

daniel BASLER



Neuronale Netze mit C# programmieren

Mit praktischen Beispielen für Machine
Learning im Unternehmenseinsatz



Beispielcode unter
plus.hanser-fachbuch.de

HANSER

Neuronale Netze mit C# programmieren



Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel kostenloses Zusatzmaterial.

Geben Sie auf plus.hanser-fachbuch.de einfach diesen Code ein:

plus-5db90-her61



Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine.

Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter:

www.hanser-fachbuch.de/newsletter



Daniel Basler

Neuronale Netze mit C# programmieren

Mit praktischen Beispielen
für Machine Learning
im Unternehmenseinsatz

HANSER

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autor und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.



Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2021 Carl Hanser Verlag München, www.hanser-fachbuch.de

Copy editing: Walter Saumweber, Ratingen

Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München

Umschlagrealisation: Max Kostopoulos

Titelmotiv: © istockphoto.com/Artystarty

Layout: Manuela Treindl, Fürth

Druck und Bindung: Kösel, Krugzell

Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702

Printed in Germany

Print-ISBN: 978-3-446-46229-8

E-Book-ISBN: 978-3-446-46426-1

E-Pub-ISBN: 978-3-446-46635-7

Inhalt

Vorwort	XIII
Aufbau des Buches	XV
1 Künstliche Intelligenz	1
1.1 Grundlagen	1
1.1.1 Schwache künstliche Intelligenz	2
1.1.2 Starke künstliche Intelligenz	3
1.1.3 Hybride künstliche Intelligenz	3
1.2 Themenfelder der künstlichen Intelligenz	4
1.2.1 Machine Learning	5
1.2.2 Deep Learning	5
1.2.3 Cognitive Computing	6
1.2.4 Big Data und Data Science	6
1.2.5 Predictive Analytics	7
1.2.6 Natural Language Processing	7
1.3 KI-Service-Plattformen	8
1.3.1 Amazon	8
1.3.2 Google	9
1.3.3 Microsoft Cognitive Services	11
1.3.4 IBM	12
1.4 Künstliche neuronale Netze	13
1.4.1 Funktionsweise	13
1.4.2 Netztypen	14
1.4.3 Anwendungsbereiche	16
1.5 Grundbaustein Neuron	16
1.5.1 Aktivierungsfunktion	17
1.5.2 Matrizendarstellung	20
1.6 Architekturprinzipien	21
2 Konzepte und Methoden von Machine Learning	23
2.1 ML - Machine Learning	23
2.2 Algorithmen und Modelle	25

2.3	Die Schritte in einem Machine-Learning-Projekt	26
2.4	Machine-Learning-Verfahren	28
2.4.1	Klassifikation	29
2.4.2	Regression	29
2.4.3	Clustering	29
2.4.4	Bayes-Klassifikation	30
2.4.5	Künstliche neuronale Netze	30
2.5	Lernformen	31
2.5.1	Überwachtes Lernen	31
2.5.2	Unüberwachtes Lernen	31
2.5.3	Semi-überwachtes Lernen	32
2.5.4	Verstärkendes Lernen	32
2.6	Machine-Learning-Algorithmen	33
2.6.1	k -Nearest-Neighbour	34
2.6.2	Support Vector Machine	35
2.6.3	Entscheidungsbäume	37
2.6.4	Decision Tree und Random-Forest	38
2.6.5	Clustering	38
2.6.5.1	K-Means Clustering	38
2.6.5.2	EM-Clustering	39
2.6.5.3	Hierarchische Clusteranalyse	39
2.7	Training und Validierung des ML-Modells	40
2.8	Das einfache neuronale Netz	41
2.9	Deep Learning	48
2.10	Einsatzgebiete und Anwendungen	49
3	Neuronale Netze	51
3.1	Vom Problem zum KNN	51
3.2	KNN-Modelle	52
3.3	Mathematik neuronaler Netze	55
3.3.1	Lineare Algebra	55
3.3.2	Vektor	56
3.3.2.1	Rechnen mit Vektoren	57
3.3.2.2	Skalarprodukt	58
3.3.3	Matrix	58
3.3.3.1	Rechnen mit Matrizen	59
3.3.3.2	Matrizenmultiplikation	59
3.3.3.3	Transponieren	61
3.3.4	Tensor	61
3.3.5	Eigenwert- und Singulärwertzerlegung	61
3.4	Mehrschichtige neuronale Netze	62
3.4.1	Multilayer Perceptron (MLP)	62
3.5	Predictive Maintenance	65

3.6	Maschinensimulation mit MLP	67
3.6.1	Datenmodellierung	67
3.6.1.1	Ziel des Feedforward-Netzes	68
3.6.1.2	Mehrklassen-Klassifikation	68
3.6.2	Entwurf	70
3.6.3	Projekt anlegen	71
3.6.4	Erfassung und Berechnung der Daten	73
3.6.5	Bias-Neuron	75
3.6.6	Die Programmierung	76
3.6.7	Aktivierungsfunktionen implementieren	85
3.6.8	Fazit	86
3.7	Lernalgorithmus für Neuronen	87
3.7.1	Kostenfunktion	87
3.7.2	Gradientenabstiegsverfahren	88
3.7.3	Backpropagation-Algorithmus	89
3.8	Backpropagation programmieren	92
3.9	Implementierung	97
4	Training von neuronalen Netzen	111
4.1	Trainings- und Testphase	111
4.1.1	Generalisierung	112
4.1.2	Dimensionsreduzierung	112
4.2	Batch-, inkrementelles und Mini-Batch-Training	113
4.2.1	Batch-Training	113
4.2.2	Inkrementelles Training	113
4.2.3	Mini-Batch-Training	114
4.3	Lernprozess beim Backpropagation-Algorithmus	114
4.3.1	Problemstellung	116
4.3.2	Vorbereiten der Daten	116
4.3.3	Das neuronale Netz programmieren	117
4.3.4	Benutzeroberfläche	118
4.3.4.1	Code-Behind der MainWindow-Klasse	121
4.3.4.2	Nutzen der Hold-Out Validation	124
4.3.5	Programmablauf	127
4.3.6	Das neuronale Netz implementieren	127
4.3.7	Auswertung ermitteln	137
4.4	Simulationsergebnis	138
4.5	Parameteranpassungen	140
5	Recurrent Neural Networks	141
5.1	Sequenzen und Rückkopplung	142
5.2	Architektur eines RNN	144
5.3	Backpropagation Through Time	147

5.4	Long Short-Term Memory Networks	149
5.4.1	Funktionsweise von LSTMs	151
5.4.1.1	Forget-Gate	152
5.4.1.2	Input-Gate	152
5.4.1.3	Output-Gate	154
5.4.1.4	Zusammenfassung	154
5.4.2	Gradient Clipping	155
5.4.3	Varianten	155
5.4.4	LSTM-Implementierung	156
6	Convolutional Neural Networks	159
6.1	Aufbau eines CNN	160
6.2	Detektionsteil	162
6.2.1	Kantenerkennung	162
6.2.2	Pooling	164
6.2.3	Schrittweite	165
6.2.4	2D- und 3D-Volumen	165
6.2.5	Aktivierungsfunktion	166
6.2.6	Ein sehr einfaches CNN	167
6.2.7	Subsampling	168
6.2.8	CNN mit Pooling Layer	169
6.3	Identifikationsteil	170
6.4	Schlussbemerkung	171
7	Machine Learning Frameworks	173
7.1	Einbindung von ML-Frameworks in C#	174
7.2	TensorFlow	175
7.2.1	Ablauf in TensorFlow	176
7.2.2	Das TensorBoard	177
7.2.3	Begriffe	178
7.2.4	TensorFlow Playground	178
7.3	Keras	179
7.4	Infer.NET	180
7.4.1	Probabilistische Programmierung	181
7.4.2	Arbeitsweise von Infer.NET	182
7.4.3	Infer.NET-Architektur	184
7.4.4	Infer.NET Modelling-API	185
7.4.5	Lernen und Trainieren	186
7.4.6	Infer.NET in der Anwendung	186
7.4.7	Das Modell entwerfen	187
7.4.8	Infer.NET anwenden	188
7.5	ML.NET mit AutoML und ModelBuilder	192
7.5.1	Einbinden von ML.NET	193

7.5.2	Was ist AutoML	193
7.5.3	Model Builder	195
7.5.4	Einbinden in das Projekt	195
7.5.5	Szenario	196
7.5.6	Daten.....	197
7.5.7	Training und Auswertung	200
7.5.8	Der Code.....	200
7.5.9	Automatisiert modellieren	201
7.5.10	Die Kommandozeile (CLI).....	207
7.5.11	Die Zukunft von AutoML	207
7.6	Benutzerdefiniertes ML.NET	208
7.6.1	ML.NET-Komponenten	209
7.6.2	Benutzerdefinierter Workflow	211
7.6.3	Erstellen einer benutzerdefinierten Anwendung.....	212
7.6.4	Datentransformation.....	214
7.6.5	ML.NET-Algorithmus	215
7.6.6	Erstellen und Trainieren eines ML-Modells.....	215
7.6.7	Modellauswertung.....	216
7.6.8	Modellbereitstellung.....	218
7.6.9	TensorFlow, ONNX und ML.NET	218
8	SciSharp Stack	221
8.1	TensorFlow.NET.....	222
8.1.1	TensorFlow.NET-SDK installieren	222
8.1.2	Tensor.....	224
8.1.3	Platzhalter	225
8.1.4	Variable	226
8.1.5	Konstante	227
8.1.6	Berechnungsgraph	228
8.1.7	Lineare Regression	229
8.1.8	Von der Theorie zum Code	230
8.2	Keras.NET	233
8.2.1	Keras.NET installieren	233
8.2.2	Modelle erstellen	234
8.3	NeuralNetwork.NET	236
9	Machine Learning as a Service.....	237
9.1	Amazon Machine Learning und KI-Services	238
9.1.1	Amazon Lex	239
9.1.2	Die Lex-Chatbot-Struktur	240
9.1.3	Entwickeln mit AWS-Lambda-Funktionen	242
9.2	Erstellen eines Lex-Chatbots für .NET	244
9.2.1	Erste Schritte	244

9.2.2	Beispiel Chatbot	245
9.2.3	Intents	247
9.2.4	Testen Sie den Bot	249
9.2.5	AWS-Lambda-Funktion	251
9.2.6	Slots	255
9.2.7	Error Handling	255
9.2.8	Konfigurieren von Cognito	256
9.2.9	Die Web-Applikation	257
9.3	Azure Cognitive Services	259
9.3.1	Intelligente kontextbasierte Suchfunktion	260
9.3.1.1	Bing-Websuche	260
9.3.1.2	Bing Suche über REST API	262
9.3.1.3	Die eigene Suchmaschine	263
9.4	Azure Machine Learning Studio	269
9.4.1	Arbeitsbereich	269

10 Anwendungen entwerfen **273**

10.1	Predictive Analytics	273
10.1.1	Fallbeispiel: Energiebranche	274
10.1.2	Zeitreihenanalyse	274
10.1.3	Beispielprogramm und Anwendung der Prognose	276
10.1.4	Definieren der Pipeline	277
10.2	Bildklassifikation	280
10.2.1	Benötigte Daten	280
10.2.2	Projekt konfigurieren	281
10.2.3	Importieren des MNIST-Datensatzes	283
10.2.4	Aktivierungsfunktion	287
10.2.5	Input Layer	288
10.2.6	Hidden Layer	289
10.2.7	Output Layer	291
10.2.8	Neural Network	293
10.2.9	Initialisierung und Auswertung	297
10.2.10	Training und Backpropagation	300
10.2.11	Auswertung und Verbesserung	301
10.3	Visuelle Muster erkennen	302
10.3.1	Aufgabenstellung	302
10.3.2	Convolutional Layer	303
10.3.3	Pooling Layer	303
10.3.4	Flatten Layer	305
10.3.5	Fully Connected Layer	306
10.3.6	Methoden	306
10.3.7	Training	307
10.4	Objekterkennung	309

10.4.1 Transferlernen mit ML.NET	310
10.4.2 Neue Bilddaten vorbereiten	311
10.4.3 Trainiertes TensorFlow-Modell verwenden	313
10.4.4 MLContext, Pipeline und Prognose	315
10.5 Natural Language Processing	317
10.5.1 Textklassifikation	318
10.5.2 Merkmalsvektoren (Feature Vectors)	320
10.5.3 Texterkennung mit CNN.....	322
10.5.4 Textklassifikation mit RNN	323
10.5.5 Word Embedding mit ML.NET	325
10.5.6 Stopwörter	328
10.6 Stanford CoreNLP für .NET	331
10.7 Sentiment-Analyse	333
10.7.1 Sentiment.....	333
10.7.2 Sentiment-Analyse mit ML.NET.....	333
10.7.3 Sentiment-Analyse mit AutoML.....	338
10.7.4 Modell erstellen mit dem Model Builder.....	338
10.7.5 Das Modell als Web-App.....	343
Referenzen und Quellen	347
Stichwortverzeichnis	351

Vorwort

Neuronale Netze sind seit einiger Zeit überall im Gespräch. Als Softwareentwickler stellt man fest, dass es zurzeit keinen anderen Bereich in der Softwareentwicklung gibt, der sich so rasant verändert und weiterentwickelt.

