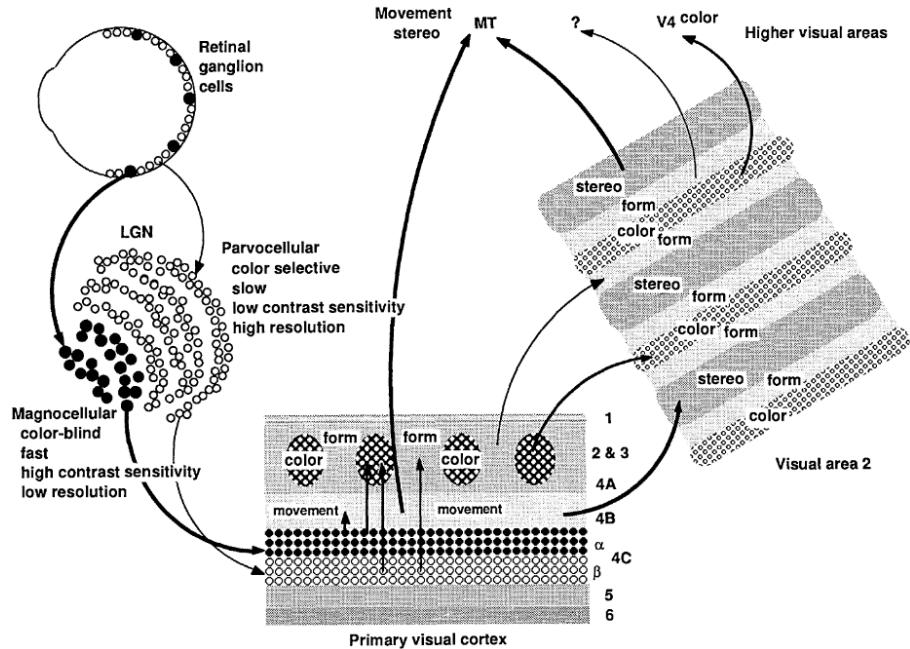


Abbildung 4.2: Funktionale Spezialisierung des visuellen Cortex [LH88]

Diese grundlegende Erkenntnis von der Funktionsweise des visuellen Cortex wurde als Inspiration bei der Entwicklung der *Convolutional Neural Network (CNN)* verwendet.

4.2 Aufbau und Schichten

Im letzten Abschnitt wurde festgestellt, dass im visuellen Cortex verschiedene Zellen auf bestimmte Funktionen bei der Signalverarbeitung spezialisiert sind und nacheinander Stufen der Verarbeitung durchlaufen werden. Beim *Feature Learning* durch *Deep Learning* werden stufenweise bestimmte *Features* erlernt (siehe Abb. 4.3).

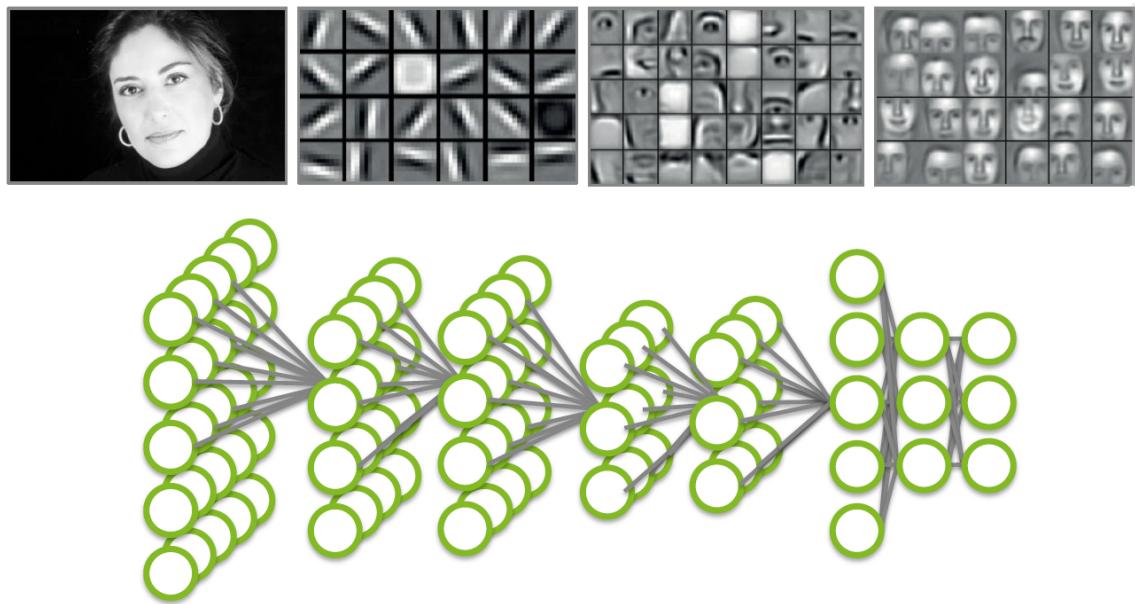


Abbildung 4.3: Feature Learning durch Deep Learning – in Anlehnung an [Lee+11]

4 Convolutional Neural Network

Betrachtet man bspw. die Gesichtserkennung als Anwendung der digitalen Bildverarbeitung, dann werden in der ersten Stufe zunächst Kanten als *Features* erkannt. In der nächsten Stufe können dann schon größere Bereiche als *Features* erkannt werden: Augen, Nase, Mund usw. In der letzten Stufe wird dann das komplette Gesicht als *Feature* gelernt.

Wie das *Multilayer Perceptron* ist auch das *Convolutional Neural Network* ein mehrschichtiges vorwärtsgekoppeltes Netzwerk. Dabei wiederholt sich die Kombination aus *Convolution Layer* und *Pooling Layer* mehrfach. Mit Hilfe dieser Kombination können *Features* erlernt werden. Abb. 4.4 zeigt den schematischen Aufbau eines CNN. Mit *Subsampling* ist das *Pooling* gemeint. Der *Convolutional Layer* besitzt mehrere Filter-Kernel. Jeder Filter-Kernel enthält die zu trainierenden Gewichte und zu jedem dieser Filter-Kernel wird eine *Feature Map* generiert. Es entsteht also ein Stapel von *Feature Maps*, die zwar alle den gleichen Input bekommen, aber aufgrund der verschiedenen Kernels auch unterschiedliche *Features* extrahieren.

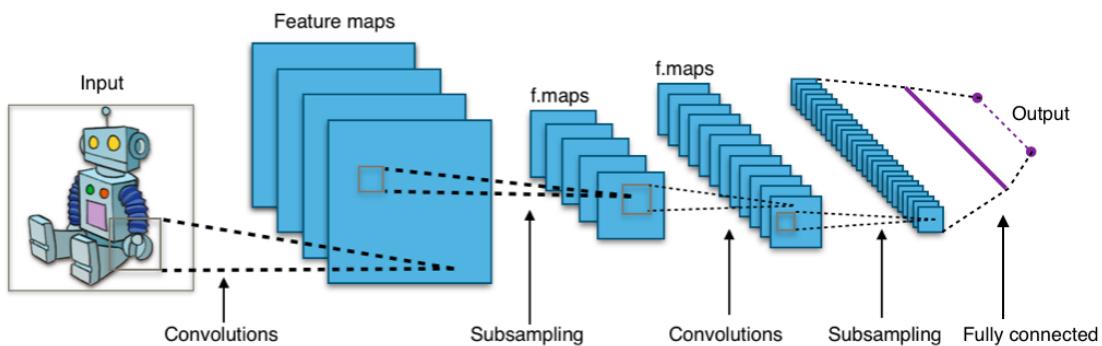


Abbildung 4.4: Aufbau eines typischen CNN [Aph15b]

Convolutional Layer Diese Schicht ist der wesentliche Baustein eines CNN. Die Neuronen bzw. Einheiten in dieser Schicht sind entsprechend der jeweiligen Anwendung angeordnet. Im Bereich Bilderkennung werden normalerweise zweidimensionale digitale Bilder verarbeitet. Somit sind die Einheiten in diesem Anwendungsfall als *2D-Convolutional Layer* angeordnet, nur ist dieser wesentlich kleiner als das gesamte Bild. Die Aktivität jeder Einheit wird durch eine diskrete Faltung (engl. *Convolution*) berechnet. Dabei wird die Faltungsmatrix, auch Filter-Kernel genannt, schrittweise über das Inputbild bewegt. Ein solcher Schritt (engl. *Stride*) kann ein oder mehrere Pixel betragen. Das innere Produkt aus der Faltungsmatrix und dem aktuellen Bildausschnitt ergibt dann gerade die Inputaktivität. Wenn die Faltungsmatrix über den Bildrand verschoben wird, kommt eine spezielle Technik zum Einsatz, das sogenannte *Padding*. Hierbei muss definiert werden, wie mit den eigentlich nicht-existierenden Bildpunkten verfahren werden soll. Beim *Zero-Padding* werden bspw. einfach Nullwerte hinzugefügt. Eine Besonderheit des *Convolutional Layers* ist, dass die Gewichte des Filter-Kernels für alle Neuronen einer *Feature Map* gleich sind, d.h. diese Parameter werden geteilt, man spricht auch von sogenannten *Shared Weights*. Deshalb ist der Einsatz von *Convolutional Layers* besonders effizient. Jede Einheit in diesem *Convolutional Layer* empfängt also nur Input-Signale von einer lokalen Bildumgebung, teilt sich aber die Parameter (Gewichte) mit allen anderen Einheiten. Dies führt dazu, dass Kanten bzw. *Features* in den Bildern besonders gut erkannt werden. Als Aktivierungsfunktion wird zu den Einheiten im *Convolutional Layer* meistens die *Rectified Linear Unit (ReLU)* verwendet.

Pooling Layer Nach jedem *Convolutional Layer* folgt normalerweise ein *Pooling Layer*. Dieser wird benötigt, um überflüssige Informationen zu entfernen. Bspw. wird bei der Objekterkennung nicht die exakte Position einer Kante bzw. eines *Features* benötigt, sondern eine ungefähre Lokalisierung ist ausreichend. Häufig werden entweder das *Max Pooling* oder das *Avg Pooling* eingesetzt. Beim *Max Pooling* wird nur die Aktivität des aktivsten Neurons bezüglich eines Kernels (z.B. der Größe 2×2) gefiltert. Beim *Avg Pooling* wird dagegen der Mittelwert der Aktivitäten bestimmt und diese mittlere Aktivität weiter verwendet. Mit Hilfe des *Poolings*, auch *Down Sampling* oder *Subsampling* genannt, lässt sich also die Dimension und damit die Komplexität reduzieren. Dadurch wird Überanpassung (engl. *Overfitting*) vorgebeugt, der Speicherbedarf sinkt und die Berechnungsgeschwindigkeit steigt. *Pooling* ist somit die Voraussetzung, wenn sehr tiefe CNN konstruiert werden sollen, um komplexe Probleme zu lösen. Abb. 4.5 zeigt ein Beispiel für das *Max Pooling*, bei dem ein Filter der Größe 2×2 mittels einer Schrittweite (*Stride*) von 2 auf eine Input-Matrix der Größe 4×4 angewendet wird.

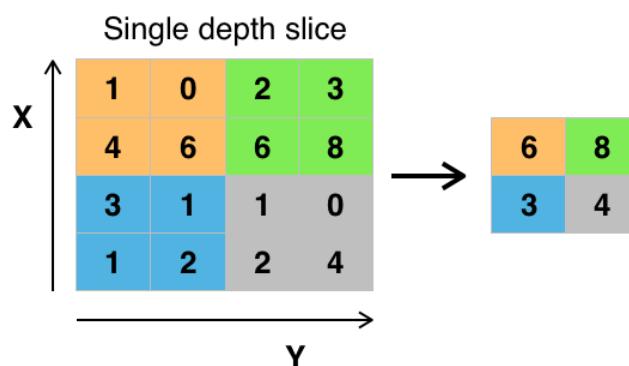


Abbildung 4.5: Max Pooling mit 2×2 -Filter und $\text{Stride}=2$ [Aph15a]

Fully-Connected Layer Das CNN wird häufig mit einer oder mehreren vollvernetzten Schichten abgeschlossen, analog zum Aufbau eines *Multilayer Perceptron*. Für eine typische Klassifikationsaufgabe im *Data Mining* gibt es in der letzten Schicht genau so viele Neuronen, wie es Klassen gibt. Es wird also die sogenannte *One Hot*-Codierung bzw. *Dummy*-Variablen verwendet, durch die jede Klasse als eine eigene Variable mit den Werten 0 und 1 für *false* und *true* abgebildet wird. Dabei geht man von nominalskalierten Merkmalen aus, d.h. man muss keine impliziten Annahmen über Ähnlichkeiten von Klassen machen. Als Aktivierungsfunktion wird meistens *Softmax* verwendet, da die Summe über alle Elemente genau eins ergibt und diese Funktion dann wie eine Normierung wirkt.

Dropout Layer Neben dem *Pooling Layer* kann auch ein *Dropout Layer* verwendet werden, um die Komplexität des Netzwerks zu verringern und um das Übertrainieren bzw. die Überanpassung zu reduzieren. In dieser Schicht werden die Aktivitäten von einigen Neuronen zufällig auf null gesetzt. Diese Einheiten fallen dann in diesem Trainingsschritt aus (engl. *drop out*). Über eine einstellbare Ausfallrate in Prozent kann dieses Verhalten gesteuert werden.

Loss Layer Normalerweise wird der *Backpropagation*-Algorithmus bzw. ein Gradientenabstiegsverfahren verwendet, um die Parameter des CNN mittels überwachten Lernens anzupassen. Der *Loss Layer* definiert dabei, wie das Training mit der Abweichung zwischen den vorhergesagten und tatsächlichen Output-Werten umgehen soll. Somit ist der

4 Convolutional Neural Network

Loss Layer als letzte Schicht des CNN zu implementieren. Je nach Aufgabe kann als *Loss-Funktion* bspw. eine der drei Funktionen verwendet werden:

- | | | | |
|---|------------------------|-----------------------|-------------------------------|
| 1 | Sigmoide Cross-Entropy | binäre Klassifikation | Werte 0 oder 1 |
| 2 | Softmax | Klassifikation | Werte $\in [0, 1]$ |
| 3 | Euklid Loss | Regression | Werte $\in [-\infty, \infty]$ |

4.3 LeNet-5

Die LeNet-5-Architektur ist eine der ersten und bekanntesten Topologien eines CNN, entwickelt von Yann LeCun, um handgeschriebene Ziffern zu erkennen [LeC+98]. Abb. 4.6 zeigt den schematischen Aufbau zu diesem CNN.

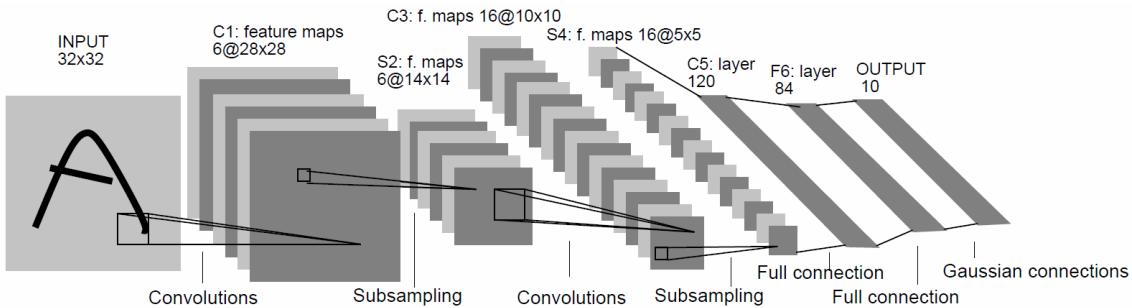


Abbildung 4.6: Aufbau des CNN LeNet-5 [LeC+98]

In Tab. 4.1 sind die verwendeten Einstellungen der LeNet-5-Architektur im Detail angegeben. Die ersten 5 Schichten des CNN dienen dem *Feature Learning*, während die letzten beiden Schichten zur Klassifikation genutzt werden.

Schicht	Typ	Maps	Größe	Kernel	Stride	Aktiv.
In	Input	1	32 x 32	-	-	-
C1	Convolution	6	28 x 28	5 x 5	1	tanh
S2	Avg Pooling	6	14 x 14	2 x 2	2	tanh
C3	Convolution	16	10 x 10	5 x 5	1	tanh
S4	Avg Pooling	16	5 x 5	2 x 2	2	tanh
C5	Convolution	120	1 x 1	5 x 5	1	tanh
F6	Fully Connected	-	84	-	-	tanh
Out	Fully Connected	-	10	-	-	RBF

Tabelle 4.1: Topologie der LeNet-5-Architektur

Eigentlich bestehen die zu verarbeitenden Bildern aus 28 x 28 Pixeln. Diese werden aber zu etwas größeren Bildern (32 x 32 Pixeln) derart vorverarbeitet, dass an den Rändern weiße Bildpunkte ergänzt werden (*Zero Padding*). Außerdem werden die Graustufenwerte (0 - 255) noch normalisiert, d.h. auf das Intervall [0, 1] gebracht, indem die Werte durch die Zahl 255 dividiert werden. Die Ausgabeschicht besteht aus 10 Neuronen, weil es 10 Objektklassen (Ziffern Null bis Neun) gibt, die in den Bildern erkannt werden sollen.

4.4 AlexNet

Die AlexNet-Architektur ist benannt nach Alex Krizhevsky, der zusammen mit Ilya Sutskever und Geoffrey E. Hinton von der Universität Toronto dieses tiefe CNN entwickelt hat [KSH12]. Es hat 2012 den Wettbewerb *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* mit einer Fehlerrate von 15,3 % sehr eindrucksvoll gewonnen und zwar mit einem Vorsprung von über 10 Prozentpunkten zum Zweitplatzierten (Fehlerrate 26,2 %) [Rus+15]. Die Aufgabe bestand darin, Objekte auf Bildern zu erkennen, die zu jeweils einer von 1000 verschiedenen Klassen gehören. Als Trainingsdaten standen 1,2 Millionen Bilder zur Verfügung, zum Validieren der Modellgüte der Klassifizierer wurden 50.000 Bilder aus einer Menge von 150.000 zufällig ausgewählt, die von der Plattform Flickr und anderen Webseiten stammen, und nicht bereits in den Trainingsdaten enthalten sind [Ima14]. Das Sieger-Netzwerk besteht aus 5 *Convolutional Layers* und 3 *Fully-connected Layers* (siehe Abb. 4.7).

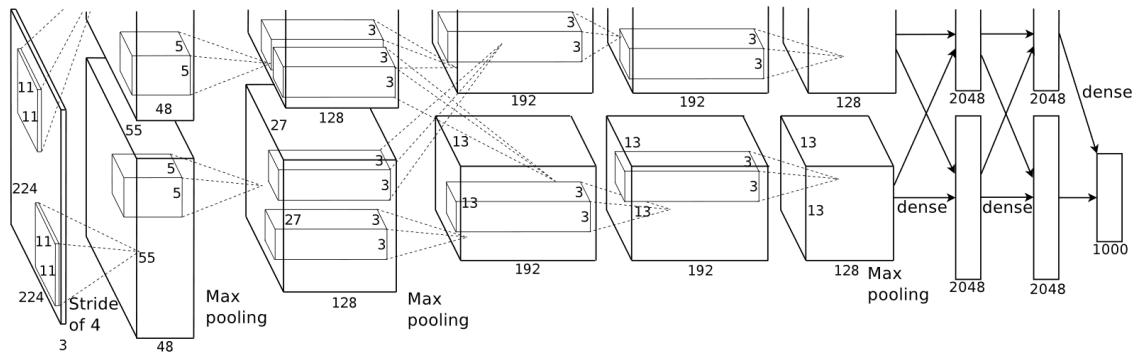


Abbildung 4.7: Aufbau des AlexNet [KSH12]

In Tab. 4.2 sind die verwendeten Einstellungen der AlexNet-Architektur detailliert angegeben. *Valid Padding* bedeutet, dass kein *Padding* angewendet wird. Beim *Same-Padding* haben Input und Output *Feature Maps* die gleiche Dimension.

Schicht	Typ	Maps	Größe	Kernel	Stride	Padding	Aktiv.
In	Input	3	224 x 224	-	-	-	-
C1	Convolution	96	55 x 55	11 x 11	4	Same	ReLU
S2	Max Pooling	96	27 x 27	3 x 3	2	Valid	-
C3	Convolution	256	27 x 27	5 x 5	1	Same	ReLU
S4	Max Pooling	256	13 x 13	3 x 3	2	Valid	-
C5	Convolution	384	13 x 13	3 x 3	1	Same	ReLU
C6	Convolution	384	13 x 13	3 x 3	1	Same	ReLU
C7	Convolution	256	13 x 13	3 x 3	1	Same	ReLU
F8	Fully Connected	-	4.096	-	-	-	ReLU
F9	Fully Connected	-	4.096	-	-	-	ReLU
Out	Fully Connected	-	1.000	-	-	-	Softmax

Tabelle 4.2: Topologie der AlexNet-Architektur

4 Convolutional Neural Network

Um das Übertrainieren bzw. die Überanpassung des Netzwerks zu reduzieren, wurden Regularisierungstechniken verwendet. Die Schichten F8 und F9 des Netzwerks wurde mittels *Dropout* mit einer Rate von 50 % während des Trainings temporär ausgedünnt. Außerdem kam *Data Augmentation* zum Einsatz, d.h. die Trainingsbilder wurden zufällig Pixelweise verschoben, gedreht, gespiegelt usw. Eine Besonderheit des AlexNet ist auch die *Local Response Normalisation (LRN)*, eine spezielle Art der Normierung, welche auf die ReLUs der Schichten C1 und C3 angewendet wird [KSH12].

4.5 GoogleLeNet

Zwei Jahre später nach dem Erfolg des AlexNet beim Wettbewerb ILSVRC (vgl. Kap. 4.4), ist es Christian Szegedy und seinen Kollegen von Google Inc. gelungen, die Fehlerrate sogar auf unter 7 % zu reduzieren [Sze+14]. Hierzu haben sie das GoogleLeNet entwickelt, welches viel tiefer als das AlexNet ist. Mit Hilfe von speziellen Einheiten, den sogenannten *Inception Modulen*, konnten Parameter viel effizienter genutzt werden, als dies in früheren Architekturen möglich war. Abb. 4.8 zeigt den Aufbau eines solchen *Inception Moduls* mit Einheiten zur Dimensionsreduktion (*Pooling*).

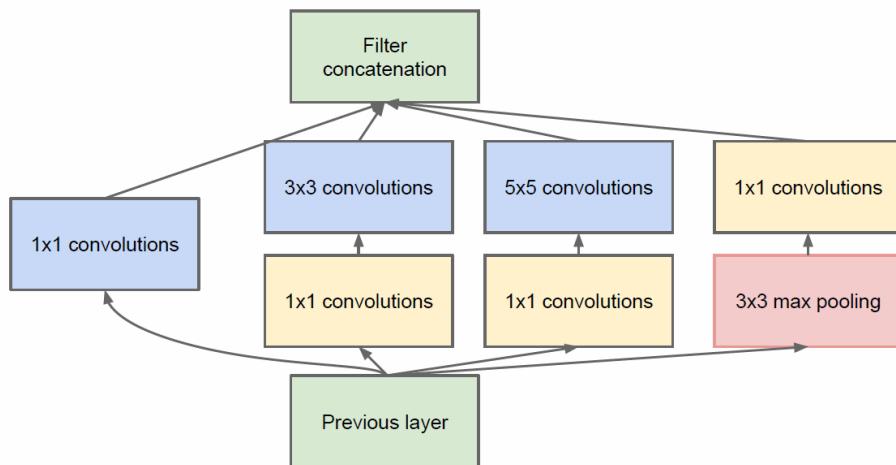


Abbildung 4.8: Schematischer Aufbau eines Inception Moduls [Sze+14]

Mit dem *Inception Modul* lassen sich also vielfache *Features* in nur einer Stufe erlernen, sowohl generelle *Features* (5×5 Kernel) als auch lokale *Features* (1×1 Kernel). Mit einem speziellen Filter werden die Ergebnisse am Ende wieder zusammengefügt und an die nächste Schicht weitergereicht. Abb. 4.9 zeigt den grafischen Aufbau des kompletten GoogleLeNet vom Input (links) zum Output (rechts), in dem neun *Inception Module* enthalten sind. Aus Darstellungsgründen ist das Netzwerk um 90 Grad gedreht.

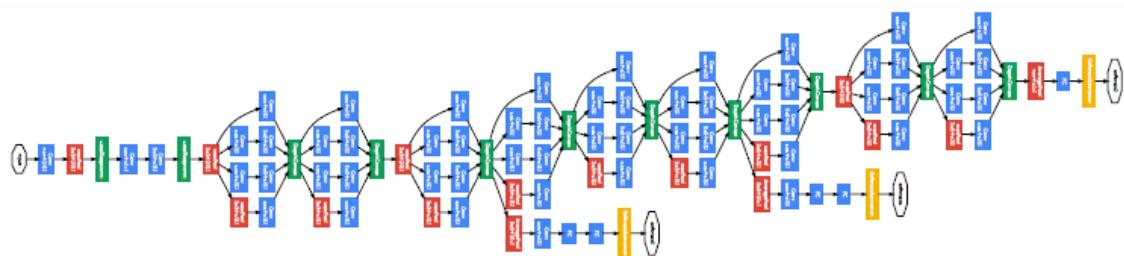


Abbildung 4.9: Graphischer Aufbau des GoogleLeNet [Sze+14]

In Abb. 4.10 ist der Aufbau von GoogleLeNet tabellarisch dargestellt.

type	patch size/ stride	output size	depth	# 1×1	# 3×3 reduce	# 3×3	# 5×5 reduce	# 5×5	pool proj	params	ops
convolution	$7 \times 7 / 2$	$112 \times 112 \times 64$	1							2.7K	34M
max pool	$3 \times 3 / 2$	$56 \times 56 \times 64$	0								
convolution	$3 \times 3 / 1$	$56 \times 56 \times 192$	2		64	192				112K	360M
max pool	$3 \times 3 / 2$	$28 \times 28 \times 192$	0								
inception (3a)		$28 \times 28 \times 256$	2	64	96	128	16	32	32	159K	128M
inception (3b)		$28 \times 28 \times 480$	2	128	128	192	32	96	64	380K	304M
max pool	$3 \times 3 / 2$	$14 \times 14 \times 480$	0								
inception (4a)		$14 \times 14 \times 512$	2	192	96	208	16	48	64	364K	73M
inception (4b)		$14 \times 14 \times 512$	2	160	112	224	24	64	64	437K	88M
inception (4c)		$14 \times 14 \times 512$	2	128	128	256	24	64	64	463K	100M
inception (4d)		$14 \times 14 \times 528$	2	112	144	288	32	64	64	580K	119M
inception (4e)		$14 \times 14 \times 832$	2	256	160	320	32	128	128	840K	170M
max pool	$3 \times 3 / 2$	$7 \times 7 \times 832$	0								
inception (5a)		$7 \times 7 \times 832$	2	256	160	320	32	128	128	1072K	54M
inception (5b)		$7 \times 7 \times 1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7 \times 7 / 1$	$1 \times 1 \times 1024$	0								
dropout (40%)		$1 \times 1 \times 1024$	0								
linear		$1 \times 1 \times 1000$	1							1000K	1M
softmax		$1 \times 1 \times 1000$	0								

Abbildung 4.10: Tabellarischer Aufbau des GoogleLeNet [Sze+14]

Das GoogleLeNet hatte trotz seiner enormen Tiefe mit 22 Schichten nur etwa 6 Millionen freie Parameter, wohingegen beim AlexNet ca. 60 Millionen freie Parameter trainiert werden mussten.

4.6 Sonstige

Neben dem LeNet-5 (vgl. Kap. 4.3), dem AlexNet (vgl. 4.4) und dem GoogleLeNet (vgl. 4.5) gibt es noch zahlreiche andere Architekturen von CNN. Es lässt sich bereits feststellen, dass die Netzwerke von Jahr zu Jahr tiefer und komplexer werden. Ein anderer Trend ist, dass ein Ensemble bzw. Komitee aus Netzwerken verwendet wird, um das Gesamtergebnis per demokratische Abstimmung aus den einzelnen Resultaten zu bestimmen.

Beispielsweise gewann das ResNet ein Jahr nach dem GoogleLeNet den Wettbewerb ILSVRC in der Kategorie Objektklassifikation. Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun trainierten 2015 dieses sehr tiefe CNN mit bis zu 152 Schichten und erzielten mit einem Ensemble von diesen Netzen eine Fehlerrate von nur 3,57 % [He+15]. ResNet ist die Abkürzung von *Residual Network*. Es trägt diesen Namen, weil von den eigentlichen Output-Werten die Input-Werte abgezogen werden, sodass dieses Netzwerk eigentlich die Residuen lernt. Umgesetzt wird diese Idee durch sogenannte *Skip Connections*, die auch als *Shortcut Connections* bezeichnet werden (siehe Abb. 4.11). Zu Beginn des Trainings sind die Parameter (Gewichte und Bias) meistens in der Nähe von null und der Output des Netzwerks ist dann ebenfalls nahe null. Verwendet man nun *Skip Connections*, dann ist der Output aber ähnlich zum Input, d.h. die Zielfunktion ist näherungsweise die Identitätsfunktion. Durch diesen Trick läuft das Training sehr viel schneller an, was bei solchen tiefen Netzwerken sehr wichtig ist.

4 Convolutional Neural Network

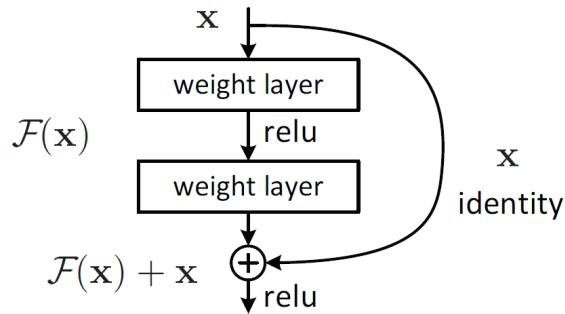


Abbildung 4.11: Funktionsweise einer Skip Connection in einer Residual Unit [He+15]

Das chinesische Siegerteam Trimp-Soushen des Wettbewerbs ILSVRC von 2016 in der Kategorie Objektklassifikation hat eigentlich keine neuen Netzwerke konstruiert, sondern im Wesentlichen die bestehenden verfeinert und ein Ensemble über das Ergebnis abstimmen lassen, sodass die Fehlerrate auf 2,99 % reduziert werden konnte [Ima16]. Ein Jahr später wurde die Fehlerrate vom WMM-Team auf 2,25 % verbessert [Ima17]. Hierzu wurden spezielle Bausteine namens *Squeeze-and-Excitation (SE)* entwickelt und anstelle der *Inception-* bzw. Residuen-Module in die Netzwerke integriert (siehe Abb. 4.12).

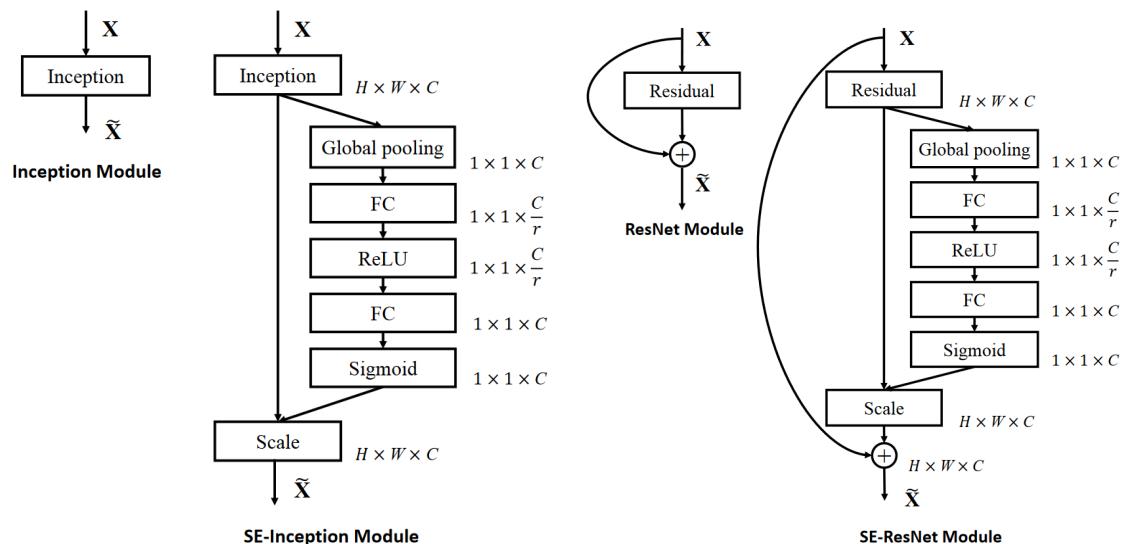


Abbildung 4.12: SE-Module ersetzen Inception- und Residuen-Module [HSS17]

Der Aufwand war jedoch enorm: Es wurden 64 Nvidia Pascal Titan X Grafikprozessoren für ein paralleles und synchrones 20-stündiges Training verwendet. Ab dem Jahr 2018 wird dieser Wettbewerb dann von der *Data Science* Plattform *Kaggle* durchgeführt [Kag18d]. Populäre und erfolgreiche CNN sowie andere KNN werden mittlerweile in sogenannten *Model Zoos* gesammelt. Je nach Softwarelösung (vgl. Kap. 6) gibt es verschiedene Implementierungen zu diesen Netzwerken.

5 Recurrent Neural Network

Die Künstlichen Neuronalen Netzwerke, die bisher näher betrachtet wurden, also das *Multilayer Perceptron* (vgl. Kap. 3) und das *Convolutional Neural Network* (vgl. Kap. 4), gehören zu den vorwärtsgekoppelten Netzwerken (engl. *Feedforward Networks*). Die Daten bzw. Signale durchlaufen diese Netzwerke von den Eingaben-Einheiten zu den Ausgaben-Einheiten. CNN wurden beispielsweise in Anlehnung an das visuelle Sehen entwickelt und werden insbes. im Bereich der Bilderkennung eingesetzt. Aber auch in anderen Bereichen wie z.B. der Sprachverarbeitung (engl. *Natural Language Processing (NLP)*) konnten CNN erste Erfolge aufweisen [Col+11]. Dabei sind andere KNN-Typen aufgrund ihrer Architektur eigentlich besser zur Problemlösung von Aufgaben geeignet, in denen Sequenzen verarbeitet werden müssen. Es sind die sogenannten rekurrenten bzw. rekursiven neuronalen Netzwerke. Im Gegensatz zu den bisher betrachteten Netzwerken gibt es in einem *Recurrent Neural Network (RNN)* auch Verbindungen zwischen den Neuronen derselben Schicht oder Verbindungen von Neuronen zu Neuronen einer vorangegangenen Schicht. Auch hier stand wieder die Natur Pate: In der Großhirnrinde, dem Neocortex, sind Neuronen auch auf diese Weise verschaltet. Der Neocortex ist für höhere Gehirnfunktionen wie Motorik und Sprache verantwortlich. Außerdem spielt er eine wichtige Rolle für Gedächtnis und Lernprozesse.

Am 13.11.2012 erschien das Buch mit dem Titel *How to Create a Mind: The Secret of Human Thought Revealed* von Ray Kurzweil [Kur12], die deutsche Übersetzung lautet: Das Geheimnis des menschlichen Denkens. Einblicke in das Reverse Engineering des Gehirns. Der hierarchische Aufbau der Welt spiegelt sich nach Meinung Kurzweils im hierarchischen Aufbau des Neocortex wider. Dieser Teil der Großhirnrinde besteht aus 300 Millionen Mustererkennern. Je nach Komplexität laufen die Mustererkennungsprozesse auf verschiedenen Ebenen ab. Kurzweil selbst hat u.a. das *Hidden Markov Model (HMM)* als Mustererkenner im Bereich der Spracherkennung verwendet. Abb. 5.1 zeigt ein Beispiel für ein solches stochastisches Modell, welches aus Markov-Ketten besteht. Die Basiselemente sind Zustände x , Beobachtungen y , Übergangswahrscheinlichkeiten a und Emissionswahrscheinlichkeiten b .