Eine attraktive Alternative zu Python für den Entwurf von neuronalen Netzen ist eine modulare und objektorientierte Programmiersprache wie Microsoft C#. Die Objektorientierung bietet den Vorteil, dass sich sehr gut modulare Machine-Learning-Modelle entwerfen lassen, die konfigurierbar, erweiterbar und wiederverwendbar sind.

Dieses Buch richtet sich an C#-Entwickler, die einen möglichst umfassenden Blick über neuronale Netze erlangen wollen. Es möchte Sie beim Kennenlernen, Experimentieren und Arbeiten mit neuronalen Netzen und Machine-Learning-Modellen anleiten und unterstützen. Dabei wendet es sich an im Umgang mit neuronalen Netzen unerfahrene Programmierer.

Sie sollten über Grundkenntnisse in der Programmierung mit C# verfügen, sodass Begriffe wie Variablen, Schleifen etc. Ihnen vertraut sind. Wegen des mathematischen Anteils müssen Sie sich keine Sorgen machen. Um die Buchinhalte zu verstehen, sind wirklich nur gute Kenntnisse in dem Konzept der linearen Algebra erforderlich.

Insgesamt erhebt das Buch auch keinen Anspruch auf Vollständigkeit. Es verzichtet auf tiefgehende mathematische und programmiertechnische Details, die nicht wirklich notwendig sind, um die Programmierung von neuronalen Netzen und Machine Learning zu verstehen.

Anhand von Anwendungsbeispielen lernen Sie neuronale Netze und Machine-Learning-Modelle zu entwickeln. Sie lernen auf diese Weise dynamische Datenstrukturen, Feedforward-Netze, Backpropagation-Algorithmen sowie Convolutional Neural Networks und Natural Language Processing kennen. Das Buch möchte Ihnen die Leistungsvielfalt neuronaler Netze vermitteln und Ihnen helfen diese in eigenen Programmierprojekten zu nutzen.

Den Mitarbeiterinnen und Mitarbeitern des Hanser-Verlages, besonders Frau Sylvia Hasselbach, danke ich für die Sorgfalt und Unterstützung bei der Veröffentlichung dieses Buches.

Ihnen, liebe Leserin und lieber Leser, wünsche ich viel Freude und Erfolg beim Kennenlernen und Arbeiten mit neuronalen Netzen mit C#.

Daniel Basler

Herford, April 2021

Aufbau des Buches

Die wichtigsten Begriffe und Konzepte werden gleich in den ersten beiden Kapiteln erläutert. Die weiteren Kapitel sind thematisch so aufgebaut, dass man das Buch von vorne nach hinten durcharbeiten kann.

Nachfolgend erhalten Sie einen Überblick über den Inhalt des Buches:

- **Kapitel 1** führt in das Thema künstliche Intelligenz ein. Dabei geht es um die Anwendungsbiete, den Aufbau von neuronalen Netzen und das Neuron als Grundbaustein.
- **Kapitel 2** beschreibt die Konzepte und Methoden von Machine Learning. Es nennt wichtige Algorithmen und zeigt ein erstes einfaches neuronales Netz als Programmierbeispiel.
- **Kapitel 3** beschreibt Schritt für Schritt den Aufbau eines künstlichen neuronalen Netzes in C# und geht auf die wichtigsten Mathematik-Grundlagen ein. Anhand von Aufgabenstellungen werden mehrschichtige neuronale Netze entwickelt. Des Weiteren beschäftigen Sie sich in diesem Kapitel mit Lernalgorithmen und der Programmierung des Backpropagation-Algorithmus.
- **Kapitel 4** zeigt das Training von neuronalen Netzen. Sie lernen hier unterschiedliche Lernprozesse kennen und implementieren Trainingsmethoden in ein neuronales Netz.
- **Kapitel 5** führt Sie weiter zu den sogenannten Recurrent Neural Networks (RNN), die hauptsächlich in der Verarbeitung von Textsequenzen oder Zeitreihen eingesetzt werden. Sie lernen, wie die einzelnen Schichten in einem RNN aufgebaut sind.
- **Kapitel 6** erläutert ausführlich den Aufbau von Convolutional Neural Networks (CNN), die vornehmlich für die Verarbeitung von Bild- und Audiodateien eingesetzt werden. Ein CNN ist ein spezielles mehrschichtiges Feedforward-Netz.
- **Kapitel 7** beschäftigt sich mit den im Moment aktuellen Machine Learning Frameworks, die Sie bei der Entwicklung von Machine-Learning-Modellen in C# unterstützen können.
- **Kapitel 8** beschreibt ein weiteres nützliches ML-Framework im .NET Umfeld. Der sogenannte SciSharp-Technologie-Stack ermöglicht es, ML-Modelle für TensorFlow mit C# zu erstellen. Des Weiteren werden in diesem Kapitel die Frameworks Keras.NET und NeuralNetwork.NET vorgestellt.
- **Kapitel 9** gibt einen Einblick in Machine Learning as a Service und stellt Amazon Lex und Azure Cognitive Services an einem Anwendungsbeispiel näher vor.
- **Kapitel 10** setzt die in den vorherigen Kapiteln aufgezeigten Methoden in einzelnen Beispieleanwendungen für Zeitreihenanalyse, Bildklassifikation, Objekterkennung und Natural Language Processing um.

Programmbeispiele

Um den Praxisbezug zu gewährleisten, wird gezeigt, wie die beschriebenen Themen in C#-Programme umgesetzt werden. In den meisten Fällen wird der vollständige Beispielcode abgebildet, um die Programmierung in C# zu verdeutlichen. Sie können die Buchbeispiele komplett von meinem GitHub-Repository herunterladen (<https://github.com/DanielBasler/NeuralNetwork>) oder von der Plus.Hanser-Webseite:



Ihr Plus – digitale Zusatzinhalte!

Auf unserem Download-Portal finden Sie zu diesem Titel die Code-Beispiele aus dem Buch. Geben Sie auf plus.hanser-fachbuch.de einfach diesen Code ein:

plus-5db90-her61

Zielsetzung des Buches

Mein Ziel ist es, Ihnen ein solides Fundament für das Entwerfen und Entwickeln von neuronalen Netzen und Machine-Learning-Modellen an die Hand zu geben, unterstützt von praktischen Beispielen. Des Weiteren möchte ich, dass Sie ein Gefühl für den Programmier-Aufwand von neuronalen Netzen und deren Leistung bekommen und die Vielfalt der Einsatzmöglichkeiten aber auch der Anforderungen kennenlernen.

Alle hier verwendeten Softwaretools, mit Ausnahme der KI-Services, sind kostenlos und Open Source. Die Beispiele lassen sich mit Visual Studio Community Version 2019 entwickeln und auf einem „normalen PC“ ohne besondere Hardware-Power erstellen und ausführen.

1

Künstliche Intelligenz

„Okay Google“, „Alexa, spiel bitte ‚Eight days a week‘ von den Beatles“ oder „hey Siri“... kennen Sie vermutlich alle. Die sogenannte künstliche Intelligenz, kurz nur noch als KI bezeichnet, ist schon längst in unserem Alltag angekommen und dringt in immer mehr Bereiche vor.

Das heißt, wir sind alle mittendrin, ob es um Sprachassistenten als Helfer auf dem Smartphone, im Auto oder zu Hause geht, die digitale Vernetzung in allen Lebensbereichen schreitet mit unglaublicher Geschwindigkeit voran. Mit der digitalen Transformation, durch die auch die KI allgegenwärtig wird bzw. ist, müssen sich auch Unternehmen auf eine Umstrukturierung und gegebenenfalls sogar auf eine Neuorientierung ihrer Geschäftsprozesse einstellen. Heute müssen Sie sich als Entwickler mit Begriffen wie digitaler Zwilling, Product Performance Management und digitale Fabrik auseinandersetzen. So stellt zum Beispiel das Zusammenspiel von Robotic Process Automation (RPA) und künstlicher Intelligenz ganz neue Symbiosen im Bereich der Automation dar.

In der Diskussion über KI tauchen heute sehr viele verschiedene Begriffen auf. Es ist die Rede von Machine Learning, neuronalen Netzen, Representation Learning, Natural Language Processing (NLP) oder auch Deep Learning. Selbst Begriffe wie Big Data und Data Science werden in die Runde geworfen. Sie sehen also, wer sich mit KI befasst, wird sehr schnell mit einem Begriffswirrwarr von Schlagwörtern überzogen, die durchaus für Verwirrung sorgen können.

Man kann KI als Überbegriff sehen, unter dem unterschiedliche Techniken und Bezeichnungen versammelt sind, und bevor wir uns dem Hauptthema des Buches – der Programmierung neuronale Netze – widmen, möchte ich einige Begriffe und Technologien klären, die in diesem Buch benutzt werden.

■ 1.1 Grundlagen

Dabei ist KI keine neue Wissenschaft oder Technologie. Die Anfänge der KI-Forschung gehen bis in die 1950er-Jahre zurück, in denen Alan Turing den Aufsatz „Computing Machinery and Intelligence“ [1] vorgelegt hat. Auf Turing geht auch der nach ihm benannte Turing-Test zurück, der dazu dient, zu unterscheiden, ob eine Maschine ein gleichwertiges Denkvermögen aufweist wie ein Mensch oder nicht.

Zum ersten Mal wurde der Begriff „artificial intelligence“ von John McCarthy [2] verwendet bzw. geprägt. Laut McCarthy ist KI eine Informations- und Ingenieurwissenschaft, die dem Herstellen „intelligenter“ Maschinen und speziellen intelligenten Computerprogrammen gewidmet ist. Für McCarthy besteht der rechnerische Teil der Intelligenz in der Fähigkeit, die Ziele in der Welt zu erreichen. Das heißt für ihn, ein Computer soll so gebaut oder programmiert werden, dass er eigenständig Programme bearbeitet und löst, aus den gemachten Fehlern lernt, Entscheidungen trifft und mit seiner Umgebung kommuniziert.

Aufgrund dessen kann man auch vereinfacht sagen, KI beschäftigt sich mit der Entwicklung von Systemen, die eigenständig Probleme lösen und analog zu menschlichen Denk- und Verhaltensmustern intelligent handeln.

Dass die KI seit den letzten Jahren einen dermaßen großen Boom erlebt, hat sie folgenden Faktoren zu verdanken:

- Die Menge der zur Verfügung stehenden Daten hat extrem zugenommen. Durch Big Data [3] erweitern sich auch die Anwendungsfelder für den Einsatz von KI.
- Die Daten die über Big Data gewonnen werden, lassen sich immer preisgünstiger speichern. Die Preise für Storage sind in den vergangenen Jahren deutlich gesunken. Das macht das Speichern und Auswerten großer Datenmengen inzwischen für Unternehmen rentabel.
- Grafikprozessoren (GPU) werden günstiger und leistungsfähiger. Durch den Einsatz von GPUs können Berechnungen massiv parallelisiert und dadurch beschleunigt werden. Vor allem in diesem Bereich waren die Fortschritte in den vergangenen Jahren enorm.
- Frameworks und Cloud Services erleichtern den Einsatz. Neben der Rechenleistung und dem Speicher in der Cloud können hier auch ML-Modelle erstellt und auf verschiedenen Rechnern in der Cloud-Infrastruktur getestet werden. Hier haben Frameworks wie TensorFlow und vorgefertigte KI-Services aus der Cloud die Einstiegsschwelle deutlich gesenkt.

Durch diese Faktoren finden immer mehr KI-Lösungen ihren Weg in die Praxis. Allerdings sprechen wir hier nur über die sogenannte schwache KI, die inzwischen sehr gut erforscht und sich in zahlreichen Produkten und Diensten etabliert hat.

1.1.1 Schwache künstliche Intelligenz

Als schwache künstliche Intelligenz (engl. *weak AI* oder *narrow AI*) werden Systeme bezeichnet, die auf die Lösung konkreter Anwendungsprobleme fokussieren.

Das heißt, bei der schwachen KI geht es um die Simulation eines gewissen Bereiches des intelligenten Verhaltens mit Mitteln der Mathematik und der Informatik. Mit schwacher KI haben wir es mittlerweile im Alltag ständig zu tun. Beispiele hierfür sind unteranderen:

- Zeichen- und Texterkennung
- Bilderkennung
- Spracherkennung
- Automatische Übersetzung
- Expertensysteme
- Navigationssysteme
- Autovervollständigung und Korrekturvorschläge bei Suchvorgängen

Besondere Aufmerksamkeit erhalten hier auch die sogenannten intelligenten Chatbots, die den Kundensupport revolutionieren sollen. Aber auch die Buchhaltung wird heute schon mit schwacher KI automatisiert und optimiert, ebenso wie der Posteingang beim E-Mail-Verkehr und Aufgaben im Backoffice allgemein.

1.1.2 Starke künstliche Intelligenz

Starke künstliche Intelligenz (engl. *strong AI* oder *general AI*) hat das Ziel, eine Intelligenz zu erschaffen, die menschliches Denken, Bewusstsein und Emotionen oder die gleichen intellektuellen Fertigkeiten von Menschen erreichen oder übertreffen kann.

Starke KI müsste somit folgende Eigenschaften aufweisen:

- Logisches Denkvermögen
- Entscheidungsfähigkeit auch bei Unsicherheit
- Planungs- und Lernfähigkeit
- Fähigkeit zur Kommunikation in natürlicher Sprache
- Kombinieren aller Fähigkeiten

Allerdings ist es bis heute noch nicht gelungen, eine starke KI zu entwickeln. Auch die Diskussion, ob eine solche Entwicklung überhaupt möglich ist, hält weiterhin an. Verstärkt finden wir heute schon Ansätze in der Form der hybriden KI.

1.1.3 Hybride künstliche Intelligenz

Bei der hybriden künstlichen Intelligenz handelt es sich um eine KI-Technologie die versucht in Verbindung mit agiler Optimierung in der Unternehmensplanung oder auch in der Operationsforschung eine Reihe von Prozess- Technologien und -Modellen mit Machine Learning und Deep Learning zu vereinen.

Das heißt, man macht sich die Algorithmen aus Machine Learning zunutze um entsprechende ML-Modelle zu entwickeln, welche wiederum mit menschlichem Expertenwissen hinsichtlich Geschäftsprozessen, Verhaltensmustern und Planungszielen erweitert und trainiert werden. Somit ist es möglich, potenzielle Entscheidungsräume innerhalb dieser Modelle in kürzester Zeit durch den Experten auszuloten und zu bewerten.

Google nutzt die hybride künstliche Intelligenz schon bei der Sprachsuche. Wird von dem ML-System eine Äußerung richtig verstanden, folgt darauf das Markieren eines Treffers durch einen menschlichen Experten. Das ML-System ist so angelegt, dass es sich durch diese Markierung selbst trainieren kann, es verbessert so seine Erkennungsmöglichkeiten, je häufiger es eingesetzt wird.

■ 1.2 Themenfelder der künstlichen Intelligenz

In den meisten Fällen wird KI als Bezeichnung für Computersysteme benutzt, die Aufgaben abarbeiten, nachdem sie mit großen Datenmengen trainiert wurden. Die Industrie fasst den Begriff KI noch weiter, hier versteht man KI als technologische Methode, die es ermöglicht, menschliche Wahrnehmung und Handeln durch Maschinen nachzubilden. Die Einsatzgebiete der KI sind dabei sehr vielfältig. So gehört auch der erfolgreiche Einsatz in der Robotik dazu, wenn es z. B. im Maschinenbau um die Bearbeitung in Bereichen von Tausendstel-Millimetern geht.

Des Weiteren spielt auch Robotic Process Automation (RPA) eine große Rolle. Hier werden Routine-Aufgaben automatisiert, indem der Software-Roboter diese nachahmt. In einigen Fällen kann sogar der gesamte Geschäftsprozess durch einen Software-Roboter abgebildet werden.

Allerdings wird KI immer nur als reiner Oberbegriff behandelt, es ist daher wichtig zu klären, was man wirklich meint, wenn man über künstliche Intelligenz, maschinelles Lernen und Deep Learning spricht und wie sich die Begriffe zueinander verhalten.

Das Mengendiagramm in Bild 1.1 zeigt, dass künstliche Intelligenz ein Umfangreiches Set an Methoden, Verfahren und Technologien bildet. So stellt jeder Kreis im Mengendiagramm eine entsprechende KI-Technologie dar und somit auch, dass die KI zwar das maschinelle Lernen und damit auch Deep Learning einschließt, darauf aber nicht begrenzt ist. Im täglichen Umgang werden die aufgezeigten Begriffe sehr häufig synonym gebraucht.

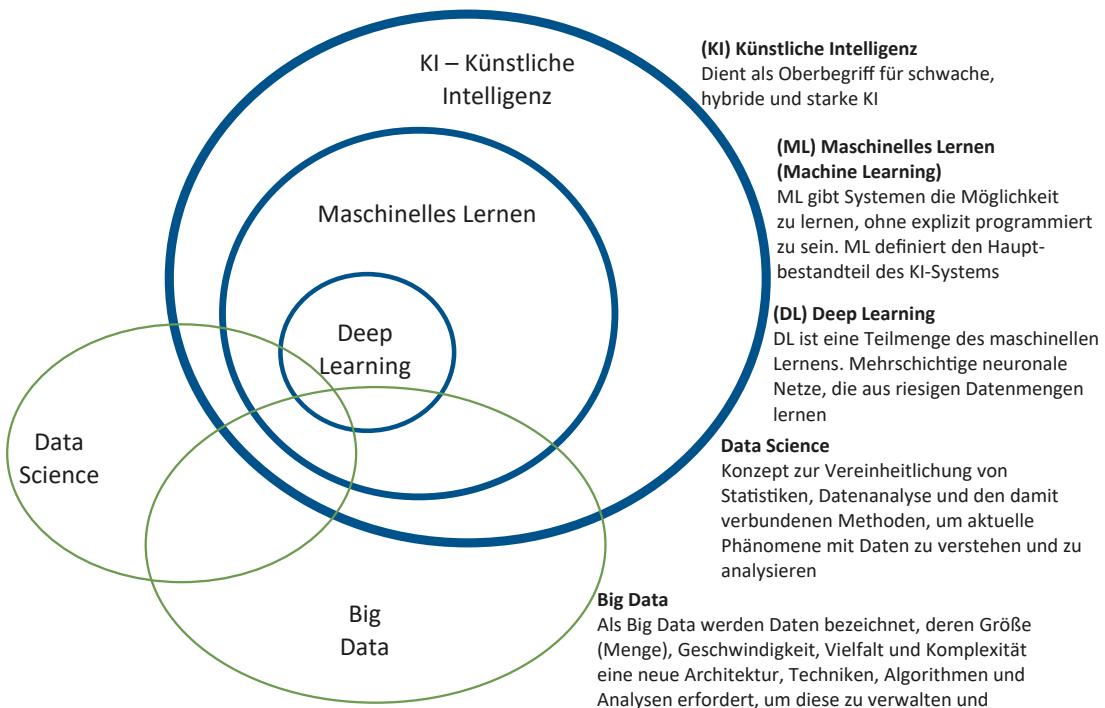


Bild 1.1 Verwandte Themenfelder in der KI

1.2.1 Machine Learning

Machine Learning, kurz als ML bzw. auch als maschinelles Lernen bezeichnet, ist eines der bekanntesten Teilgebiete der Künstlichen Intelligenz und eine Datenanalyse-Methode. ML verfolgt das Ziel, aus Daten zu lernen und möglichst treffende Vorhersagen zu generieren. Das heißt, ein künstliches System lernt aus Beispielen und kann nach Beendigung der Lernphase das Gelernte verallgemeinern. Es lernt hierbei aber die Beispiele nicht einfach auswendig, sondern es erkennt in den Lerndaten Gesetzmäßigkeiten.

Daher werden im Machine Learning Daten gesammelt und aufbereitet, um sie mithilfe von speziellen Machine-Learning-Algorithmen auf einem oder mehreren Rechnern zu trainieren. Dabei leitet der Algorithmus aus den Daten ein Modell ab, das in der Lage ist, bestimmte Eigenschaften in den Daten zu erkennen. Das Ergebnis eines ML-Algorithmus ist immer ein Machine-Learning Model. Fast alle Modelle, die ML-Algorithmen aus Beispielen erzeugen, sind letztlich statistische Modelle, weshalb Statistik ein Fundament für die Theorie des maschinellen Lernens bildet.

Für den Softwarebereich hat Thomas Mitchell es wie folgt beschrieben: Ein Computerprogramm lernt beim Lösen einer bestimmten Klasse von Aufgaben (T), wenn seine messbare Leistung (P) sich mit der Erfahrung (E) im Lauf der Zeit erhöht [4]. Diese Logik zu finden wird maschinelles Lernen genannt. Ausführlicher wird ML im nachfolgenden Kapitel 2 erläutert.

1.2.2 Deep Learning

Deep Learning, kurz als DL oder auch als tiefgehendes Lernen bezeichnet, ist eine Teildisziplin des maschinellen Lernens und wird derzeit am häufigsten im Zusammenhang mit dem Begriff KI verwendet. Die Grundlage von DL sind künstliche neuronale Netze und somit eine spezielle Methode der Informationsverarbeitung. DL wird besonders auf große Datenmengen angewandt, die mithilfe von neuronalen Netzen analysiert werden. Über diese Technologie schafft es das System, Strukturen zu erkennen, Informationen zu sortieren und zu evaluieren. Dabei vollzieht sich ein permanenter Prozess, das Gelernte wird immer wieder mit neuen Inhalten verknüpft und erweitert. Dies führt mit hoher Wahrscheinlichkeit dazu, dass ein richtiges Ergebnis erkannt und ausgegeben wird. DL kommt zum Einsatz, wenn andere ML-Verfahren an ihre Grenzen stoßen.

Auch der Prozess und die benötigten Ressourcen unterscheiden sich im Vergleich zum Machine Learning. Bei ML gibt der Mensch Testdaten mit korrekten Antworten vor, sodass der Algorithmus basierend auf diesen manuell klassifizierten Daten lernt. Beim Deep Learning fällt die Vorgabe der Lösung weg, stattdessen wird die Zuordnung und Klassifizierung der Daten automatisiert. Mehr Informationen zu DL finden Sie in Abschnitt 2.9.

1.2.3 Cognitive Computing

Cognitive Computing stellt einen Teilbereich der KI dar, der eine natürliche, möglichst menschliche Interaktion mit Maschinen anstrebt. Hierfür werden bei Cognitive Computing Technologien der KI genutzt, um menschliche Denkprozesse zu simulieren. Dabei geht es darum, auf Basis von Erfahrungen eigene Lösungen und Strategien zu entwickeln.

Mithilfe von KI und Cognitive Computing soll ein System entstehen, das Bilder und Sprache interpretiert und auch noch schlüssig antworten kann. Die wichtigste Voraussetzung für das Cognitive Computing ist demnach die Fähigkeit, aus den gemachten Erfahrungen selbstständig zu lernen. Gleichzeitig soll das System die eigenen Lösungsansätze ständig hinterfragen.

Ein wesentliches Merkmal des Cognitive Computing ist, dass riesige Datenmengen unterschiedlichster Art gespeichert und binnen kürzester Zeit verarbeitet werden müssen. Da die Daten in der Regel in unstrukturierter Form vorliegen, lassen sich herkömmliche relationale arbeitende Systeme nicht für das Cognitive Computing effizient einsetzen. Daher kommen hier Techniken aus dem Big-Data-Umfeld in Verbindung mit Data Science zum Einsatz.

Das bekannteste Beispiel für Cognitive Computing ist wohl der Sieg 2011 von IBM Watson bei Jeopardy. Hierbei war das System in der Lage, Fragen selbstständig und sinnvoll zu beantworten.