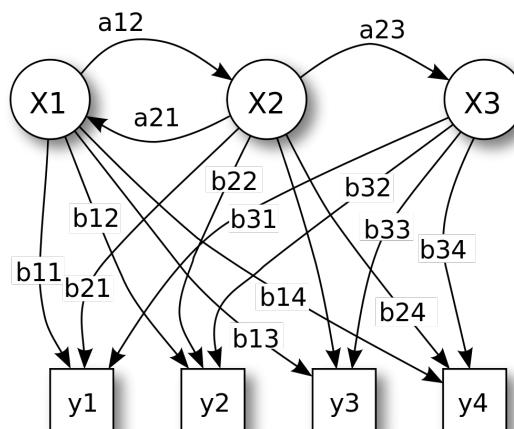


Abbildung 5.1: Beispiel für ein Hidden Markov Model [Raz12]

Auch wenn Kurzweil kein Freund von Künstlichen Neuronalen Netzwerken ist, so stellen die RNN jedoch ebenfalls sehr gute Mustererkennner im Bereich der Sprachanalyse dar. In den folgenden Abschnitten werden zunächst die einzelnen Bausteine, d.h. das rekurrente Neuron bzw. die *Memory Cell* vorgestellt, anschließend Typen von Input-Output-Sequenzen differenziert, spezielle Lernverfahren präsentiert und abschließend zwei neue und bedeutende Konzepte erläutert: *Long Short-Term Memory (LSTM)* und *Gated Recurrent Unit (GRU)*.

5.1 Bausteine

Neben den *Feedforward*-Verbindungen zwischen den Neuronen bzw. Einheiten der Künstlichen Neuronalen Netzwerke, sind auch drei Typen von *Feedback*-Verbindungen möglich (siehe Abb. 5.2):

- | | | | | |
|---|------------------------|--------------------------|-------|------|
| 1 | Direkte Rückkopplung | <i>Direct Feedback</i> | w_d | blau |
| 2 | Indirekte Rückkopplung | <i>Indirect Feedback</i> | w_i | grün |
| 3 | Seitliche Rückkopplung | <i>Lateral Feedback</i> | w_l | rot |

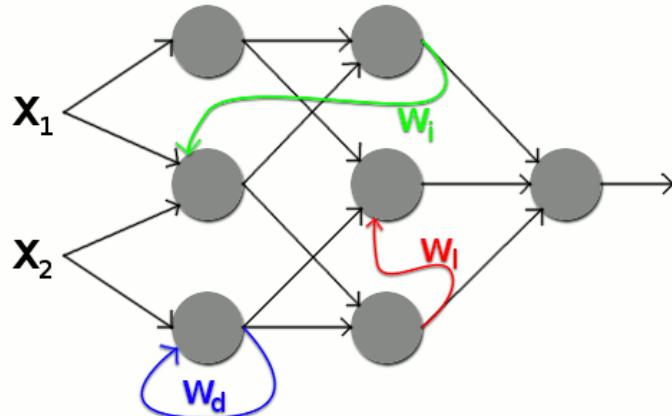


Abbildung 5.2: Drei Feedback-Typen [Mer16]

Die Basiseinheit eines RNN, das rekurrente Neuron (engl. *Recurrent Neural Unit*), ist eine Einheit mit einer direkten Rückkopplung, d.h. der eigene Ausgang wird als weiterer Eingang verwendet (siehe blaue Verbindung in Abb. 5.2 bzw. die neue Abb. 5.3).

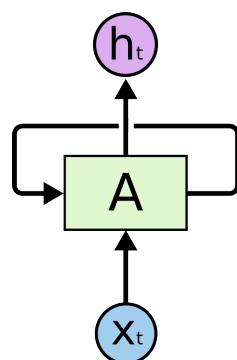


Abbildung 5.3: Basiseinheit im RNN [Ola15]

In jedem Zeitschritt t bzw. *Frame* bekommt dieses spezielle Neuron A neben den normalen Eingaben \vec{x}_t auch die Ausgabe vom vorherigen Zeitschritt h_{t-1} . Wenn man nun viele solcher Zeitschritte bzw. *Frames* betrachtet, und für jeden Zeitschritt die RNN-Basiseinheit nacheinander darstellt, ergibt sich das folgende Bild (siehe Abb. 5.4).

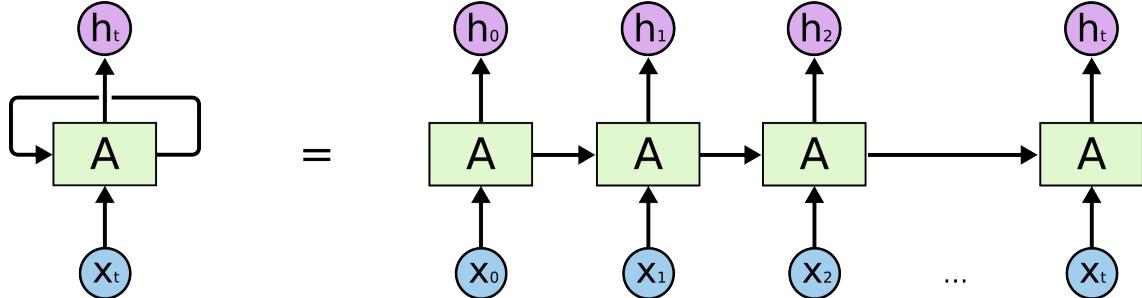


Abbildung 5.4: Ausgerollte RNN-Basiseinheit [Ola15]

Diese Darstellungsform nennt man auch ausgerollte RNN-Einheit. Für den Fall $t = 0$ wird noch keine Rückkopplung verwendet bzw. diese wird auf null gesetzt. Das Verhalten dieser RNN-Einheit erinnert an eine Kette. Deshalb sind RNN auch besonders gut geeignet, um Sequenzen oder Listen von Daten zu verarbeiten. Im Zeitschritt t fließen also nicht nur die Eingaben \vec{x}_t , sondern auch alle vorherigen Eingaben \vec{x}_0 bis \vec{x}_{t-1} indirekt in Form der Ausgaben h_0 bis h_{t-1} in die Berechnung der neuen Ausgabe h_t mit ein. Diese RNN-Einheit wirkt also wie ein Gedächtnis. Deshalb wird diese Einheit auch als *Memory Cell* bezeichnet.

5.2 Sequenzen

Eine Sequenz ist eine Folge von normalerweise gleichartigen Elementen. Bspw. kann eine Sequenz eine Folge von vektoriellen Eingabedaten \vec{x}_0 bis \vec{x}_t sein. Betrachten wir nun die Verarbeitung solcher Sequenzen, die aus Vektoren bestehen. In Abb. 5.5 sind fünf Fälle unterschieden.

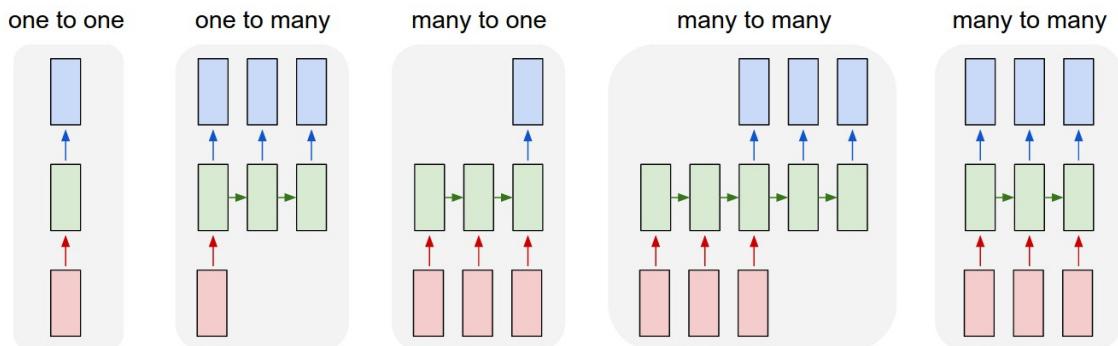


Abbildung 5.5: Verarbeitung von Sequenzen [Kar15]

Jedes Rechteck stellt einen Vektor dar, wobei die Farben verschiedene Bedeutungen zu kommen: Rot für die Eingabe (*Input*), Grün für das Verborgene (*Hidden*) und Blau für die Ausgabe (*Output*) - analog also zum Aufbau und den Schichten von KNN. Die Pfeile repräsentieren mathematische Operationen, z.B. Matrix-Multiplikationen. Im ersten Fall (*One to One*) kommen noch keine Sequenzen vor. Hier wird ein Eingabe-Vektor auf einen

5 Recurrent Neural Network

Ausgabe-Vektor abgebildet. Dies ist bspw. bei der Objektklassifizierung der Bilderkennung der Fall: Das 2D-Bild als Input kann auch als Vektor dargestellt werden und der Output-Vektor enthält die unterschiedlichen Objektklassen. Im zweiten Fall (*One to Many*) bekommt man als Output eine Sequenz von Vektoren. Eine Beispielanwendung ist das automatische Generieren von Bildunterschriften (*Image Captioning*): Als Input wird das Bild verwendet, als Output kommt eine Folge von Wörtern heraus. Im dritten Fall (*Many to One*) sind Input und Output im Vergleich zum zweiten Fall gerade vertauscht. Die *Sentiment Analysis* ist eine typische Anwendung: Ein Text, also eine Folge von Wörtern, wird hinsichtlich verschiedener Stimmungen (2 Klassen: positiv, negativ) untersucht. Der vierte Fall (*Many to Many*) mit vielen verborgenen Einheiten beschreibt bspw. das maschinelle Übersetzen: Ein englischer Satz, bestehend aus n Wörtern, wird ins Deutsche übertragen (m Wörter). Der letzte Fall sieht auf dem ersten Blick so ähnlich aus wie der vierte. Gemeint ist damit aber die synchrone Verarbeitung der Sequenzen. Eine typische Anwendung ist die automatische Objektklassifikation in Videos. Im ersten Fall wurden Objekte auf Bildern klassifiziert. Ein Video besteht aus einer Folge von Bildern bzw. *Frames*. Zu jedem Bild erfolgt nun eine synchrone Objektklassifizierung.

Man kann also feststellen: Die Anwendungsmöglichkeiten der Verarbeitung von Sequenzen beliebiger Längen ist sehr vielfältig [Kar15]. Mit Hilfe von RNN lassen sich diese und noch andere Anwendungsfälle durchführen. Deshalb sind RNN sehr mächtige Werkzeuge im Rahmen des maschinellen Lernens.

5.3 Lernverfahren

Das Standard-Lernverfahren, um ein *Feedforward*-Netzwerk zu trainieren, ist der *Backpropagation*-Algorithmus (vgl. Kap. 3.2). Beim überwachten Lernen wird dabei versucht, den Netzwerkfehler zu minimieren, wobei eine Kostenfunktion bzw. *Loss*-Funktion zur Berechnung eingesetzt wird, welche die Abweichung zwischen der Netzwerkausgabe und dem tatsächlichen Wert bewertet. Es handelt sich also um ein Optimierungsproblem, das bspw. mit Hilfe des Gradientenabstiegsverfahrens gelöst werden kann.

In RNN, also rekurrenten Netzwerken, kann das Lernverfahren *Backpropagation Through Time* (BPTT) zum Trainieren eingesetzt werden [Wer88]. Dabei wird das Netzwerk nach der Zeit ausgerollt. Dies entspricht dem Ausrollen von vielen RNN-Basiseinheiten (vgl. Kap. 5.1). Die Inputs und Outputs jeder Kopie des ausgerollten Netzwerks sind zwar unterschiedlich, aber es werden die gleichen Parameter (Gewichte, Bias) verwendet, d.h. diese werden somit im Zeitverlauf geteilt. Beim Training des Netzwerks werden die Parameter dann so eingestellt, dass die Kostenfunktion möglichst minimal wird. Dies kann wieder mit Hilfe des Gradientenabstiegsverfahrens erfolgen, also mit dem bereits bekannten *Backpropagation*-Algorithmus. Daher hat der Algorithmus für RNN auch den Namen BPTT bekommen.

Für lange Sequenzen kann dieses RNN als sehr tiefes KNN verstanden werden. Beim Gradientenabstiegsverfahren in tiefen neuronalen Netzen gibt es jedoch das Problem der verschwindenden Gradienten (vgl. Kap. 3.6). Eine Lösung für dieses Problem stellen spezielle Architekturen wie *Long Short-Term Memory* (vgl. Kap. 5.4) oder *Gated Recurrent Unit* (vgl. Kap. 5.5) dar.

5.4 Long Short-Term Memory

Das Netzwerk *Long Short-Term Memory (LSTM)*, zu Deutsch: langes Kurzzeitgedächtnis, ist ein spezielles RNN, das 1997 von Sepp Hochreiter und Jürgen Schmidhuber entwickelt wurde und das Problem verschwindender Gradienten löst [HS97]. Zwei Jahre später wurde noch ein Baustein hinzugefügt, damit auch Informationen wieder vergessen werden können bzw. eine Art *Reset* durchgeführt werden kann [GSC99]. Die Besonderheit einer LSTM-Einheit besteht darin, dass sie eine innere Struktur aufweist (vgl. Abb. 5.6).

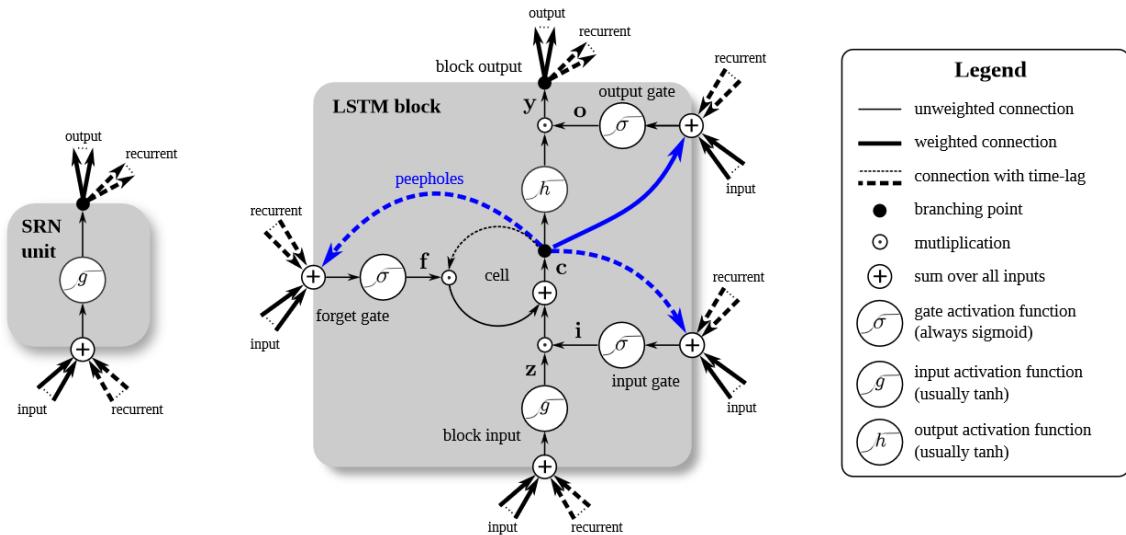


Abbildung 5.6: Aufbau einer LSTM-Einheit [Gre+15]

Eine Zelle ist der zentrale Bestandteil, um Informationen zu verarbeiten und drei Tore bzw. Schranken (engl. *Gates*) steuern und regeln dabei den Informationsfluss:

- | | |
|--------------------|---------------------------------------|
| <i>Input Gate</i> | Informationsfluss in die Zelle |
| <i>Forget Gate</i> | Informationen behalten bzw. vergessen |
| <i>Output Gate</i> | Informationsfluss aus der Zelle |

Die drei *Gates* benutzen die sigmoide Aktivierungsfunktion, während ansonsten für die anderen Elemente der *Tangens Hyperbolicus* verwendet wird. Die ausgerollte LSTM-Einheit ist in Abb. 5.7 dargestellt.

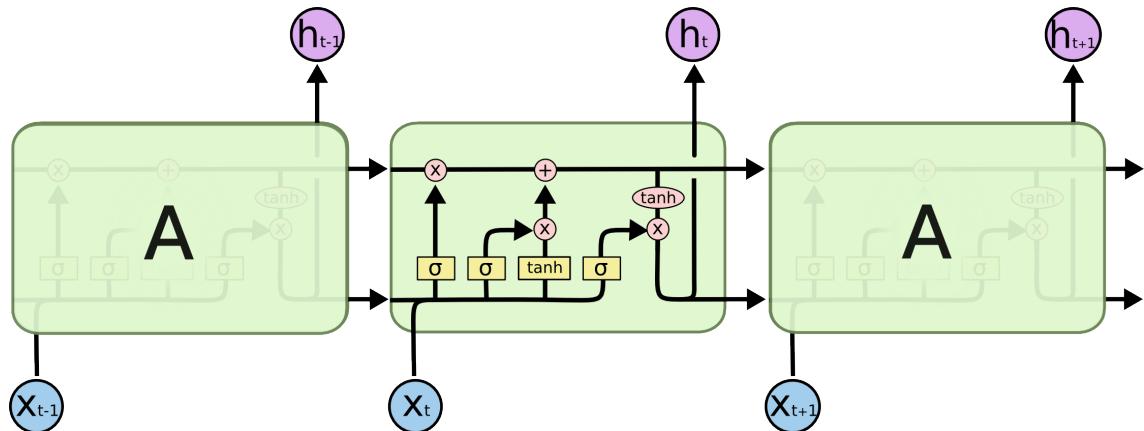


Abbildung 5.7: Ausgerollte LSTM-Einheit [Ola15]

5 Recurrent Neural Network

Die Funktionsweise der LSTM-Einheit ist in den folgenden Abbildungen skizziert.

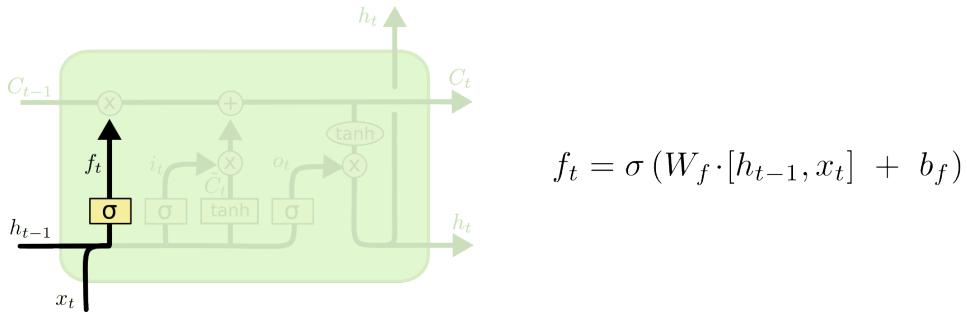


Abbildung 5.8: Schritt 1: Forget-Gate [Ola15]

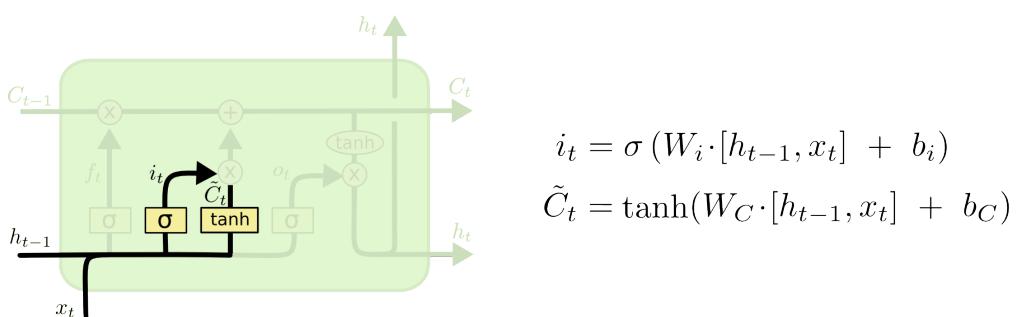


Abbildung 5.9: Schritt 2: Input-Gate [Ola15]

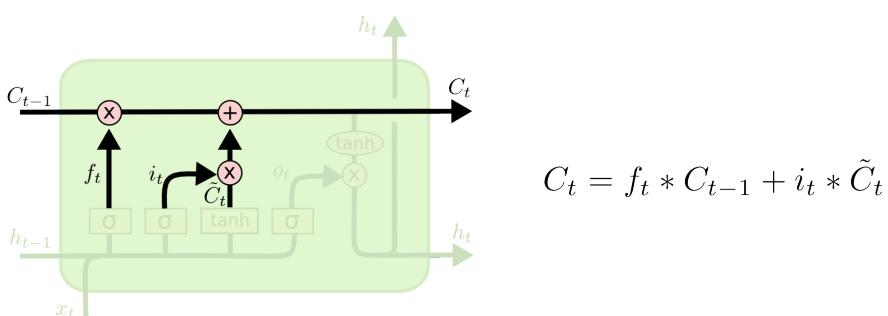


Abbildung 5.10: Schritt 3: Zelle [Ola15]

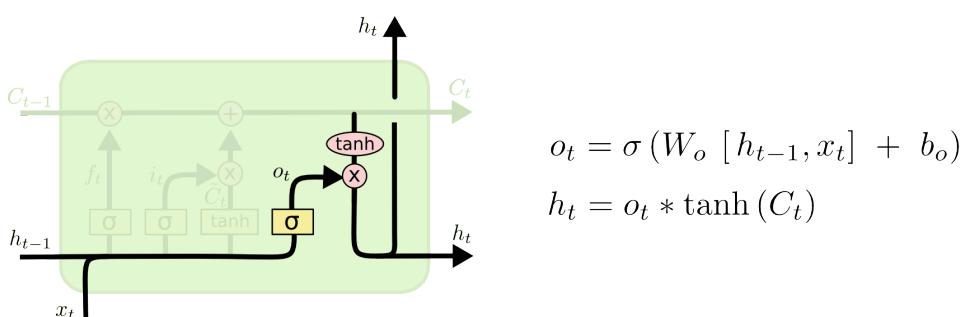


Abbildung 5.11: Schritt 4: Output-Gate [Ola15]

Im ersten Schritt (siehe Abb. 5.8) entscheidet die LSTM-Einheit, welche Informationen der Zelle nicht mehr benötigt werden. Die Eingaben x_t und h_{t-1} werden dabei verarbeitet, wobei h_{t-1} die Ausgabe dieser rekurrenten Zelle im vorherigen Zeitschritt repräsentiert. Die sigmoide Aktivierungsfunktion des *Forget Gate* liefert kontinuierliche Werte zwischen 0 und 1, wobei 0 bedeutet, dass alle Informationen vergessen werden, während 1 bedeutet, dass die Informationen komplett behalten werden. Als Zwischenergebnis wird der Wert f_t bestimmt.

Im nächsten Schritt (siehe Abb. 5.9) entscheidet dann die LSTM-Einheit, welche neuen Informationen in der Zelle gespeichert werden sollen. Auch hier werden wieder die Eingaben x_t und h_{t-1} verarbeitet. Einerseits wird von der sigmoiden Aktivierungsfunktion des *Input Gate* das Zwischenergebnis i_t berechnet, andererseits wird mittels des *Tangens Hyperbolicus* der neue Kandidat für den Zellenwert \tilde{C}_t bestimmt.

In Schritt 3 (siehe Abb. 5.10) wird der Zellenwert C_t nun aktualisiert. Hierzu wird der alte Zellenwert C_{t-1} mit der Ausgabe des *Forget Gate* f_t gewichtet. Der neue Kandidat \tilde{C}_t wird mit dem Zwischenergebnis des *Input Gate* i_t gewichtet. Dann werden beide Teile addiert und als C_t gespeichert.

Im vierten und letzten Schritt (siehe Abb. 5.11) muss die LSTM-Einheit noch entscheiden, welcher Wert schließlich weitergegeben werden soll. Hierzu werden zunächst die ursprünglichen Eingaben x_t und h_{t-1} durch die sigmoiden Aktivierungsfunktion des *Output Gate* geschickt und o_t berechnet. Anschließend wird der *Tangens Hyperbolicus* auf den aktuellen Zellzustand C_t angewendet und die Ausgabe h_t als Produkt (Matrix-Multiplikation) dieser beiden Teilergebnisse berechnet.

5.5 Varianten

Im letzten Abschnitt wurde das Netzwerk *Long Short-Term Memory* vorgestellt. Zu diesem Netzwerk gibt es einige Varianten, je nachdem wie die einzelnen inneren Elemente miteinander verschaltet sind.

Das Besondere am Gucklock-LSTM (engl. *Peephole-LSTM*) ist, dass die *Gates* den Status der Zelle sehen und somit auch Informationen aus der Zelle mitverarbeiten können [GS00]. In Abb 5.12 ist der Aufbau schematisch dargestellt. Der Unterschied zur normalen LSTM-Einheit ist, dass die sigmoiden Aktivierungsfunktionen der drei *Gates* auch die Zellwerte C_{t-1} bzw. C_t als Input verarbeiten.

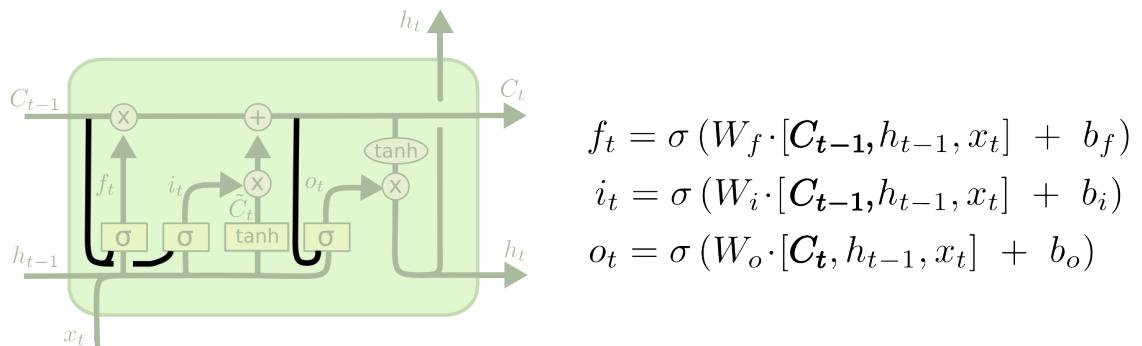


Abbildung 5.12: Peephole-LSTM [Ola15]

5 Recurrent Neural Network

In einer anderen Variante sind *Forget Gate* und *Input Gate* gekoppelt. Dies vereinfacht den Aufbau, da nun nur noch eine sigmoide Aktivierungsfunktion benötigt wird, um den neuen Zellenzustand C_t zu bestimmen. Statt also zwei unabhängige Entscheidungen darüber zu treffen, welche Informationen von x_t und h_{t-1} vergessen bzw. behalten werden sollen, werden diese Entscheidungen in dieser speziellen LSTM-Einheit nun zusammen getroffen. Es werden nur dann alte Informationen vergessen, wenn diese durch neue Informationen ersetzt werden. Oder anders ausgedrückt: Es werden nur dann neue Informationen verwendet, wenn dafür alte Informationen aussortiert werden. In Abb. 5.13 ist der prinzipielle Aufbau dieser LSTM-Variante dargestellt.

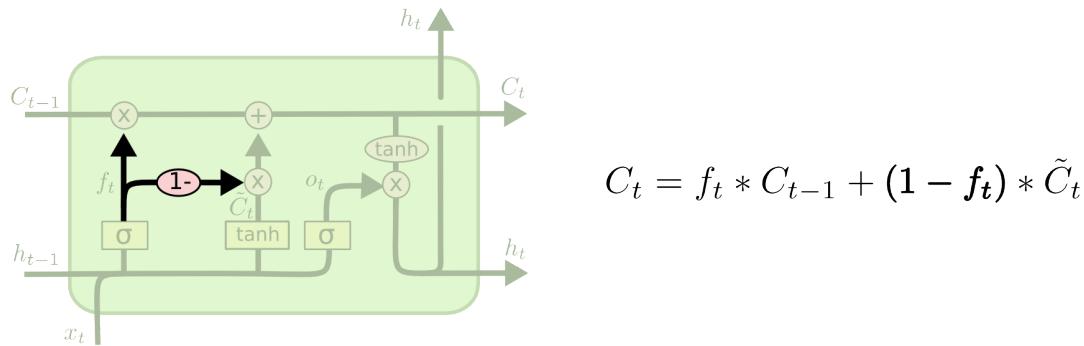


Abbildung 5.13: Gekoppelte LSTM-Einheit [Ola15]

Die bekannteste LSTM-Variante ist die GRU, die *Gated Recurrent Unit* [Cho+14]. Auch in dieser Variante wird auf eine sigmoide Aktivierungsfunktion verzichtet, die Änderungen sind aber gravierender. Das *Forget Gate* und das *Input Gate* werden zu einem *Update Gate* kombiniert. Außerdem werden der Zellzustand C_t und der verborgene Zustand h_t zusammengeführt. Abb. 5.14 zeigt den schematischen Aufbau der GRU. Diese Variante ist einfacher als die Original-LSTM-Einheit aber ähnlich leistungsstark.

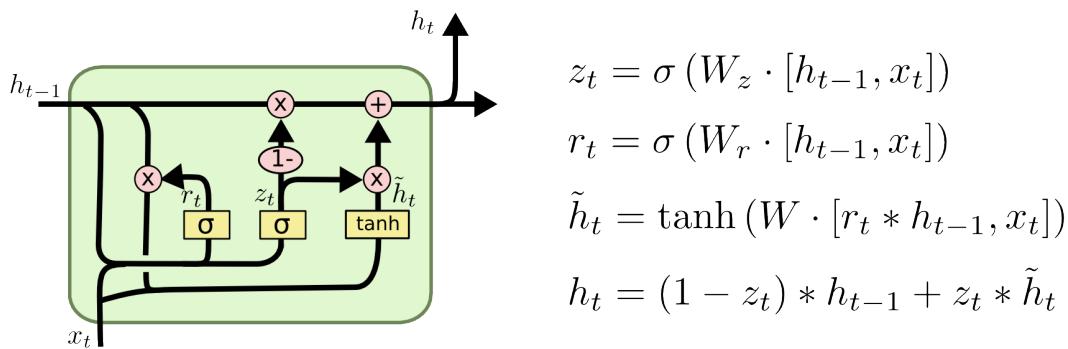


Abbildung 5.14: Gated Recurrent Unit (GRU) [Ola15]

Xingjian Shi und Kollegen der Universität Hongkong haben 2015 die beiden Architekturen CNN und RNN zu einem *Convolutional LSTM* verbunden [Shi+15]. Daneben existieren noch viele andere LSTM-Varianten [Gre+15] [JZS15], die im Rahmen dieser Arbeit aber nicht weiter betrachtet werden .

5.6 Einsatzgebiete

Sehr stark vereinfacht ausgedrückt, werden CNN im Bereich der Bilderkennung und RNN im Bereich der Spracherkennung eingesetzt. Die Anwendungsbreite von RNN im Allgemeinen bzw. von LSTM oder GRU im Speziellen ist eigentlich aber viel größer. Diese Netzerweke können in den folgenden Domänen eingesetzt werden: Robotersteuerung, Zeitreihenanalyse und Prognose, Spracherkennung, Rhythmus-Lernen, Musik-Komponieren, Grammatik-Lernen, Handschriften-Erkennung, Erkennung menschlicher Aktionen, Gebärdensprache-Übersetzung, Homologie-Modellieren von Proteinen, Überwachung von betrieblichen Geschäftsprozessen, Vorhersagen im medizinisch-klinischen Behandlungs pfad usw. [Wik18j; Wik18f].

Trotzdem bleibt die Spracherkennung das primäre Einsatzgebiet. Der Prozess der Spracherkennung kann in mehrere Phasen bzw. Schritte unterteilt werden (siehe Abb. 5.15).

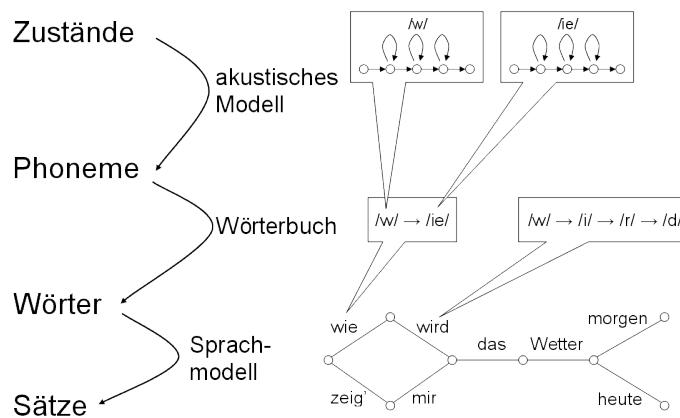


Abbildung 5.15: Modell des Spracherkennungsprozesses [Mwk04]

Zunächst werden die akustischen Eingangssignale analysiert, um Phoneme zu finden. Dies sind die kleinsten unterscheidbaren Einheiten der gesprochenen Sprache. In der deutschen Sprache werden bspw. lange (/a:/) und kurze (/a/) Vokale, nasale (/m/) und stimmlose (/p/) Konsonanten usw. unterschieden. Im nächsten Schritt werden dann aus diesen Lauten mit Hilfe eines Wörterbuchs der zugrundeliegenden Sprache einzelne Wörter gebildet. Das Sprachmodell versucht dann, aus den Wörtern einen Satz zu bilden. Hierbei können bspw. auch Grammatikmodelle oder statistische Analysen eingesetzt werden, denn bestimmte Wortkombinationen treten häufiger auf als andere. Die Spracherkennung beschäftigt sich aber nicht mehr damit, den Sinn und die Bedeutung des gebildeten Satzes zu entschlüsseln. Hierzu kommen dann andere Methoden aus dem Bereich *Natural Language Processing (NLP)* zum Einsatz.

Mit Hilfe des von Ray Kurzweil favorisierten *Hidden Markov Model (HMM)* konnten sehr erfolgreich Phoneme erkannt werden. Die gesprochenen Laute werden dabei als versteckte Zustände aufgefasst, während die tatsächlich hörbaren Töne als Emissionen betrachtet werden (vgl. Abb. 5.1).

Alternativ werden auch KNN in der Spracherkennung verwendet. Diese konnten sich aber erst in den letzten Jahren gegenüber dem HMM durchsetzen, weil es durch *Deep Learning* möglich war, auch große Datenmengen in tiefen KNN in annehmbarer Zeit auf Prozessoren (CPUs und insbes. GPUs) zu verarbeiten. Die präferierten Netzwerk-Architekturen sind LSTM und seine Varianten. Inzwischen setzen die großen und nam-

5 Recurrent Neural Network

haften US-amerikanischen Technologieunternehmen LSTM in ihren digitalen Produkten bzw. intelligenten Assistenten ein [Sch17], also bspw. Apple in Siri [Efr16], Amazon in Polly und in Alexa [Vog16] sowie Google in seine Smartphone-Spracherkennung [Bea15] und in Allo [Kha16].

6 Software

Nachdem nun in den letzten Kapiteln die Grundlagen zum Thema KNN und *Deep Learning* gelegt und drei erfolgreiche Netzwerk-Architekturen vorgestellt wurden, sollen in diesem Kapitel ausgewählte Softwarelösungen zu diesem Thema präsentiert werden. Als Software soll eine *Open Source* Lösung eingesetzt werden. Darunter versteht man Software, deren Quelltext (engl. *Source Code*) öffentlich ist und die von Dritten eingesehen, genutzt und verändert werden darf. Somit fallen für diese Art von Software keine Lizenzgebühren für deren Nutzung an. Die Software ist aber trotzdem nicht kostenlos. Nach dem Konzept *Total Cost of Ownership (TCO)* entfallen bspw. auch Kosten auf Hardware, Installation, Konfiguration, Betrieb, Pflege und Wartung, also für Sachmittel, Personal und Energie. Auf der englischsprachigen Wikipedia-Website https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software [Wik18c] findet man eine Übersicht von Softwarelösungen zum Thema *Deep Learning*. Die meisten dieser Lösungen sind *Open Source* und auf der Internetplattform *GitHub* vorhanden, auf denen Quelltexte öffentlich bereit gestellt werden [Wik18e]. Der Name dieser Plattform setzt sich zusammen aus den Worten *Git* und *Hub*. *Git* ist ein verteiltes Versionsverwaltungssystem [Wik18d]. Ein *Hub* ist der zentrale Knoten in einem Netzwerk. Am 08.03.2018 wurde eine Suche auf GitHub zu dem Begriff *Deep Learning* durchgeführt [Git18]. 28.104 Repositories wurden gefunden. Repositories sind Projektverzeichnisse, die mit Git verwaltet und auf der Plattform bereitgestellt werden. Die meisten dieser Repositories können zu einer Programmiersprache zugeordnet werden. Einige Repositories enthalten aber auch gar keine Quelltexte, sondern sind bspw. als Sammlung nützlicher Links angelegt. Tabelle 6.1 zeigt die Suchergebnisse, aufgeschlüsselt nach den Top 10 Programmiersprachen.