1.2.4 Big Data und Data Science

Big Data ist ein allgemeiner Begriff, der für die Beschreibung umfangreicher Mengen unstrukturierter und semi-strukturierter Daten verwendet wird. Demzufolge bezeichnet Big Data die immer rasanter wachsenden Datenmengen zum Beispiel aus den sozialen Netzwerken. Die wichtigsten Datenquellen für Big Data sind:

- Mobile Internetnutzung
- Social Media
- Geo-Tracking
- Cloud Computing
- Vitaldaten-Messung
- Media-Streaming

Der Begriff Big Data meint aber nicht nur die Daten selbst, sondern auch deren Analyse und Nutzung. Hierfür wird Big Data auch mit Cloud Computing und Machine Learning in Verbindung gebracht, um die Echtzeit-Analyse von großen Datenmengen zu bewerkstelligen. Da die gesammelten Datenmengen allerdings komplex, schnelllebig und unstrukturiert sind, stellt sich die Aufbereitung und Analyse sehr häufig als schwierig und aufwendig dar. An dieser Stelle kommt dann Data Science zum Einsatz.

Bei Data Science handelt es sich um eine Wissenschaft, die sich mit der Extraktion von Wissen aus großen Datenmengen (auch Big Data) beschäftigt. Mit verschiedenen Methoden aus der Datenanalyse und Visualisierung werden die gesammelten Daten erfasst und die benötigten relevanten Informationen extrahiert. Gegenwärtig werden im Data Science Techniken und Methoden aus der Mathematik, der Statistik, der Stochastik und der Informatik angewendet.

1.2.5 Predictive Analytics

Predictive Analytics stellt eine Analysemethode dar, die sowohl neue als auch historische Daten zur Vorhersage von Aktivitäten, Verfahren und Trends verwendet. Man nutzt hier Machine Learning um Vorhersagemodelle zu erstellen, die einen numerischen Wert für die Wahrscheinlichkeit des Eintretens eines bestimmten Ereignisses berechnen. Bei Predictive Analytics kommen folgende Methoden zum Einsatz:

- **Logistische Regression:** Eine Analysemethode, die zur Vorhersage eines Datenwertes auf Grundlage früherer Beobachtungen eines Datensatzes verwendet wird.
- **Zeitreihenanalyse:** Eine Darstellung von Datenpunkten in aufeinanderfolgenden Zeitintervallen.
- **Entscheidungsbaum:** Ein Diagramm, das mithilfe einer Verzweigungsmethode jedes mögliche Ergebnis einer Entscheidung darstellt.

Durch den Einsatz von ML hat Predictive Analytics in den letzten Jahren große Fortschritte gemacht und viel neue Aufmerksamkeit erfahren.

1.2.6 Natural Language Processing

Natural Language Processing, kurz NLP, beschreibt Techniken und Methoden zur maschinellen Verarbeitung natürlicher Sprache. Das heißt, NLP versucht, natürliche Sprache zu erfassen und mithilfe von Regeln und maschinengestütztem Lernen zu verarbeiten. NLP kommt als Teilbereich in folgenden Anwendungen zum Einsatz:

- Spracherkennung
- Segmentierung zuvor erfasster Sprache in einzelne Wörter und Sätze
- Erkennen der Grundformen der Wörter und Erfassung grammatischer Informationen
- Extraktion der Bedeutung von Sätzen und Satzteilen

Ein praktisches Beispiel für den Einsatz von NLP finden Sie in Kapitel 8.

Bild 1.2 zeigt noch einmal alle Teilbereiche der KI in Bezug auf Handeln, Wahrnehmung und Lernen.

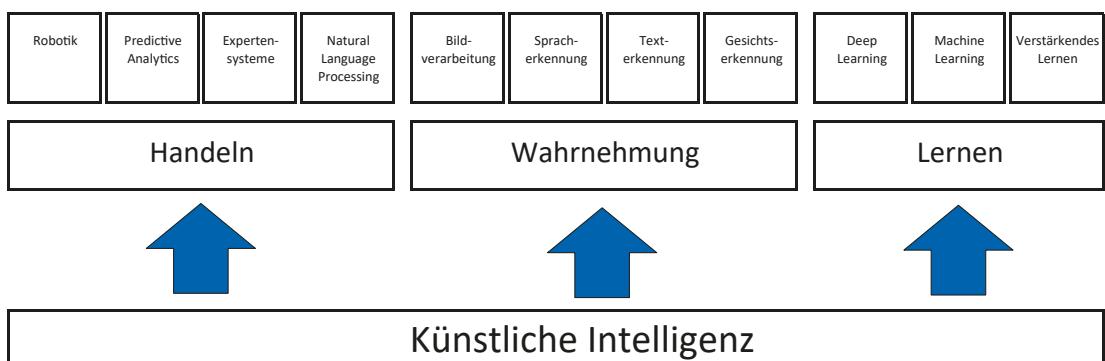


Bild 1.2 Die Teilbereiche Handeln, Wahrnehmen und Lernen in der KI

■ 1.3 KI-Service-Plattformen

Alle großen Internetkonzerne bieten inzwischen eine Zusammenstellung verschiedener APIs (Application Programming Interface), SDKs (Software Development Kit) und Services (Dienste) an, um Ihnen als Entwickler dabei zu helfen, ihre Programme intelligenter und benutzerfreundlicher zu machen.

Über diese KI-Plattformen können Sie auf ausgereifte und schon fertige Services für Lösungen auf Basis von Machine-Learning-Modellen oder Services für Sprach- und Text-Erkennung zurückgreifen. Neben den verschiedenen KI-Services und -Tools bieten diese Plattformen durch die Nutzung in der Cloud auch eine gute Unterstützung bei der Infrastruktur, der Sicherheit, der Verfügbarkeit und natürlich auch den Vorteil der zur Verfügung stehenden Rechenleistung.

Neben diesen KI-Services veröffentlichen die Anbieter wie Google, Amazon, Microsoft und IBM auch den entsprechenden Code ihrer Open Source ML- und DL-Frameworks. So können Sie als Entwickler an der direkten Weiterentwicklung der Frameworks teilnehmen und sich in der jeweiligen Community austauschen.

1.3.1 Amazon

Amazon betreibt mit seinem Web Service (AWS) den größten Cloud Computing Dienst weltweit. Neben den bekannten Infrastruktur-Diensten bietet Amazon auch Services für künstliche Intelligenz an.

Mit Deep Scalable Sparse Tensor Network Engine (DSSTNE) hat Amazon seine Programm-bibliothek für maschinelles Lernen als Open Source zu Verfügung gestellt. Der Schwerpunkt von DSSTNE liegt auf Verfahren, die bei der Entwicklung von ML-Modellen mit nur wenigen eigenen Trainingsdaten auskommen, da das erstellte ML-Modell über einen Cloud Service verteilt trainiert werden kann. Die weiteren KI Services von Amazon bieten die folgenden Möglichkeiten.

Amazon SageMaker

Der Amazon SageMaker ist ein von Amazon vollständig verwalteter Service, der jedem Entwickler die Möglichkeit bietet, schnell Modelle für Machine Learning zu erstellen und diese Modelle zu trainieren, um sie dann für eine Anwendung bereitzustellen. Über dieses Tool wird der gesamte ML-Zyklus von der Datenaufbereitung bis zur Bereitstellung der ML-Modelle abgedeckt.

Amazon Lex

Amazon Lex ist ein Service zur Erstellung einer Schnittstelle für Sprache und Text. Amazon Lex bietet automatische Spracherkennung zur Umwandlung von Sprache in Text und eine Erkennung der Textabsicht, um eine realistische Gesprächsinteraktion in der Anwendung zu gewährleisten.

Amazon Lex bildet den Kern von Amazon Alexa und stellt somit jedem Entwickler diese Technologie zur Verfügung. Somit soll es möglich sein, schnell und einfach komplexe Chatbots zu entwickeln.

Amazon Translate

Amazon Translate ist ein neuraler, maschineller Übersetzungsservice, der kostengünstig Übersetzungen liefert. Hierbei bedeutet neuronale maschinelle Übersetzung, dass Amazon Deep-Learning-Modelle verwendet, um im Vergleich zu herkömmlichen regelbasierten Übersetzungsalgorithmen eine genauere und natürlicher klingende Übersetzung zu liefern.

Amazon Polly

Amazon Polly ist ein Service, der Text in realistische Sprachausgabe verwandelt. Hier kommt über den Text-to-Speech-Service Deep Learning zum Einsatz, um natürlich klingende menschliche Sprache zu synthetisieren. Somit soll es möglich sein, neue sprachfähige Produkte zu entwickeln.

Amazon Rekognition

Hinter Amazon Rekognition steht ein hochgradig skalierbares Deep-Learning-Modell, das es ermöglicht Objekte, Menschen, Text, Szenen und Aktivitäten in Bildern und Videos zu identifizieren. Der Service bietet weiterhin auch eine sehr genaue Gesichtsanalyse- und Gesichtsfunktionen an, mit denen Sie Gesichter analysieren und vergleichen können. Sie benötigen für diesen Service keinerlei Machine-Learning-Kenntnisse.

Amazon RoboMaker

Mit dem Amazon RoboMaker steht eine umfassende Cloud-Lösung für die Entwicklung von Robotersystemen zur Verfügung. RoboMaker ermöglicht das Simulieren, Testen und das sichere Bereitstellen von Roboteranwendungen im großen Umfang.

Apache MXNet

Amazon bietet einige optimierte Services für das Deep-Learning-Framework Apache MXNet an. Bei MXNet handelt es sich um ein schnelles und skalierbares Schulungs- und Interferenz-Framework mit einer kompakten und entwicklerfreundlichen API für Machine Learning. Apache MXNet bietet des Weiteren interessante Einblicke in den Aufbau tiefgreifender Deep-Learning-Systeme für Entwickler.

1.3.2 Google

Google gehört heute zu den führenden Unternehmen im Bereich KI. Neben der Forschung betreibt Google auch eigene KI-Systeme auf der Google Cloud Plattform, die Ihnen als Entwickler zur Verfügung stehen. Die bekannteste Programmiersprache für ML von Google ist TensorFlow.

TensorFlow

TensorFlow ist Googles plattformunabhängiges Open-Source-Framework für maschinelles Lernen. Der Schwerpunkt des Systems liegt in der Verarbeitung von Sprache und Bildern. In der Forschung und im Produktivbetrieb wird das Framework derzeit in verschiedenen Google-Produkten eingesetzt. Hierzu zählen die Google-Spracherkennung, Gmail, Google Fotos, Google Maps und die Google-Suche. Mehr Informationen zu TensorFlow finden Sie in Abschnitt 7.2 „TensorFlow“.

Google Vision API

Die Google Vision API bietet über eine REST API-Schnittstelle leistungsstarke vorabtrainierte Modelle für Machine Learning. Sie können über die Schnittstelle Bilder mit Labels versehen und diese Bilder dann in kurzer Zeit Millionen von vordefinierten Kategorien zuordnen. Außerdem ist es möglich, Objekte und Gesichter zu erkennen sowie gedruckten und handgeschriebenen Text zu lesen.

Cloud AI Platform

Die Cloud AI Platform ermöglicht es Entwicklern, den vollständigen Zyklus der Machine-Learning-Entwicklung in einem Projekt zu managen. Sie unterstützt mit einem integrierten Toolset alle Prozessketten von der Erstellung der Datenbasis bis zu Ausführung und Bereitstellung des ML-Projekts.

AutoML Video Intelligence

AutoML Video Intelligence stellt eine grafische Benutzeroberfläche zur Verfügung, mit der Sie eigene ML-Modelle zur Klassifizierung und Nachverfolgung von Objekten in Videos trainieren können.

Translation API Basic

Die Google Translation API übersetzt Texte aus Websites und Anwendungen in mehr als 100 Sprachen. Translation API verwendet hierfür vortrainierte neuronale Machine-Learning-Übersetzungen und liefert damit schnelle, dynamische Ergebnisse.

AutoML Natural Language

AutoML Natural Language bietet eine Benutzeroberfläche, um eigene ML-Modelle zum Klassifizieren, Extrahieren und Erkennen von Stimmungen trainieren zu können. Sie können Ihre Trainingsdaten direkt über die Oberfläche hochladen und anschließend einfach Ihr erstelltes ML-Modell testen.

Vortrainierte Modelle

Für viele gängige Anwendungsfälle im Bereich Sprachen- und Textanalyse sowie in der Bildverarbeitung stehen Ihnen von Google schon vortrainierte ML-Modelle als API-Schnittstelle für die Nutzung in eigenen Anwendungen zur Verfügung. Da Google laufend weitere Fortschritte in der KI-Forschung erzielt, haben Sie immer Zugriff auf die neusten vollständig trainierten ML-Modelle.

1.3.3 Microsoft Cognitive Services

Neben dem bekannten Open-Source-Framework ML.NET bietet das Softwareunternehmen Microsoft weitere APIs, SDKs und Dienste rund um das Thema KI an. Diese werden als Cognitive Services bezeichnet und sind über die Azure-Plattform erreichbar.

Gesichtserkennungs-API

Die Gesichtserkennungs-API nutzt Algorithmen, um menschliche Gesichter auf Bildern zu erkennen. Diese API bietet Erweiterungen wie Gesichtserkennung, Gesichtsabgleich und Gesichtsgruppierung, um Gesichter basierend auf ihrer Ähnlichkeit in Gruppen einzufügen.

Video API

Die Video API ermöglicht das Erkennen von gesprochenen Wörtern, geschriebenem Text, Gesichtern, Emotionen, Marken oder Szenen automatisch aus Video- oder Audiodateien zu extrahieren. Über eine einfache Benutzeroberfläche kann die API-Schnittstelle konfiguriert werden.

Speech-Service

Der Speech-Service dient zur Vereinheitlichung von Spracherkennung, Sprachsynthese und Sprachübersetzung. Bis Anfang 2020 war dieser Bereich noch in einzelne API-Schnittstellen aufgeteilt. Auch Sprachassistenten können mit diesem umfangreichen Service entwickelt werden.

Freihanderkennungs-API

Diese KI-API unterstützt Sie bei der Erkennung digitaler Freihandinhalte wie handschriftlicher Texte, Formen und Layouts geschriebener Dokumente. Es handelt sich hierbei aber nicht um eine optische Zeichenerkennung wie bei OCR (Optical Character Recognition), sondern um eine Analyse der Daten anhand der Bewegung über Eingabertools wie digitaler Stifte oder Fingerzeichen. Der erkannte Inhalt wird dann über die API-Schnittstelle als JSON-Antwort zurückgeliefert.

Cognitive Toolkit

Auch das Cognitive Toolkit ist ein kostenloses Open-Source-Deep-Learning-Framework, mit dem künstliche neuronale Netze trainiert werden können. Das Toolkit beschreibt hierbei neuronale Netze als eine Reihe von Rechenschritten über einen gerichteten Graphen. Sie können mit dem Cognitive Toolkit beliebige ML-Modelltypen einfach realisieren und diese auch kombinieren. Das Toolkit ist mit vielen verschiedenen Programmiersprachen nutzbar und ermöglicht auch einen zuverlässigen Betrieb mit großen Datenmengen.

1.3.4 IBM

Auch IBM bietet über seine Watson Data Platform verschiedene Dienste für KI-Aktivitäten an. Diese werden als Watson Developer Cloud Services zur Verfügung gestellt.

Watson Natural Language Classifier

Der Natural Language Classifier unterstützt Sie, die Sprache kurzer Texte zu verstehen. Ein Klassifikationsmerkmal, der Classifier, lernt anhand Ihrer Beispieldaten und kann anschließend Informationen zu Texten zurückgeben, anhand derer das ML-Modell nicht trainiert wurde.

Watson Natural Language Understanding

Der Natural Language Understanding-Service dient zur Analyse von sprachlichen Merkmalen in Texteingaben. Es wird versucht, Kategorien, Konzepte oder auch Emotionen aus dem Text zu extrahieren.

Watson Visual Recognition

Auch bei IBMs Visual Recognition werden Deep-Learning-Algorithmen verwendet, um in Bildern Objekte und Szenen erkennen zu können.

Vortrainierte Modelle

Wie Google, stellt auch IBM vortrainierte Modelle für Machine Learning zur Verfügung. Hier haben die ML-Modelle aber einen eindeutigen Bezug auf entsprechende Anwendungsfälle. So gibt es ML-Modelle für die Fertigung, für Versicherungen, den Einzelhandel und den Schulungsbereich.

Watson Assistant

Bei IBMs Watson Assistant handelt es sich um eine sogenannte Konversations-KI-Plattform für die Erstellung von virtuellen Assistenten (Chatbots). Hier ist der Assistent ein kognitiver Bot, den Sie an Ihre Anforderungen anpassen und bereitstellen können, um Ihren Anwendern kontextbezogene Unterstützung anzubieten. Watson Assistant erstellt dynamisch ein Modell für Machine Learning, das auf Ihre definierten aber auch auf ähnliche Benutzeranfragen abgestimmt ist.

Wie Sie sehen, bieten die Konzerne schon eine beachtliche Anzahl von nützlichen Schnittstellen und KI-Diensten an, die auch entsprechend weiterentwickelt werden, und es kommen ständig neue APIs oder KI-Services hinzu.

Durch das Zur-Verfügung-stellen einer entsprechenden API oder auch eines Service ist es für Sie sehr viel einfacher, auf schon vorhandene Funktionen und Modelle zurückzugreifen. Profitieren Sie hier einfach von den Weiterentwicklungen und dem Austausch in der jeweiligen Community, um Ihre Programmentwicklung noch effektiver zu gestalten.

Dieses Buch möchte Sie darin unterstützen, selber Modelle für entsprechende Frameworks zu erstellen oder schon vorhandene zu modifizieren. Nutzen Sie hierfür aber auch die Vorteile der großen Lernplattformen von Microsoft, Google und Co.

■ 1.4 Künstliche neuronale Netze

Unter den lernenden Algorithmen sind die künstlichen neuronalen Netze die Schlüsseltechnologie für Machine und Deep Learning. Um aber die Möglichkeit von ML und DL auszukosten und richtig zu nutzen, ist es hilfreich, wenn man weiß, wie neuronale Netze tatsächlich funktionieren.

Die Zusammenhänge im Detail versteht man am besten, wenn man ein eigenes neuronales Netz programmiert. Das so erworbene Wissen lässt sich dann bei der Verwendung eines KI-Services oder eines entsprechenden Frameworks sicher und effektiv nutzen.

1.4.1 Funktionsweise

Künstliche neuronale Netze (engl. *artificial neural network* – ANN), kurz als KNN bezeichnet, bestehen aus Knoten, die Sie als Analogie zu menschlichen Nervenzellen sehen können. Diese Knoten stellen die einzelnen Neuronen dar, die auch gerne als Units oder Einheiten bezeichnet werden.

Sie dienen dazu, Informationen aus der Umwelt oder von anderen Neuronen aufzunehmen und diese wiederum an andere Neuronen in modifizierter Form weiterzuleiten. Diese Neuronen sind in Schichten, den sogenannten Layers angeordnet, in denen sich mehrere Neuronen befinden. Jedes Neuron einer Schicht (Layer) ist mit allen Neuronen der nächsten Schicht verbunden. Somit besteht ein KNN aus Neuronen mit gerichteten und gewichteten Verbindungen. Bild 1.3 zeigt die schematische Darstellung eines neuronalen Netzes.

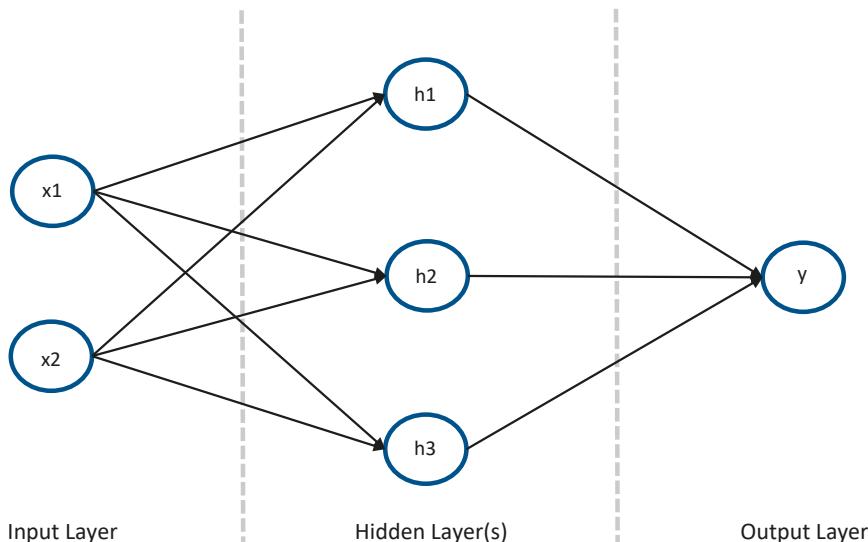


Bild 1.3 Schematische Darstellung eines künstlichen neuronalen Netzes (KNN)

Die Schichten eines neuronalen Netzes werden im Regelfall in folgende Kategorien unterteilt:

- **Input Layer:** Die Eingabeschicht repräsentiert die Eingabedaten in das neuronale Netz, also die gewünschten Daten, auf die eine Klassifikation angewendet wird.
- **Hidden Layer:** Die verborgene Zwischenschicht bildet sich wie folgt: Zwischen den Neuronen der Eingangsschicht und deren Ausgangsschicht befinden sich eine oder auch mehrere versteckte Schichten. Gibt es mehr als nur eine versteckte Schicht, handelt es sich um ein sogenanntes tiefes neuronales Netz, auch bekannt unter dem Begriff Deep Neural Network. Theoretisch ist die Anzahl der möglichen verborgenen Schichten in einem künstlichen neuronalen Netz unbegrenzt. In der Praxis bewirkt jede hinzukommende verborgene Schicht jedoch auch einen Anstieg der benötigten Rechenleistung.
- **Output Layer:** Die Ausgangsschicht ist mit den Neuronen der versteckten Schicht verbunden und repräsentiert das Ergebnis der Informationsverarbeitung.

1.4.2 Netztypen

Je nachdem, welche Aufgabe ein neuronales Netz erledigen soll, ob eine einfache Klassifikation, Bildanalysen oder Übersetzungen, werden neuronale Netze nach unterschiedlichen Gesichtspunkten aufgebaut. Dabei unterscheidet man zwischen grundsätzlich verschiedenen Strukturen, die als Topologien bezeichnet werden. Die Topologie eines Netzes, also die Zuordnung der Verbindungen zwischen den Neuronen, muss abhängig von der Aufgabenstellung gewählt werden. Die meisten heute im Einsatz befindlichen neuronalen Netze kombinieren verschiedene Formen der Netz-Architektur. Die folgende Aufzählung zeigt die wichtigsten Netztypen:

- *Single-Layer Perceptron (Perzeptron)* ist die einfachste Form eines neuronalen Netzes. Es besteht nur aus einem einzigen Neuron. Mehr dazu erfahren Sie in Abschnitt 2.8.
- *Feedforward Neural Network (vorwärtsgekoppelte Netze)* sind Netze, in denen die Verbindungen nur in eine Richtung gehen, von der Eingabe zur Ausgabe.
- *Recurrent Neural Network (rekurrente Netze)* sind Netze, in denen es auch Verbindungen in Rückwärtsrichtung gibt, sodass Rückkopplungen entstehen können (siehe auch Kapitel 5).
- *Convolutional Neural Network* sind Netze, die eine Sonderform des künstlichen neuronalen Netzes darstellen. Sie werden insbesondere im Bereich der Bild- und Audioverarbeitung eingesetzt. Dieses Netz lernen Sie in Kapitel 6 genauer kennen.
- *Asynchrone Netze* sind Netze, in denen die Neuronen nicht alle auf einmal (synchron), sondern nacheinander in zufälliger Reihenfolge aktiviert werden.
- *Symmetrische Netze* sind Netze, die von jedem Betrachtungspunkt eines Neurons gleich aussehen. Das heißt, die Neuronen in einem symmetrischen Netz verhalten sich identisch. Die Symmetrie erleichtert die Programmierung, da keine besonderen Fälle bei der Verbindung der Neuronen zu beachten sind.
- *Selbstassoziative Netze* sind Netze, in denen Eingabe- und Ausgabe-Neuronen übereinstimmen.
- *Zyklische Netze* sind Netze, in denen sich einige Neuronen gegenseitig stimulieren. Auch hier gibt es die Möglichkeit einer Rückkopplung.

Eine Betrachtung und Beschreibung aller möglichen KNNs würde den Rahmen dieses Buches sprengen. Daher beschränken wir uns hier auf die Beschreibung der grundlegenden Strukturen von Feedforward Neural Networks, Recurrent Neural Networks und den Convolutional Neural Networks.