Nr	Sprache	Anzahl	Anteil
1	Python	8.579	41,3 %
2	Jupyter Notebook	7.641	36,8 %
3	HTML	2.392	11,5 %
4	C++	499	2,4 %
5	Matlab	442	2,1 %
6	Java	333	1,6 %
7	JavaScript	294	1,4 %
8	Lua	220	1,1 %
9	TeX	197	0,9 %
10	Cuda	193	0,9 %

Tabelle 6.1: Übersicht von DL-Repositories auf der Plattform GitHub [Git18] (Stand: 08.03.2018)

Es fällt auf, dass die meisten Repositories zu *Deep Learning* in der Programmiersprache Python geschrieben sind (41,4 %). Knapp dahinter folgt Jupyter Notebook mit 36,8 %. Jupyter Notebook ist eigentlich keine eigene Programmiersprache, sondern eine *Open Source* Webapplikation, mit der Dokumente erstellt und ausgetauscht werden können.

6 Software

Sehr häufig wird diese Anwendung benutzt, um Python-Quelltexte zu schreiben und auszuführen. Somit wird sicherlich auch ein großer Teil der Jupyter Notebook Repositories Python-Skripte beinhalten. Hypertext Markup Language (HTML) ist genaugenommen keine Programmiersprache sondern eine Auszeichnungssprache. Andere bekannte Programmiersprachen wie bspw. C++ oder Java spielen also nur eine untergeordnete Rolle. Python ist also die dominante Programmiersprache im Bereich *Deep Learning*.

Die Programmiersprache Python wurde 1991 von Guido van Rossum entwickelt, wird aber mittlerweile von der Python Software Foundation weiterentwickelt und veröffentlicht [Wik18i]. Sie zählt sowohl zu den objektorientierten Programmiersprachen als auch zu den Skriptsprachen. Python zeichnet sich durch seine Einfachheit, Übersichtlichkeit und Erweiterbarkeit aus. Die Sprache kommt mit relativ wenigen Schlüsselwörtern (ver einfacht: Vokabeln) aus. Eine Besonderheit ist, dass Anweisungsblöcke durch Einrückungen strukturiert werden. Mit dem Paketmanager PIP lassen sich sehr leicht Programmbibliotheken als zusätzliche Pakete vom zentralen Repository *Python Package Index (PyPI)* installieren. Im Bereich *Data Science* gibt es eine große Auswahl an wissenschaftlichen Bibliotheken wie z.B. NumPy, Pandas, Matplotlib usw. Die Python-Distribution Anaconda enthält bereits viele dieser Pakete und außerdem das Jupyter Notebook. Python ist für die gängigen Betriebssysteme Linux, Mac OSX und Windows frei erhältlich und wird aktuell in zwei Versionslinien entwickelt. Python 2 gibt es seit dem Jahr 2000 und aktuell wird die Version 2.7.14 ausgeliefert. Python 3 ist seit 2008 verfügbar und 3.6.4 ist die aktuellste Version dieser Linie. Beide Versionslinien sind jedoch nicht kompatibel zueinander, d.h. man muss sich für eine entscheiden. Falls man die freie Wahl hat, sollte man die neuere, also Version 3, benutzen.

In den nächsten Kapiteln werden ausgewählte Repositories und die dahinterliegenden Projekte vorgestellt. Aufgrund der Vielzahl der Repositories ist es unmöglich, alle vorzustellen. Hierzu muss also eine Auswahl getroffen werden. Die Ergebnisse auf GitHub lassen sich nach der besten Übereinstimmung zum Suchbegriff (*Best Match*), den meisten positiven Bewertungen (*Most Stars*), den meisten privaten Repository-Kopien (*Most Forks*) oder der neusten Aktualisierung (*Recently updated*) sortieren. Außerdem kann man statistische Daten zu jedem Repository abrufen, deren Attribute in Tab. 6.2 kurz erklärt sind.

Nr	Attribut	Erklärung
1	Watch	Projekt beobachten
2	Star	Positive Bewertung
3	Fork	Eigene, private Variante des Repositories
4	Commits	Einreichen von Quellcode (neu, geändert)
5	Contributors	Am Projekt beteiligte Mitglieder
6	Issues Open	Frage stellen, Fehler melden
7	Issues Closed	Issue beantwortet
8	Pull Requests Open	Anfrage an den Admin zwecks Commit
9	Pull Requests Closed	Anfrage abgeschlossen
10	Release (Date)	Nr. der Release-Version und Datum
11	Last Commit	Zeitstempel der letzten Änderung

Tabelle 6.2: Übersicht von Attributen zu einem Repository auf GitHub

Tabelle 6.3 zeigt eine Übersicht von ausgewählten Softwarelösungen zum Thema *Deep Learning* in alphabetischer Reihenfolge. Die Anzahl der Sterne (engl. *Stars*) sind positive Bewertungen von Mitgliedern der Plattform GitHub für die jeweilige Softwarelösung (Stand: 08.03.2018).

Nr	Name	Stars	Organisation
1	Caffe	23.110	UC Berkeley
2	Caffe2	7.499	Facebook
3	Chainer	3.561	Preferred Networks
4	CNTK	13.979	Microsoft
5	Darknet	6.115	(Joseph Redmon)
6	Deepwater	252	H2O.ai
7	DL4J	8.447	Skymind
8	Dlib	4.365	(Davis E. King)
9	Gluon	2.094	Apache
10	DSSTNE	4.057	Amazon
11	Keras	26.583	(François Chollet)
12	Lasagne	3.384	Open Community
13	MXNet	13.292	Apache
14	Neon	3.422	Intel
15	OpenNN	442	Artelnics
16	Paddle	6.519	Baidu
17	PyTorch	12.777	-
18	SINGA	1.317	Apache
19	TensorFlow	91.783	Google
20	TFLearn	7.705	Open Community
21	Theano	7.976	Université de Montréal
22	Torch	7.728	-

Tabelle 6.3: Übersicht ausgewählter DL-Softwarelösungen auf der Plattform GitHub [Git18]

Feststellen lässt sich bereits, dass die Softwarelösung *TensorFlow* innerhalb der Plattform GitHub mit deutlichem Abstand am beliebtesten und bekanntesten ist. Die Lösung TF-Learn ist eine Ergänzung zu TensorFlow. Keras ist eine Software, die auf TensorFlow oder Theano basiert. Die Lösungen von großen, namhaften Unternehmen wie Facebook, Microsoft, Amazon, Intel, Baidu und Google sind vertreten. Aus dem wissenschaftlichen Bereich der Universitäten gibt es nur zwei Lösungen: Caffe und Theano. In den nachfolgenden Abschnitten werden diese 22 ausgewählten Bibliotheken und Frameworks nun nacheinander kurz vorgestellt.

6.1 Caffe

Caffe wurde von Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama und Trevor Darrell vom Berkeley Artificial Intelligence Research (BAIR) und Berkeley Vision and Learning Center (BVLC) an der Kalifornischen Universität Berkeley entwickelt [Jia+14]. Die Bibliothek verfügt über eine Vielzahl von Möglichkeiten, Künstliche Neuronale Netzwerke (KNN) aus Schichten (engl. *Layers*) zu erstellen und diese dann mit diversen Algorithmen zu trainieren. Zu den Schichten gehören:

- 1 *Data Layers*
- 2 *Vision (Convolutional) Layers*
- 3 *Recurrent Layers*
- 4 *Common Layers*
- 5 *Normalization Layers*
- 6 *Activation / Neuron Layers*
- 7 *Utility Layers*
- 8 *Loss Layers*

Damit lassen sich u.a. bekannte DL-Architekturen wie MLP, CNN, RNN und LSTM verwenden. Aus Gründen der *Performance* ist die Software in C++ entwickelt worden, wobei CuDNN von Nvidia unterstützt wird. Programmierschnittstellen werden zu Python und Matlab angeboten [JS18]. Das Unternehmen Yahoo hat Caffe in die *Big Data* Anwendung Apache Spark eingebunden, um *Deep Learning* als verteilte Softwarelösung zu verwenden.

Tabelle 6.4 fasst die wichtigsten Merkmale der Caffe-Bibliothek zusammen [She+18].

Name	Caffe
Organisation	UC Berkeley
Webseite	https://github.com/BVLC/caffe
Lizenz	BSD 2-Clause
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C/C++
APIs	Python, Matlab
Statistik vom	08.03.2018
Watch Star Fork	2.156 23.110 14.116
Commits Contributors	4.110 264
Issues: Open Closed	536 3.622
Requests: Open Closed	241 1.872
Release (Date)	1.0 (18.04.2017)
Last Commit	07.03.2018

Tabelle 6.4: Steckbrief zur Caffe-Bibliothek [She+18]

6.2 Caffe2

Caffe2 ist ein leichtgewichtiges, modular aufgebautes und skalierbares Framework für *Deep Learning*, das von Yangqing Jia, Bram Wasti, Pieter Noordhuis, Luke Yeager, Simon Layton, Dmytro Dzhulgakov und anderen auf Basis von Caffe (vgl. Kap. 6.14) beim US-Unternehmen und sozialen Netzwerk Facebook entwickelt wurde [Sou18]. Das Framework ist auf *Performance* optimiert, es verwendet die Bibliotheken cuDNN, cuBLAS und NCCL von Nvidia, d.h. die Rechnungen lassen sich auch auf vielen GPUs ausführen. Programmierschnittstellen werden zu C++ und Python in der Version 2 angeboten, für Python in der Version 3 ist die Unterstützung momentan nur experimentell.

Die Modelle von Caffe lassen sich leicht zu Caffe2 konvertieren und damit wiederverwenden. Hierzu gibt es auch Skripte, die diese Anpassung automatisch ausführen. Statt der *Layers* in Caffe sind in Caffe2 die Operatoren (engl. *Operators*) die zentralen Elemente, mit denen gearbeitet wird (vgl. Abb. 6.1). Es gibt über 400 verschiedene Operatoren in Caffe2.

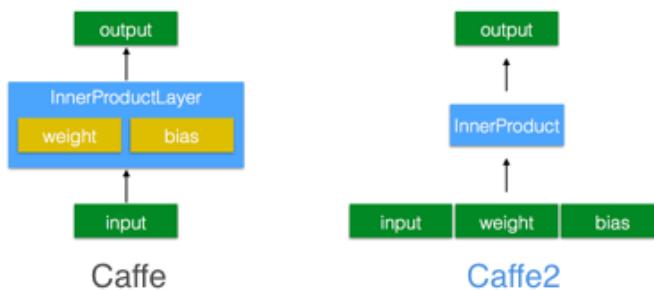


Abbildung 6.1: Vergleich Caffe und Caffe2 [Tok+15]

Tabelle 6.5 fasst die wichtigsten Merkmale des Caffe2-Frameworks zusammen [Jia+18].

Name	Caffe2
Organisation	Facebook
Webseite	https://github.com/caffe2/caffe2
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++
APIs	C++, Python
Statistik vom	08.03.2018
Watch Star Fork	543 7.499 1.763
Commits Contributors	175 3.466
Issues: Open Closed	517 626
Requests: Open Closed	73 980
Release (Date)	0.8.1 (08.08.2017)
Last Commit	08.03.2018

Tabelle 6.5: Steckbrief zum Caffe2-Framework [Jia+18]

6.3 Chainer

Chainer ist ein DL-Framework, das von Yuya Unno, Seiya Tokui, Ryosuke Okuta, Masayuki Takagi und Shunta Saito bei Preferred Networks entwickelt wurde [Net18]. Moderne Netzwerke wie das *Convolutional Neural Network* und das *Recurrent Neural Network* werden unterstützt. Eine Python-API wird zur Programmierung zur Verfügung gestellt. GPU-Unterstützung durch CuPy, welches auf CUDA und cuDNN basiert, wird ebenfalls angeboten [Tok+15]. Modelle, die mit Caffe (vgl. Kap. 6.14) erstellt wurden, lassen sich importieren.

Bei der Entwicklung der Modelle und dem Training der KNN wird eine neuartiger Ansatz verwendet, der sogenannte *Define-by-Run*, der sich gegenüber dem klassischen Verfahren *Define-and-Run* insofern unterscheidet, dass nun dynamische statt statische Graphen zum Einsatz kommen (vgl. Abb. 6.2).

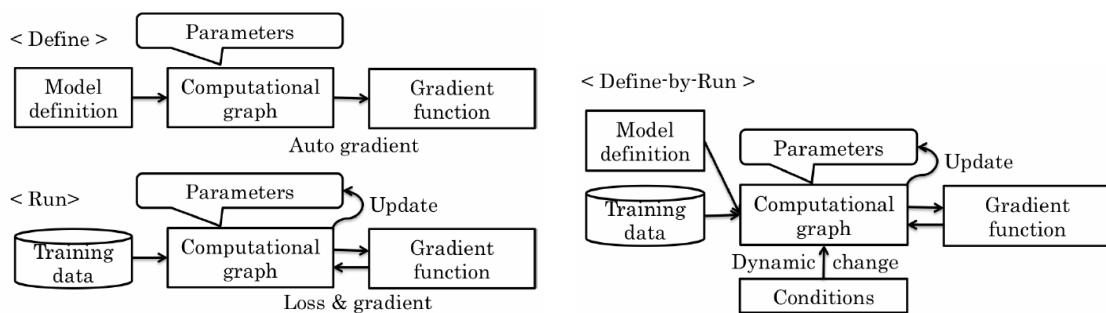


Abbildung 6.2: Vergleich Define-and-Run (links) mit Define-by-Run (rechts) [Tok+15]

Tabelle 6.6 fasst die wichtigsten Merkmale des Chainer-Frameworks zusammen [Unn+18].

Name	Chainer
Organisation	Preferred Networks
Webseite	https://github.com/chainer/chainer
Lizenz	MIT License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	305 3.561 937
Commits Contributors	12.233 157
Issues: Open Closed	135 1.032
Requests: Open Closed	73 3.212
Release (Date)	4.0.0b4 (20.02.2018)
Last Commit	08.03.2018

Tabelle 6.6: Steckbrief zum Chainer-Framework [Unn+18]

6.4 CNTK

Das Microsoft Cognitive Toolkit (CNTK) ist ein DL-Framework, das von Frank Seide, Willi Richert, Mark Hillebrand, Amit Agarwal, Jean Baptiste Faddoul, Zhou Wang, William Darling und anderen bei Microsoft Research entwickelt wurde und seit April 2015 als *Open Source* Lösung angeboten wird [Bas+17]. KNN lassen sich als Serie von Rechnungsschritten in einem gerichteten Graphen modellieren. Es lassen sich MLP, CNN und RNN bzw. LSTM konstruieren und trainieren (z.B. SGD, Backpropagation), auch parallel auf mehreren GPUs. Die Basis-Architektur zum CNTK-Framework ist in Abb. 6.3 dargestellt.

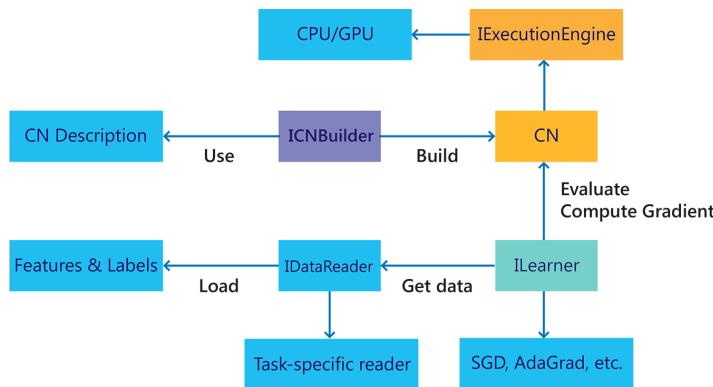


Abbildung 6.3: CNTK-Architektur [Bas+17]

Das CNTK ist zwar in C++ programmiert, für Python existiert aber ebenfalls eine API. Außerdem lässt es sich als *Backend* für das DL-Framework Keras (vgl. Kap. 6.11) verwenden. Insbesondere wird das Austauschformat *Open Neural Network Exchange (ONNX)* unterstützt, mit dem sich bspw. Modelle zwischen den DL-Frameworks CNTK, Caffe2 (vgl. Kap. 6.2), MXNet (vgl. Kap. 6.13) und PyTorch (vgl. Kap. 6.17) austauschen lassen.

Tabelle 6.7 fasst die wichtigsten Merkmale des CNTK-Frameworks zusammen [Sei+18].

Name	CNTK
Organisation	Microsoft
Webseite	https://github.com/Microsoft/CNTK
Lizenz	MIT License
Plattformen	Linux, Windows
Geschrieben in	C++
APIs	C#, C++, Python
Statistik vom	08.03.2018
Watch Star Fork	1.330 13.979 3.705
Commits Contributors	15.538 173
Issues: Open Closed	406 2.217
Requests: Open Closed	47 350
Release (Date)	2.4 (01.02.2018)
Last Commit	08.03.2018

Tabelle 6.7: Steckbrief zum CNTK-Framework [Sei+18]

6.5 Darknet

Mit dem *Darknet* ist nicht der manuelle Zusammenschluss von Rechnern zu einem so genannten *Peer-to-Peer-Overlay-Netzwerk* gemeint, sondern das quelloffene Framework zum Thema Künstliche Neuronale Netzwerke, welches von Joseph C. Redmon in der Programmiersprache C und mit Hilfe der Nvidia-Bibliothek CUDA entwickelt wurde [Red18]. Aus dem Bereich *Deep Learning* werden die gängigen Modelle wie CNN und RNN unterstützt. Die folgenden Beispiele sind vorhanden:

- | | |
|----------------|---------------------------------|
| 1 YOLO | Echtzeit-Objekterkennung |
| 2 ImageNet | Objekt-Klassifizierung |
| 3 Nightmare | Bildgenerierung (vgl. Abb. 6.4) |
| 4 RNN | Textgenerierung |
| 5 DarkGo | Spiel Go |
| 6 Tiny Darknet | Bild-Klassifizierung |
| 7 CIFAR-10 | Bild-Klassifizierung |



Abbildung 6.4: Nightmare: Der Schrei
(Edvard Munch) [Red18]

Tabelle 6.8 fasst die wichtigsten Merkmale des Darknet-Frameworks zusammen [Red+18].

Name	Darknet
Organisation	(Joseph Redmon)
Webseite	https://github.com/pjreddie/darknet
Lizenz	Public Domain
Plattformen	Linux, MacOSX
Geschrieben in	C
APIs	C
Statistik vom	08.03.2018
Watch Star Fork	512 6.115 2.976
Commits Contributors	392 3
Issues: Open Closed	303 90
Requests: Open Closed	87 35
Release (Date)	-
Last Commit	26.11.2017

Tabelle 6.8: Steckbrief zum Darknet-Framework [Red+18]

6.6 Deep Water

Die DL-Bibliothek *Deep Water* wurde von Wen Phan, Magnus Stensmo, Mateusz Dymczyk, Arno Candel und Qiang Kou des Unternehmens H2O.ai als Erweiterung der H2O-Plattform entwickelt [Pha+18]. Als *Backend* lassen sich bspw. Caffe (vgl. Kap. 6.14), MXNet (vgl. Kap. 6.13) oder TensorFlow (vgl. Kap. 6.19) verwenden. Abb. 6.5 zeigt den schematischen Aufbau der skalierbaren Client/Server-Architektur von Deep Water.

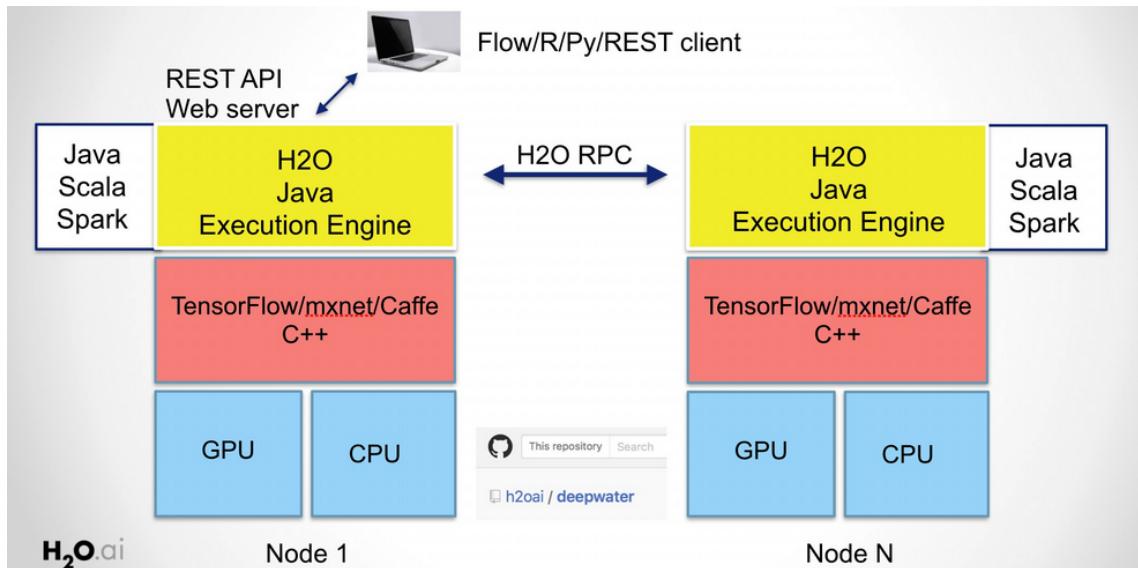


Abbildung 6.5: Deep Water Architektur [Can+18]

Tabelle 6.9 fasst die wichtigsten Merkmale der Deep Water-Bibliothek zusammen [Can+18].

Name	Deep Water
Organisation	H2O.ai
Webseite	https://github.com/h2oai/deepwater
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Java
APIs	Flow, R, Python, Java, Scala, REST
Statistik vom	08.03.2018
Watch Star Fork	99 252 83
Commits Contributors	15 724
Issues: Open Closed	7 32
Requests: Open Closed	0 23
Release (Date)	1.0.1 (21.10.2016)
Last Commit	20.02.2018

Tabelle 6.9: Steckbrief zur Deepwater-Bibliothek [Can+18]

6.7 DL4J

Deep Learning for Java (DL4J) ist eine Bibliothek, die von Alex Black, Adam Gibson, Melanie Warrick, Max Pumperla, Justin Long, Samuel Audet and Eron Wright von Skymind ursprünglich für die Programmiersprache Java entwickelt wurde [Bla+18]. Die mathematischen Kernberechnungen sind in C/C++ und CUDA implementiert. Neben Java werden auch APIs für Scala und Clojure angeboten. Python wird indirekt durch die Verwendung von DL4J als *Backend* in Keras (vgl. Kap. 6.11) unterstützt. Mit Keras ist es sogar möglich, Modelle aus anderen Lösungen wie Caffe (vgl. Kap. 6.14), TensorFlow (vgl. Kap. 6.19), Theano (vgl. Kap. 6.21) oder Torch (vgl. Kap. 6.22) zu importieren.

Verschiedene Typen von Künstlichen Neuronalen Netzwerken lassen sich mit DL4J erstellen und trainieren, die zugehörigen Algorithmen lassen sich auch parallel auf Apache Hadoop und Spark ausführen [GNP+18].

- | | | |
|---|------|---------------------------------------|
| 1 | MLP | <i>Multilayer Perceptron</i> |
| 2 | CNN | <i>Convolutional Neural Network</i> |
| 3 | RNN | <i>Recurrent Neural Network</i> |
| | LSTM | <i>Long Short-Term Memory</i> |
| 4 | GAN | <i>Generative Adversarial Network</i> |
| 5 | RBM | <i>Restricted Boltzmann Machine</i> |
| 6 | DBN | <i>Deep Belief Network</i> |
| 7 | DAE | <i>Deep Autoencoder</i> |
| 8 | SDA | <i>Stacked Denoising Autoencoder</i> |

Tabelle 6.10 fasst die wichtigsten Merkmale der DL4J-Bibliothek zusammen [Bla+18].

Name	Deep Learning for Java
Organisation	Skymind
Webseite	https://github.com/deeplearning4j/deeplearning4j
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C/C++, Java
APIs	Java, Scala, Clojure, (Python)
Statistik vom	08.03.2018
Watch Star Fork	790 8.447 4.075
Commits Contributors	9.468 140
Issues: Open Closed	619 2.033
Requests: Open Closed	7 2.115
Release (Date)	0.9.2 (08.12.2017)
Last Commit	08.03.2018

Tabelle 6.10: Steckbrief zur DL4J-Bibliothek [Bla+18]

6.8 Dlib

Dlib ist eine Bibliothek und ein Werkzeug für Maschinenlernen, welches von Davis E. King in der Programmiersprache C++ bereits ab 2002 entwickelt wurde [Kin09]. Die Software ist Komponentenbasiert aufgebaut (vgl. Abb. 6.6).

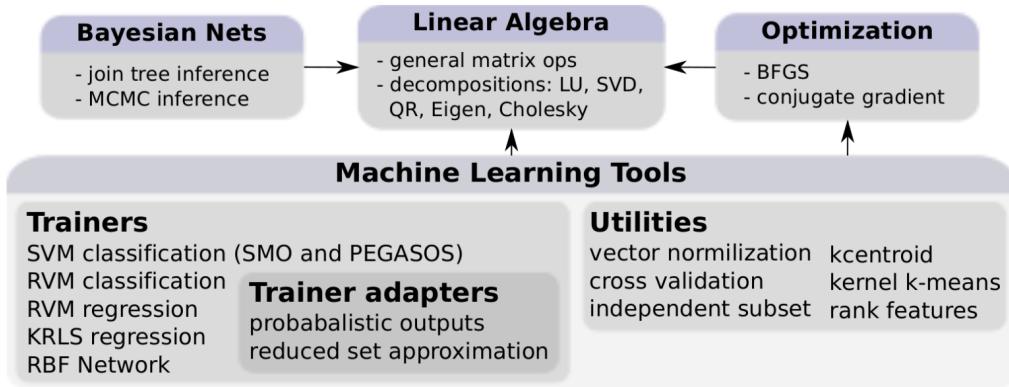


Abbildung 6.6: Komponenten von Dlib [Kin09]

Neben den klassischen Architekturen von Künstlichen Neuronalen Netzwerken wie bspw. Support Vector Machine (SVM), Multilayer Perceptron (MLP) oder RBF-Netze, haben Techniken des *Deep Learning* ebenfalls Einzug in Dlib gefunden [Kin18]. Mittlerweile wird auch eine API für Python bereitgestellt.

Tabelle 6.11 fasst die wichtigsten Merkmale der Dlib-Bibliothek zusammen [Kin+18].

Name	Dlib
Organisation	(Davis E. King)
Webseite	https://github.com/davisking/dlib
Lizenz	Boost Software License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++
APIs	C++, Python
Statistik vom	08.03.2018
Watch Star Fork	339 4.365 1.299
Commits Contributors	7.193 101
Issues: Open Closed	119 819
Requests: Open Closed	6 233
Release (Date)	19.9 (23.01.2018)
Last Commit	04.03.2018

Tabelle 6.11: Steckbrief zur Dlib-Bibliothek [Kin+18]

6.9 DSSTNE

Deep Scalable Sparse Tensor Network Engine (DSSTNE) – gesprochen *Destiny* – wurde von Rejith Joseph, Tristan Penman u.a. bei Amazon entwickelt [JP+18]. Im Mittelpunkt dieser Bibliothek steht daher die *Big Data* Anwendung, personalisierte Produktempfehlungen aus großen Datenmengen zu generieren. Hierzu werden DL-Techniken verwendet und tiefe KNN trainiert, wobei die Berechnungen parallel auf GPUs ausgeführt und dabei die AWS-Dienste EMR und ECS verwendet werden (siehe Abb. 6.7).

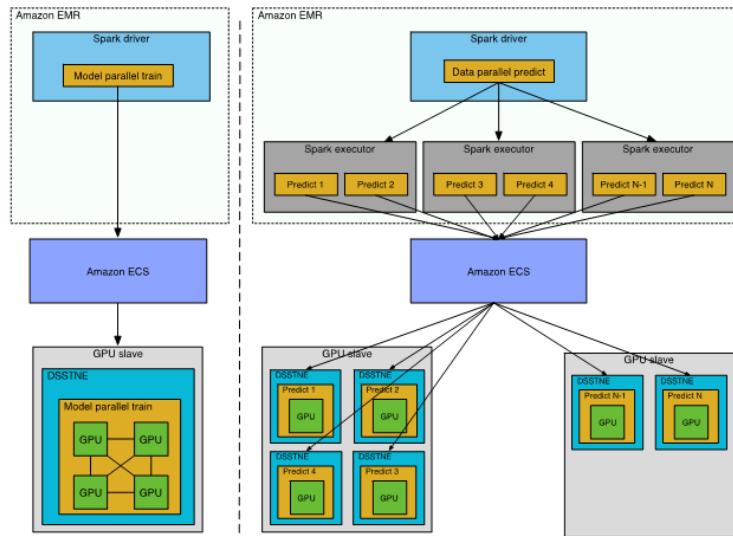


Abbildung 6.7: Architektur mit DSSTNE [Chu16]

Statt einer API werden Skripte bereitgestellt, mit denen sich KNN mittels *Neural Network Layer Definition Language* im JSON-Format beschreiben lassen.

Tabelle 6.12 fasst die wichtigsten Merkmale der DSSTNE-Bibliothek zusammen [JP+18].

Name	DSSTNE
Organisation	Amazon
Webseite	https://github.com/amzn/amazon-dsstne
Lizenz	Apache 2.0
Plattformen	Linux
Geschrieben in	C++
APIs	-
Statistik vom	08.03.2018
Watch Star Fork	349 4.057 666
Commits Contributors	30 287
Issues: Open Closed	27 60
Requests: Open Closed	0 73
Release (Date)	-
Last Commit	01.03.2018

Tabelle 6.12: Steckbrief zur DSSTNE-Bibliothek [JP+18]

6.10 Gluon

Gluon ist ein *High-Level Python-API* für MXNet (vgl. Kap. 6.13), welches von Mitgliedern der Apache Software Foundation entwickelt und von Steffen Rochel auf GitHub bereitgestellt wird [Roc+18]. Selbst komplexe Modelle von Künstlichen Neuronalen Netzwerken lassen sich einfach erstellen und trainieren, wobei prinzipiell immer nach dem gleichen Schema vorgegangen wird:

- 1 Netzwerk definieren
- 2 Parameter initialisieren
- 3 Schleife über die Eingaben
- 4 Propagation, Output berechnen
- 5 Fehler- bzw. Kosten bestimmen
- 6 Backpropagation der Gradienten
- 7 Parameter aktualisieren

Für typische Anwendungen gibt es hierzu bereits die passenden Bausteine wie bspw. vordefinierte *Layers*, *Optimizers*, *Initializers* usw. Die Dokumentation von Gluon enthält wiederum mehrere APIs: Gluon Neural Network Layers, Gluon Recurrent Neural Network API, Gluon Loss API, Gluon Data API, Gluon Model Zoo und Gluon Contrib API [Apa18b]. Außerdem wurde das Online-Buch mit dem Titel *Deep Learning - The Straight Dope* und vielen Code-Beispielen in Gluon erstellt, welches laufend erweitert und aktualisiert wird [Apa18a].

Tabelle 6.13 fasst die wichtigsten Merkmale der Gluon-Bibliothek zusammen [Roc+18].

Name	Gluon
Organisation	Apache Software Foundation
Webseite	https://github.com/gluon-api/gluon-api
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	172 2.094 211
Commits Contributors	13 6
Issues: Open Closed	1 10
Requests: Open Closed	1 4
Release (Date)	-
Last Commit	27.02.2018

Tabelle 6.13: Steckbrief zur Gluon-Bibliothek [Roc+18]

6.11 Keras

Keras ist eine *High-Level Python-API* zum Thema *Deep Learning*, die vom Google-Ingenieur François Chollet im Rahmen eines Forschungsprojekts entwickelt wurde [Cho+18b]. Als *Backend* kann dabei bspw. TensorFlow (vgl. Kap. 6.19), CNTK (vgl. Kap. 6.4), MXNet (vgl. Kap. 6.13) oder Theano (vgl. Kap. 6.21) eingesetzt werden. Keras bietet Bausteine an, mit denen sich Modelle einfach erstellen und wiederverwenden lassen:

<i>layers</i>	Schichten
<i>objectives</i>	Ziel(funktionen)
<i>activation functions</i>	Aktivierungsfunktionen
<i>optimizers</i>	Optimierer
<i>tools</i>	Werkzeuge

Eine Besonderheit von Keras ist auch, dass die entwickelten Modelle sich leicht auf einer Vielzahl von Plattformen ausführen lassen (vgl. Tab. 6.14).

Plattform	Voraussetzung / Umsetzung
iOS	Apple CoreML
Android	TensorFlow Android Runtime
Webbrowser	JavaScript (Keras.js) und WebDNN
Google Cloud	TensorFlow Serving
Python WebApp	Flask App
JVM	DL4J (vgl. Kap. 6.7)
Raspberry Pi	-

Tabelle 6.14: Übersicht zu Keras-Verwendungsmöglichkeiten

Tabelle 6.15 fasst die wichtigsten Merkmale der Keras-Bibliothek zusammen [CRZ+18].

Name	Keras
Organisation	François Chollet
Webseite	https://github.com/keras-team/keras
Lizenz	MIT License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	1.577 26.583 9.698
Commits Contributors	4.402 638
Issues: Open Closed	1.120 5.718
Requests: Open Closed	23 2.726
Release (Date)	2.1.5 (06.03.2018)
Last Commit	08.03.2018

Tabelle 6.15: Steckbrief zur Keras-Bibliothek [CRZ+18]

6.12 Lasagne

Lasagne ist eine leichtgewichtige DL-Bibliothek bzw. *High-Level Python-API*, die von Eric Battenberg, Sander Dieleman, Daniel Nouri, Eben Olson, Aäron van den Oord, Colin Raffel, Jan Schlüter und Søren Kaae Sønderby auf Basis des Theano-Frameworks (vgl. Kap. 6.21) entwickelt wurde [Bat+18]. Analog zu Keras (vgl. Kap. 6.11) werden dem Programmierer bereits viele fertige Bausteine zur Modellierung und zum Trainieren der Künstlichen Neuronalen Netzwerke angeboten, wie z.B. *Layers*, *Regularizers*, *Optimizers* usw. Die Hauptmerkmale von Lasagne bezüglich der Programmierung sind in Tab. 6.16 dargestellt.

Nr	Merkmal	Beispiele
1	KNN	Multi-Input-Multi-Output, MLP
2	DL	CNN, RNN mit LSTM
3	Optimierer	Nesterov Momentum, RMSprop, Adam
4	Kostenfunktion	Automatische symbolische Differentiation
5	Prozessor	CPU und GPU

Tabelle 6.16: Programmieren mit der Lasagne-Bibliothek

Die Optimierer verwenden häufig Gradientenabstiegsverfahren, um die Fehler- bzw. Kostenfunktionen zu minimieren. Dabei müssen die ersten partiellen Ableitungen gebildet werden. Weil als Backend Theano verwendet wird, können mittels symbolischer Differentiation beliebige Kostenfunktionen automatisch partiell abgeleitet werden.

Tabelle 6.17 fasst die wichtigsten Merkmale der Lasagne-Bibliothek zusammen [Sch+18].

Name	Lasagne
Organisation	Open Community
Webseite	https://github.com/Lasagne/Lasagne
Lizenz	MIT License
Plattformen	Linux, MacOSX, (Windows)
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	216 3.384 916
Commits Contributors	1.150 64
Issues: Open Closed	113 388
Requests: Open Closed	25 376
Release (Date)	0.1 (13.08.2015)
Last Commit	01.03.2018

Tabelle 6.17: Steckbrief zur Lasagne-Bibliothek [Sch+18]

6.13 MXNet

MXNet ist ein DL-Framework der Apache Software Foundation, das maßgeblich von Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang und Zheng Zhang entwickelt wurde [Che+15]. Es bietet sehr viele Programmierschnittstellen (APIs) an: C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl und Wolfram [Apa18c]. Weitere Merkmale sind in Tab. 6.18 dargestellt, der schichtenweise, modulare Aufbau ist in Abb. 6.8 skizziert.

Nr	Merkmal	Beispiele
1	Programmierung	Symbolisch & Imperativ
2	DL	CNN, LSTM
3	Prozessor	Multi-CPU und Multi-GPU
4	Cloud Computing	Amazon S3, Apache HDFS, Microsoft Azure
5	Portabilität	Mobile (Amalgamation), IoT (AWS Greengrass)

Tabelle 6.18: Programmieren mit dem MXnet-Framework

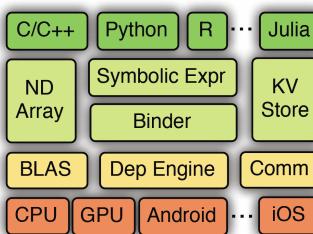


Abbildung 6.8: MXNet: Technologie-Stack [Che+15]

Tabelle 6.19 fasst die wichtigsten Merkmale der MXNet-Bibliothek zusammen [Xie+18].

Name	MXNet
Organisation	Apache Software Foundation
Webseite	https://github.com/apache/incubator-mxnet
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++
APIs	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl
Statistik vom	08.03.2018
Watch Star Fork	1.130 13.292 4.889
Commits Contributors	6.722 490
Issues: Open Closed	762 4.755
Requests: Open Closed	72 4.412
Release (Date)	1.1.0 (19.02.2018)
Last Commit	08.03.2018

Tabelle 6.19: Steckbrief zur MXNet-Bibliothek [Xie+18]

6.14 Neon

Das DL-Framework Neon wurde ursprünglich von Alex Park, Scott Leishman, Anil Thomas, Urs Köster und anderen bei Nervana Systems entwickelt [Par+18], 2016 dann aber vom US-amerikanischen Halbleiterhersteller Intel übernommen. Die typischen KNN-Architekturen wie MLP, CNN, RNN, LSTM und GRU werden unterstützt, aber auch spezielle Formen wie bspw. *Deep Autoencoder (DAE)* können verwendet werden. Ein zentrales Gestaltungsmittel sind dabei viele verschiedene *Layers*-Objekte (vgl. Abb. 6.9), die zu einem Künstlichen Neuronalen Netzwerk angeordnet werden können.

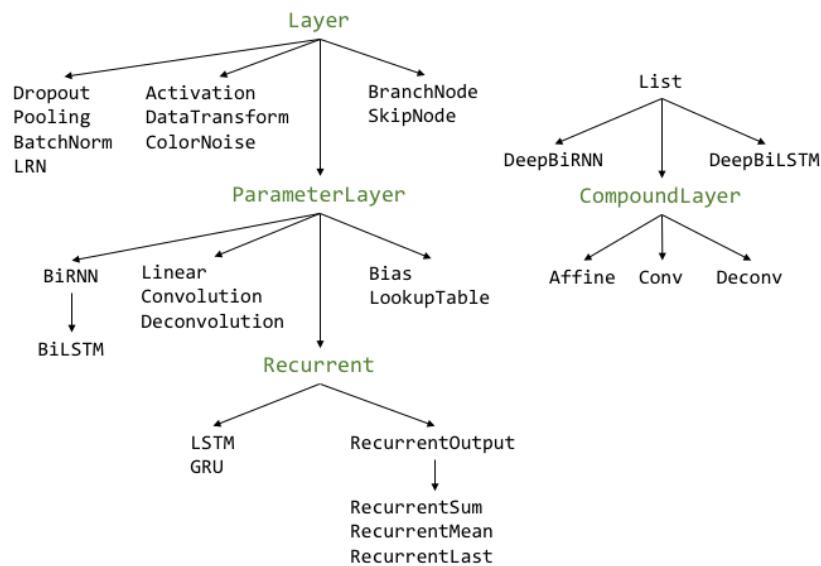


Abbildung 6.9: Neon: Layer Taxonomie [Sys18]

Tabelle 6.20 fasst die wichtigsten Merkmale der Neon-Bibliothek zusammen [Par+18].

Name	Neon
Organisation	Intel
Webseite	https://github.com/NervanaSystems/neon
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C, CUDA, Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	348 3.422 775
Commits Contributors	1.112 78
Issues: Open Closed	64 300
Requests: Open Closed	5 77
Release (Date)	2.6.0 (05.01.2018)
Last Commit	08.02.2018

Tabelle 6.20: Steckbrief zur Neon-Bibliothek [Par+18]

6.15 OpenNN

Die Bibliothek OpenNN für *Advanced Analytics* wurde von Fernando P. Gomez beim Unternehmen Artelnics in der Programmiersprache C++ entwickelt [Gom+18]. Im Mittelpunkt stehen Anwendungen zum Thema *Business Intelligence (BI)* wie z.B. Kundensegmentierung, Abwanderungsanalyse (*Churn Prevention*) und Vorausschauende Wartung (*Predictive Maintenance*) [Art18]. Hierzu werden Künstliche Neuronale Netzwerke eingesetzt. Es lassen sich zwar MLP konstruieren, aber keine CNN, RNN oder LSTM. Abb. 6.10 zeigt das zentrale Klassendiagramm zu der Software-Bibliothek.

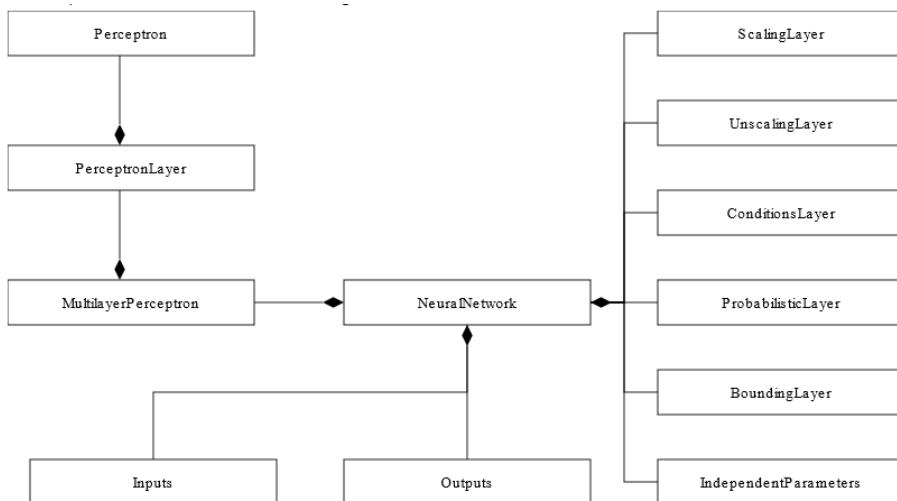


Abbildung 6.10: OpenNN: Klassendiagramm [Art18]

Tabelle 6.21 fasst die wichtigsten Merkmale der OpenNN-Bibliothek zusammen [Gom+18].

Name	OpenNN
Organisation	Artelnics
Webseite	https://github.com/Artelnics/OpenNN
Lizenz	LGPL 3.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++
APIs	C++
Statistik vom	08.03.2018
Watch Star Fork	96 442 156
Commits Contributors	70 5
Issues: Open Closed	12 9
Requests: Open Closed	13 7
Release (Date)	-
Last Commit	20.04.2017

Tabelle 6.21: Steckbrief zur OpenNN-Bibliothek [Gom+18]

6.16 Paddle

Paddle steht für *PArallel Distributed Deep LEarning* – eine DL-Plattform, die von Yu Yang, Qiao Longfei, Tao Luo, QI JUN, JiayiFeng und anderen Wissenschaftlern und Ingenieuren des chinesischen Unternehmens und der gleichnamigen Suchmaschine Baidu in C++ entwickelt wurde [Yan+18]. Die Software unterstützt eine Reihe von KNN-Architekturen, darunter CNN, RNN, LSTM und GRU. Außerdem gibt es bereits einige vordefinierte Modelle mit zugehörigen Beispieldaten, die sich sehr leicht verwenden und anpassen lassen [Bai18]:

1	<i>Word Embedding</i>	Merkalsextraktion (<i>Word Vector Learning</i>)
2	<i>RNN language model</i>	Text-Generierung
3	<i>Click-Through Rate prediction</i>	Erfolg von Online-Werbung vorhersagen
4	<i>Text classification</i>	Stimmungserkennung (<i>Sentiment Analysis</i>)
5	<i>Learning to rank</i>	Empfehlungssystem (<i>Recommender Engine</i>)
6	<i>Semantic model</i>	Suchmaschine
7	<i>Sequence tagging</i>	Eigennamenerkennung (<i>Named Entity Recognition</i>)
8	<i>Sequence to sequence learning</i>	Übersetzung (<i>Neural Machine Translation</i>)
9	<i>Image classification</i>	Objekterkennung

Paddle ist optimiert für mathematische Operationen (*Math Kernel Library (MKL)*, *Basic Linear Algebra Subprograms (BLAS)*) und skalierbar (Multi-CPUs/GPUs). Mit der Python-API lassen sich DL-Projekte wesentlich einfacher umsetzen.

Tabelle 6.22 fasst die wichtigsten Merkmale der Paddle-Bibliothek zusammen [Yan+18].

Name	Paddle
Organisation	Baidu
Webseite	https://github.com/PaddlePaddle/Paddle
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++, Go
APIs	C++, Python
Statistik vom	08.03.2018
Watch Star Fork	551 6.519 1.721
Commits Contributors	12.380 103
Issues: Open Closed	992 3.929
Requests: Open Closed	245 3.721
Release (Date)	0.11.0 (09.12.2017)
Last Commit	08.03.2018

Tabelle 6.22: Steckbrief zur Paddle-Bibliothek [Yan+18]

6.17 PyTorch

PyTorch ist eine *High-Level* Python-API und DL-Bibliothek auf Basis von Torch (vgl. Kap. 6.22), die von Adam Paszke, Soumith Chintala, Gregory Chanan, Sam Gross, Edward Z. Yang, Zachary DeVito, Trevor Killeen und anderen entwickelt wurde [Pas+18]. Diese Softwarelösung unterstützt Tensor-Rechnungen mit starker GPU-Beschleunigung sowie eine spezielle Form des automatischen Differenzierens (*Reverse-Mode Auto-Differentiation*). Im Vergleich zu Torch werden bspw. im CNN die Zwischenzustände nicht in Modulen sondern im Graphen selbst gespeichert (vgl. Abb. 6.11). Dadurch lassen sich Gewichte leichter teilen und Module einfacher wiederverwenden.

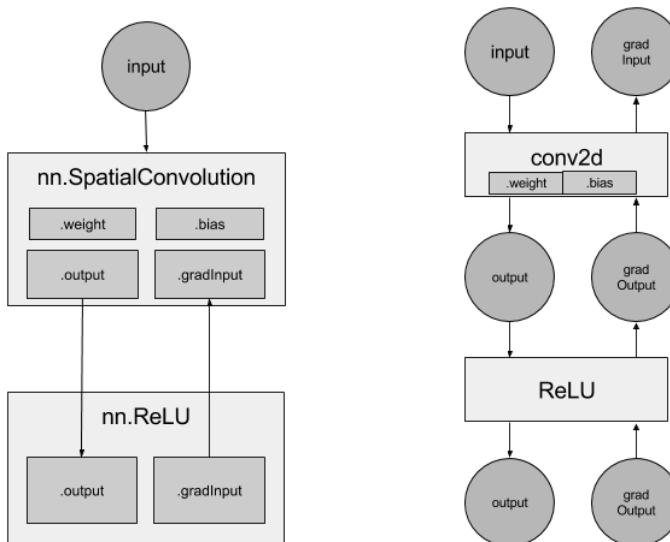


Abbildung 6.11: Vergleich Torch (links) und PyTorch (rechts) [PyT18]

Tabelle 6.23 fasst die wichtigsten Merkmale der PyTorch-Bibliothek zusammen [Pas+18].

Name	PyTorch
Organisation	Open Community
Webseite	https://github.com/pytorch/pytorch
Lizenz	BSD License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	679 12.777 2.720
Commits Contributors	6.594 414
Issues: Open Closed	690 2.152
Requests: Open Closed	116 2.672
Release (Date)	0.3.1 (13.02.2018)
Last Commit	08.03.2018

Tabelle 6.23: Steckbrief zur PyTorch-Bibliothek [Pas+18]

6.18 SINGA

Die DL-Bibliothek SINGA wurde ursprünglich von der *DB System Group* an der *National University of Singapore* in Zusammenarbeit mit der *Database Group* an der *Zhejiang University* entwickelt [Ooi+15] [Wan+15]. Mittlerweile wird die Software durch die *Apache Software Foundation (ASF)* und dort maßgeblich durch Wei Wang, Zhongle Xie und Wang Sheng weiterentwickelt [WX+18]. Abb. 6.12 zeigt den technologischen Software-Stapel.

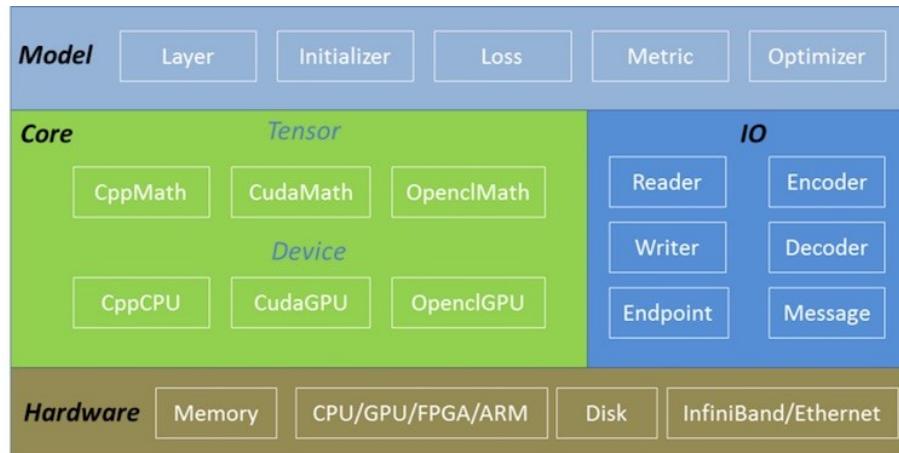


Abbildung 6.12: Apache SINGA Software Stack [Apa18d]

Auf der obersten Ebene stehen dem Programmierer bzw. Anwender die Klassen *Layers*, *Initializer*, *Loss*, *Metric* und *Optimizer* zur Verfügung.

Tabelle 6.24 fasst die wichtigsten Merkmale der SINGA-Bibliothek zusammen [WX+18].

Name	SINGA
Organisation	Apache Software Foundation (ASF)
Webseite	https://github.com/apache/incubator-singa
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++
APIs	C++, Python
Statistik vom	08.03.2018
Watch Star Fork	119 1.317 309
Commits Contributors	863 30
Issues: Open Closed	0 0
Requests: Open Closed	18 332
Release (Date)	1.1.1 (29.07.2017)
Last Commit	22.01.2018

Tabelle 6.24: Steckbrief zur SINGA-Bibliothek [WX+18]

6.19 TensorFlow

Die Software-Bibliothek TensorFlow wurde von Benoit Steiner, Shanqing Cai, Vijay Vasudevan, Derek Murray, Gunhan Gulsoy und anderen des Google Brain Teams entwickelt und ist seit dem 09.11.2015 *Open Source* [Ste+18]. Mathematische Operationen mit Tensoren (bzw. Vektoren, Matrizen) werden graphenbasiert formuliert. Jeder Knoten repräsentiert dabei eine mathematische Operation und jede Kante ein Tensor. Der Fluss (engl. *Flow*) durch den Graphen spiegelt dann die Berechnung wieder [Ten18].

Die TensorFlow-Bibliothek ist in mehreren Ebenen aufgebaut (vgl. Abb. 6.13). Mit Hilfe der *Mid-Level API* lassen sich eigene, tiefe Künstliche Neuronale Netzwerke konstruieren und trainieren. Die *High-Level API* bietet auch bereits vorgefertigte Modelle und typische Anwendungen des Maschinenlernens an. Außerdem ist mit dem TensorBoard ein Werkzeug zum Visualisieren von Daten enthalten.

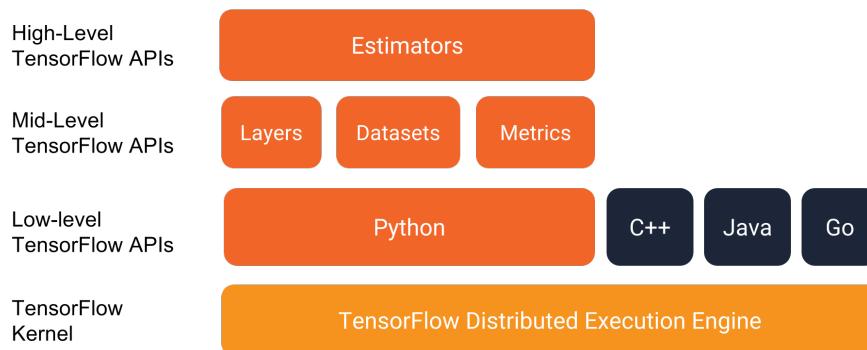


Abbildung 6.13: TensorFlow Technologie-Stapel [Ten18]

Tabelle 6.25 fasst die wichtigsten Merkmale der TensorFlow-Bibliothek zusammen [Ste+18].

Name	TensorFlow
Organisation	Google
Webseite	https://github.com/tensorflow/tensorflow
Lizenz	Apache 2.0
Plattformen	Linux, MacOSX, Windows
Geschrieben in	C++, CUDA
APIs	C++, Python, Java, Go
Statistik vom	08.03.2018
Watch Star Fork	7.555 91.783 59.230
Commits Contributors	29.717 1.357
Issues: Open Closed	1.254 9.397
Requests: Open Closed	203 6.600
Release (Date)	1.6.0 (02.03.2018)
Last Commit	08.03.2018

Tabelle 6.25: Steckbrief zur TensorFlow-Bibliothek [Ste+18]

6.20 TFLearn

TFLearn ist eine DL-Bibliothek und *High-Level* Python-API auf Basis von TensorFlow (vgl. Kap. 6.19), die von Aymeric Damien, Will Ballard und anderen Mitstreitern einer *Open Community* entwickelt wurde [DB+18] [Dam+18]. Künstliche Neuronale Netzwerke lassen sich aus einer Vielzahl von *Layers* konstruieren:

<i>core</i>	input_data, fully_connected, dropout, custom_layer, reshape, flatten, activation, single_unit, highway, one_hot_encoding, time_distributed
<i>conv</i>	conv_2d, conv_2d_transpose, max_pool_2d, avg_pool_2d, conv_1d, conv_3d, max_pool_3d, highway_conv_1d, global_avg_pool, ...
<i>recurrent</i>	simple_rnn, lstm, gru, bidirectionnal_rnn, dynamic_rnn
<i>embedding</i>	embedding
<i>normalization</i>	batch_normalization, local_response_normalization, l2_normalize
<i>merge</i>	merge, merge_outputs
<i>estimator</i>	regression

Des Weiteren lassen sich viele Operatoren verwenden:

<i>activations</i>	linear, tanh, sigmoid, softmax, softplus, softsign, relu, relu6, prelu, elu
<i>objectives</i>	softmax_categorical_crossentropy, mean_square, hinge_loss, ...
<i>optimizers</i>	SGD, RMSProp, Adam, Momentum, AdaGrad, Ftrl, AdaDelta
<i>metrics</i>	Accuracy, Top_k, R2
<i>initializations</i>	zeros, uniform, uniform_scaling, normal, truncated_normal, xavier, ...
<i>losses</i>	l1, l2

Tabelle 6.26 fasst die wichtigsten Merkmale der TFLearn-Bibliothek zusammen [DB+18].

Name	TFLearn
Organisation	Open Community
Webseite	https://github.com/tflearn/tflearn
Lizenz	MIT License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	440 7.705 1.818
Commits Contributors	587 110
Issues: Open Closed	476 325
Requests: Open Closed	16 208
Release (Date)	0.3.2 (18.06.2017)
Last Commit	10.02.2018

Tabelle 6.26: Steckbrief zur TFLearn-Bibliothek [DB+18]

6.21 Theano

Theano ist eine Python-Bibliothek für numerisches Rechnen, die von Frédéric Bastien, Pascal Lamblin, Ian Goodfellow, Aaron Courville, Yoshua Bengio und anderen Wissenschaftlern am *Montreal Institute for Learning Algorithms (MILA)* der Universität von Montréal entwickelt wurde [AR+16]. Das Standardwerk zum Thema *Deep Learning* stammt ebenfalls von dieser Forschungsgruppe [GBC16]. Theano verwendet Graphen, um mathematische Berechnungen symbolisch darzustellen, wobei der Graph aus Variablen, Operatoren und internen *Apply*-Objekten konstruiert wird (vgl. Abb. 6.14).

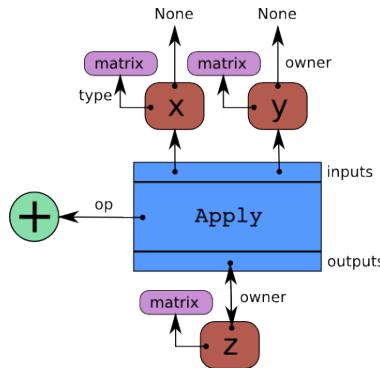


Abbildung 6.14: Theano: Beispiel-Graph mit Apply-Objekt [Ben+17]

Am 28.09.2017 hat Pascal Lamblin eine Nachricht vom Leiter der MILA-Forschungsgruppe, Yoshua Bengio, verbreitet, dass die Weiterentwicklung von Theano nach der Release-Version 1.0 aufgrund von starker Konkurrenz seitens der Industrie eingestellt wird [Lam17]. Damit wird vermutlich auf die Macht von Google und dem Produkt TensorFlow (vgl. Kap. 6.19) angespielt. Die Version 1.0 wurde am 15.11.2017 veröffentlicht. Seitdem wurde nur noch ein *Update* 1.0.1 nachgereicht, das Fehlerkorrekturen enthält.

Tabelle 6.27 fasst die wichtigsten Merkmale der Theano-Bibliothek zusammen [BLG+18].

Name	Theano
Organisation	Université de Montréal
Webseite	https://github.com/Theano/Theano
Lizenz	BSD License
Plattformen	Linux, MacOSX, Windows
Geschrieben in	Python
APIs	Python
Statistik vom	08.03.2018
Watch Star Fork	571 7.976 2.418
Commits Contributors	27.942 328
Issues: Open Closed	529 2.003
Requests: Open Closed	101 3.934
Release (Date)	1.0.1 (07.12.2017)
Last Commit	07.03.2018

Tabelle 6.27: Steckbrief zur Theano-Bibliothek [BLG+18]

6.22 Torch

Torch ist ein Framework für wissenschaftliches Rechnen und Maschinenlernen, das ursprünglich von Ronan Collobert, Koray Kavukcuoglu und Clement Farabet entwickelt wurde und mittlerweile in Version 7 von Soumith Chintala, Keren Zhou, Nicholas Léonard und anderen unterstützt wird [Chi+18]. Als API wird Lua verwendet (vgl. Abb. 6.15). Dies ist eine Programmiersprache, die häufig in Computerspielen zum Einsatz kommt, um Spielcharaktere mittels KI-Algorithmen zu steuern. Mit PyTorch (vgl. Kap. 6.17) gibt es eine Erweiterung, die auch eine Programmierschnittstelle zu Python anbietet.

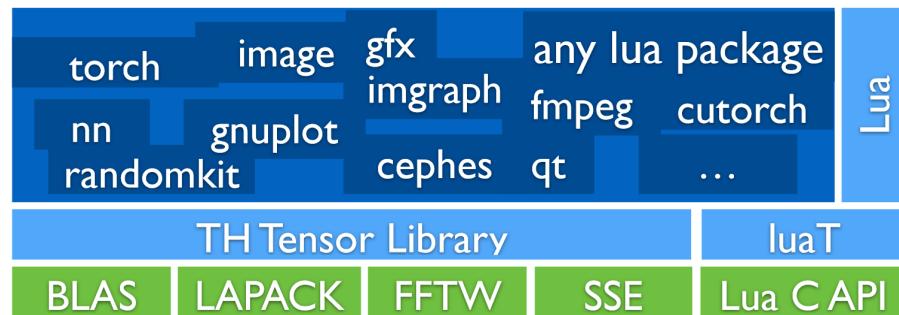


Abbildung 6.15: Torch 7 Technologie-Stapel [Col+18]

Das in Torch 7 enthaltene Paket *nn* kann verwendet werden, um Künstliche Neuronale Netzwerke zu erstellen und zu trainieren. Die Bausteine sind dabei: *Module*, *Container*, *Transfer function*, *Simple layer*, *Table layer*, *Convolution layer* und *Criterion*.

Tabelle 6.28 fasst die wichtigsten Merkmale der Torch-Bibliothek zusammen [Chi+18].

Name	Torch
Organisation	Open Community
Webseite	https://github.com/torch/torch7
Lizenz	BSD License
Plattformen	Linux, MacOSX
Geschrieben in	C, CUDA
APIs	Lua
Statistik vom	08.03.2018
Watch Star Fork	677 7.728 2.250
Commits Contributors	1.335 133
Issues: Open Closed	216 412
Requests: Open Closed	13 494
Release (Date)	-
Last Commit	26.09.2017

Tabelle 6.28: Steckbrief zur Torch-Bibliothek [Chi+18]

7 Anwendungen

In Kapitel 6 wurden 22 Softwarelösungen zum Thema *Deep Learning* vorgestellt. Aufgrund der Statistik der Plattform GitHub nimmt die Lösung TensorFlow (vgl. Kap. 6.19) eine dominante Stellung ein. Einige Softwareentwickler haben aber bereits auch Schwächen bzw. Lücken von TensorFlow identifiziert und versuchen diese nun zu schließen, indem sie eine spezielle *High-Level API* auf Basis von TensorFlow anbieten. Zu diesen Lösungen gehören Deep Water (vgl. Kap. 6.6), Keras (vgl. Kap. 6.11) und TFLearn (vgl. Kap. 6.20). Von diesen drei Lösungen ist Keras wiederum die bekannteste bzw. populärste. Eine gewisse Nähe dieser beiden Software-Lösungen ist unverkennbar, denn Keras wurde vom Google-Ingenieur François Chollet entwickelt.

Um in diesem Kapitel typische Anwendungen im Bereich *Deep Learning* durchzuführen, werden somit die Software-Bibliotheken TensorFlow und Keras verwendet. Das Ziel ist nicht, neue Rekorde hinsichtlich der Genauigkeit und Güte der trainierten Modelle auf den Testdaten zu erzielen. Dies wäre auch gar nicht möglich, weil hierzu ein hoher Aufwand in der Modellbildung und eine enorme Rechenleistung des Systems nötig wäre. Stattdessen wird der Fokus auf die Verwendbarkeit der *High-Level* Bibliothek Keras gerichtet. Die Frage ist also: Ist es möglich, mit nur wenigen Programmzeilen in Python, ein Künstliches Neuronales Netzwerk zu konfigurieren und zu trainieren, sodass die erzielten Ergebnisse, gemessen auf den Testdaten, akzeptabel sind.

Hierzu werden drei typische Problemstellungen betrachtet, die mit Hilfe von mindestens drei unterschiedlichen Typen von KNN mit Hilfe der DL-Bibliothek Keras bearbeitet werden sollen.

1	MNIST	Bilderkennung	Klassifikation	MLP
2	CIFAR-10	Bilderkennung	Klassifikation	CNN
3	IMDb	Textanalyse	Klassifikation	LSTM

Alle drei Anwendungsbeispiele fallen somit in die *Data Mining* Kategorie Klassifikation. Während in den Bildern Objekte erkannt und unterschieden werden sollen, geht es bei der Textanalyse darum, Stimmungen zu erkennen und diese zu klassifizieren. Das Ergebnis eines Klassifizierers, also eines trainierten Modells, welches eine Klassifikation vornimmt, kann bspw. bewertet werden, indem die Genauigkeit auf einer Testdatenmenge bestimmt wird. Diese Testdatenmenge ist disjunkt zur Trainingsdatenmenge, d.h. in der Testdatenmenge sind andere Datensätze enthalten als in der Trainingsdatenmenge. Diese Testdaten sind also dem Modell, bspw. also dem Künstlichen Neuronalen Netzwerk, nicht bekannt. Somit kann dann ein aussagekräftiges Gütekriterium zur Bewertung bzw. Evaluation des Modells auf dieser Testdatenmenge bestimmt werden. Im Fall der Klassifikation ist dies die Genauigkeit (engl. *Accuracy*) in Prozent. Sie ist definiert als Quotient aus der Anzahl der korrekten Klassifikationen zu der Anzahl der Datensätze in der Testdatenmenge. Je näher dieser Wert an 100 % liegt, desto besser ist das Modell. Je nach Anwendungsbereich kann eine Genauigkeit von 80 % bereits eine sehr gute Leistung sein. Die Wahrheitsmatrix bzw. Konfusionsmatrix (engl. *Confusion Matrix*) kann zusätzlich benutzt werden, um die Klassifikationsergebnisse detaillierter zu untersuchen.

7.1 MNIST

Die MNIST-Datenbank (*Modified National Institute of Standards and Technology*) enthält 70.000 Datensätze von Bildern zu handgeschriebenen Ziffern Null bis Neun. Die Datensätze wurden von der ursprünglichen Datenbank des *National Institute of Standards and Technology (NIST)* neu zusammengestellt, weil die Trainings- und Testdaten aus unterschiedlichen Quellen stammten: Die Trainingsdaten kamen von Angestellten des *American Census Bureau*, während die Testdaten durch amerikanische *Highschool*-Studenten erstellt wurden. Außerdem wurden die ursprünglich schwarz-weißen Bilder nun in ein einheitliches Format von 28 x 28 Pixeln in Graustufen mit Kantenglättung (Anti-Aliasing) gebracht [LCB18]. 60.000 Bilder sind zum Trainieren vorgesehen, während 10.000 Bilder zum Testen, d.h. Evaluieren des Modells, benutzt werden können. Abbildung 7.1 zeigt einige Beispielbilder der zehn Ziffern aus der Testmenge der MNIST-Datenbank.



Abbildung 7.1: Beispielbilder des MNIST Test-Datensatzes [Ste17]

Die MNIST-Datenbank ist sehr populär und wird zum Trainieren von Bilderkennungssystemen eingesetzt [Wik18h]. Die Aufgabe fällt in die *Data Mining* Kategorie Klassifikation. Denn das trainierte Modell soll die Bilder zu einer der zehn Klassen (Ziffern Null bis Neun) möglichst korrekt zuordnen. Somit sind die Genauigkeit bzw. die Fehlerrate die Gütekriterien zu den Klassifizierern. Die Anwendung gehört auch zum Gebiet *Optical Character Recognition (OCR)*. Als Algorithmen können bspw. klassische Methoden wie *Naive Bayes*, *K Nearest Neighbor*, Entscheidungsbäume, *Support Vector Machine* eingesetzt oder moderne Techniken des Maschinenlernens wie bspw. Künstliche Neuronale Netzwerke angewendet werden.

1998 haben Yann LeCun, Léon Bottou, Yoshua Bengio und Patrick Haffner in ihrer wissenschaftlichen Studie [LeC+98] u.a. die Ergebnisse von 27 trainierten Klassifizierern (vgl. Abb. 7.2) zur MNIST-Datenbank zusammengestellt. Sehr einfache lineare Klassifizierer kommen dabei auf Fehlerraten im Bereich von 10 %, während der beste Klassifizierer bereits eine Fehlerrate von nur 0,7 % erreicht, also die Ziffern der Testmenge mit einer Genauigkeit von 99,3 % korrekt klassifiziert. Dieser beste Klassifizierer basiert auf einem Ensemble von drei Netzwerken vom Typ *Convolutional Neural Network*. Jedes dieser drei CNNs ist folgendermaßen aufgebaut: Die 784 Eingaben (28 x 28 Pixel) werden auf 4 erste *Feature Maps* und 8 *Subsampling Maps* abgebildet. Danach folgen 16 *Feature Maps* und 16

Subsampling Maps sowie eine vollständig vernetzte Schicht mit 120 Neuronen und schließlich die Ausgabeschicht mit 10 Neuronen für die 10 Klassen. Insgesamt besteht ein solches CNN aus ca. 260.000 Verbindungen und es hat etwa 17.000 freie Parameter.

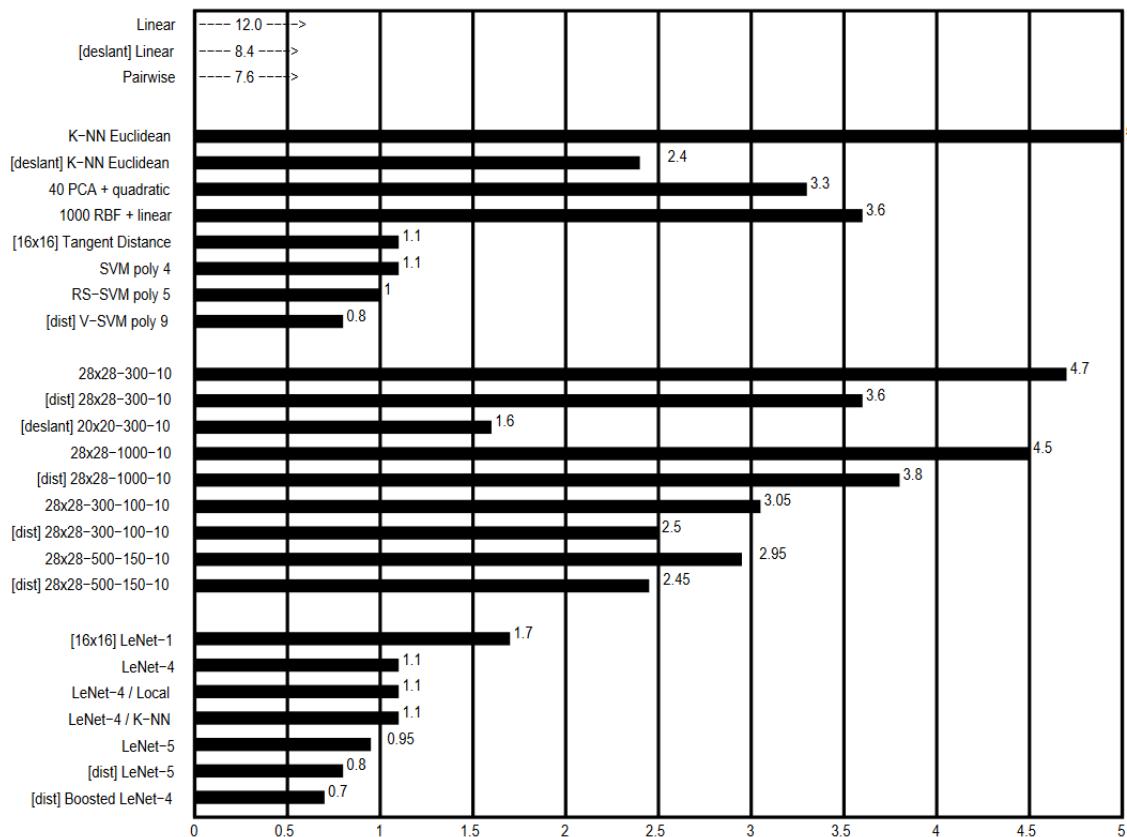


Abbildung 7.2: Fehlerraten für verschiedene Klassifizierer zur MNIST Datenbank [LeC+98]

In den folgenden Jahren konnte die Genauigkeit noch weiter verbessert werden. 2003 erreichten Patrice Y. Simard, Dave Steinkraus und John C. Platt ebenfalls die Fehlerrate von 0,7 %, wobei diesmal allerdings ein einfaches *Multilayer Perceptron* mit nur einer verborgenen Schicht von 800 Neuronen, also dem topologischen Aufbau 784-800-10, erzielt werden konnte [SSP03]. Der Trick bestand u.a. darin, die Anzahl der Trainingsbeispiele zu erhöhen, indem aus den bestehenden Bildern mit Hilfe von elastischen Verzerrungen (engl. *Elastic Distortions*) weitere Beispiele generiert wurden. 2010 haben Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella und Jürgen Schmidhuber mit einem tiefen KNN der Topologie 784-40-80-500-1000-2000-10 eine Fehlerrate von 0,35 % erzielt [Cir+10]. Die gleiche Forschungsgruppe hat 2012 sogar eine Fehlerrate von 0,23 % erreicht [CMS12]. Diesmal allerdings mit einem Ensemble bzw. Komitee von 35 CNNs der Topologie 1-20-P-40-P-150-10.

Die *Data Science* Plattform *Kaggle* hat die MNIST-Klassifikationsaufgabe inzwischen als Wettbewerb in sein Portfolio aufgenommen und beschreibt sie als "Hallo Welt"-Aufgabe im Bereich des maschinellen Sehens in Anspielung auf ein "Hallo Welt"-Computerprogramm als Standard-Beispiel zu einer Programmiersprache [Kag18b].