Feedforward Neural Network

Ein Feedforward Neural Network, kurz FNN, besitzt beliebig viele Layer (Schichten). Die Daten werden hierbei von einem Layer über verschiedene gewichtete Verbindungen zum nächsten Layer weitergeleitet. Der Informationsfluss findet ausschließlich vorwärtsgerichtet von den Input Layer über die Hidden Layer zu den Output Layer statt. Es existieren somit keine Verbindungen, welche zurückführen. Des Weiteren handelt es sich bei einem FNN immer um ein vollständig verbundenes (fully connected) neuronales Netz, da immer sämtliche Neuronen einer Schicht mit allen der darauffolgenden verbunden sind. Das FNN wird sehr häufig für die Ermittlung einer Prognose verwendet. Eine praktische Programmieraufgabe mit diesem Netz finden Sie in Abschnitt 3.4.

Recurrent Neural Network

Ein Recurrent Neural Network, kurz RNN, kann, im Gegensatz zu einem FNN, Informationen wieder in vorherige Layers (Schichten) leiten. Somit existieren in RNNs Verbindungen, bei denen Informationen bestimmte Neuronen-Verbindungen des neuronalen Netzes rückwärts und anschließend erneut vorwärts durchlaufen können. In den meisten Fällen werden RNN in der Spracherkennung, Übersetzungen und Handschrifterkennung eingesetzt.

Hierbei sind folgende Varianten möglich:

- **Direkte Rückkopplung:** Ein Neuron nutzt seinen Output als erneuten Input.
- **Indirekte Rückkopplung:** Der Output eines Neurons wird als Input eines Neurons in einer vorgelagerten Schicht (Layer) verwendet.
- **Seitliche Rückkopplung:** Der Output eines Neurons wird als Input eines Neurons in derselben Schicht (Layer) verwendet.
- **Vollständige Verbindung:** Der Output eines Neurons wird von jedem anderen Neuron im Netz als zusätzlicher Input verwendet.

Dieses Netz lernen Sie ausführlich in Kapitel 5, „Recurrent Neural Network“ kennen.

Convolution Neural Network

Ein Convolutional Neural Network, kurz CNN, ist ein spezielles neuronales Netz. Es besitzt mehrere Faltungsschichten und ist dadurch in der Lage, Input in Form einer Matrix zu verarbeiten. Die Architektur eines CNN unterscheidet sich deutlich von der eines klassischen FNN.

Die Topologie gestaltet sich bei CNNs aus den Convolutional Layers und Pooling Layers. Hierdurch ist eine Untersuchung des Inputs aus verschiedenen Perspektiven möglich und es eignet sich daher besonders für das maschinelle Lernen im Bereich Bild- und Spracherkennung. Mehr zu CNNs erfahren Sie in Kapitel 6.

1.4.3 Anwendungsbereiche

Die Bandbreite der Anwendungsbereiche für künstliche neuronale Netze hat entsprechend der Weiterentwicklung von Hard- und Software zugenommen. KNNs werden heute in den unterschiedlichsten Bereichen zur Informationsverarbeitung eingesetzt. Typische bekannte Anwendungen sind:

- Bilderkennung
- Spracherkennung
- Mustererkennung
- Schrifterkennung
- Frühwarnsysteme
- Simulation komplexer Systeme

Vor allem Bild-, Muster- und Spracherkennung werden in Industrieanwendungen aus den Bereichen Qualitätssicherung, Konstruktion und Entwicklung erfolgreich eingesetzt. In der Automatisierung und Produktion sorgen sie für eine Optimierung der Prozesse.

■ 1.5 Grundbaustein Neuron

Ein künstliches neuronales Netz ist ein mathematisches Modell, das durch das Vorbild verzweigter biologischer Nervenzellen, den sogenannten Neuronen, inspiriert worden ist. Somit stellt das Neuron auch in dem mathematischen Modell den Grundbaustein dar. Aus mathematischer Sicht besteht das Neuron im Wesentlichen aus einer gewichteten Verknüpfung der Eingänge, einer Aktivierungsberechnung und einer Ausgabefunktion. Das künstliche Neuron im Modell stellt einfach einen Prozessor mit einer bestimmten Anzahl von Ein- und Ausgaben dar. Hierbei unterscheidet man bei der Aktivität eines Neurons zwischen dem Aktiv-Zustand, dem Senden eines Signals, und dem Inaktiv- oder Ruhezustand.

Jede Verbindung der Neuronen kennzeichnet den Datenfluss von einem Neuron zu einem anderen. Bild 1.4 zeigt die Notwendigen Elemente in einem Neuron. Die übermittelten Daten sind einfach Zahlwerte, die unterschiedlich gewichtet sind. Das Neuron verknüpft die eingehenden Werte X_1, \dots, X_n und die Gewichtungen W_1, \dots, W_n mathematisch.

Das heißt, die Eingabesignale X_1, \dots, X_n werden mit den Gewichtsfaktoren W_1, \dots, W_n multipliziert. Alle Eingabeinformationen mit den Gewichten der Verbindungen sind somit zu einer einzigen Netzeingabe verknüpft. Die Netzeingabe stellt somit die Summe aller Informationen dar, die aus dem Netz an das Neuron weitergegeben werden. Dementsprechend erhält man über die verwendete Propagierungsfunktion $p(X_1, \dots, X_n)$ das zusammengefasste Gesamt- bzw. das Netto-Eingabesignal.

Wenn die Summe über das Produkt aller Eingaben mit den entsprechenden Gewichten größer als der Schwellenwert für die Aktivierungsfunktion ist, wird die Ausgabe auf aktiv gesetzt, sonst ist die Ausgabe passiv. Der Schwellenwert kann auch durch einen weiteren Eingang, dem sogenannten *Bias bn*, bestimmt werden.

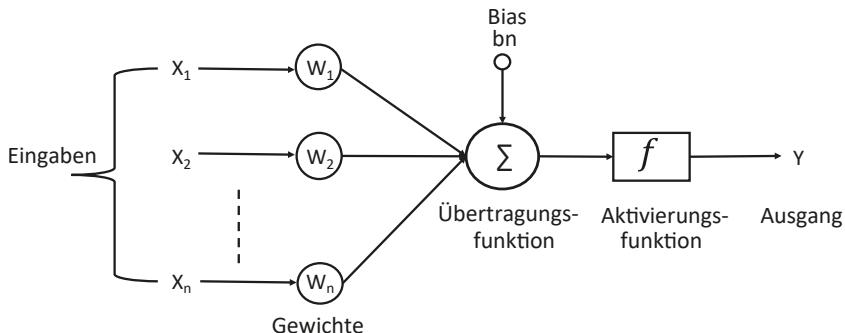


Bild 1.4 Schema des künstlichen Neurons

Bei dem Bias handelt es sich um einen Initialwert. Das Resultat entscheidet darüber, ob das Neuron aktiviert ist und somit zum Ergebnis beitragen soll. Es zählt hierbei die einfache Regel, dass ein Wert $v \leq 0$ einer Nicht-Aktivierung und ein Wert $v > 0$ einer Aktivierung entspricht.

Somit kann man das Neuron in folgende Teile untergliedern:

- Eine oder mehrere eingehende Verbindungen X_1 bis X_n , welche numerische Ausgangssignale von anderen Neuronen empfangen. Dabei wird jeder Verbindung eine Gewichtung W_1 bis W_n zugewiesen, die verwendet wird, um das jeweilige gesendete Signal zu verstärken bzw. zu drosseln.
- Eine Aktivierungsfunktion, die den numerischen Wert des Ergebnis-Signals bestimmt, welches das Neuron aussendet.
- Eine oder eventuell mehrere Ausgangsverbindungen, die das Ergebnis-Signal zu den anderen Neuronen übertragen.



Bias

Je nachdem, welches Problem mit dem neuronalen Netz gelöst werden soll, kann es sein, dass eine bestimmte Aktivierungsfunktion nicht optimal ist. Hierfür gibt es den Bias-Wert. Dieser Wert ist ein zusätzliches Gewicht, das bestimmt, wie groß eine gewichtete Summe sein muss, um den Schwellenwert der Aktivierungsfunktion zu überschreiten. Der Bias-Wert verschiebt somit das Grundniveau der Aktivierungsfunktion. In der Praxis spricht man beim Bias auch von der systematischen Verzerrung, wobei dieser Wert für die Berechnung einfach als weiteres Gewicht behandelt wird.

1.5.1 Aktivierungsfunktion

Die Aktivierungsfunktion, auch als Transfer- oder Übertragungsfunktion bekannt, stellt die Verbindung zwischen dem Netzinput und dem Aktivitätslevel eines Neurons dar.

Hierbei handelt es sich um eine mathematische Funktion, die auf das Zwischenergebnis, also die Summe der Eingaben multipliziert mit den Gewichtungen (Bild 1.4), angewendet wird.

Somit entscheidet die Aktivierungsfunktion, ab welchem Potenzial das Neuron aktiv wird. Dementsprechend lässt sich für das Neuron ein Aktivierungspotenzial simulieren. Zusätzlich werden die Ausgabewerte der Neuronen einer Schicht normalisiert.

Die Aktivierungsfunktion hat für gewöhnlich einen dynamischen Ausgabebereich zwischen -1 und 1 oder zwischen 0 und 1, wobei bei einigen Aktivierungsfunktionen die Ausgabegrenzen breiter sein können bzw. auch bis unendlich gehen. Die verschiedenen Aktivierungsfunktionen unterscheiden sich in ihrer Komplexität und Ausgabeart. Bild 1.5 zeigt die am häufigsten verwendeten Aktivierungsfunktionen für neuronale Netze.

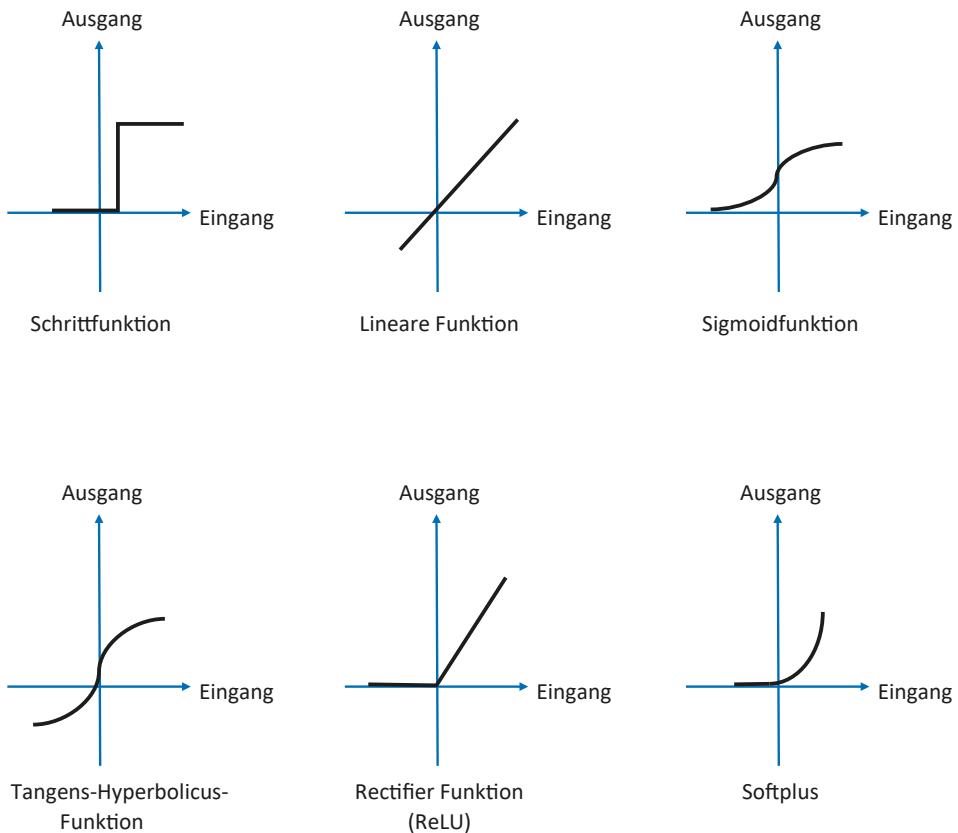


Bild 1.5 Aktivierungsfunktionen

Bei der Auswahl einer Aktivierungsfunktion gibt es keine einfache Faustregel. Alles hängt von den Netzeingabedaten ab und auch davon in welcher Form die Daten nach dem Aktivierungsfunktionsaufruf umgewandelt werden sollen. In der Praxis haben sich aus mathematischer Sicht die Sigmoid-, die Tangens-Hyperbolus (\tanh)-, aber auch die ReLU-Funktion als nützlich erwiesen, da sie sich besonders gut für die Klassifizierung und Mustererkennung eignet. Insgesamt unterscheidet man zwischen folgenden Aktivierungsfunktionen:

- **Schrittfunction (binäre Schwellenfunktion):** Hier gibt es nur zwei Zustände des Aktivitätslevels, 0 (bzw. manchmal auch -1) oder 1.

- **Lineare Funktion:** Hier ist der Zusammenhang zwischen Netzinput und Aktivitätslevel linear. Die lineare Funktion ermöglicht auch das Arbeiten mit einer Schwelle. Das heißt, bevor der Zusammenhang zwischen den beiden Größen linear wird, muss eine zuvor festgelegte Schwelle überschritten werden.
- **Sigmoidfunktion:** Diese Art von Aktivierungsfunktion wird in vielen Modellen verwendet, die kognitive Prozesse in KNNs simulieren. Es handelt sich um eine differenzierbare reelle Funktion mit einer durchweg positiven oder negativen ersten Ableitung und genau einem Wendepunkt. Mit der Sigmoidfunktion sind Berechnungen wesentlich leichter durchzuführen als mit anderen s-förmigen Funktionen. Sie werden die Sigmoidfunktion in den Praxisbeispielen in diesem Buch des Öfteren einsetzen.
- **Tangens-Hyperbolicus-Funktion (tanh):** Diese Aktivierungsfunktion verhält sich relativ ähnlich wie die Sigmoidfunktion. Die Funktion $\tanh(x)$ ist ebenso sigmoid, stetig und differenzierbar und hat als Wertebereich das Intervall $(-1, 1)$. Auch diese Funktion werden Sie später noch in den Praxisbeispielen kennenlernen.
- **Rectifier-Funktion (ReLU):** Diese Funktion steht im Kontext künstlicher neuronaler Netze als Gleichrichter und definiert eine Aktivierungsfunktion, die als Positivteil ihres Arguments definiert ist. Für den Einsatz im Deep Learning ist diese Funktion dabei, die weit verbreitete Sigmoidfunktion abzulösen. Haupteinsatzzweck ist das Convolutional Neural Network (CNN).
- **Softplus-Funktion:** Auch Softplus ist eine neuere Funktion als Sigmoid oder tanh. Sowohl die ReLU-Funktion als auch Softplus sind weitgehend identisch, außer in der Nähe von 0, hier ist die Softplus-Funktion glatt und differenzierbar. Softplus bietet eine Alternative zu den anderen Funktionen, da sie schnell differenzierbar und ihre Ableitung leicht nachvollziehbar ist.

Wie Sie sehen, weisen die beschriebenen Aktivierungsfunktionen allesamt große mathematische Ähnlichkeiten auf. Es gibt keine universelle Regel für die Auswahl einer Aktivierungsfunktion und es gibt auch keine Aktivierungsfunktion, die für alle Anwendungsfälle funktioniert. So ist die lineare Aktivierungsfunktion ideal für die Lösung eines Regressionsproblems. Die Sigmoidfunktion kann den linearen Übergang nur annähernd beschreiben, jedoch niemals exakt abbilden. Die ReLU-Funktion ist die gängigste Aktivierungsfunktion in Convolutional Neural Networks, die bei der Erkennung von Texten und Objekten in Bildern verwendet wird. Persönlich verwende ich sehr gerne die tanh-Funktion, weil sie gut begrenzt und sehr schnell zu berechnen ist, aber vor allem, weil sie für meine Anwendungsbereiche am besten funktioniert. Sie können aber auch Aktivierungsfunktionen mischen und anpassen, um eine spezielle Lösung für Ihren Anwendungsbereich zu entwickeln.

Am Ende jeden Neurons steht die Ausgabefunktion. Da diese Funktion fast ausschließlich nur ihr Argument zurückgibt, wird sie auch als identische Abbildung oder Identität bezeichnet. An einem einfachen Beispiel sehen Sie in Bild 1.6, wie man sehr schnell und einfach errechnen kann, ob das Neuron aktiv wird oder ruht. Als Aktivierungsfunktion wird die einfache Schrittfunktion verwendet.

Um jetzt zu bestimmen, ob das Neuron N_1 aktiv wird, müssen Sie zuerst die Netto-Eingabe ermitteln. Die Netto-Eingabe (net) entspricht, wie schon erläutert der Summe der gewichteten Eingangssignale. Daher kann sie wie folgt berechnet werden:

$$net = X_1 \cdot W_1 + X_2 \cdot W_2 = 1 \cdot 0,6 + 1 \cdot (-0,4) = 0,2.$$

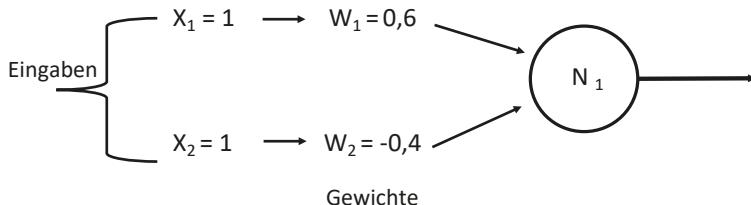


Bild 1.6 Neuron mit Eingabewerte und Gewichten

Die Netto-Eingabe net mit dem Wert 0,2 stellt das Argument der Schrittfunktion dar, die bestimmt, ob das Neuron aktiv wird. Die Berechnung $f(net) = f(0,2) = 1$ stellt das Neuron aktiv, da $0,2 > 0$ und dadurch bestimmt wird, dass das Neuron aktiv wird.

Das Beispiel zeigt das einfachste künstliche neuronale Netz mit nur einem Neuron. Es wird als Linear Threshold Unit, kurz LTU, bezeichnet und stellt mathematisch ein Schwellenwertelement dar, das eine Verarbeitungseinheit für Zahlen mit n Eingängen X_1, \dots, X_n und einem Ausgang bildet. Das Element hat einen Schwellenwert und jeder Eingang ist mit einem Gewicht versehen. Mit den geeigneten Gewichten und einem entsprechenden Schwellenwert lässt sich mit einer LTU eine logische UND-Verknüpfung nachbilden.

Wie die Aktivierungsfunktion praktisch eingesetzt wird, lernen Sie in Kapitel 3, wenn es um den Aufbau eines ersten neuronalen Netzes in C# geht.

1.5.2 Matrizendarstellung

Wie Sie an dem einfachen Beispiel gesehen haben, kann die Berechnung der Eingaben, Gewichte und der Ausgaben bei einem größeren Netz zu sehr viel Arbeit führen. Stellen Sie sich den Rechenaufwand bei einem neuronalen Netz mit drei Schichten zu je 50 Knoten vor. Allein das Notieren aller erforderlichen Berechnungen wäre eine riesige Aufgabe. Hier kann die Matrizenmultiplikation sehr nützlich sein.

Künstliche neuronale Netze lassen sich glücklicherweise auch sehr schnell als Matrizen darstellen. Bild 1.7 zeigt die schematische Darstellung und die entsprechende Matrzenschreibweise.

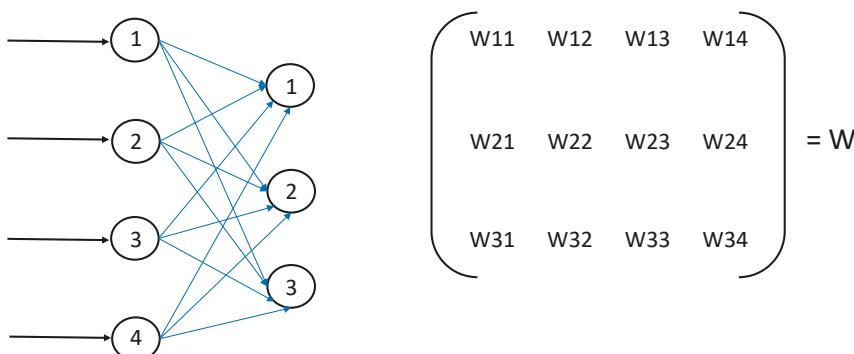


Bild 1.7 Matrzenschreibweise

Diese Darstellung hat den Vorteil, dass die zuvor gezeigten Berechnungen mathematisch relativ einfach sind und zusammenfassend vorgenommen werden können. Infolgedessen stellt die Matrix eine mathematische Einheit dar, genau wie eine einzelne Zahl.

So besteht eine Matrix W aus einer Menge von Elementen W_{ij} . In der hier verwendeten Darstellung bedeutet W_{ij} die Verbindung von Zelle i nach Zelle j . Da das Lernen in neuronalen Netzen in den Gewichten stattfindet und diese das gelernte Wissen des Netzes speichern, werden die Netzgewichte als Matrix dargestellt. In der Schreibweise $W = w_{ij}$ gilt dann:

- Ein Eintrag in der Matrix W mit $W_i = 0$ gibt an, dass keine Verbindung zwischen i und j existiert.
- Ein Eintrag in der Matrix W mit $W_i < 0$ gibt an, dass Neuron i seinen Nachfolger j durch ein Gewicht der Stärke w_{ij} kennt.
- Ein Eintrag in der Matrix W mit $W_i > 0$ gibt an, dass Neuron i seinen Nachfolger j durch ein Gewicht der Stärke w_{ij} anregt.

Beachten Sie auch, dass die Matrixelemente nicht zwingend Zahlen sein müssen, sie können auch Größen sein, denen Sie einen Namen geben, aber keinen numerischen Wert zuweisen.

Bei der Berechnung von mehrschichtigen Netzen, zum Beispiel von einem dreischichtigen Netz mit einem Hidden Layer, würden Sie zwei Gewichtsmatrizen benötigen, bei zwei Hidden Layern drei Matrizen usw. Die Multiplikation von Matrizen ist etwas gewöhnungsbedürftig. Als Voraussetzung für die Durchführbarkeit der Multiplikation muss die Anzahl der Spalten der linkssstehenden Matrix gleich der Anzahl der Zeilen der rechtsstehenden Matrix sein.

Die sich ergebende Matrix hat so viele Zeilen wie die rechte Matrix und so viele Spalten wie die linke Matrix. Aber keine Angst vor der Mathematik. Die Programmiersprache C# stellt mit der Methode `Matrix.Multiply(Matrix, Matrix)` eine sehr schnelle und effektive Matrixberechnung für unsere Aufgabenstellungen zur Verfügung.

■ 1.6 Architekturprinzipien

Die Struktur eines neuronalen Netzes, ob als FNN, RNN oder auch CNN, besitzt eine große Bedeutung für die Ergebnisse des Lernprozesses. Die Netzarchitektur bestimmt somit das zu verwendende Lernverfahren zur Berechnung der Netzgewichte zwischen den Neuronen.

Das einfachste Prinzip finden Sie bei den einschichtigen Netzen, die nur aus einer Neuronenschicht bestehen und bei denen eine bestimmte Anzahl von Neuronen einen Input-Vektor aufnehmen und andere Neuronen einen Output-Vektor ausgeben. Die Erweiterung des Prinzips stellen dann die geschichteten Netze dar. Diese repräsentieren eine Hierarchie und sind damit leistungsfähiger als einschichtige Netze.

Die jeweilige Topologie sollten Sie daher auf jeden Fall vor der Entwicklung eines neuronalen Netzes festlegen:

- Größe der Eingangs-/Ausgangsschicht steht fest
- Anzahl der verdeckten Schichten ist variabel

- Größe der verdeckten Schichten ist variabel
- Entweder strenge Schichtenarchitektur oder beliebige vorwärts gerichtete Schichtenarchitektur

Leider gibt es hier kaum Standard-Regeln. Sie können aber, wenn Sie Ihr erstes eigenes einfaches Netz entwickeln mit folgenden Annahmen experimentieren:

- Es gibt eine verdeckte Schicht und eine Ausgangsschicht.
- Der Eingabevektor hat meist 20 bis 200 Elemente.
- Der Ausgabevektor hat meist 2 bis 100 Elemente.
- Die Anzahl der Neuronen in der verdeckten Schicht liegt bei 50 bis 500.

Des Weiteren sollten Sie die neuronale Netzstruktur hinsichtlich des Anwendungsbereichs festlegen, welcher einer der folgenden sein kann:

- Mustererkennung
- Optimierungsprobleme
- Roboterkontrolle und Überwachung
- Entscheidungstheorie und Klassifizierung

Aber keine Angst, Sie lernen in den nachfolgenden Kapiteln, welche neuronale Netzstruktur für welchen Verwendungszweck am besten geeignet ist.