Mit Hilfe der DL-Bibliothek Keras auf Basis des *Backends* TensorFlow wurde ein einfaches MLP mit der Topologie 784-256-128-10 trainiert. Der Python-Quelltext hierzu ist in dem folgenden Listing gegeben und *inline* kommentiert.

7 Anwendungen

```
# Importiere Keras-Bibliotheken und -Funktionen
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.callbacks import EarlyStopping
# Importiere sonstige Bibliotheken und Funktionen
import matplotlib.pyplot as plt
import time

# Trainingsparameter
batch_size = 128
num_classes = 10
epochs = 100

# Bild-Dimensionen
img_rows, img_cols = 28, 28
input_size = img_rows * img_cols

# Partitionierung: Training vs Test
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Datenkonvertierung 1: X-Werte
x_train = x_train.reshape(x_train.shape[0], input_size)
x_test = x_test.reshape(x_test.shape[0], input_size)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Datenkonvertierung 2: Y-Werte (Klassenvektoren => Binäre Klassenmatrizen)
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Modell
model = Sequential()
model.add(Dense(256, activation='relu', input_shape=(input_size,)))
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy', optimizer=RMSprop(),
              metrics=['accuracy'])

# Training
start_time = time.time()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=
    epochs, verbose=2, validation_data=(x_test, y_test), callbacks=[EarlyStopping(min_delta=0.00001, patience=10)])
end_time = time.time()
print("Zeitdauer[s]: {}".format((end_time - start_time)))

# Test / Validierung
loss, acc = model.evaluate(x_test, y_test, verbose=0)
print("Loss: {}".format(loss))
print("Genauigkeit[%]: {}".format(acc * 100))

# Visualisierung: Genauigkeit
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Modell-Genauigkeit')
```

```

plt.ylabel('Genauigkeit')
plt.xlabel('Epoche')
plt.legend(['Training', 'Test'], loc='lower right')
plt.show()

# Visualisierung: Fehler (Loss-Funktion)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Modell-Fehler (Loss-Funktion)')
plt.ylabel('Fehler bzw. Loss')
plt.xlabel('Epoche')
plt.legend(['Training', 'Test'], loc='upper right')
plt.show()

```

Listing 7.1: MNIST MLP

Der Quelltext ist mehrere Abschnitte gegliedert und durch Überschriften kommentiert. Im Modell wird als Aktivierungsfunktion der Neuronen der Eingabeschicht und der verborgenen Schicht *Rectified Linear Unit (ReLU)* eingesetzt. In der Ausgabeschicht wird dagegen die *Softmax*-Funktion verwendet. Als Gradientenabstiegsverfahren wird der Algorithmus RMSprop benutzt und als Fehlerfunktion bzw. *Loss*-Funktion *Categorical Crossentropy*. Die Anzahl der freien Parameter dieses vollvernetzten Netzwerks setzt sich aus den Gewichten der Verbindungen und den Bias-Werten der Neuronen zusammen und beträgt:

$$(784 + 1) \cdot 512 + (512 + 1) \cdot 128 + (128 + 1) \cdot 10 = 235.146$$

Das Training wird durch *Early Stopping* abgebrochen, wenn sich in 10 Epochen keine Verbesserung gemäß des berechneten Fehlers (*Loss*) auf der Testmenge einstellt. Die Ausgaben auf der Konsole bei Ausführung des obigen Python-Programms sehen wie folgt aus.

```

-----
Layer (type)          Output Shape         Param #
-----
dense_1 (Dense)      (None, 256)          200960
dense_2 (Dense)      (None, 128)           32896
dense_3 (Dense)      (None, 10)            1290
-----
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0

-----
Train on 60000 samples, validate on 10000 samples
Epoch 1/100
- 1s - loss: 0.2633 - acc: 0.9211 - val_loss: 0.1137 - val_acc: 0.9642
Epoch 2/100
- 1s - loss: 0.1025 - acc: 0.9686 - val_loss: 0.0785 - val_acc: 0.9749
Epoch 3/100
- 1s - loss: 0.0680 - acc: 0.9792 - val_loss: 0.0858 - val_acc: 0.9745
Epoch 4/100
- 1s - loss: 0.0496 - acc: 0.9843 - val_loss: 0.0724 - val_acc: 0.9781
Epoch 5/100
- 1s - loss: 0.0370 - acc: 0.9887 - val_loss: 0.0896 - val_acc: 0.9739
Epoch 6/100
- 1s - loss: 0.0287 - acc: 0.9906 - val_loss: 0.0684 - val_acc: 0.9806

```

7 Anwendungen

```
Epoch 7/100
- 1s - loss: 0.0219 - acc: 0.9931 - val_loss: 0.0870 - val_acc: 0.9766
Epoch 8/100
- 1s - loss: 0.0177 - acc: 0.9943 - val_loss: 0.0785 - val_acc: 0.9805
Epoch 9/100
- 1s - loss: 0.0142 - acc: 0.9957 - val_loss: 0.0856 - val_acc: 0.9806
Epoch 10/100
- 1s - loss: 0.0117 - acc: 0.9962 - val_loss: 0.0850 - val_acc: 0.9816
Epoch 11/100
- 1s - loss: 0.0098 - acc: 0.9970 - val_loss: 0.0885 - val_acc: 0.9801
Epoch 12/100
- 1s - loss: 0.0083 - acc: 0.9976 - val_loss: 0.0920 - val_acc: 0.9816
Epoch 13/100
- 1s - loss: 0.0061 - acc: 0.9980 - val_loss: 0.0995 - val_acc: 0.9811
Epoch 14/100
- 1s - loss: 0.0069 - acc: 0.9979 - val_loss: 0.1021 - val_acc: 0.9803
Epoch 15/100
- 1s - loss: 0.0057 - acc: 0.9983 - val_loss: 0.1068 - val_acc: 0.9807
Epoch 16/100
- 1s - loss: 0.0046 - acc: 0.9985 - val_loss: 0.1177 - val_acc: 0.9803
Zeitdauer[s]: 11.67345929145813
Loss: 0.11772279915162692
Genauigkeit[%]: 98.03
```

Für das Training der 16 Epochen dieses Netzwerks werden nur knapp 12 Sekunden auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt dabei bereits 98 %, d.h. die Fehlerrate ist somit 2 %. Abb. 7.3 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten.

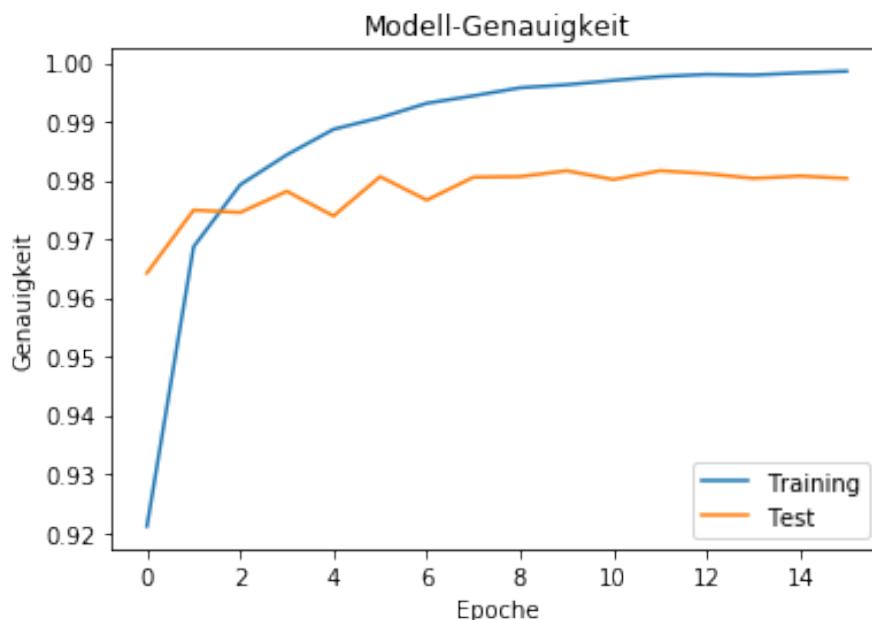


Abbildung 7.3: Genauigkeit des MLP für die Klassifikation der MNIST-Datenbank

In Abb. 7.4 sind die zeitlichen Verläufe der Netzwerkfehler, d.h. der berechnete *Loss*, auf den Trainings- und Testdaten dargestellt.

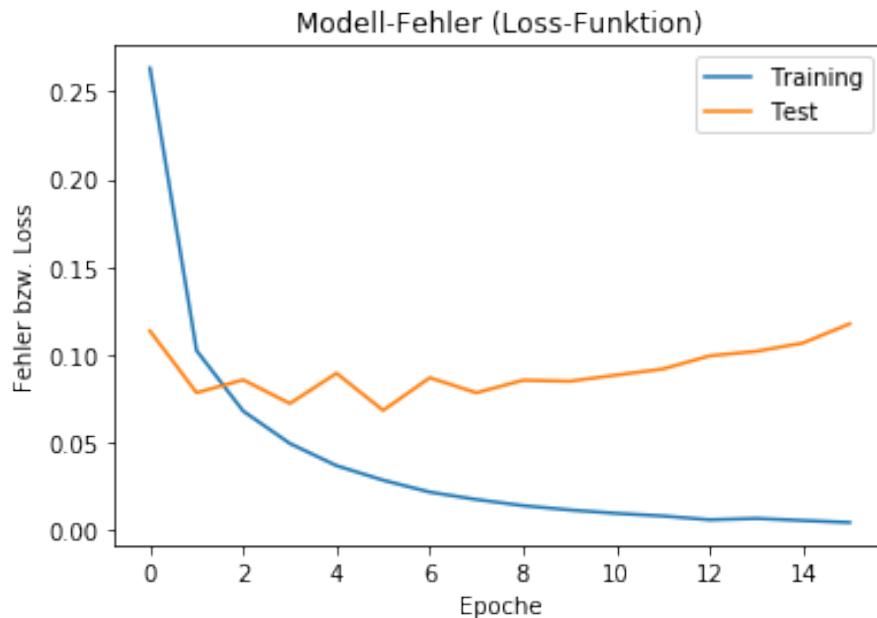


Abbildung 7.4: Fehler bzw. Loss des MLP für die Klassifikation der MNIST-Datenbank

Eine längere Trainingsdauer, gemessen an der Anzahl der Epochen, führt mit dieser Netztopologie nicht unbedingt zu einer Verbesserung der Genauigkeit der Klassifikation auf den Testdaten. Trotzdem ist dies ein gutes Ergebnis, wenn man bedenkt, dass es sich bei dem Künstlichen Neuronalen Netzwerk um ein kleines MLP handelt.

Mit der Verwendung eines *Convolutional Neural Network* kann dieses Ergebnis noch verbessert werden. Der Aufbau des hierzu verwendeten CNN ist in Tab. 7.1 dargestellt.

Schicht	Typ	Maps	Größe	Kernel	Stride	Aktiv.
In	Input	1	28 x 28	-	-	-
C1	Convolution	24	28 x 28	5 x 5	1	ReLU
S2	Max Pooling	24	14 x 14	2 x 2	2	-
C3	Convolution	48	14 x 14	3 x 3	1	ReLU
S4	Max Pooling	48	7 x 7	2 x 2	2	-
F5	Fully Connected	-	2.352	-	-	ReLU
F6	Fully Connected	-	256	-	-	ReLU
Out	Fully Connected	-	10	-	-	Softmax

Tabelle 7.1: Topologie des CNN für die MNIST-Datenbank

Das folgende Listing zeigt nur die Quelltextblöcke, die in dem ursprünglichen Listing geändert werden müssen.

```
# Importiere Keras-Bibliotheken und -Funktionen
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

# Bild-Dimensionen
img_rows, img_cols = 28, 28
input_shape = (img_rows, img_cols, 1)
```

7 Anwendungen

```
# Datenkonvertierung 1: X-Werte
x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)

# Modell
model = Sequential()
model.add(Conv2D(24, kernel_size=(5, 5), strides=(1, 1), padding='same',
    activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(48, kernel_size=(3, 3), strides=(1, 1), padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy', optimizer=RMSprop(),
    metrics=['accuracy'])
```

Listing 7.2: MNIST CNN

Die Konsolen-Ausgaben sehen dann wie folgt aus:

```
-----  
Layer (type)          Output Shape         Param #  
-----  
conv2d_1 (Conv2D)      (None, 28, 28, 24)      624  
-----  
max_pooling2d_1 (MaxPooling2D) (None, 14, 14, 24) 0  
-----  
conv2d_2 (Conv2D)      (None, 14, 14, 48)      10416  
-----  
max_pooling2d_2 (MaxPooling2D) (None, 7, 7, 48) 0  
-----  
flatten_1 (Flatten)    (None, 2352)           0  
-----  
dense_1 (Dense)        (None, 256)            602368  
-----  
dropout_1 (Dropout)    (None, 256)           0  
-----  
dense_2 (Dense)        (None, 10)             2570  
-----  
Total params: 615,978  
Trainable params: 615,978  
Non-trainable params: 0  
-----  
Train on 60000 samples, validate on 10000 samples  
Epoch 1/100  
- 3s - loss: 0.2092 - acc: 0.9356 - val_loss: 0.0500 - val_acc: 0.9855  
Epoch 2/100  
- 2s - loss: 0.0652 - acc: 0.9803 - val_loss: 0.0280 - val_acc: 0.9905  
...  
Epoch 19/100  
- 2s - loss: 0.0168 - acc: 0.9952 - val_loss: 0.0291 - val_acc: 0.9928  
Epoch 20/100  
- 2s - loss: 0.0176 - acc: 0.9949 - val_loss: 0.0269 - val_acc: 0.9935  
Zeitdauer[s]: 40.922513246536255  
Loss: 0.02688839722765547  
Genauigkeit[%]: 99.35000000000001
```

Diesmal müssen also schon 615.978 Parameter trainiert werden. Für das Training von 20 Epochen des CNN werden nun knapp 41 Sekunden auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt dabei nun 99,35 %, d.h. die Fehlerrate liegt somit nur bei 0.65 %. Das ist bereits ein hervorragendes Ergebnis, gemessen an dem Programmier- und Trainingsaufwand. Abb. 7.5 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten. Die zeitlichen Verläufe der Netzwerkfehler (*Loss*) auf den Trainings- und Testdaten sind in Abb. 7.6 dargestellt.

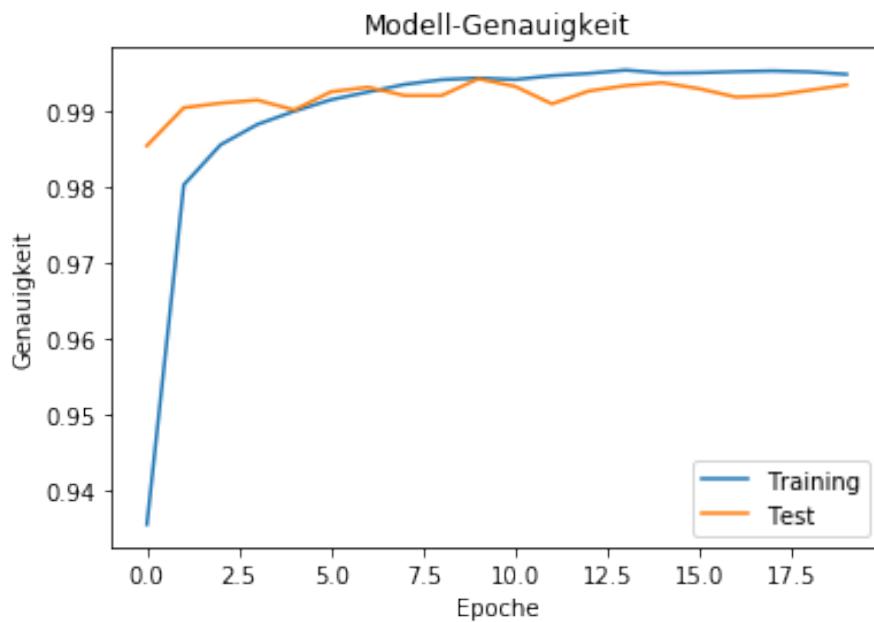


Abbildung 7.5: Genauigkeit des CNN für die Klassifikation der MNIST-Datenbank

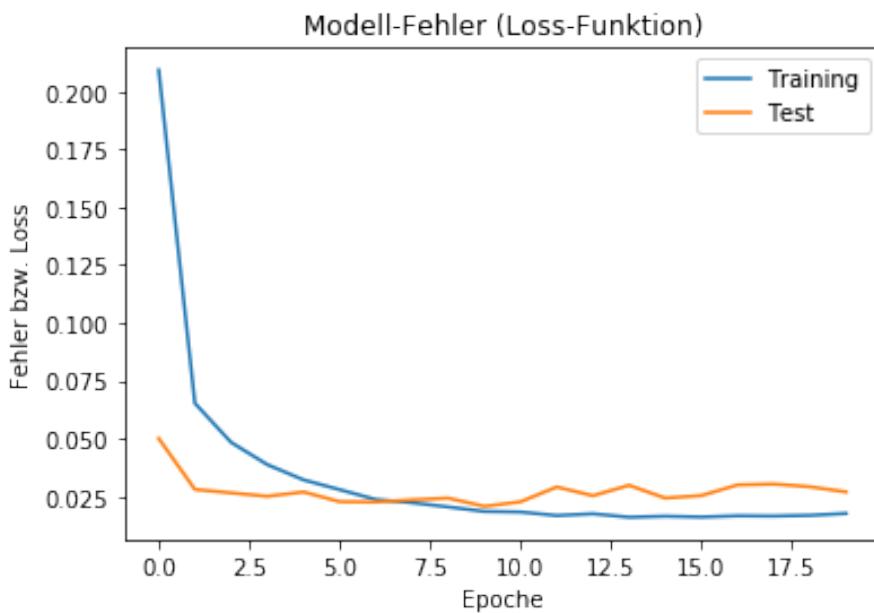


Abbildung 7.6: Fehler bzw. Loss des CNN für die Klassifikation der MNIST-Datenbank

7.2 CIFAR-10

Die CIFAR-10-Datenbank des *Canadian Institute For Advanced Research* ist eine Sammlung von 60.000 Farb-Bildern im Format 32 x 32 Pixeln, die von Alex Krizhevsky, Vinod Nair und Geoffrey E. Hinton zusammengestellt wurde [Kri09]. Die Bilder zeigen 10 unterschiedliche Klassen von Objekten: Flugzeuge, Autos, Vögel, Katzen, Rehe, Hunde, Frösche, Pferde, Schiffe und LKWs. Die Datensätze zu den Klassen sind gleichverteilt, d.h. zu jeder Klasse gibt es 6.000 Bilder. Außerdem ist in jedem Bild nur ein Objekt einer dieser Klassen zu erkennen. Abb. 7.7 zeigt zu diesen 10 Klassen einige Beispielbilder.

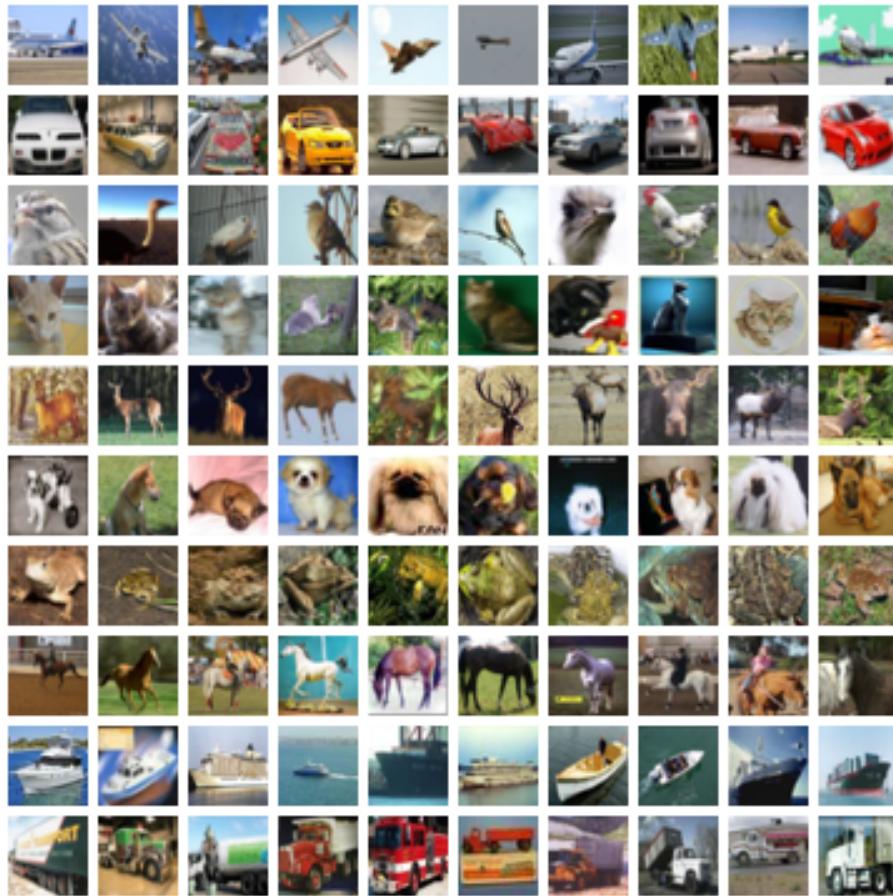


Abbildung 7.7: Beispielbilder der CIFER-10 Datenbank [Kri18]

Ähnlich wie die MNIST-Datenbank (vgl. Kap. 7.1) kann die CIFAR-10-Datenbank im Bereich maschinelles Sehen, insbesondere der Objekterkennung, eingesetzt werden, um neue Modelle und Algorithmen zu entwickeln, zu testen und miteinander zu vergleichen [Wik18b]. Auch die *Data Science* Plattform *Kaggle* bietet zu dieser Datenbank ebenfalls einen Wettbewerb an [Kag18a]. Die Aufgabe ist aber schon wesentlich schwieriger im Vergleich zum MNIST-Datensatz, in dem die zehn Ziffern als Klassen erkannt werden mussten. Mit einem einfachen *Multilayer Perceptron* lässt sich diese Aufgabe nicht adäquat lösen. Hierzu muss schon ein *Convolutional Neural Network* eingesetzt werden. 2010 betrug die Fehlerrate noch 21,1 %, was einer Genauigkeit von 78,9 % entspricht, wobei bereits eine spezielle Art eines CNN, dem *Deep Belief Network (DBN)*, eingesetzt wurde [Kri10]. 2018 konnte die Fehlerrate sogar auf 2,13 % reduziert werden [Rea+18]. Dabei mussten allerdings auch 34,9 Millionen freie Parameter des sogenannten AmoebaNet-B (6, 128) trainiert werden.

Abb. 7.8 zeigt schematisch ein typisches *Convolutional Neural Network* mit seinen verschiedenen Arten von Schichten zur Verarbeitung der Bilddaten der CIFAR-10-Datenbank und zur anschließenden Klassifikation der 10 Objekte.

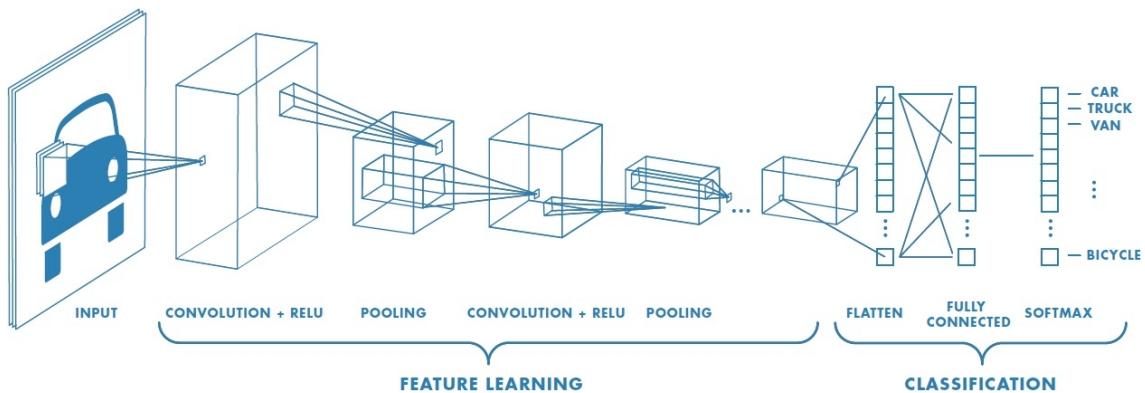


Abbildung 7.8: CNN verarbeitet Bilder der CIFER-10 Datenbank [Mat18]

Tab. 7.2 zeigt die Parameter des Aufbaus des verwendeten *Convolutional Neural Network*.

Schicht	Typ	Maps	Größe	Kernel	Stride	Padding	Aktiv.
In	Input	1	32 x 32	-	-	-	-
C1	Convolution	32	30 x 30	3 x 3	1	valid	ReLU
C2	Convolution	32	30 x 30	3 x 3	1	same	ReLU
P3	Max Pooling	32	15 x 15	2 x 2	2	-	-
C4	Convolution	64	13 x 13	3 x 3	1	valid	ReLU
C5	Convolution	64	13 x 13	3 x 3	1	same	ReLU
P6	Max Pooling	64	6 x 6	2 x 2	2	-	-
C7	Convolution	128	4 x 4	3 x 3	1	valid	ReLU
C8	Convolution	128	4 x 4	3 x 3	1	same	ReLU
P9	Max Pooling	126	2 x 2	2 x 2	2	-	-
F10	Fully Connected	-	512	-	-	-	ReLU
F11	Fully Connected	-	1.024	-	-	-	ReLU
Out	Fully Connected	-	10	-	-	-	Softmax

Tabelle 7.2: Topologie des CNN für die CIFAR-10-Datenbank

Das folgende Python-Programm verwendet wieder die Kombination aus Keras und TensorFlow, um dieses CNN auf die CIFAR-10-Datenbank anzusetzen.

```
# Importiere Keras-Bibliotheken und -Funktionen
import keras
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
```

7 Anwendungen

```
# Importiere sonstige Bibliotheken und Funktionen
import matplotlib.pyplot as plt
import time

# Trainingsparameter
batch_size = 32
num_classes = 10
epochs = 100

# Partitionierung: Training vs Test
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Datenkonvertierung 1: X-Werte
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Datenkonvertierung 2: Y-Werte (Klassenvektoren => Binäre Klassenmatrizen)
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Modell
model = Sequential()
# In, C1, C2, P3
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='valid',
    activation='relu', input_shape=x_train.shape[1:]))
model.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.2))
# C4, C5, P6
model.add(Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='valid',
    activation='relu'))
model.add(Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.2))
# C7, C8, P9
model.add(Conv2D(128, kernel_size=(3, 3), strides=(1, 1), padding='valid',
    activation='relu'))
model.add(Conv2D(128, kernel_size=(3, 3), strides=(1, 1), padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.2))
# F10, F11, Out
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
#
model.summary()
model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=0.0001,
    decay=1e-6), metrics=['accuracy'])

# Training
start_time = time.time()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=
    epochs, verbose=2, validation_data=(x_test, y_test), shuffle=True,
    callbacks=[EarlyStopping(min_delta=0.00001, patience=10)])
end_time = time.time()
print("Zeitdauer[s]: {}".format((end_time - start_time)))
```

```

# Test / Validierung
loss, acc = model.evaluate(x_test, y_test, verbose=0)
print("Loss: {}".format(loss))
print("Genauigkeit [%]: {}".format(acc*100))

# Visualisierung: Genauigkeit
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Modell-Genauigkeit')
plt.ylabel('Genauigkeit')
plt.xlabel('Epoche')
plt.legend(['Training', 'Test'], loc='lower right')
plt.show()

# Visualisierung: Fehler (Loss-Funktion)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Modell-Fehler (Loss-Funktion)')
plt.ylabel('Fehler bzw. Loss')
plt.xlabel('Epoche')
plt.legend(['Training', 'Test'], loc='upper right')
plt.show()

```

Listing 7.3: CIFAR-10 CNN

Die Konsolenausgabe sieht dann wie folgt aus:

Layer (type)	Output Shape	Param #
conv2d_13 (Conv2D)	(None, 30, 30, 32)	896
conv2d_14 (Conv2D)	(None, 30, 30, 32)	9248
max_pooling2d_7 (MaxPooling2)	(None, 15, 15, 32)	0
dropout_5 (Dropout)	(None, 15, 15, 32)	0
conv2d_15 (Conv2D)	(None, 13, 13, 64)	18496
conv2d_16 (Conv2D)	(None, 13, 13, 64)	36928
max_pooling2d_8 (MaxPooling2)	(None, 6, 6, 64)	0
dropout_6 (Dropout)	(None, 6, 6, 64)	0
conv2d_17 (Conv2D)	(None, 4, 4, 128)	73856
conv2d_18 (Conv2D)	(None, 4, 4, 128)	147584
max_pooling2d_9 (MaxPooling2)	(None, 2, 2, 128)	0
dropout_7 (Dropout)	(None, 2, 2, 128)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 1024)	525312
dropout_8 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 10)	10250

7 Anwendungen

```
Total params: 822,570
Trainable params: 822,570
Non-trainable params: 0

-----
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
- 8s - loss: 1.8083 - acc: 0.3184 - val_loss: 1.4979 - val_acc: 0.4422
Epoch 2/100
- 8s - loss: 1.4655 - acc: 0.4573 - val_loss: 1.3259 - val_acc: 0.5153
Epoch 3/100
- 8s - loss: 1.3139 - acc: 0.5226 - val_loss: 1.1934 - val_acc: 0.5682
...
Epoch 45/100
- 8s - loss: 0.2950 - acc: 0.8936 - val_loss: 0.6073 - val_acc: 0.8077
Epoch 46/100
- 8s - loss: 0.2865 - acc: 0.8978 - val_loss: 0.6424 - val_acc: 0.8129
Epoch 47/100
- 8s - loss: 0.2803 - acc: 0.8997 - val_loss: 0.6085 - val_acc: 0.8178
Zeitdauer[s]: 373.5088930130005
Loss: 0.6085280065774917
Genauigkeit[%]: 81.78
```

Für das Training der 47 Epochen hat dieses Netzwerks mit seinen 822.570 freien Parametern etwas mehr als 6 Minuten auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt 81,78 %, d.h. die Fehlerrate ist somit 18,22 % und damit sogar etwas besser als das Ergebnis von Alex Krizhevsky aus dem Jahr 2010. Damals hat das Training allerdings noch 3 Tage und 9 Stunden auf der Grafikkarte Nvidia GTX 280 gedauert [Kri10].

Abb. 7.9 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten.

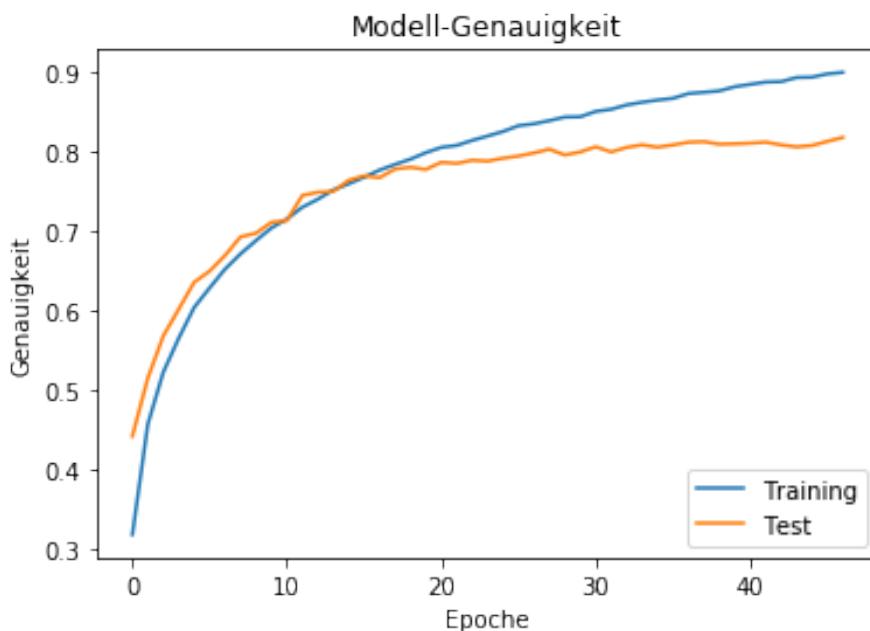


Abbildung 7.9: Genauigkeit des CNN für die Klassifikation der CIFAR-10-Datenbank

Die zeitlichen Verläufe der Netzwerkfehler (*Loss*) auf den Trainings- und Testdaten sind in Abb. 7.10 dargestellt.

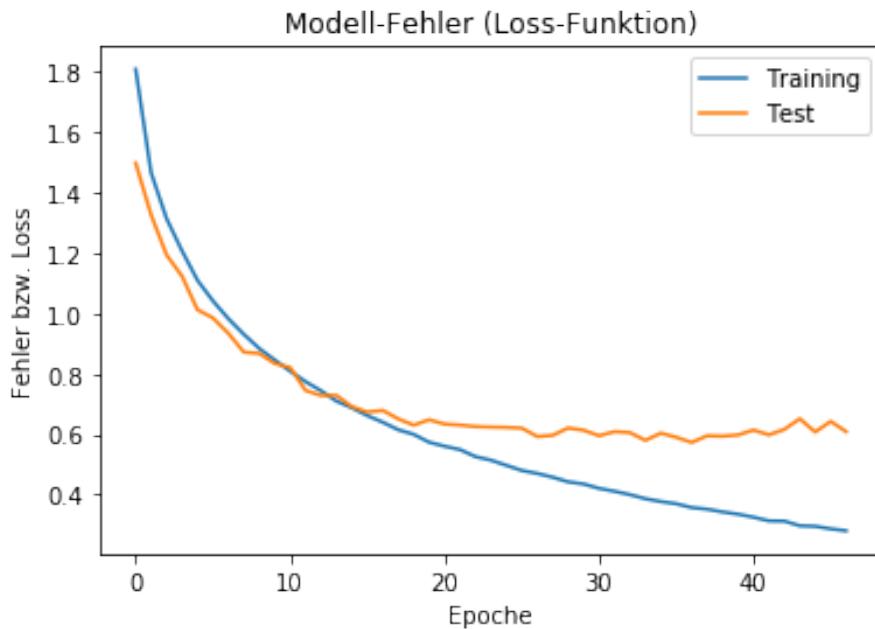


Abbildung 7.10: Fehler bzw. Loss des CNN für die Klassifikation der CIFAR-10-Datenbank

7.3 IMDb

Die *Internet Movie Database (IMDb)* ist die größte Plattform rund um das Thema Film [IMD18a]. Die technische Basis ist eine Datenbank, in der Informationen zu Kino-Filmen, TV-Filmen und -Serien, Videoproduktionen, Computerspielen und Internet-Streams gesammelt werden. Nicht nur Produkte, sondern auch Personen wie Schauspieler, Regisseure, Produzenten usw. werden präsentiert. Die IMDb gehört zum Web 2.0, weil die Nutzer maßgeblich Inhalte erstellen (*User Generated Content*), und zwar in Form von Online-Rezensionen mit Bewertungen. Aktuell (Stand 01.05.2018) gibt es über 91 Millionen registrierte Nutzer und Einträge zu mehr als 4,7 Millionen Titeln und 8,7 Millionen Personen [IMD18b]. Besitzer und Betreiber der Plattform ist der Online-Händler Amazon.

Die verwendeten IMDb-Daten enthalten 50.000 Nutzer-Kritiken (engl. *Reviews*), die jeweils sehr eindeutig eine positive oder negative Meinung vertreten [Cho+18a]. Mit Hilfe dieser Daten lässt sich eine Stimmungserkennung (engl. *Sentiment Analysis*) durchführen [Kag18c]. Diese Analyseart gehört zum Bereich *Text Mining*, ein Teilgebiet des *Data Mining*. Zunächst müssen dabei nämlich unstrukturierte Daten, die Rezensionstexte, verarbeitet werden. Anschließend werden diese dann klassifiziert, wobei es nur zwei Klassen gibt: positive oder negative Rezension. Man spricht deshalb auch von binärer Klassifikation. Des Weiteren kann man diesem Anwendungsbeispiel auch das Etikett *Big Data* geben. Wenn man nämlich alle Rezensionen analysieren würde, wären dies enorme Datenmengen, die verarbeitet werden müssten. Stattdessen wird hier aber nur eine Stichprobe benutzt.

Die 50.000 Daten sind in zwei gleiche Hälften aufgeteilt: Trainings- und Testdaten. Insgesamt werden in diesen Rezensionstexten 85.585 verschiedene Wörter benutzt. Im Mittel besteht ein Rezensionstext aus 234,76 Wörtern mit einer Standardabweichung von 172,91 Wörter [Bro16a]. Die Klassen sind entweder mit dem Etikett (engl. *Label*) 0 für eine negative Rezension oder 1 für eine positive Rezension versehen. Es gibt zu beiden Klassen gleich viele Datensätze. Im Folgenden sind zwei Beispiele dargestellt.

7 Anwendungen

Label 0

My dear Lord, what a movie! Let's talk about the special effects first. Don't get me wrong here, I am not one of those effect fanatics but I was truly thinking that superimposition was a practice of the long gone past, mainly the 60's. So for some time I thought they might have recorded this movie a long time ago and it took them forever to cut and release it. But as far as I know they did not have cell phones in the 60's...

What I am looking for in movies is mainly a good story with a really good message. Acting is secondary, effects are secondary, I do not even mind a few little inconsistencies. However, in a movies like this bad acting, incredibility, etc. add up to make a bad movie even worse - that's what happened for me with the Celestine Prophecy.

My wife said the book was actually really good and even though I am not into all that spiritual stuff I can somehow see that it can be brought across in a believable way - the movie failed to do so.

There could be one single reason to watch this one though. If you really love cheesy movies it'll be the right one for you. If the IMDb stars were for cheesiness instead of quality I MUST have rated this movie ten stars.

By the way, three stars are for the fact that there are worse movies out there, like "Critical Mass" (look up the comments on that one - hilarious). The Celestine Prophecy is at least entertaining to a certain degree.

Label 1

I loved this movie. It is a definite inspirational movie. It fills you with pride. This movie is worth the rental or worth buying. It should be in everyone's home. Best movie I have seen in a long time. It will make you mad because everyone is so mean to Carl Brashear, but in the end it gets better. It is a story of romance, drama, action, and plenty of funny lines to keep you tuned in. I love a lot of the quotes. I use them all the time. They help keep me on task of what I want to do. It shows that anyone can achieve their dreams, all they have to do is work for it. It is a long movie, but every time I watch it, I never notice that it is as long as it is. I get so engrossed in it, that it goes so quick. I love this movie. I watch it whenever I can.

Wenn man sich die Beispiele genauer betrachtet und die Texte liest, stellt man fest, dass die zweite Rezension sehr eindeutig und leicht als positive Kritik erkannt werden kann. Direkt in den ersten Sätzen werden Schlüsselwörter wie *loved, inspirational, pride, worth, best* benutzt. Das erste Beispiel dagegen ist sehr viel schwieriger zu interpretieren, denn es enthält sowohl Schlüsselwörter wie *bad* und *worse* aber genauso *good* und *love*. Außerdem sind im ersten Beispiel noch sogenannte Tags wie bspw.
 enthalten. Dies sind Textformatierungen in *Hypertext Markup Language (HTML)*, die also keinerlei Bedeutung haben. In den Modellen, die in dieser Arbeit für die *Sentiment Analysis* zum Einsatz kommen, werden aber keinerlei Hinweise gegeben, welche (Schlüssel-)Wörter welche Bedeutung haben, auch nicht, ob diese Wörter etwas positiv oder negativ beschreiben.

In Keras sind diese 50.000 IMDb-Datensätze bereits vorverarbeitet. Jeder Rezensionstext ist dabei als Sequenz von Wörtern dargestellt. Jedes Wort wird als ganze Zahl (*Integer*) ausgedrückt, abhängig davon, wie häufig es insgesamt vorkommt. D.h. das häufigste Wort bekommt die Nummer 1, das zweithäufigste die Nummer 2 usw. Die Zahl 0 steht nicht für ein spezielles Wort, sondern für irgendein unbekanntes Wort, das nicht im zugrundeliegenden Wörterbuch enthalten ist. Die Länge dieser Sequenzen ist für jeden Datensatz verschieden, da die Rezensionstexte unterschiedlich lang sind.

Eine der ersten Aufgaben ist es, diese Sequenzen auf die gleiche Länge zu bringen, damit sie weiterverarbeitet werden können. Nach einer festgelegten Länge von Wörtern wird die jeweilige Sequenz einfach abgeschnitten. Falls der Rezensionstext weniger Wörter enthält, dann wird mit Nullwerten aufgefüllt (*Padding*). Ein weiterer, wichtiger Schritt ist die Technik *Word Embedding*. Dabei werden Sequenzen von Wörtern, die als diskrete, ganze

Zahlen repräsentiert sind, auf einen Vektor abgebildet, der aus Elementen von kontinuierlichen Gleitkommazahlen besteht. Dies ist notwendig, weil KNN diese kontinuierlichen Werte besser verarbeiten können.

In den Modellen dieser Arbeit werden nur die 1.000 häufigsten Wörter betrachtet und die Rezensionen werden auf die ersten 100 Wörter gekürzt. Beim *Word Embedding* werden dann die Sequenzen der 100 Wörter auf Vektoren der Dimension 16 abgebildet. Da es sehr viele Trainingsdaten gibt, wird als *batch_size* hier kein Wert zwischen 32 und 256, sondern sogar mit den Längen von 1.024 bzw. 4.096 gearbeitet. Man könnte daher auch von einer *Maxi-Batch* statt von *Mini-Batch* sprechen (vgl. Kap. 3.3). Diese Einstellgröße hat Auswirkungen auf das Lernen und somit muss man mit den Einstellungen des Lernalgorithmus, insbes. der Lernrate, experimentieren. Als Optimierer des Gradientenabstiegs wird Adam verwendet. Vier unterschiedliche Netzwerkmodelle (MLP, CNN, LSTM und GRU) werden ausprobiert und die jeweiligen Quellcodes, Ausgaben und grafischen Ergebnisse im Folgenden dargestellt. Das Training wird abgebrochen, wenn in den letzten 20 Epochen keine signifikante Verbesserung erzielt werden konnte.

MLP Das MLP ist eines der einfachsten, aber auch populärsten KNN (vgl. Kap. 3). In dieser Stimmungsanalyse bzw. Klassifikationsaufgabe wird die Topologie 256-16-1 verwendet. Die Output-Einheit benutzt die sigmoide Aktivierungsfunktion. Dadurch wird das Ergebnis im Intervall $[0, 1]$ abgebildet. Ein Wert nahe null entspricht also einer negativen Rezension, während ein Wert nahe eins für eine positive Kritik steht. Um das Übertrainieren zu vermindern, kommt die *Dropout*-Technik zum Einsatz, in der beim Training zufällig eine vorgegebene Zahl von Parametern (Gewichte, Schwellenwerte) auf null gesetzt wird (vgl. Kap. 2.7).

```
# Importiere Keras-Bibliotheken und -Funktionen
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Embedding, Flatten
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.datasets import imdb

# Importiere sonstige Bibliotheken und Funktionen
import matplotlib.pyplot as plt
import time

# Trainingsparameter
dictionary_length = 1000 # Wörterbuch: Die 1.000 häufigsten Wörter in den Rezensionen
max_review_length = 200 # Betrachte die ersten 200 Wörter je Rezension
embedding_dim = 16 # Jede Rezension wird auf einen Vektor (Dimension 16) abgebildet
batch_size = 1024 # Maxi-Batch
epochs = 1000

# Partitionierung: Training vs Test
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=dictionary_length)

# Datenkonvertierung: X-Werte => Feste Dimension max_review_length
x_train = sequence.pad_sequences(x_train, maxlen=max_review_length)
x_test = sequence.pad_sequences(x_test, maxlen=max_review_length)

# Modell
model = Sequential()
model.add(Embedding(dictionary_length, embedding_dim, input_length=max_review_length))
```

7 Anwendungen

```

model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.summary()
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.001, decay
=1e-5), metrics=['accuracy'])

# Training
start_time = time.time()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=
epochs, verbose=2, validation_data=(x_test, y_test), callbacks=[
EarlyStopping(min_delta=1e-6, patience=20)])
end_time = time.time()
print("Zeitdauer[s]: {}".format((end_time - start_time)))

# Test / Validierung
loss, acc = model.evaluate(x_test, y_test, verbose=0)
print("Loss: {}".format(loss))
print("Genauigkeit[%]: {}".format(acc*100))

# Visualisierung: Genauigkeit
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Modell-Genauigkeit')
plt.ylabel('Genauigkeit')
plt.xlabel('Epoch')
plt.legend(['Training', 'Test'], loc='lower right')
plt.show()

# Visualisierung: Fehler (Loss-Funktion)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Modell-Fehler (Loss-Funktion)')
plt.ylabel('Fehler bzw. Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Test'], loc='upper right')
plt.show()

```

Listing 7.4: IMDB MLP

Ergebnis:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_38 (Embedding)	(None, 200, 16)	16000
<hr/>		
dropout_120 (Dropout)	(None, 200, 16)	0
<hr/>		
flatten_38 (Flatten)	(None, 3200)	0
<hr/>		
dropout_121 (Dropout)	(None, 3200)	0
<hr/>		
dense_109 (Dense)	(None, 256)	819456
<hr/>		
dropout_122 (Dropout)	(None, 256)	0
<hr/>		

```

dense_110 (Dense)           (None, 16)      4112
-----
dropout_123 (Dropout)       (None, 16)      0
-----
dense_111 (Dense)           (None, 1)       17
=====
Total params: 839,585
Trainable params: 839,585
Non-trainable params: 0
-----
Train on 25000 samples, validate on 25000 samples
Epoch 1/1000
- 1s - loss: 0.6940 - acc: 0.5091 - val_loss: 0.6921 - val_acc: 0.5242
Epoch 2/1000
- 0s - loss: 0.6910 - acc: 0.5235 - val_loss: 0.6905 - val_acc: 0.5502
Epoch 3/1000
- 0s - loss: 0.6853 - acc: 0.5526 - val_loss: 0.6759 - val_acc: 0.6090
...
Epoch 36/1000
- 0s - loss: 0.3136 - acc: 0.8684 - val_loss: 0.3288 - val_acc: 0.8596
Epoch 37/1000
- 0s - loss: 0.3123 - acc: 0.8710 - val_loss: 0.3272 - val_acc: 0.8596
Epoch 38/1000
- 0s - loss: 0.3103 - acc: 0.8681 - val_loss: 0.3280 - val_acc: 0.8602
Zeitdauer[s]: 9.14800477027893
Loss: 0.3280345226287842
Genauigkeit[%]: 86.016

```

Für das Training der 38 Epochen des MLP werden knapp 10 Sekunden auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt 86 %, d.h. die Fehlerrate ist somit 14 %. Beim zufälligen Raten und der Gleichverteilung beider Klassen würde man eine Genauigkeit von 50 % erzielen. Da die *IMDb Sentiment Analysis* aber keine Standard-Aufgabe oder Bestandteil eines Wettbewerbs ist, wie dies bspw. bei den MNIST-Daten (vgl. Kap. 7.1) oder den CIFAR-10-Daten (vgl. 7.2) der Fall ist, fehlen aussagekräftige Referenzwerte zum Vergleichen der Ergebnisse. Abb. 7.11 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten.

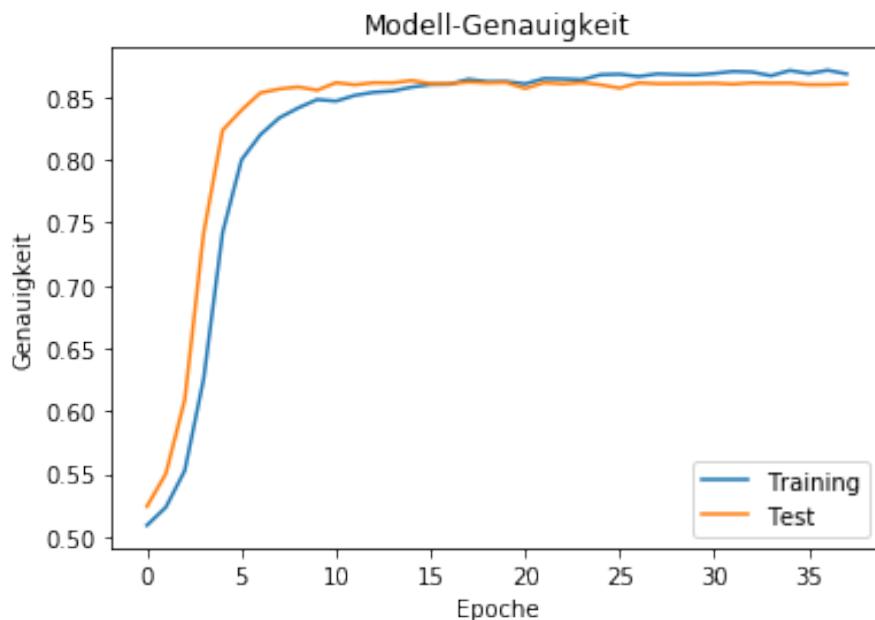


Abbildung 7.11: Genauigkeit des MLP für die Klassifikation der IMDb-Datenbank

7 Anwendungen

In Abb. 7.12 sind die zeitlichen Verläufe der Netzwerkfehler, d.h. der berechnete *Loss*, auf den Trainings- und Testdaten dargestellt.

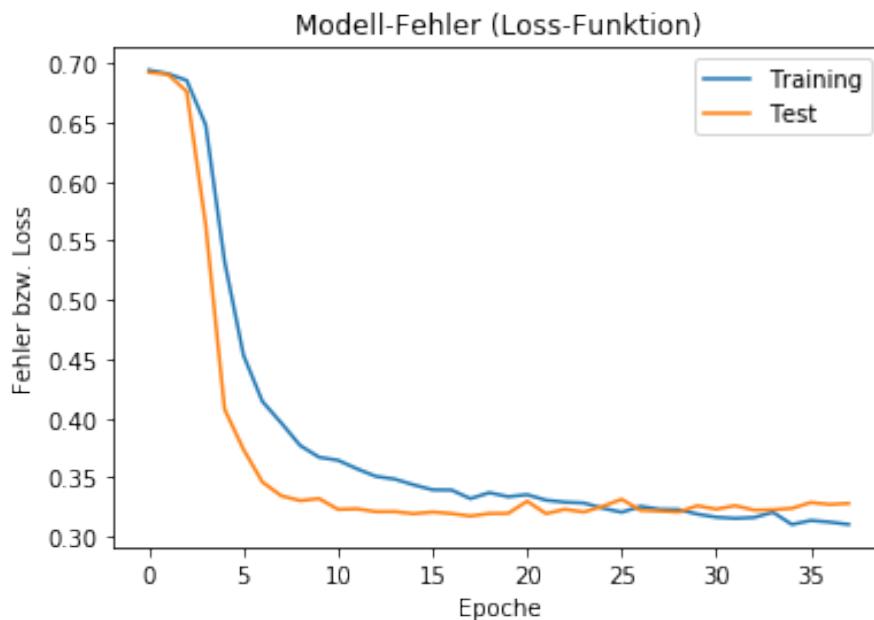


Abbildung 7.12: Fehler bzw. Loss des MLP für die Klassifikation der IMDb-Datenbank

CNN Ein *Convolutional Neural Network* wird meistens im Bereich Bilderkennung eingesetzt (vgl. Kap. 4). Trotzdem kann man es auch verwenden, um aus den Rohdaten sogenannte *Features* zu generieren und auf Basis dieser *Features* eine Klassifizierung vorzunehmen. In dieser *Sentiment Analysis* können diese *Features* als Kombinationen von Wörtern interpretiert werden, die häufig zusammen verwendet werden, um entweder eine positive oder eine negative Stimmung auszudrücken. Hierzu wird ein eindimensionaler *Convolutional Layer* mit 128 *Feature Maps* und einem *Kernel* der Dimension 3 verwendet. Auf dieser Schicht folgt dann ein dazu passender *Pooling Layer*. Zur Aktivierung werden ReLUs verwendet. Im folgenden Quelltext sind nur die Änderungen gegenüber dem ersten Modell dargestellt.

```
# Importiere Keras-Bibliotheken und -Funktionen
from keras.layers import Dense, Dropout, Embedding, Flatten, Conv1D,
GlobalMaxPooling1D

# Modell
model = Sequential()
model.add(Embedding(dictionary_length, embedding_dim, input_length=
    max_review_length))
model.add(Dropout(0.3))
model.add(Conv1D(128, 3, padding='valid', activation='relu', strides=1))
model.add(GlobalMaxPooling1D())
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.001, decay
=1e-5), metrics=['accuracy'])
```

Listing 7.5: IMDB CNN

Ergebnis:

```

-----  

Layer (type)          Output Shape       Param #  

=====  

embedding_29 (Embedding)    (None, 200, 16)      16000  

-----  

dropout_57 (Dropout)        (None, 200, 16)      0  

-----  

conv1d_29 (Conv1D)         (None, 198, 128)     6272  

-----  

global_max_pooling1d_28 (GlobalMaxPooling1D) (None, 128)      0  

-----  

dense_57 (Dense)           (None, 16)            2064  

-----  

dropout_58 (Dropout)        (None, 16)            0  

-----  

dense_58 (Dense)           (None, 1)             17  

=====  

Total params: 24,353  

Trainable params: 24,353  

Non-trainable params: 0  

-----  

Train on 25000 samples, validate on 25000 samples  

Epoch 1/1000  

- 1s - loss: 0.6922 - acc: 0.5311 - val_loss: 0.6892 - val_acc: 0.7021  

Epoch 2/1000  

- 0s - loss: 0.6824 - acc: 0.6437 - val_loss: 0.6640 - val_acc: 0.7370  

Epoch 3/1000  

- 0s - loss: 0.6234 - acc: 0.7354 - val_loss: 0.5544 - val_acc: 0.7790  

...
Epoch 54/1000  

- 0s - loss: 0.2447 - acc: 0.9023 - val_loss: 0.3099 - val_acc: 0.8704  

Epoch 55/1000  

- 0s - loss: 0.2379 - acc: 0.9067 - val_loss: 0.3186 - val_acc: 0.8693  

Epoch 56/1000  

- 1s - loss: 0.2388 - acc: 0.9072 - val_loss: 0.3115 - val_acc: 0.8709  

Zeitdauer[s]: 27.976366996765137  

Loss: 0.3114780087661743  

Genauigkeit[%]: 87.092

```

Für das Training der 56 Epochen des CNN wird knapp eine halbe Minute auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt 87 %, d.h. die Fehlerrate ist somit 13 %. D.h. das CNN schlägt das MLP also um einen Prozentpunkt.

Abb. 7.13 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten. In Abb. 7.14 sind die zeitlichen Verläufe der Netzwerkfehler, d.h. der berechnete *Loss*, auf den Trainings- und Testdaten dargestellt. Nach ca. 12 Epochen kreuzen sich die Kurven, d.h. das Training führt ab dann zu immer besseren Werten, während die Messung auf der Testmenge keine nennenswerten Verbesserungen mehr bringt. Dies ist ein Zeichen für Übertrainieren. Mit Hilfe von anderen Strategien, z.B. Regularisierung, kann die Generalisierungsleistung dieses Modells also ggf. noch gesteigert werden.

7 Anwendungen

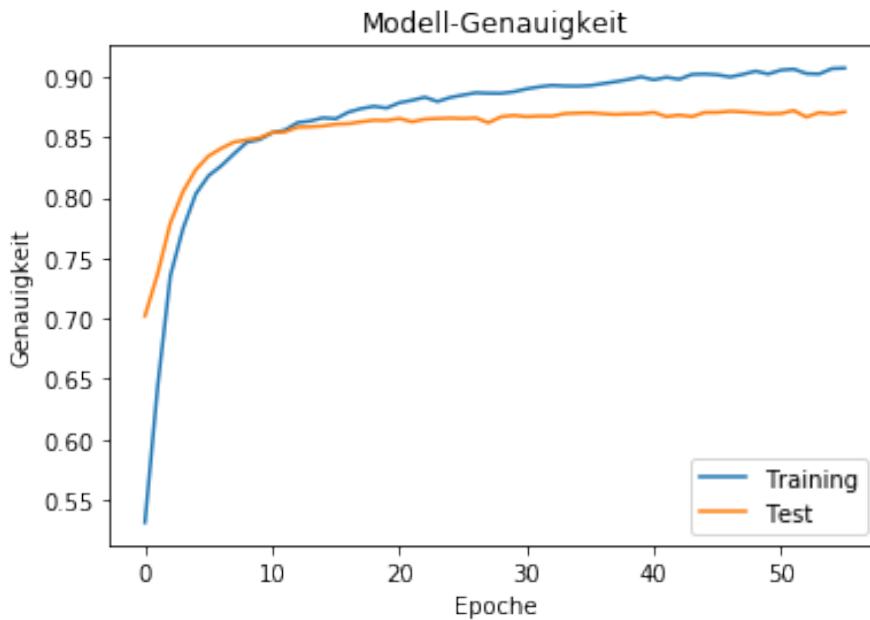


Abbildung 7.13: Genauigkeit des CNN für die Klassifikation der IMDb-Datenbank

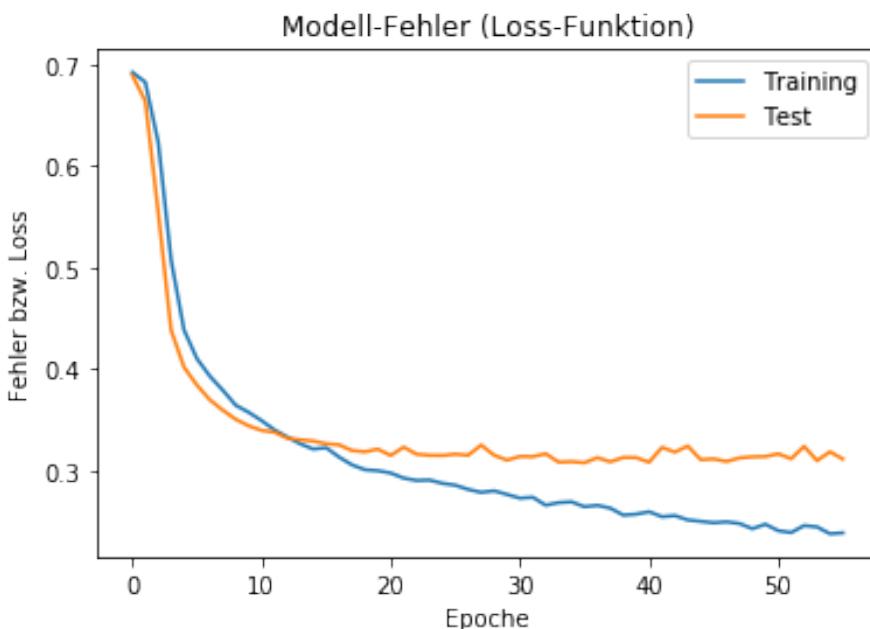


Abbildung 7.14: Fehler bzw. Loss des CNN für die Klassifikation der IMDb-Datenbank

LSTM Da in dieser Anwendung Sequenzen verwendet werden und diese Sequenzen aus Wörter bestehen, sind RNN eigentlich prädestiniert, um solche Probleme anzugehen. Die bekanntesten Vertreter dieser rekurrenten Netzwerke sind LSTM (vgl. Kap. 5.4) und seine Varianten wie bspw. GRU (vgl. Kap 5.5). Betrachten wir zunächst die *Long Short-Term Memory* Einheiten. Generell sind rekurrente Netzwerke sehr schwierig zu trainieren. Die *batch_size* wird auf 4.096 und die Lernrate auf 0,02 erhöht, um ein möglichst schnelles Training zu erreichen. Dafür ist das Training dann ggf. nicht so robust und größere Ausschläge, d.h. Varianzen, werden in Kauf genommen. Es wird eine Schicht mit 16 LSTM-Einheiten trainiert. Im Folgenden sind wieder nur die Änderungen im Quelltext im Vergleich zum ersten Modell dargestellt.

```

# Importiere Keras-Bibliotheken und -Funktionen
from keras.layers import Dense, Dropout, Embedding, LSTM

# Trainingsparameter
batch_size = 4096 # Maxi-Batch

# Modell
model = Sequential()
model.add(Embedding(dictionary_length, embedding_dim, input_length=
    max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(16, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(4, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.02, decay=1
    e-5), metrics=['accuracy'])

```

Listing 7.6: IMDB LSTM

Ergebnis:

```

-----  

Layer (type)          Output Shape         Param #  

-----  

embedding_1 (Embedding)    (None, 200, 16)       16000  

-----  

dropout_1 (Dropout)        (None, 200, 16)       0  

-----  

lstm_1 (LSTM)             (None, 16)           2112  

-----  

dense_1 (Dense)            (None, 4)            68  

-----  

dropout_2 (Dropout)        (None, 4)            0  

-----  

dense_2 (Dense)            (None, 1)            5  

-----  

Total params: 18,185  

Trainable params: 18,185  

Non-trainable params: 0  

-----  

Train on 25000 samples, validate on 25000 samples  

Epoch 1/1000  

- 2s - loss: 0.6871 - acc: 0.5389 - val_loss: 0.6639 - val_acc: 0.6695  

Epoch 2/1000  

- 1s - loss: 0.6588 - acc: 0.6185 - val_loss: 0.6156 - val_acc: 0.6651  

Epoch 3/1000  

- 1s - loss: 0.5979 - acc: 0.6576 - val_loss: 0.5346 - val_acc: 0.7394  

...
Epoch 124/1000  

- 1s - loss: 0.2681 - acc: 0.8943 - val_loss: 0.3187 - val_acc: 0.8709  

Epoch 125/1000  

- 1s - loss: 0.2635 - acc: 0.8948 - val_loss: 0.3230 - val_acc: 0.8711  

Epoch 126/1000  

- 1s - loss: 0.2642 - acc: 0.8952 - val_loss: 0.3213 - val_acc: 0.8694  

Zeitdauer[s]: 190.6857328414917  

Loss: 0.3212622569513321  

Genauigkeit[%]: 86.936

```

7 Anwendungen

Für das Training der 126 Epochen der LSTM-Einheiten werden bereits ca. 3 Minuten auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt 87 %, d.h. die Fehlerrate ist somit 13 %. Dieses Ergebnis ist also mit dem des CNN vergleichbar, wenn auch das Training hierfür ca. 6 Mal so lange dauert. Abb. 7.15 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten. In Abb. 7.16 sind die zeitlichen Verläufe der Netzwerkfehler, d.h. der berechnete *Loss*, auf den Trainings- und Testdaten dargestellt. Man erkennt in beiden Abbildungen sehr deutlich die Ausschläge aufgrund der hohen Werte für die *batch_size* und Lernrate.

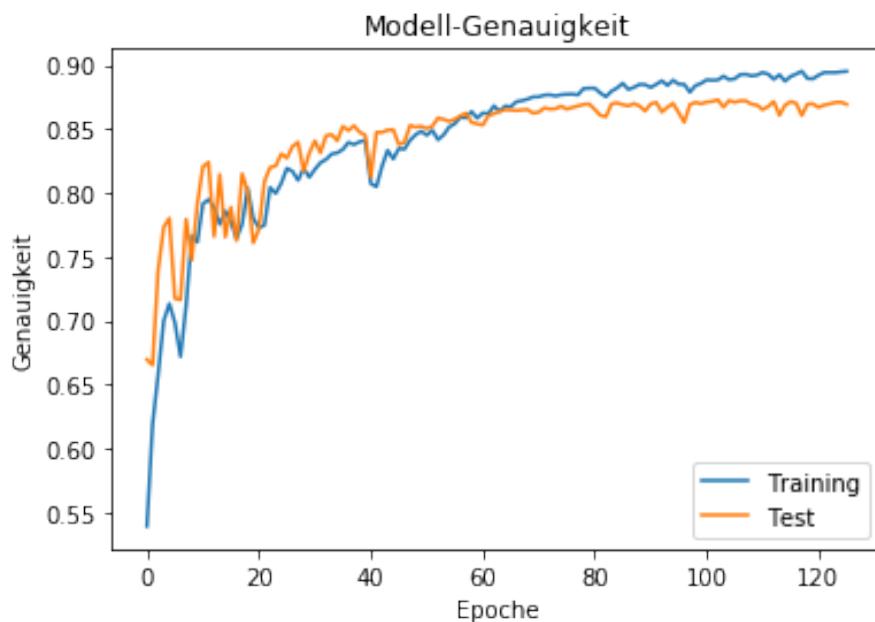


Abbildung 7.15: Genauigkeit des LSTM für die Klassifikation der IMDb-Datenbank

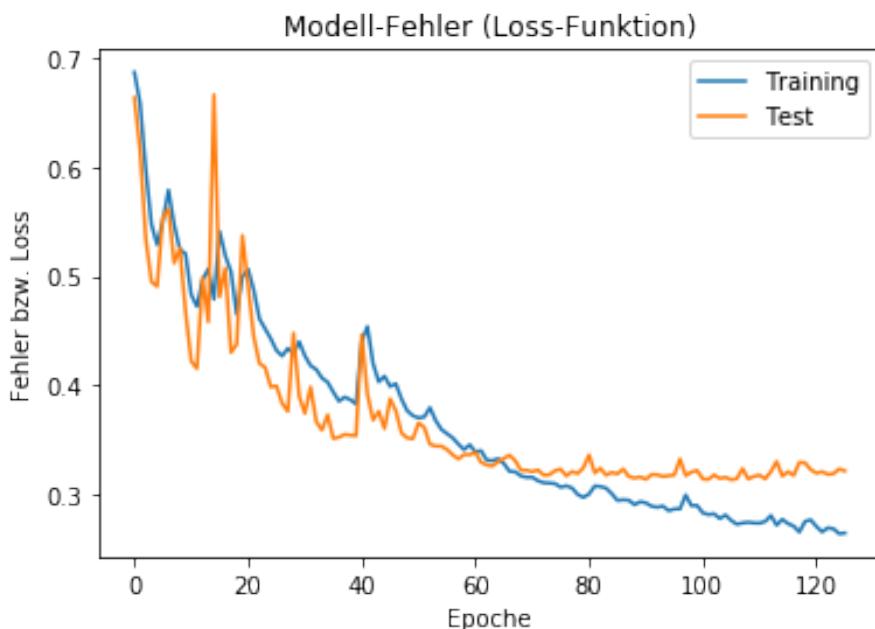


Abbildung 7.16: Fehler bzw. Loss des LSTM für die Klassifikation der IMDb-Datenbank

GRU Das letzte Modell verwendet die Variante *Gated Recurrent Unit* statt der LSTM-Einheit. Das ist auch schon die einzige Änderung im Quelltext.

```
# Importiere Keras-Bibliotheken und -Funktionen
from keras.layers import Dense, Dropout, Embedding, GRU

# Trainingsparameter
batch_size = 4096 # Maxi-Batch

# Modell
model = Sequential()
model.add(Embedding(dictionary_length, embedding_dim, input_length=
    max_review_length))
model.add(Dropout(0.2))
model.add(GRU(16, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(4, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.summary()
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.02, decay=1e-5), metrics=['accuracy'])
```

Listing 7.7: IMDB GRU

Ergebnis:

```
-----  
Layer (type)          Output Shape         Param #  
=====-----  
embedding_19 (Embedding)    (None, 200, 16)      16000  
-----  
dropout_34 (Dropout)       (None, 200, 16)      0  
-----  
gru_19 (GRU)             (None, 16)           1584  
-----  
dense_33 (Dense)          (None, 4)            68  
-----  
dropout_35 (Dropout)       (None, 4)            0  
-----  
dense_34 (Dense)          (None, 1)            5  
-----  
Total params: 17,657  
Trainable params: 17,657  
Non-trainable params: 0  
-----  
Train on 25000 samples, validate on 25000 samples  
Epoch 1/1000  
- 3s - loss: 0.6799 - acc: 0.5566 - val_loss: 0.6020 - val_acc: 0.6840  
Epoch 2/1000  
- 1s - loss: 0.6167 - acc: 0.6534 - val_loss: 0.5922 - val_acc: 0.6800  
Epoch 3/1000  
- 1s - loss: 0.5927 - acc: 0.6916 - val_loss: 0.5799 - val_acc: 0.6916  
...  
Epoch 70/1000  
- 1s - loss: 0.2713 - acc: 0.8931 - val_loss: 0.3253 - val_acc: 0.8568  
Epoch 71/1000  
- 1s - loss: 0.2795 - acc: 0.8881 - val_loss: 0.3020 - val_acc: 0.8728  
Epoch 72/1000  
- 1s - loss: 0.2752 - acc: 0.8898 - val_loss: 0.3010 - val_acc: 0.8743  
Zeitdauer[s]: 88.92190265655518  
Loss: 0.30100429022312164  
Genauigkeit[%]: 87.432
```

7 Anwendungen

Für das Training der 72 Epochen des GRU-Netzwerks werden knapp 1,5 Minuten auf der Grafikkarte Nvidia GTX 1080 Ti benötigt. Die Genauigkeit, gemessen auf den Testdaten, beträgt 87,4 %, d.h. die Fehlerrate ist somit 12,6 %. D.h. einerseits konnte die Trainingsdauer gegenüber dem LSTM-Netzwerk um die Hälfte reduziert werden, andererseits erzielte dieses Netzwerk das beste Ergebnis. Abb. 7.17 zeigt die zeitlichen Verläufe der Genauigkeiten des Klassifizierers auf den Trainings- und Testdaten. In Abb. 7.18 sind die zeitlichen Verläufe der Netzwerkfehler, d.h. der berechnete *Loss*, auf den Trainings- und Testdaten dargestellt.

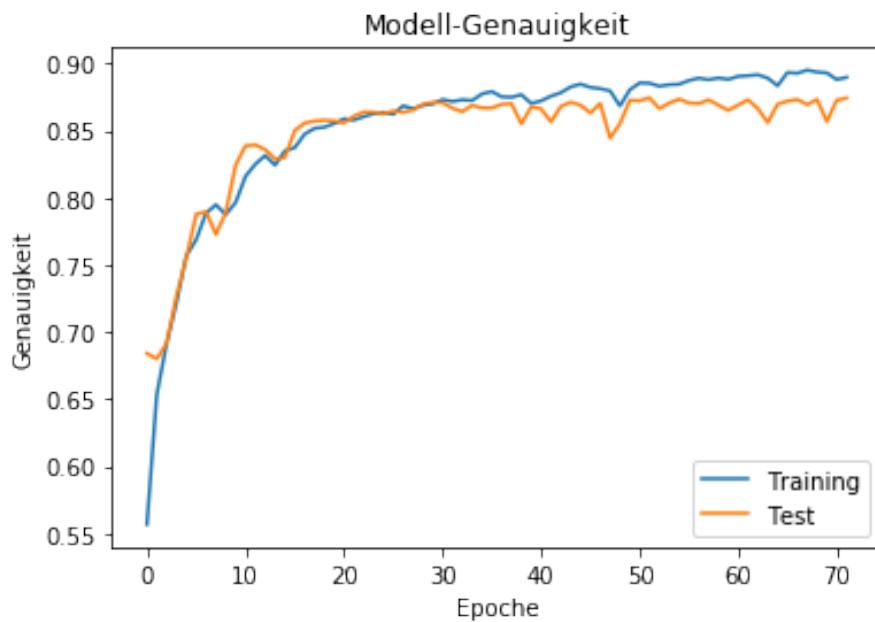


Abbildung 7.17: Genauigkeit des GRU für die Klassifikation der IMDb-Datenbank

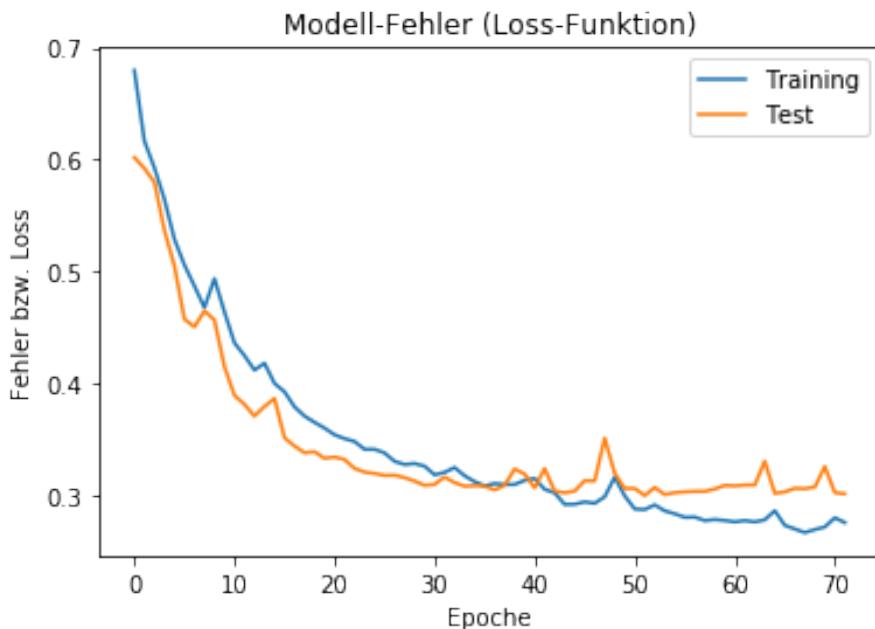


Abbildung 7.18: Fehler bzw. Loss des GRU für die Klassifikation der IMDb-Datenbank

Insgesamt lässt sich feststellen, dass mit allen vier Modellen sehr ähnliche Ergebnisse (Genauigkeiten von 86 bis 87,4 %) erzielt werden konnten. In den *Tutorials* von Jason Brownlee wird ein anderer Ansatz verfolgt [Bro16a; Bro16a]: Hier werden 5.000 statt nur 1.000 der häufigsten Wörter betrachtet und die Rezensionen werden auf 500 Wörter statt auf 100 gekürzt. Auch die Dimension des Vektors beim *Word Embedding* ist mit 32 größer als in dieser Arbeit, nämlich genau doppelt so groß. Dafür werden für die *batch_size* Werte von 64 oder 128 verwendet und das Training bereits nach 2 oder 3 Epochen abgebrochen. Der wesentliche Nachteil dieses Ansatzes ist allerdings, dass eine grafische Analyse des Trainingsfortschritts bei so wenigen Epochen nicht möglich ist. In Tab. 7.3 sind die Ergebnisse dieser Arbeit und die Ergebnisse aus den *Tutorials* gegenübergestellt.

Modell	Diese Arbeit	Brownlee	Quelle
Multilayer Perceptron	86,02 %	86,94 %	[Bro16a]
Convolutional Neural Network	87,09 %	87,79 %	[Bro16a]
Long Short-Term Memory	86,94 %	86,79 %	[Bro16b]
Gated Recurrent Unit	87,43 %	-	-

Tabelle 7.3: Vergleich der Ergebnisse

Die Unterschiede zwischen den Ergebnissen sind nicht sehr groß. Die Modelle dieser Arbeit sind sehr viel einfacher hinsichtlich der genannten Parameter (Länge des Wörterbuchs, Länge der Rezensionen, Dimension des Vektors zum *Word Embedding*). Der Einfluss dieser Parameter auf das Ergebnis könnte in weiteren Studien näher untersucht werden. Hierzu eignen sich insbes. sogenannte Hyperparameter-Optimierungen. Dabei kommen bspw. die Gittersuche (engl. *Grid Search*) oder Evolutionäre Algorithmen zum Einsatz, um die optimale Parameter-Kombination zu finden. Nachgelagert können dann noch an den eigentlichen Netzparametern (Anzahl der Schichten und Einheiten) und Trainingsparametern (Aufteilung der Teilmengen, Lernrate) weitere Studien zur Optimierung durchgeführt werden.

Das Anwendungsbeispiel *IMDb Sentiment Analysis* zeigt die Möglichkeiten des Maschinellen Lernens im Bereich *Big Data* auf. Aus unstrukturierte Daten wird neues Wissen generiert, nämlich beliebige Rezensionen automatisch möglichst korrekt klassifizieren zu können. Einerseits konnten hierzu vier unterschiedliche Architekturen Künstlicher Neuronaler Netzwerke auf dieses Problem angesetzt werden, d.h. aus didaktischen Gründen ist dieses Anwendungsbeispiel bereits sehr interessant, andererseits sind die mit diesen Modellen erzielten Ergebnissen bereits recht gut, die Fehlerquote liegt unter 15 Prozent, sodass man die trainierten Modelle auch tatsächlich zur automatisierten Klassifikation einsetzen könnte.

7 Anwendungen

Ähnliche Anwendungsfälle in diesem Bereich sind denkbar, einige Beispiele sind hier abschließend noch kurz genannt:

- 1 Produktbewertungen in Online-Portalen klassifizieren
- 2 Hasskommentare in sozialen Netzwerken identifizieren
- 3 Spam-E-Mails filtern
- 4 Kunden-E-Mails an den richtigen Service weiterleiten
- 5 Kunden-Zufriedenheit bezgl. Produkte & Dienstleistungen messen
- 6 Wahlergebnisse anhand von Stimmungsbildern vorhersagen
- 7 Auswirkungen von Wirtschaftsnachrichten auf Aktienkurse ermitteln

8 Zusammenfassung und Ausblick

Digitalisierung bedeutet im engeren Sinn die Transformation von analoge in digitale Daten. Im weiteren Sinn wird darunter die digitale Revolution verstanden, die unser Arbeits- und Privatleben radikal verändert. Als Treiber für diese Entwicklung sind die technologischen Fortschritte in den folgenden Bereichen zu nennen: Mobile Anwendungen (Apps), *Cloud Computing*, *Big Data*, Künstliche Intelligenz (KI), *Internet of Things* (IoT) sowie Industrie 4.0. Die KI nimmt dabei eine besondere Rolle ein. Aufgrund der intensiven Nutzung und großen Verbreitung mobiler Endgeräte, der Auslagerung von IT-Services an *Cloud*-Anbieter, dem Zusammenschluss vieler Geräte im Internet und dem massiven Einsatz sensorischer Maschinen in *Smart Factories* werden enorme Mengen an digitalen Daten generiert (*Big Data*), die ohne den Einsatz intelligenter Algorithmen nicht mehr verarbeitet und analysiert werden können. Genau hier setzen moderne Methoden der KI an. Künstliche Neuronale Netzwerke (KNN) und insbes. *Deep Learning* (DL) bilden die technologische Grundlage von vielen selbstlernenden Systemen. Auch aufgrund von Fortschritten im Bereich der grafischen Prozessoren (GPUs) können diese KI-Technologien inzwischen kostengünstig eingesetzt werden und / oder komplexe Probleme der Bild- und Spracherkennung lösen. Ein Beispiel ist das autonome Fahren, bei denen die Videodaten in Echtzeit analysiert und Objekte (Autos, Schilder, Fußgänger usw.) eindeutig erkannt, d.h. klassifiziert, werden müssen. Digitale Sprachassistenten wie bspw. Apples Siri oder Amazons Alexa basieren auf den Fortschritten der Spracherkennung. Hierdurch wird eine neue Mensch-Maschine-Schnittstelle ermöglicht, wodurch in den nächsten Jahren viele neue Anwendungen und Innovationen entstehen werden.

Das Ziel dieser Arbeit war es, typische Anwendungen durchzuführen, bei denen Künstliche Neuronale Netzwerke bzw. Techniken des *Deep Learning* zum Einsatz kommen, die dann als Basis in der Lehre und der angewandten Forschung an der Hochschule für Technik und Wirtschaft des Saarlandes (htw saar) dienen können. Zunächst wurden deshalb Grundlagen zu den wesentlichen Konzepten von KNN vorgestellt. Auf die biologischen Analogien wurde eingegangen und die Ideen zu den ersten Netzwerk-Modellen und Lernverfahren inklusive deren Probleme wurden erläutert. Drei bedeutende KNN-Architekturen wurde jeweils ein eigenes Kapitel gewidmet. Das *Multilayer Perceptron* (MLP) ist ein vollständig-vernetztes vorwärtsgekoppeltes KNN, welches mit dem populären *Backpropagation*-Algorithmus trainiert werden kann. Lernen bedeutet, dass die Parameter des Netzwerks, also die Gewichte der Verbindungen und die Schwellenwerte der Neuronen, angepasst werden. Hierzu werden die bewerteten Differenzen aus Netzfehlern und tatsächlichen Werten schichtweise rückwärts durch das Netzwerk propagiert. Dieses Lernverfahren gehört zum überwachten Lernen und basiert auf der Methode des steilsten Gradienten. Es handelt sich also um ein Optimierungsproblem mit seinen typischen Problemen wie flache Plateaus, Oszillationen, Steckenbleiben in lokalen Minima usw. Netzwerke mit vielen Parametern in vielen verborgenen Schichten können durch das Training zu einer sehr guten Anpassung kommen, ggf. aber schlecht generalisieren, wenn neue Daten verwendet werden. Man spricht dann von einer Überanpassung (*Overfitting*). Werden die Netzwerke immer größer und tiefer, dann kommt noch das Problem der verschwindenden Gradienten hinzu und das Training gerät ins Stocken.

8 Zusammenfassung und Ausblick

Tiefe KNN werden verwendet, um große Datenmengen besser verarbeiten zu können. Um die beschriebenen Probleme zu vermindern bzw. zu lösen, werden Techniken des *Deep Learning* eingesetzt. Hierzu wurden u.a. neuartige KNN-Architekturen entwickelt, Neuronen mit speziellen Aktivierungsfunktionen (ReLU) verwendet, grafische Prozessoren (GPUs) zur schnellen Matrix-Multiplikation eingesetzt usw. Das *Convolutional Neural Network (CNN)* ist ein Netzwerk mit dem visuellen Cortex als biologisches Vorbild. Es wird deshalb auch häufig im Bereich Bilderkennung eingesetzt. Durch die Kombination von sogenannten *Convolutional Layer* und *Pooling Layer* werden die Bildinformationen schrittweise abgetastet, um daraus Stufe für Stufe *Features* zu generieren. Im Fall der Gesichtserkennung würden in der ersten Stufe zunächst Kanten im Bild identifiziert werden, also Bereiche in denen es große Helligkeitsunterschiede gibt. In der zweiten Stufe können Merkmale wie Augen, Nase, Mund, Ohren usw. identifiziert werden. In der letzten Stufe werden dann die kompletten Gesichter analysiert. Das CNN generiert diese *Features* allerdings allein, d.h. ohne fremde, menschliche Hilfe. Erfolgreiche CNN-Architekturen wie bspw. das LeNet-5, das AlexNet und das GoogleLeNet, die internationale Wettbewerbe gewonnen haben, wurden exemplarisch vorgestellt.

Rekurrente bzw. rekursive Netzwerke werden dagegen häufig im Bereich Sprach- oder Texterkennung eingesetzt. Das *Recurrent Neural Network (RNN)* ist ein Netzwerk mit Rückkopplungen. In diesem werden die Signale nicht nur von den Eingabeneuronen über die verdeckten Neuronen zu den Ausgabeneuronen transportiert, sondern auch in entgegengesetzter Richtung. Das biologische Pendant ist der Neocortex mit den höheren Gehirnfunktionen wie Motorik oder Sprache, der auch eine wichtige Rolle für das Gedächtnis spielt. Die Eingabedaten werden bei diesen Netzwerken sequenzweise verarbeitet, wobei eine Sequenz bspw. einen Satz aus Wörtern darstellen kann. Einheiten mit Verbindungen auf sich selbst, also mit einfachen Rückkopplungen, lassen sich hinsichtlich dieser Sequenz wie ein ausgerolltes Netzwerk betrachten und mit einem *Backpropagation*-Algorithmus trainieren, den man nun *Backpropagation Through Time (BPTT)* nennt. Um auch hier den Problemen der verschwindenden Gradienten entgegenzuwirken, wurden die *Long Short-Term Memory (LSTM)* Einheiten entwickelt. Diese besitzen eine innere Struktur aus einer Zelle und drei *Gates*, um den Signalfluss zu steuern und eine Art Gedächtnis abzubilden. Eine Variante davon ist die *Gated Recurrent Unit (GRU)*, die etwas einfacher aufgebaut ist, aber zu ähnlich erfolgreichen Leistungen fähig ist.

Nach den ausführlichen Präsentationen dieser erfolgreicher KNN-Modelle, wurden 22 aktuell verfügbare *Open Source* Softwarelösungen zum Thema *Deep Learning (DL)* vorgestellt, die auf der Internet-Plattform *GitHub* vorhanden sind. Neben den Beschreibungen wurden auch die auf GitHub gesammelten statistischen Daten analysiert. Die mit Abstand populärste DL-Bibliothek ist TensorFlow, die von Mitarbeitern des Google Brain Teams entwickelt wurde. Mathematische Operationen mit Tensoren werden graphenbasiert formuliert. Typische Operationen wie Matrix-Multiplikationen, die beim Lernverfahren ausgeführt werden müssen, lassen sich so einfach formulieren und durch das Verwenden von grafischen Prozessoren auch parallelisiert auf diesen sehr effizient ausführen. TensorFlow benutzt hierfür bspw. CUDA von Nvidia. Die Bibliothek bietet zwar eine Schnittstelle für die Programmiersprache Python an, jedoch steht mit der *High Level API Keras* eine Erweiterung zur Verfügung, die sehr viele Bausteine enthält, die leicht eingesetzt und wiederverwendet werden können. Als *Backend* könnte theoretisch auch Thenao benutzt werden. Diese von der Universität Montréal entwickelte Lösung wird aber seit Ende 2017 nicht mehr weiterentwickelt. Somit wurde die Kombination der Softwarepakete TensorFlow und Keras verwendet, um drei typische DL-Anwendungen durchzuführen.

Die erste Anwendung stammt aus dem Bereich Bilderkennung und Objektklassifikation. Die Datenbank MNIST enthält 70.000 Datensätze von Graustufen-Bildern im Format 28 x 28 Pixel zu handgeschriebenen Ziffern Null bis Neun. Das Ziel ist es also, ein Modell zu trainieren, das die jeweilige Ziffer automatisch identifizieren kann. Somit stellt diese Anwendung einen Schritt im Prozess *Optical Character Recognition (OCR)* dar. Zwei KNN wurden hierzu mit 60.000 Bildern trainiert, ein klassisches MLP mit der Topologie 784-256-128-10 und ein modernes CNN mit einer komplexeren Topologie. Das Training wurde auf der Grafikkarte Geforce GTX 1080 Ti von Nvidia ausgeführt. Nach 12 Sekunden und 16 Epochen erreichte das MLP eine Genauigkeit von 98 % auf den 10.000 Bildern der Testdaten. Das CNN kam sogar auf eine Genauigkeit von 99,35 %, wobei 20 Epochen in 41 Sekunden trainiert wurden. Das sind bereits hervorragende Klassifikationsergebnisse.

Auch die zweite Anwendung ist dem Bereich Bilderkennung zuzuordnen. Diesmal standen mit der CIFAR-10-Datenbank eine Sammlung von 60.000 Farb-Bildern im Format 32 x 32 Pixel zur Verfügung, auf denen zehn Klassen von Objekten zu erkennen sind: Flugzeuge, Autos, Vögel, Katzen, Rehe, Hunde, Frösche, Pferde, Schiffe und LKWs. Als Modell wurde ein komplexes CNN mit 822.570 freien Parametern verwendet und 47 Epochen in etwas über 6 Minuten trainiert. Die Genauigkeit betrug knapp 82 %. Das ist schon sehr ordentlich, wenn man bedenkt, dass diese Anwendung viel komplizierter als das Erkennen der Ziffern ist.

Anwendung Nummer Drei ist eine Stimmungserkennung (engl. *Sentiment Analysis*), d.h. sie gehört zum Bereich *Text Mining*. Als Datenbasis werden 50.000 Nutzer-Kritiken zu Filmen der Internetplattform IMDb verwendet. Diese unstrukturierten Text-Rezensionen sind sehr eindeutig hinsichtlich einer positiven oder negativen Kritik. Letztendlich wird bei dieser Analyse also auch wieder eine Klassifikation durchgeführt, diesmal als binäre Klassifikation mit den zwei Klassen positiv und negativ. Zunächst mussten die Daten vorverarbeitet werden. Die Wort-Sequenzen unterschiedlicher Länge der Rezensionen enthielten bereits ganze Zahlen für die unterschiedlichen Wörter, die nach der auftretenden Häufigkeit sortiert waren. Jede Sequenz wurde zunächst auf 100 Wörter gekürzt und dann auf einen Vektor der Dimension 16 abgebildet, der kontinuierliche Werte enthält (*Word Embedding*). Es wurden dann vier verschiedene KNN mit 25.000 Rezensionen trainiert und dabei folgende Ergebnisse auf der Testmenge von ebenfalls 25.000 Kritiken erzielt. Mit dem MLP der Topologie 256-16-1 konnte nach 38 Epochen bzw. 10 Sekunden eine Genauigkeit von 86 % erzielt werden. Das CNN schaffte nach 56 Epochen in knapp 30 Sekunden 87 %. Für das Training der LSTM-Einheiten wurden bereits 3 Minuten benötigt, wobei nach den 126 Epochen ebenfalls eine Genauigkeit von 87 % erreicht wurde. Schließlich kam das GRU-Netzwerk in der Hälfte der Zeit und 72 Epochen auf eine Genauigkeit von 87,4 %.

Insgesamt kann festgestellt werden, dass mit nur wenigen Zeilen Python-Quelltext verschiedene Anwendungen mit Hilfe von mehreren KNN bearbeitet werden konnten, d.h. die Modelle wurden konfiguriert, trainiert und evaluiert. Das Training auf der *High-End* Grafikkarte von Nvidia verlief schnell und die Modelle zeigten bereits gute Genauigkeiten. Die Kombination aus TensorFlow und Keras ist somit eine gute Basis für den Einsatz in der Lehre und der angewandten Forschung an der htw saar. Das Ziel dieser Arbeit wurde also vollumfänglich erreicht.

8 Zusammenfassung und Ausblick

Die bisher eingesetzten Netzwerke können sicherlich noch bezüglich der Genauigkeit auf den Testdaten verbessert werden. Optimierungstechniken wie Kreuz-Validierung und Regularisierung kamen bislang noch gar nicht zum Einsatz. Hinsichtlich der Topologie und den Trainingseinstellungen lassen sich auch sogenannte Hyperparameterstudien durchführen, in denen automatisch nach den optimalen Parametern gesucht wird, wobei bspw. die Gittersuche oder evolutionäre Algorithmen verwendet werden.

Neben den drei durchgeführten Beispielanwendungen aus den Bereichen Bild- und Texterkennung könnten auch noch weitere Anwendungen untersucht werden, um das Wissen zu vertiefen. Beispielsweise stellt die *Data Science* Plattform Kaggle in zahlreichen Wettbewerben Daten zur Verfügung, die sich mittels Techniken des Maschinenlernens bearbeiten lassen. Betrachtet man die magischen Quadranten zum Thema *Data Science* und *Maschine Learning* des Marktforschungs- und Beratungsunternehmens Gartner (siehe Abb. 8.1), so fällt auf, dass keine der in dieser Arbeit präsentierten Softwarelösungen dort zu erkennen ist. Das liegt daran, dass in dieser Arbeit der Fokus auf die Themen KNN und *Deep Learning* gelegt wurde und es in diesem Bereich momentan nur Softwarelösungen gibt, die dem Anwender bzw. *Data Scientist* Python oder eine andere Programmiersprache als Schnittstelle zur Verfügung stellen. Es muss also noch programmiert werden.



Abbildung 8.1: Gartners magische Quadranten zu Data Science und Maschine Learning [Ido+18]

Die *Data Science* Software KNIME Analytics Plattform, die in den magischen Quadranten als eine der führenden Lösungen angesehen wird, wird bereits seit mehreren Jahren in der Fakultät für Wirtschaftswissenschaften der htw saar in der Lehre erfolgreich eingesetzt.

Diese *Open Source* Software, die an der Universität Konstanz entwickelt wurde, bietet die Möglichkeit, grafische Workflows zu erstellen, um sehr effizient und benutzerfreundlich Aufgaben im Bereich *Data Science* zu bearbeiten (siehe Abb. 8.2). Der Anwender muss dabei also nicht über Programmierkenntnisse verfügen. Mittlerweile gibt es auch zahlreiche Erweiterungen für KNIME, sogenannte *Extensions*, mit denen sich auch andere Software-Bibliotheken einbinden und nutzen lassen. Zum Thema *Deep Learning* werden Erweiterungen zu DL4J und Keras angeboten [KNI18]. Diese Erweiterungen sollten zukünftig im Hinblick auf den Einsatz in der Lehre intensiv getestet werden.

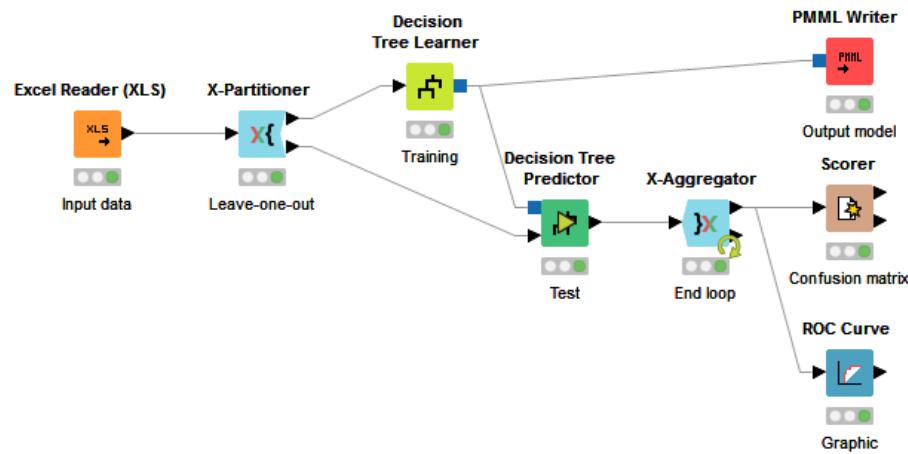


Abbildung 8.2: Workflow der KNIME Analytics Platform

In den magischen Quadranten von Gartner fällt außerdem auf, dass die Lösung H2O.ai als besonders visionär angesehen wird. In Kap. 6.6 wurde die DL-Bibliothek *Deep Water* vorgestellt, die als Erweiterung der H2O-Plattform entwickelt wurde. Als *Backend* zu H2O lässt sich bspw. auch TensorFlow verwenden. Auch diese Softwarelösung ist interessant und sollte ausgiebig getestet werden, um den möglichen Einsatz in der Lehre oder angewandten Forschung zu evaluieren.

Das Thema *Data Science* wird derzeit an der htw saar in den folgenden Master-Modulen gelehrt:

- Angewandte Methoden der Informationsbeschaffung (MMF-130)
- Angewandte Informatik (DFMMS-222)
- Big Data Analysis (MAMS-120)
- Data Science (MASCM-141)

Außerdem wäre dieses Thema und insbesondere Künstliche Neuronale Netzwerke und *Deep Learning* aber auch für den Schwerpunkt Wirtschaftsinformatik im Bachelor-Studiengang Betriebswirtschaft interessant. Im 5. und 6. Semester vertiefen sich die Studierenden in zwei von fünf Schwerpunkten. Vier Module der Vertiefung Wirtschaftsinformatik werden derzeit angeboten. *Data Science* oder KI-Themen sind aber inhaltlich noch nicht Bestandteil. Zusätzlich könnten diese neuen Themen auch im Rahmen von Projektarbeiten (Modul BBWL-622) oder Abschlussarbeiten durch Studierende bearbeitet werden.

Problematisch sind dabei allerdings mehrere Aspekte. Damit das Training von KNN effizient ausgeführt werden kann, wird eine *High End* Grafikkarte benötigt. Die in dieser Arbeit verwendete Nvidia-Grafikkarte Geforce GTX 1080 Ti kostet ca. 750 Euro netto. Möchte man also ein Labor der Fakultät für Wirtschaftswissenschaften (WiWi) damit ausstatten,

8 Zusammenfassung und Ausblick

so wären dies Kosten in Höhe von ca. 30.000 Euro brutto, wenn man 34 Rechner betrachtet (32 Studierenden-PCs, 1 Dozenten-PC, 1 Master-PC). Auf dem DL-Rechner, der in dieser Arbeit verwendet wurde, ist Ubuntu als Betriebssystem installiert. Zur Administration dieses Systems fehlt aber in der Fakultät WiWi das notwendige Know-how. Auf fast allen Rechnern der Fakultät ist Windows als Betriebssystem installiert. Für diese Rechner gibt es administrative Unterstützung durch das dezentrale IT-Service-Team der Fakultät. Der Einsatz von CUDA, cuDNN, TensorFlow und Keras unter Windows muss noch getestet werden.

Ein weiteres Problem besteht darin, dass Python als Programmiersprache benutzt werden muss. Dies kann bei den Studierenden aber nicht vorausgesetzt werden und Grundlagen der Programmierung lassen sich auch nicht in kurzer Zeit vermitteln. Somit bleibt also nur das Modul BBWL-622, um in einem ersten Pilotprojekt ausgewählten Studierenden des Schwerpunkts Wirtschaftsinformatik an diese aktuellen Themen heranzuführen. Im 5. Semester werden bereits Grundlagen der Programmierung vermittelt, allerdings am Beispiel der objektorientierten Programmiersprache Java. Der Umstieg auf die Skriptsprache Python ist aber einfacher als Programmierkenntnisse von null aufzubauen. Für dieses Pilotprojekt bräuchte man auch nur einen Rechner mit der Nvidia-Grafikkarte. D.h. die Kosten halten sich in überschaubare Grenzen. Die Installation und Konfiguration des Rechners könnten die Studierenden sogar selbst bewerkstelligen.

Eine andere Verwendungsmöglichkeit der TensorFlow-Keras-Kombination ist der Einsatz in der angewandten Forschung. Drei Ideen werden im Folgenden kurz skizziert.

Soziale Netzwerke: Hasskommentare Rezensionen, Kritiken oder Meinungsäußerungen findet man im Internet bzw. Web 2.0 auf ganz verschiedenen Plattformen: Im sozialen Netzwerk Facebook oder beim Kurznachrichtendienst Twitter schreiben viele Nutzer ihre persönliche Meinung in Nachrichten und Kommentaren, häufig auch zu gesellschaftspolitischen Themen. Dabei wird manchmal auch eine Grenze überschritten und Hetze oder gefälschte Meldungen (*Fake News*) verbreitet. Nach dem Netzwerkdurchsetzungsgesetz (NetzDG), welches in Deutschland seit dem 1. Oktober 2017 in Kraft getreten ist, müssen die großen sozialen Netzwerke mit mehr als zwei Millionen registrierten Nutzern, offensichtlich rechtswidrige Inhalte innerhalb von 24 Stunden nach Eingang einer Beschwerde löschen oder sperren.

Eine spezielle Form der *Sentiment Analysis* könnte also auf Kommentare in sozialen Netzwerken angewendet werden, um Hasskommentare automatisch zu identifizieren. Ein Klassifizierer als selbstlernendes System könnte also im ersten Schritt dabei helfen, um zu erkennen, ob es sich bei dem Text, um einen Hasskommentar handelt oder nicht. Hierzu müssten zunächst viele Beispiele und Gegenbeispiele gesammelt und manuell klassifiziert werden, um diese dann als Trainings- und Testdaten verwenden zu können. Doch was genau ist ein Hasskommentar? Anders formuliert: An wann wird aus einer freien Meinungsäußerung ein Hasskommentar? Hier gibt es keine einheitliche Definition oder Richtlinie. Nehmen wir aber an, diese Probleme lassen sich lösen, und genügend Datensätze stehen zum Trainieren bereit, dann könnten ähnliche Verfahren wie bei der Klassifikation der positiven und negativen Filmkritiken der IMDb angewendet werden.

Der Gründer von Facebook, Mark Zuckerberg, hat im April 2018 bei seiner Anhörung vor dem US-Kongress zum Datenschutzskandal in Aussicht gestellt, dass die Künstliche Intelligenz in zehn Jahren in der Lage ist, *Hate Speech* bei Facebook zu erkennen und auto-

matisch zu löschen [Bor18]. Auch Google arbeitet an solchen Lösungen und treibt diese Entwicklungen mit seinem Projekt *Perspective* voran, um sogenannte toxische Nachrichten zu identifizieren [Goo18]. Eine große Schwierigkeit ist aber immer noch Sarkasmus in den Nachrichten, wodurch der die jeweilige Aussage relativiert wird. Ob der Klassifizierer dann auch direkt die potenziellen Kandidaten von Hasskommentaren löschen soll, ist eine andere Frage, die geklärt werden muss. Kein Klassifizierer arbeitet perfekt, d.h. es wird immer einige Fehler geben. Entweder wird ein Hasskommentar nicht erkannt oder ein normaler Kommentar wird versehentlich als Hasskommentar klassifiziert. Wenn dieser automatisch vom KI-System entfernt wird, dann käme dies einer Zensur gleich und gefährdet wiederum die freie Meinungsäußerung in einer Demokratie. Dieses spannende Forschungsvorhaben ist also sehr komplex: Es ist technisch sehr anspruchsvoll und gesellschaftspolitisch äußerstbrisant.

Produktion und Logistik: Predictive Maintenance In der vorausschauenden Wartung (engl. *Predictive Maintenance*) geht es darum, vorherzusagen, wann eine Maschine ausfallen wird. Das kann bspw. ein Roboter in der Produktion, ein LKW als Transportmittel auf der Straße oder ein Windrad, welches Energie erzeugt, sein. Es gibt also sehr unterschiedliche Typen von Maschinen, die ausfallen könnten. Durch einen Ausfall kann es im ersten Beispiel zu einem völligen Stillstand der Produktion kommen, im zweiten Beispiel werden die Güter nicht rechtzeitig zum Empfänger geliefert und im dritten Beispiel könnte ein Stromengpass die Folge sein. Mit diesen Maschinenausfällen sind also gewisse Risiken verbunden, die gesteuert werden müssen. Damit Maschinenausfälle überhaupt prognostiziert werden können, müssen zunächst Daten gesammelt werden. Die Sensoren der Maschinen liefern diese Daten. Des Weiteren gibt es Wartungsprotokolle, in denen dokumentiert ist, wann es zu Störungen oder Ausfällen gekommen ist.

Prinzipiell gibt es zwei unterschiedliche Ansätze, ein Modell zu erstellen: Den *White Box* und den *Black Box* Ansatz. Im ersten Fall wird ein Strukturgleichungsmodell aufgestellt, d.h. kausale Zusammenhänge des Systems werden modelliert. Diese basieren auf physikalische Gesetzmäßigkeiten, also wird z.B. das Schwingungs- und Vibrationsverhalten einer Maschine abgebildet. Anschließend werden Simulationen zum kritischen Verhalten der Maschine auf Basis dieses Modells durchgeführt. Der Nachteil dieses Ansatzes ist, dass für jede Maschine ein eigenes Modell erstellt werden muss, da andere physikalische Gesetzmäßigkeiten berücksichtigt werden müssen. Ein weiteres Problem besteht darin, dass Maschinen nicht nur durch Überbeanspruchung und Verschleiß ausfallen können, sondern weil der Mensch sie falsch bedient – absichtlich oder unwissend. Solche Gründe lassen sich in diesen Modellen nicht abbilden.

Der *Black Box* Ansatz benötigt dagegen nur die Eingabedaten (z.B. Sensordaten) und die Ausgabedaten (z.B. Maschinenzustand). Mit Hilfe des maschinellen Lernens wird nun das Verhalten der Maschine nachgebildet, ohne die innere Struktur und die physikalischen Gesetzmäßigkeiten zu kennen. Auch menschliche Anwendungsfehler werden indirekt durch die Eingabedaten erfasst und sind im Modell berücksichtigt. Insbesondere KNN und *Deep Learning* können in diesen Anwendungen eingesetzt werden. *Predictive Maintenance* lässt sich sogar als Klassifikationsaufgabe formulieren: Fällt die Maschine in der nächsten aus oder nicht? In der Fakultät für Wirtschaftswissenschaften der htw saar verfügen die Kollegen des Clusters Logistik (Prof. Dr. Thomas Bousonville, Prof. Dr. Steffen Hütter, Prof. Dr. Thomas Korne, Prof. Dr. Teresa Melo) über das entsprechende Fachwissen in diesem Bereich. Somit wäre eine Zusammenarbeit mit diesen Kollegen in der angewandten Forschung zu diesem Thema möglich.

8 Zusammenfassung und Ausblick

Wirtschaftsinformatik: IT-Security IT-Sicherheit (engl. *IT-Security*) nimmt in einer digitalen Welt einen sehr hohen Stellenwert ein. Eine wichtige Aufgabe ist es, potenzielle Hacker-Angriffe auf ein System zu erkennen, um schnell Gegenmaßnahmen einleiten zu können. Ein sogenanntes *Intrusion Detection System (IDS)* wird eingesetzt, um die Log-Daten des Systems und des Netzwerkverkehrs zu analysieren, d.h. nach Mustern in diesen Daten zu suchen, die auf einen Angriff hindeuten. Letztendlich ist es also wieder eine binäre Klassifikation: Angriff oder kein Angriff. Mit dem Einsatz von KNN und Techniken des *Deep Learning* können selbstlernende IDS auf grafischen Prozessoren implementiert werden, die in Echtzeit ein System analysieren und ggf. Warnungen (engl. *Alerts*) versenden. Fachexperte für solche Systeme ist der Kollege Prof. Dr. Christian Liebig von der Fakultät für Wirtschaftswissenschaften, der ebenfalls im Schwerpunkt Wirtschaftsinformatik in der Lehre eingesetzt ist. Ein gemeinsames Forschungsprojekt zu diesem Thema könnte insbes. auch den mittelständischen Unternehmen im Saarland helfen, neben einer Firewall zukünftig noch eine weitere effektive IT-Sicherheitslösung einzusetzen.

Quellenverzeichnis

- [AES85] D. H. Ackley, Hinton G. E. und T. J. Sejnowski. „A Learning Algorithm for Boltzmann Machines“. In: *Cognitive Science* 9 (1985), S. 147–169.
- [AR+16] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller u. a. „Theano: A Python Framework for Fast Computation of Mathematical Expressions“. In: *CoRR* abs/1605.02688 (2016). URL: <http://arxiv.org/abs/1605.02688>.
- [And70] J. A. Anderson. „Two Models for Memory Organization“. In: *Mathematical Bioscience* 8 (1970), S. 137–160.
- [And72] J. A. Anderson. „A Simple Neural Network Generating an Interactive Memory“. In: *Mathematical Bioscience* 14 (1972), S. 197–220.
- [Apa18a] Apache. *Deep Learning – The Straight Dope*. 2018. URL: <http://gluon.mxnet.io> (besucht am 08. 03. 2018).
- [Apa18b] Apache. *Gluon*. 2018. URL: <https://mxnet.incubator.apache.org/api/python/gluon.html> (besucht am 08. 03. 2018).
- [Apa18c] Apache. *MXNet: A Flexible and Efficient Library for Deep Learning*. 2018. URL: <https://mxnet.apache.org> (besucht am 08. 03. 2018).
- [Apa18d] Apache. *SINGA*. 2018. URL: <https://singa.apache.org> (besucht am 08. 03. 2018).
- [Aph15a] Aphex34. *Max Pooling with 2x2 Filter and Stride = 2, 16*. Dez. 2015. URL: https://commons.wikimedia.org/wiki/File:Max_pooling.png (besucht am 21. 04. 2018).
- [Aph15b] Aphex34. *Typical CNN Architecture*. 16. Dez. 2015. URL: https://commons.wikimedia.org/wiki/File:Typical_cnn.png (besucht am 21. 04. 2018).
- [Art18] Artelnics. *OpenNN: High Performance library for Advanced Analytics*. 2018. URL: <http://www.opennn.net> (besucht am 08. 03. 2018).
- [BLG+18] F. Bastien, P. Lamblin, I. Goodfellow u. a. *Theano*. 2018. URL: <https://github.com/Theano/Theano> (besucht am 08. 03. 2018).
- [Bai18] Baidu. *PArallel Distributed Deep LEarning*. 2018. URL: <http://www.paddlepaddle.org> (besucht am 08. 03. 2018).
- [Bas+17] C. Basoglu u. a. *The Microsoft Cognitive Toolkit*. 22. Jan. 2017. URL: <https://docs.microsoft.com/en-us/cognitive-toolkit> (besucht am 08. 03. 2018).
- [Bat+18] E. Battenberg, S. Dieleman, D. Nouri, E. Olson, A. van den Oord, C. Raffel, J. Schlüter und S. K. Sønderby. *Lasagne*. 2018. URL: <http://lasagne.readthedocs.io> (besucht am 08. 03. 2018).
- [Bea15] F. Beaufays. *The Neural Networks behind Google Voice Transcription*. 11. Aug. 2015. URL: <https://research.googleblog.com/2015/08/the-neural-networks-behind-google-voice.html> (besucht am 28. 04. 2018).
- [Ben+17] Y. Bengio, F. Bastien, P. Lamblin, I. Goodfellow, R. Pascanu, N. Léonard u. a. *Theano*. 21. Nov. 2017. URL: <http://deeplearning.net/software/theano> (besucht am 08. 03. 2018).

Quellenverzeichnis

- [Bla+18] A. Black, A. Gibson, M. Warrick, M. Pumperla, J. Long, S. Audet, E. Wright u. a. *Deep Learning for Java, Scala & Clojure on Hadoop & Spark With GPUs*. 2018. URL: <https://github.com/deeplearning4j/deeplearning4j> (besucht am 08.03.2018).
- [Bor18] T. Borgböhmer. *Wie künstliche Intelligenz gegen Hass im Netz funktioniert: eine moderne Technik mit vielen Fragezeichen*. 13. Apr. 2018. URL: <http://meedia.de/2018/04/13/wie-kuenstliche-intelligenz-gegen-hass-im-netz-funktioniert-eine-moderne-technik-mit-vielen-fragezeichen> (besucht am 04.05.2018).
- [Bri18] S. Brin. *2017 Founders' Letter*. 27. Apr. 2018. URL: <https://abc.xyz/investor/founders-letters/2017/index.html> (besucht am 01.05.2018).
- [Bro16a] J. Brownlee. *Predict Sentiment From Movie Reviews Using Deep Learning*. 4. Juli 2016. URL: <https://machinelearningmastery.com/predict-sentiment-movie-reviews-using-deep-learning> (besucht am 17.04.2018).
- [Bro16b] J. Brownlee. *Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras*. 26. Juli 2016. URL: <https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras> (besucht am 17.04.2018).
- [Bro17] A Bronshtein. *Train/Test Split and Cross Validation in Python*. 17. Mai 2017. URL: <https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6> (besucht am 04.05.2018).
- [CMS12] D. C. Ciresan, U. Meier und J. Schmidhuber. „Multi-Column Deep Neural Networks for Image Classification“. In: *CoRR* abs/1202.2745 (2012). URL: <https://arxiv.org/abs/1202.2745>.
- [CRZ+18] F. Chollet, F. Rahman, O. Zabluda u. a. *Keras: Deep Learning for Humans*. 2018. URL: <https://github.com/keras-team/keras> (besucht am 08.03.2018).
- [CYL+14] C.-Y. Cheng-Yuan Liou, W.-C. Cheng, J.-W. Liou und D.-R. Liou. „Autoencoder for Words“. In: *Neurocomputing* 139 (2014), S. 84–96.
- [Can+18] A. Candel, Q. Kou, M. Stensmo, M. Dymczyk, F. Milo u. a. *Deepwater: Deep Learning in H2O using Native GPU Backends*. 2018. URL: <https://github.com/h2oai/deepwater> (besucht am 08.03.2018).
- [Che+15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang und Z. Zhang. „MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems“. In: *CoRR* abs/1512.01274 (2015). URL: <http://arxiv.org/abs/1512.01274>.
- [Chi+18] S. Chintala, R. Collobert, K. Kavukcuoglu, K. Zhou, C. Farabet u. a. *Torch*. 2018. URL: <https://github.com/torch/torch7> (besucht am 08.03.2018).
- [Cho+14] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk und Y. Bengio. „Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation“. In: *CoRR* abs/1406.1078 (2014). URL: <https://arxiv.org/abs/1406.1078>.
- [Cho+18a] F. Chollet u. a. *Datasets: IMDb Movie Reviews Sentiment Classification*. 2018. URL: <https://keras.io/datasets> (besucht am 17.04.2018).
- [Cho+18b] F. Chollet u. a. *Keras: The Python Deep Learning library*. 2018. URL: <https://keras.io> (besucht am 08.03.2018).

- [Chu16] K. Chung. *Generating Recommendations at Amazon Scale with Apache Spark and Amazon DSSTNE*. 9. Juli 2016. URL: <https://aws.amazon.com/de/blogs/big-data/generating-recommendations-at-amazon-scale-with-apache-spark-and-amazon-dsstne> (besucht am 08.03.2018).
- [Cir+10] D. C. Ciresan, U. Meier, L. M. Gambardella und J. Schmidhuber. „Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition“. In: *CoRR* abs/1003.0358 (2010). URL: <https://arxiv.org/abs/1003.0358>.
- [Col+11] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu und P. Kuksa. „Natural Language Processing (almost) from Scratch“. In: *CoRR* abs/1103.0398 (2011). URL: <https://arxiv.org/abs/1103.0398>.
- [Col+18] R. Collobert, C. Farabet, K. Kavukcuoglu, S. Chintala u. a. *Torch: A Scientific Computing Framework for LuaJIT*. 2018. URL: <http://torch.ch> (besucht am 08.03.2018).
- [DB+18] A. Damien, W. Ballard u. a. *Deep Learning Library Featuring a Higher-Level API for TensorFlow*. 2018. URL: <https://github.com/tflearn/tflearn> (besucht am 08.03.2018).
- [Dam+18] A. Damien u. a. *TFLearn*. 2018. URL: <http://tflearn.org> (besucht am 08.03.2018).
- [Dav14] S. Davies. *Hawking warns on Rise of the Machines*. 2. Dez. 2014. URL: <https://www.ft.com/content/9943bee8-7a25-11e4-8958-00144feabdc0> (besucht am 01.05.2018).
- [Efr16] A. Efrati. *Apples Machines can Learn too*. 13. Juni 2016. URL: <https://www.theinformation.com/articles/apples-machines-can-learn-too> (besucht am 28.04.2018).
- [Elm90] J. L. Elman. „Finding Structure in Time“. In: *Cognitive Science* 14 (1990), S. 179–211. URL: <https://crl.ucsd.edu/~elman/Papers/fsit.pdf>.
- [Ert16] W. Ertel. *Grundkurs Künstliche Intelligenz*. 4. Aufl. Wiesbaden: Springer Vieweg Fachmedien, 2016.
- [FPS91] W. J. Frawley und G. Piatetsky-Shapiro. *Knowledge Discovery in Databases*. 1. Aufl. Menlo Park, California, USA: The MIT Press, 1991.
- [Fay+96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth und R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. 1. Aufl. Menlo Park, California, USA: The MIT Press, 1996.
- [Fuk80] K. Fukushima. „Neocognitron: A Self-Organized Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position“. In: *Biological Cybernetics* 36 (1980), S. 193–202.
- [Fü96] K. Füser. *Neuronale Netze in der Finanzwirtschaft*. 1. Aufl. Wiesbaden: Gabler, 1996.
- [GB10] X. Glorot und Y. Bengio. „Understanding the Difficulty of Training Deep Feedforward Neural Networks“. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Hrsg. von Y. W. Teh und M. Titterington. Bd. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, S. 249–256. URL: <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.
- [GBC16] I. Goodfellow, Y. Bengio und A. Courville. *Deep Learning*. 1. Aufl. Cambridge, Massachusetts, USA: The MIT Press, 2016.

Quellenverzeichnis

- [GJ79] M. Garey und D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1. Aufl. New York: Freeman, 1979.
- [GNP+18] A. Gibson, C. Nicholson, J. Patterson u. a. *DL4J: Deep Learning for Java*. 2018. URL: <https://deeplearning4j.org> (besucht am 08.03.2018).
- [GS00] F. Gers und J. Schmidhuber. „Recurrent Nets that Time and Count“. In: *Proceedings of the International Joint Conference on Neural Networks*. Bd. 3. Como, Italy, Feb. 2000, S. 189–194. URL: <ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf>.
- [GSC99] F. A. Gers, J. Schmidhuber und F. Cummins. „Learning to Forget: Continual Prediction with LSTM“. In: *Neural Computation* 12 (1999), S. 2451–2471.
- [Git18] GitHub. *GitHub*. 2018. URL: <https://github.com> (besucht am 08.03.2018).
- [Gom+18] F. P. Gomez u. a. *OpenNN: Open Neural Networks Library*. 2018. URL: <https://github.com/Artelnics/OpenNN> (besucht am 08.03.2018).
- [Goo+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville und Y. Bengio. „Generative Adversarial Networks“. In: *CoRR* abs/1406.2661 (2014). URL: <https://arxiv.org/abs/1406.2661>.
- [Goo18] Google. *Perspective*. 2018. URL: <http://perspectiveapi.com> (besucht am 04.05.2018).
- [Gre+15] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink und J. Schmidhuber. „LSTM: A Search Space Odyssey“. In: *CoRR* abs/1503.04069 (2015). URL: <https://arxiv.org/abs/1503.04069>.
- [Gro76] S. Grossberg. „Adaptive Pattern Classification and Universal Recording I + II“. In: *Biological Cybernetics* 23 (1976), S. 187–202.
- [Gé17] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. 1. Aufl. Sebastopol (CA), USA: O'Reilly, 2017.
- [HFZ92] F. Herget, W. Finnoff und H. G. Zimmermann. *A Comparison of Weight Elimination Methods for Reducing Complexity in Neural Networks*. Techn. Ber. München: Siemens AG, Corporate Research und Development, 1992.
- [HN87a] R. Hecht-Nielsen. „Counterpropagation Networks“. In: *Applied Optics* 26 (1987), S. 4979–4984.
- [HN87b] R. Hecht-Nielsen. „Kolmogorov's Mapping Neural Network Existence Theorem“. In: *Proceedings of the First IEEE International Conference on Neural Networks*. Bd. III. San Diego (CA), 1987, S. 11–14.
- [HOT16] G. E. Hinton, S. Osindero und Y.-W. Teh. „A Fast Learning Algorithm for Deep Belief Nets“. In: *Neural Computation* 18 (2016), S. 1527–1554. URL: <https://arxiv.org/abs/1211.5063>.
- [HS97] S. Hochreiter und J. Schmidhuber. „Long Short-Term Memory“. In: *Neural Computation* 9.8 (1997), S. 1735–1780. URL: <http://www.bioinf.jku.at/publications/older/2604.pdf>.
- [HSS14] G. E. Hinton, N. Srivastava und K. Swersky. *Neural Networks for Machine Learning – Lecture 6a: Overview of Mini-Batch Gradient Descent*. 6. Feb. 2014. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (besucht am 28.04.2018).
- [HSS17] J. Hu, L. Shen und G. Sun. „Squeeze-and-Excitation Networks“. In: *CoRR* abs/1709.01507 (2017). URL: <https://arxiv.org/abs/1709.01507>.

- [Hah+00] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas und H. S. Seung. „Digital Selection and Analogue Amplification Coexist in a Cortex-inspired Silicon Circuit“. In: *Nature* 405 (2000), S. 947–951. URL: http://www.bioguider.com/ebook/biology/pdf/hahnloser_n947.pdf.
- [He+15] K. He, X. Zhang, S. Ren und J. Sun. „Deep Residual Learning for Image Recognition“. In: *CoRR* abs/1512.03385 (2015). URL: <https://arxiv.org/abs/1512.03385>.
- [Heb49] D. O. Hebb. *The Organization of Behavior*. 1. Aufl. New York: John Wiley & Sons, 1949.
- [Hoc91] S. Hochreiter. „Untersuchungen zu dynamischen neuronalen Netzen“. Diplomarbeit. München: Institut für Informatik, Technische Universität München, 1991.
- [Hop82] J. J. Hopfield. „Neural Networks and Physical Systems with Emergent Collective Computational Abilities“. In: *Proceedings of the National Academy of Sciences* 79 (1982), S. 2554–2558.
- [IMD18a] IMDb. *About IMDb*. 2018. URL: <https://www.imdb.com/pressroom/about> (besucht am 01.05.2018).
- [IMD18b] IMDb. *Statistics*. 2018. URL: <https://www.imdb.com/pressroom/stats> (besucht am 01.05.2018).
- [IS15] S. Ioffe und C. Szegedy. „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *CoRR* abs/1502.03167 (2015). URL: <https://arxiv.org/abs/1502.03167>.
- [Ido+18] C. J. Idoine, P. Krensky, E. Brethenoux, J. Hare, S. Sicular und S. Vashisth. *Magic Quadrant for Data Science and Machine-Learning Platforms*. 22. Feb. 2018. URL: <https://www.gartner.com/doc/reprints?id=1-4RMUF5K&ct=180222> (besucht am 04.05.2018).
- [Ima14] ImageNet. *ImageNet Large Scale Visual Recognition Challenge 2012*. 2. Sep. 2014. URL: <http://www.image-net.org/challenges/LSVRC/2012> (besucht am 21.04.2018).
- [Ima16] ImageNet. *ImageNet Large Scale Visual Recognition Challenge 2016*. 26. Sep. 2016. URL: <http://www.image-net.org/challenges/LSVRC/2016> (besucht am 21.04.2018).
- [Ima17] ImageNet. *ImageNet Large Scale Visual Recognition Challenge 2017*. 26. Juli 2017. URL: <http://www.image-net.org/challenges/LSVRC/2017> (besucht am 21.04.2018).
- [JP+18] Rejith J., T. Penman u. a. *Deep Scalable Sparse Tensor Network Engine (DSSTNE) – An Amazon Developed Library for Building Deep Learning (DL) Machine Learning (ML) Models*. 2018. URL: <https://github.com/amzn/amazon-dsstne> (besucht am 08.03.2018).
- [JS18] Y. Jia und E. Shelhamer. *Caffe*. 2018. URL: <http://caffe.berkeleyvision.org> (besucht am 08.03.2018).
- [JZS15] R. Jozefowicz, W. Zaremba und I. Sutskever. „An Empirical Exploration of Recurrent Network Architectures“. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning*. Bd. 37. ICML'15. Lille, France: JMLR.org, 2015, S. 2342–2350. URL: <http://proceedings.mlr.press/v37/jozefowicz15.pdf>.

Quellenverzeichnis

- [Jia+14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama und T. Darrell. „Caffe: Convolutional Architecture for Fast Feature Embedding“. In: *CoRR* abs/1408.5093 (2014). URL: <http://arxiv.org/abs/1408.5093>.
- [Jia+18] Y. Jia, B. Wasti, P. Noordhuis, L. Yeager, S. Layton, D. Dzhulgakov u. a. *Caffe2 – A Lightweight, Modular, and Scalable Deep Learning Framework*. 2018. URL: <https://github.com/caffe2/caffe2> (besucht am 08. 03. 2018).
- [Jor86] M. I. Jordan. *Serial Order: A Parallel Distributed Processing Approach*. Techn. Ber. ICS Report 8604. San Diego (CA): Institute for Cognitive Science, University of California, 1986.
- [Jor97] M. I. Jordan. „Serial Order: A Parallel Distributed Processing Approach“. In: *Advances in Psychology* 121 (1997), S. 471–495.
- [KB14] D. P. Kingma und J. Ba. „Adam: A Method for Stochastic Optimization“. In: *CoRR* abs/1412.6980 (2014). URL: <https://arxiv.org/abs/1412.6980>.
- [KNI18] KNIME. *Deep Learning*. 2018. URL: <https://www.knime.com/nodeguide/analytics/deep-learning> (besucht am 04. 05. 2018).
- [KSH12] A. Krizhevsky, I. Sutskever und G. E. Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems* 25. Hrsg. von F. Pereira, C. J. C. Burges, L. Bottou und K. Q. Weinberger. Curran Associates, Inc., 2012, S. 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [Kag18a] Kaggle. *CIFAR-10*. 2018. URL: <https://www.kaggle.com/c/cifar-10> (besucht am 17. 04. 2018).
- [Kag18b] Kaggle. *Digit Recognizer: Learn Computer Vision Fundamentals with the Famous MNIST Data*. 2018. URL: <https://www.kaggle.com/c/digit-recognizer> (besucht am 17. 04. 2018).
- [Kag18c] Kaggle. *Sentiment Analysis on IMDb Movie Reviews*. 2018. URL: <https://www.kaggle.com/c/sentiment-analysis-on-imdb-movie-reviews> (besucht am 17. 04. 2018).
- [Kag18d] Kaggle. *The Home of Data Science & Machine Learning*. 2018. URL: <https://www.kaggle.com> (besucht am 21. 04. 2018).
- [Kar15] A. Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. 21. Mai 2015. URL: <http://karpathy.github.io/2015/05/21/rnn-effectiveness> (besucht am 24. 04. 2018).
- [Kar18] A. Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2018. URL: <http://cs231n.github.io/neural-networks-3> (besucht am 28. 04. 2018).
- [Kha16] P. Khaitan. *Chat Smarter with Allo*. 18. Mai 2016. URL: <https://research.googleblog.com/2016/05/chat-smarter-with-allo.html> (besucht am 28. 04. 2018).
- [Kin+18] D. E. King u. a. *Dlib: A Toolkit for Making Real World Machine Learning and Data Analysis Applications in C++*. 2018. URL: <https://github.com/davisking/dlib> (besucht am 08. 03. 2018).

- [Kin09] D. E. King. „Dlib-ml: A Machine Learning Toolkit“. In: *Journal of Machine Learning Research* 10 (2009), S. 1755–1758. URL: <http://www.jmlr.org/papers/volume10/king09a/king09a.pdf>.
- [Kin18] D. E. King. *Dlib C++ Library*. 22. Jan. 2018. URL: <http://dlib.net> (besucht am 08. 03. 2018).
- [Koh72] T. Kohonen. „Correlation Matrix Memories“. In: *IEEE Transactions on Computers* 21 (1972), S. 353–359.
- [Koh82] T. Kohonen. „Self-Organized Formation of Topologically Correct Feature Maps“. In: *Biological Cybernetics* 43 (1982), S. 59–69.
- [Kol57] A. N. Kolmogorov. „On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition“. In: *Doklady Akademii Nauk SSR* 114 (1957), S. 953–956.
- [Kos87] B. Kosko. „Adaptive Bidirectional Associative Memories“. In: *Applied Optics* 26 (1987), S. 4947–4960.
- [Kri09] A. Krizhevsky. „Learning Multiple Layers of Features from Tiny Images“. Diplomarbeit. Toronto, Kanada: Department of Computer Science, University of Toronto, 2009. URL: <http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
- [Kri10] A. Krizhevsky. *Convolutional Deep Belief Networks on CIFAR-10*. Techn. Ber. August. Toronto, Kanada: Department of Computer Science, 2010. URL: <https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf>.
- [Kri18] A. Krizhevsky. *The CIFAR-10 Dataset*. 2018. URL: <http://www.cs.toronto.edu/~kriz/cifar.html> (besucht am 17. 04. 2018).
- [Kru+15] R. Kruse, C. Borgelt, C. Braune, F. Klawonn, C. Moewes und M. Steinbrecher. *Computational Intelligence - Eine methodische Einführung in Künstliche Neuro-nale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. 2. Aufl. Wiesbaden: Springer Vieweg Fachmedien, 2015.
- [Kur12] R. Kurzweil. *How to Create a Mind: The Secret of Human Thought Revealed*. 1. Aufl. New York City, USA: Viking Penguin, 2012.
- [LCB18] Y. LeCun, C. Cortes und C. J. C. Burges. *The MNIST Database of Handwritten Digits*. 2018. URL: <http://yann.lecun.com/exdb/mnist/index.html> (besucht am 17. 04. 2018).
- [LDS89] Y. LeCun, J. S. Denker und S. A. Solla. „Optimal Brain Damage“. In: *Advances in Neural Information Processing Systems* 2. Hrsg. von R. P. Lippmann, J. E. Moody und D. S. Touretzky. San Francisco (CA), USA: Morgan Kaufmann, 1989, S. 598–605. URL: <http://papers.nips.cc/paper/250-optimal-brain-damage.pdf>.
- [LH88] M. Livingstone und D. Hubel. „Segregation of Form, Color, Movement, and Depth: Anatomy, Physiology, and Perception“. In: *Science* 240 (1988), S. 740–749. URL: <http://www.hms.harvard.edu/bss/neuro/bornlab/nb204/papers/livingstone-hubel-segregation-science1988.pdf>.
- [Lam17] P. Lamblin. *MILA and the Future of Theano*. 28. Sep. 2017. URL: <https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY> (besucht am 08. 03. 2018).

Quellenverzeichnis

- [LeC+98] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner. „Gradient-Based Learning Applied to Document Recognition“. In: *Proceedings of the IEEE*. Institute of Electrical und Electronics Engineers, Inc., 1998. URL: <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>.
- [LeC86] Y. LeCun. „Learning Processes in an Asymmetric Threshold Network“. In: *Disordered Systems and Biological Organization*. Hrsg. von E. Bienenstock, F. Fogelman-Soulie und G. Weisbuch. Berlin: Springer, 1986.
- [Lee+11] H. Lee, R. Grosse, R. Ranganath und A. Y. Ng. „Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks“. In: *Communications of the ACM* 54.10 (2011), S. 95–103. URL: <https://www.cs.princeton.edu/~rajeshr/papers/cacm2011-researchHighlights-convDBN.pdf>.
- [Ley12] A. Leyh. *Convolutional Neural Network*. 26. Sep. 2012. URL: <https://www.dasgehirn.info/entdecken/meilensteine/den-rasen-mit-der-nagelschere-schneiden> (besucht am 21.04.2018).
- [MP43] W. S. McCulloch und W. Pitts. „A Logical Calculus of the Ideas Immanent in Nervous Activity“. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133.
- [MP69] M. Minsky und S. Papert. *Perceptrons*. 1. Aufl. Cambridge (MA): MIT Press, 1969.
- [Mad16] Mads00. *Neural Pathway Diagram*. 6. Juni 2016. URL: https://commons.wikimedia.org/wiki/File:Neural_pathway_diagram.svg (besucht am 21.04.2018).
- [Mat18] Mathworks. *Convolutional Neural Network*. 2018. URL: <https://de.mathworks.com/discovery/convolutional-neural-network.html> (besucht am 17.04.2018).
- [Mer16] Mercyse. *Three Type of Feedback in RNN*. 4. März 2016. URL: <https://commons.wikimedia.org/wiki/File:Neuronal-Networks-Feedback.png> (besucht am 24.04.2018).
- [Mou16] A. Moujahid. *A Practical Introduction to Deep Learning with Caffe and Python*. 26. Juni 2016. URL: <http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/> (besucht am 28.04.2018).
- [Mus17] E. Musk. *Competition for AI*. 4. Sep. 2017. URL: <https://twitter.com/elonmusk/status/904638455761612800> (besucht am 01.05.2018).
- [Mwk04] Mwka. *Modell einer Spracherkennung nach Waibel*. 12. Okt. 2004. URL: <https://commons.wikimedia.org/wiki/File:Spracherkennung-modell.png> (besucht am 28.04.2018).
- [Net18] Preferred Networks. *Chainer: A Powerful, Flexible, and Intuitive Framework for Neural Networks*. 2018. URL: <https://chainer.org> (besucht am 08.03.2018).
- [Ola15] C. Olah. *Understanding LSTM Networks*. 27. Aug. 2015. URL: <http://colah.github.io/posts/2015-08-Understanding-LSTMs> (besucht am 24.04.2018).
- [Ooi+15] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang und K. Zheng. „SINGA: A Distributed Deep Learning Platform“. In: *ACM Multimedia*. 2015.
- [PMB12] P. Pascanu, T. Mikolov und Y. Bengio. „On the Difficulty of Training Recurrent Neural Networks“. In: *CoRR* abs/1211.5063 (2012). URL: <https://arxiv.org/abs/1211.5063>.

- [Par+18] A. Park, S. Leishman, A. Thomas, U. Köster u. a. *Neon: Reference Deep Learning Framework*. 2018. URL: <https://github.com/NervanaSystems/neon> (besucht am 08.03.2018).
- [Par85] D. Parker. *Learning Logic*. Techn. Ber. TR-87. MIT, Cambridge (MA): Center for Computational Research in Economics und Management Science, 1985.
- [Pas+18] A. Paszke, S. Chintala, G. Chanan, S. Gross, E. Z. Yang, Z. DeVito, T. Killeen u. a. *PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration*. 2018. URL: <https://github.com/pytorch/pytorch> (besucht am 08.03.2018).
- [Pha+18] W. Phan, M. Stensmo, M. Dymczyk, A. Candel und Q. Kou. *Deep Learning with Deep Water*. Techn. Ber. Mountain View (CA), USA: H2O.ai, 2018. URL: <http://docs.h2o.ai/h2o/latest-stable/h2o-docs/booklets/DeepWaterBooklet.pdf>.
- [PyT18] PyTorch. *PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration*. 2018. URL: <http://pytorch.org> (besucht am 08.03.2018).
- [RB92] M. Riedmiller und H. Braun. „Rprop – A Fast Adaptive Learning Algorithm“. In: *Proceedings of the 7th International Symposium of Computer and Information Science (ISCIS)*. Antalya, Turkey, 1992, S. 279–286.
- [RB93] M. Riedmiller und H. Braun. „A Direct Adaptive Method for Faster Back-propagation Learning: The RPROP Algorithm“. In: *Proceedings of the IEEE International Conference on Neural Networks*. IEEE Press, 1993, S. 586–591. URL: <http://www.cs.cmu.edu/afs/cs/user/bhiksha/WWW/courses/deeplearning/Fall.2016/pdfs/Rprop.pdf>.
- [RHW86a] D. E. Rumelhart, G. E. Hinton und R. J. Williams. „Learning Internal Representations by Error Propagation“. In: *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*. Hrsg. von D. E. Rumelhardt und J. L. McClelland. Cambridge (MA): MIT Press, 1986.
- [RHW86b] D. E. Rumelhart, G. E. Hinton und R. J. Williams. „Learning Representations by Back-Propagating Errors“. In: *Nature* 323 (1986), S. 533–536.
- [Raz12] Razorbliss. *Hidden Markov Model with Output*. 22. Jan. 2012. URL: <https://commons.wikimedia.org/wiki/File:HiddenMarkovModel.svg> (besucht am 24.04.2018).
- [Rea+18] E. Real, A. Aggarwal, Y. Huang und Q. V. Le. „Regularized Evolution for Image Classifier Architecture Search“. In: *CoRR* abs/1802.01548 (2018). URL: <https://arxiv.org/abs/1802.01548>.
- [Red+18] J. Redmon u. a. *Darknet: Convolutional Neural Networks*. 2018. URL: <https://github.com/pjreddie/darknet> (besucht am 08.03.2018).
- [Red18] J. Redmon. *Darknet: Open Source Neural Networks in C*. 2018. URL: <https://pjreddie.com/darknet> (besucht am 08.03.2018).
- [Roc+18] S. Rochel u. a. *Gluon: A Clear, Concise, Simple yet Powerful and Efficient API for Deep Learning*. 2018. URL: <https://github.com/gluon-api/gluon-api> (besucht am 08.03.2018).
- [Ros58] F. Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain“. In: *Psychological Review* 65 (1958), S. 386–408.

Quellenverzeichnis

- [Rud16] S. Ruder. „An Overview of Gradient Descent Optimization Algorithms“. In: *CoRR* abs/1609.04747 (2016). URL: <https://arxiv.org/abs/1609.04747>.
- [Rus+15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg und L. Fei-Fei. „ImageNet Large Scale Visual Recognition Challenge“. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), S. 211–252. URL: <https://arxiv.org/abs/1409.0575>.
- [SSP03] P. Y. Simard, D. Steinkraus und J. C. Platt. „Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis“. In: *Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR 2003)*. Institute of Electrical und Electronics Engineers, Inc., 2003. URL: <https://www.microsoft.com/en-us/research/publication/best-practices-for-convolutional-neural-networks-applied-to-visual-document-analysis/>.
- [Sch+18] J. Schlüter, S. Dieleman, C. Raffel, E. Olson u. a. *Lasagne: Lightweight Library to Build and Train Neural Networks in Theano*. 2018. URL: <https://github.com/Lasagne/Lasagne> (besucht am 08. 03. 2018).
- [Sch14] J. Schmidhuber. *Deep Learning in Neural Networks: An Overview*. Techn. Ber. IDSIA-03-14. Manno-Lugano, Switzerland: Istituto Dalle Molle di Studi sull’Intelligenza Artificiale, University of Lugano & SUPSI, 8. Okt. 2014. URL: <https://arxiv.org/abs/1404.7828>.
- [Sch17] J. Schmidhuber. *Our Impact on the World’s Most Valuable Public Companies*. 17. Aug. 2017. URL: <http://people.idsia.ch/~juergen/impact-on-most-valuable-companies.html> (besucht am 28. 04. 2018).
- [Sei+18] F. Seide, W. Richert, M. Hillebrand, A. Agarwal, J. B. Faddoul, Z. Wang, W. Darling u. a. *Microsoft Cognitive Toolkit (CNTK)*. 2018. URL: <https://github.com/Microsoft/CNTK> (besucht am 08. 03. 2018).
- [She+18] E. Shelhamer, J. Donahue, Y. Jia, J. Long, S. Guadarrama u. a. *Caffe: A Fast Open Framework for Deep Learning*. 2018. URL: <https://github.com/BVLC/caffe> (besucht am 08. 03. 2018).
- [Shi+15] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. Wong und W. Woo. „Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting“. In: *CoRR* abs/1506.04214 (2015). URL: <https://arxiv.org/abs/1506.04214>.
- [Sou18] Facebook Open Source. *Caffe2 – A New Lightweight, Modular, and Scalable Deep Learning Framework*. 2018. URL: <https://caffe2.ai> (besucht am 08. 03. 2018).
- [Ste+18] B. Steiner, S. Cai, V. Vasudevan, D. Murray, G. Gulsoy u. a. *TensorFlow: Computation using Data Flow Graphs for Scalable Machine Learning*. 2018. URL: <https://github.com/tensorflow/tensorflow> (besucht am 08. 03. 2018).
- [Ste17] J. Steppan. *Sample Images from MNIST Test Dataset*. 14. Dez. 2017. URL: <https://commons.wikimedia.org/wiki/File:MnistExamples.png> (besucht am 17. 04. 2018).
- [Ste93] R. Stein. „Selecting Data for Neural Networks“. In: *AI Expert* 2 (1993), S. 42–47.
- [Sys18] Nervana Systems. *Neon*. 2018. URL: <http://neon.nervanasys.com> (besucht am 08. 03. 2018).

- [Sze+14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke und A. Rabinovich. „Going Deeper with Convolutions“. In: *CoRR* abs/1409.4842 (2014). URL: <https://arxiv.org/abs/1409.4842>.
- [Ten18] TensorFlow. *An Open-Source Software Library for Machine Intelligence*. 2018. URL: <https://www.tensorflow.org> (besucht am 08. 03. 2018).
- [Toe07] M. W. Toews. *Normal Distribution Curve that illustrates Standard Deviations*. 7. Apr. 2007. URL: https://commons.wikimedia.org/wiki/File:Standard_deviation_diagram.svg (besucht am 28. 04. 2018).
- [Tok+15] S. Tokui, K. Oono, S. Hido und J. Clayton. „Chainer: a Next-Generation Open Source Framework for Deep Learning“. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*. 2015. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.
- [Unn+18] Y. Unno, S. Tokui, R. Okuta, M. Takagi, S. Saito u. a. *Chainer: A Flexible Framework of Neural Networks for Deep Learning*. 2018. URL: <https://github.com/chainer/chainer> (besucht am 08. 03. 2018).
- [Vin+08] P. Vincent, H. Larochelle, Y. Bengio und P. A. Manzagol. „Extracting and Composing Robust Features with Denoising Autoencoders“. In: *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*. Helsinki, Finland, 2008, S. 1096–1103. URL: <http://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf>.
- [Vog16] W. Vogels. *Bringing the Magic of Amazon AI and Alexa to Apps on AWS*. 30. Nov. 2016. URL: <https://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html> (besucht am 28. 04. 2018).
- [WH60] B. Widrow und M. E. Hoff. *Adaptive Switching Circuits*. 1. Aufl. New York: IRE WESCON Convention Record, 1960.
- [WL90] B. Widrow und M. A. Lehr. „30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation“. In: *Proceedings of the IEEE* 78 (1990), S. 1415–1441.
- [WRH91] A. S. Weigend, D. E. Rumelhart und B. A. Huberman. „Generalization by Weight-Elimination with Application to Forecasting“. In: *Advances in Neural Information Processing Systems 3*. Hrsg. von R. P. Lippmann, J. E. Moody und D. S. Touretzky. San Francisco (CA), USA: Morgan-Kaufmann, 1991, S. 875–882. URL: <http://papers.nips.cc/paper/323-generalization-by-weight-elimination-with-application-to-forecasting.pdf>.
- [WX+18] W. Wang, Z. Xie u. a. *Apache Singa (Incubating)*. 2018. URL: <https://github.com/apache/incubator-singa> (besucht am 08. 03. 2018).
- [Wan+15] W. Wang, G. Chen, T. T. A Dinh, J. Gao, B. C. Ooi, K.-L. Tan und S. Wang. „SINGA: Putting Deep Learning in the Hands of Multimedia Users“. In: *ACM Multimedia*. 2015.
- [Wer74] P. J. Werbos. „Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences“. Diss. Harvard University, 1974.
- [Wer88] P. J. Werbos. „Generalization of Backpropagation with Application to a Recurrent Gas Market Model“. In: *Neural Networks* 1.4 (1988), S. 339–356.
- [Wik18a] Wikipedia. *AlphaGo gegen Lee Sedol*. 2018. URL: https://de.wikipedia.org/wiki/AlphaGo_gegen_Lee_Sedol (besucht am 17. 01. 2018).

Quellenverzeichnis

- [Wik18b] Wikipedia. *CIFAR-10*. 2018. URL: <https://en.wikipedia.org/wiki/CIFAR-10> (besucht am 17.04.2018).
- [Wik18c] Wikipedia. *Comparison of Deep Learning Software*. 2018. URL: https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software (besucht am 08.03.2018).
- [Wik18d] Wikipedia. *Git*. 2018. URL: <https://de.wikipedia.org/wiki/Git> (besucht am 08.03.2018).
- [Wik18e] Wikipedia. *GitHub*. 2018. URL: <https://de.wikipedia.org/wiki/GitHub> (besucht am 08.03.2018).
- [Wik18f] Wikipedia. *Long Short-Term Memory*. 2018. URL: https://en.wikipedia.org/wiki/Long_short-term_memory (besucht am 17.04.2018).
- [Wik18g] Wikipedia. *Loss Functions for Classification*. 2018. URL: https://en.wikipedia.org/wiki/Loss_functions_for_classification (besucht am 28.04.2018).
- [Wik18h] Wikipedia. *MNIST Database*. 2018. URL: https://en.wikipedia.org/wiki/MNIST_database (besucht am 17.04.2018).
- [Wik18i] Wikipedia. *Python*. 2018. URL: [https://de.wikipedia.org/wiki/Python_\(Programmiersprache\)](https://de.wikipedia.org/wiki/Python_(Programmiersprache)) (besucht am 08.03.2018).
- [Wik18j] Wikipedia. *Recurrent Neural Network*. 2018. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network (besucht am 17.04.2018).
- [Wis18] Spektrum der Wissenschaft. *Lexikon der Neurowissenschaft: Hypersäule*. 2018. URL: <https://www.spektrum.de/lexikon/neurowissenschaft/hypersaeule/5808> (besucht am 21.04.2018).
- [Xie+18] E. J. Xie, T. Chen, M. Li, B. Xu, C. Zhang, Y. Liu u. a. *Apache MXNet (incubating) for Deep Learning*. 2018. URL: <https://github.com/apache/incubator-mxnet> (besucht am 08.03.2018).
- [Yan+18] Y. Yang, Q. Longfei, T. Luo, Q. Jun, J. Feng u. a. *PArallel Distributed Deep LEarning*. 2018. URL: <https://github.com/PaddlePaddle/Paddle> (besucht am 08.03.2018).
- [ZHF92] H. G. Zimmermann, F. Herget und W. Finnoff. *Neuron Pruning and Merging Methods for Use in Conjunction with Weight Elimination*. Techn. Ber. München: Siemens AG, Corporate Research und Development, 1992.

Kolophon

Dieses Dokument wurde mit der L^AT_EX-Vorlage für Abschlussarbeiten an der htw saar der Fakultät für Wirtschaftswissenschaften im Bereich Wirtschaftsinformatik erstellt (Version 1.0). Die Vorlage wurde von Stefan Selle erstellt und basiert weitestgehend auf der entsprechenden Vorlage der Fakultät für Ingenieurwissenschaften, die von Yves Hary, André Miede, Thomas Kretschmer, Helmut G. Folz und Martina Lehser entwickelt wurde.