2

Konzepte und Methoden von Machine Learning

Wie bereits in Kapitel 1 erläutert handelt es sich bei Machine Learning (ML) um ein Teilgebiet der künstlichen Intelligenz (KI). Der Begriff Machine Learning wurde schon 1959 von Arthur Samuel [5] geprägt und somit ist Machine Learning eines der ältesten Gebiete innerhalb der künstlichen Intelligenz.

ML hat sich als sehr praktisch für viele heute verfügbaren Anwendungen erwiesen. Jeden Tag wird auf der Welt eine Unmenge von Daten generiert, von denen viele in unstrukturierter Form vorliegen. Dabei kann es sich um Audio- oder Videodaten, Dokumente oder um Sensordaten in Produktionsmaschinen handeln.

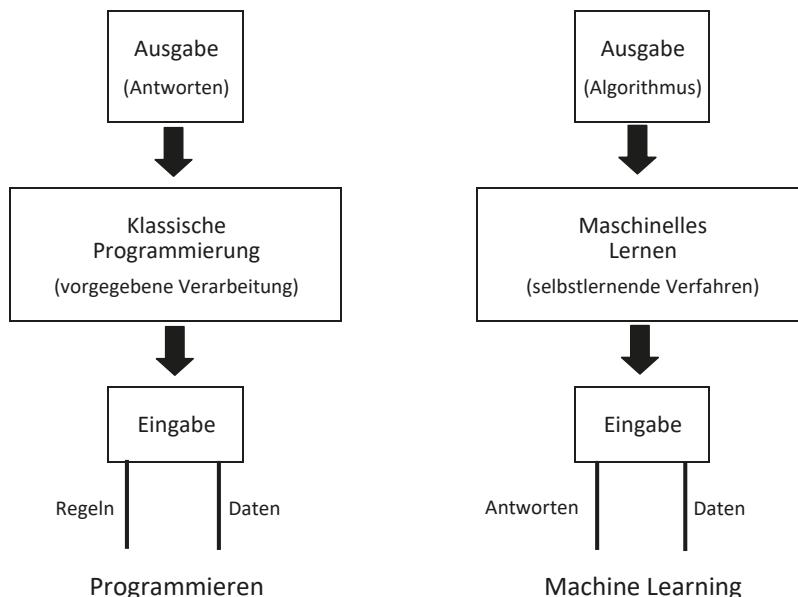
Durch die unüberschaubare Menge von Daten ist der Mensch nicht mehr in der Lage die Daten für die Lösung bestimmter Aufgaben in einer angemessenen Zeit zu analysieren. Somit versucht man heute die Analyse zu automatisieren, um die gewonnenen Daten nutzen zu können. Hier schlägt dann die Stunde von Machine Learning.

In diesem Kapitel lernen Sie die Grundlagen von Machine Learning, inklusive der Lernformen kennen. Es werden der Machine Learning Workflow und die wichtigsten ML-Algorithmen vorgestellt. Des Weiteren erforschen Sie die Möglichkeiten eines einfachen neuronalen Netzwerks sowie die Trainings- und Validierungsmöglichkeiten eines Machine-Learning-Modells.

■ 2.1 ML – Machine Learning

Beim Machine Learning geht es darum, Muster und Gesetzmäßigkeiten in großen Datensätzen zu erkennen. Diese Muster können dann wiederum genutzt werden, um spätere Entscheidungen und Handlungen abzuleiten. Das heißt, dass Ziel von ML ist es, Daten intelligent miteinander zu verknüpfen, Zusammenhänge zu erkennen, Rückschlüsse zu ziehen um dann entsprechende Vorhersagen treffen zu können.

Aus Sicht eines Entwicklers handelt es sich bei ML um eine Sammlung mathematischer Methoden, die für jede Problemlösung eine Vielzahl verschiedener Algorithmen zur Verfügung stellt. Diese Algorithmen sind der Schlüssel zu ML, während man in der klassischen Programmierung (Bild 2.1) über das Schreiben von Programmcode das gewünschte Ziel erreichen möchte, wird dies bei ML über selbstlernende Algorithmen erreicht.

**Bild 2.1** Programmierung vs. Machine Learning

Damit ML aber funktioniert und der Algorithmus Entscheidungen treffen kann, muss ein Anwender den Algorithmus trainieren. Daher müssen für diesen sogenannten maschinellen Lernprozess entsprechende Vorleistungen erbracht werden. Das ML-System muss mit relevanten Daten sowie mit dem geeigneten Algorithmus versorgt werden. Zusätzlich müssen Regeln für die Datenanalyse sowie die Mustererkennung definiert und festgehalten werden. Durch das Bereitstellen von Trainings- und Beispieldaten, wird die Grundlage dafür geschaffen, dass der Algorithmus Muster und Zusammenhänge erkennen und somit aus den Daten lernen kann. Diesen Prozess bezeichnet man in der Praxis als Modelltraining, da der Algorithmus ein statistisches Modell entwickelt. Dieses Modell kann dann zur Lösung der praktischen Aufgabe eingesetzt werden.

Verfügen Sie also über Trainings- und Beispieldaten, so können Sie ML Algorithmen z. B. für folgende Aufgaben einsetzen:

- Vorhersage von Werten aus Basis der analysierten Daten treffen
- Berechnung von Wahrscheinlichkeiten
- Erkennen von Gruppen in einem Datensatz
- Erkennen von Zusammenhängen in Sequenzen
- Optimierung von Prozessen

Ein beliebtes Anwendungsbeispiel für Machine Learning ist die Bilderkennung (Mustererkennung). ML ermöglicht es Inhalte auf Bildern zu identifizieren, weil Algorithmen zuvor anhand von Millionen von Bilddaten darauf trainiert wurden, die jeweiligen Strukturen in den Datenmassen zu erkennen, und daraus die Schlussfolgerung zu ziehen, dass es sich zum Beispiel um einen Hund, eine Katze oder Gesichter oder Zahlen handelt – je nachdem, wie die Aufgabenstellung lautet.

■ 2.2 Algorithmen und Modelle

Machine Learning nutzt statistische und mathematische Methoden sowie Algorithmen um Aufgaben zu lösen. Es existieren inzwischen eine Vielzahl von Modelltypen und Algorithmen, die jeweils für unterschiedliche Aufgaben geeignet sind. Des Weiteren unterscheidet man zwischen unterschiedlichen Lernformen, wie zum Beispiel überwachtes Lernen, unüberwachtes Lernen und bestärkendes Lernen, die für jeweils andere Zwecke geeignet sind.

Bei der Auswahl der passenden Lernform wie auch des Algorithmus ist im Wesentlichen darauf zu achten, welchen Nutzen die Art des Lernens und der Algorithmus für den jeweiligen Anwendungsfall bietet und welche Daten dafür benötigt werden. Bild 2.2 zeigt die zurzeit aktuellen Lernformen und verwendeten Algorithmen in einer Übersicht.

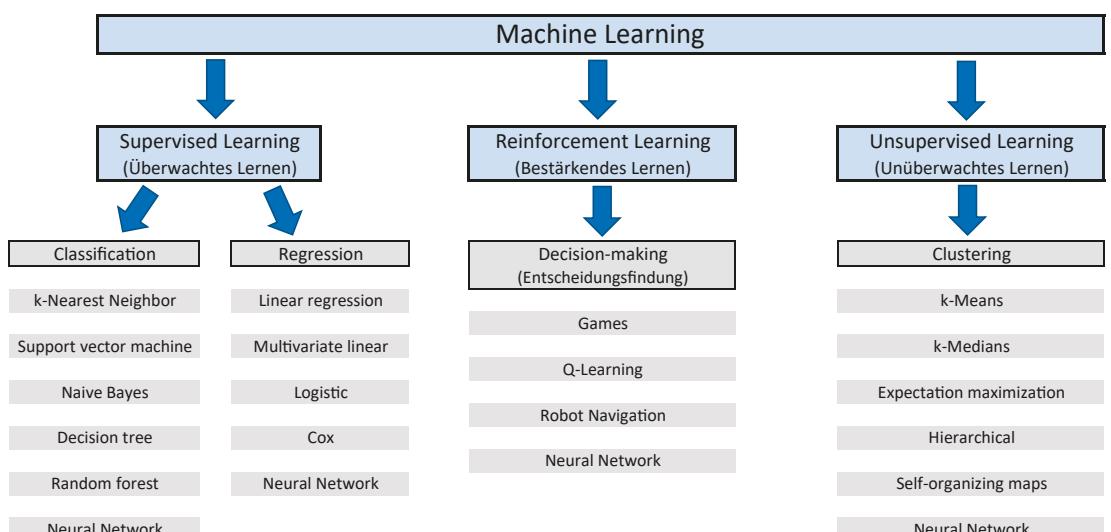


Bild 2.2 Arten des maschinellen Lernens

Die Wahl der passenden Lernmethode und des für die Problemlösung optimalen Algorithmus ist abhängig von den Anforderungen an Geschwindigkeit, Vorhersagegenauigkeit, der Trainingszeit, der Datenmenge, die für das Training benötigt wird, und davon wie einfach es ist den Algorithmus zu implementieren.

Darüber hinaus gibt es auch Problemstellungen für die Machine Learning nicht der passende Lösungsansatz ist. Wenn Sie für Ihre Lösung einen Zielwert bestimmen können, indem Sie Regeln, Berechnungen oder vorbestimmte Schritte programmieren, bietet sich weiterhin die klassische Programmierung an. Verwenden Sie ML, wenn folgende Voraussetzungen zutreffen:

- Es kann keine Regel programmiert werden:

Das betrifft Aufgaben, die nicht durch festgelegte, regelbasierte Programmierung gelöst werden können. Dies ist meist der Fall, wenn eine zu große Anzahl an Faktoren das Ergebnis beeinflusst.

- Es kann nicht skaliert werden:

Manuell oder mit Hilfe eines Programms kann nur eine begrenzte Menge an Daten bewertet und kategorisiert werden. Millionen von Daten können lassen sich auf diese Weise nicht mehr sinnvoll verarbeiten. ML ist für die Verarbeitung von sehr großen Datenmengen sehr effizient.

■ 2.3 Die Schritte in einem Machine-Learning-Projekt

Das Machine-Learning-Projekt sollte nach Möglichkeit nach einem standardisierten Prozess ablaufen, durch den eine kontinuierliche Verbesserung des ML-Modells angestrebt wird. In der Regel besteht ein ML-Projekt aus folgenden Schritten:

- Daten importieren
- Daten aufbereiten
- Trainings-Algorithmus auswählen
- Hyperparameter und Optimierungsmetriken festlegen (siehe Kasten weiter unten)
- Modell trainieren
- Modell bewerten

Datenimport und Aufbereitung

Im initialen Prozess des Datenimports und der Datenaufbereitung liegt der Fokus auf einem tiefgehenden Verständnis für die Problemstellung und die Analyse der bereitgestellten Daten. Es ist bei Machine Learning essenziell, dass man sich über die Problemstellung ausreichend Gedanken macht und daraus entsprechende Ziele ableitet. Neben der Problemstellung ist es auch sehr wichtig, die zu verarbeitenden Daten zu verstehen.



Hyperparameter

Unter Hyperparametern versteht man Parameter, die den Trainingsprozess eines ML-Modells steuern und deren Werte festgelegt werden, bevor das Training beginnt. Hyperparameter stehen damit im Kontrast zu Parametern, deren Werte während des laufenden Trainingsprozesses gesetzt und optimiert werden. Welche Hyperparameter zur Anwendung kommen, hängt vom eingesetzten Trainingsalgorithmus ab.



Optimierungsmetriken

Eine Optimierungsmetrik ist eine mathematische Funktion, mit welcher der Erfolg eines ML-Modells während der Trainingsphase gemessen wird. Je nach Problemtyp werden unterschiedliche Metriken eingesetzt.

Im ersten Schritt werden die Datensätze importiert. Beim Machine Learning benötigt man für den Lernprozess normalerweise drei Datenarten: Trainingsdaten, Validierungsdaten und Testdaten.

Trainingsdaten

Die Trainingsdaten sind die Wissensbasis, mit der ein ML-Modell trainiert wird. Vereinfacht ausgedrückt, handelt es sich um Beispiele, anhand derer ein Algorithmus lernt, eine bestimmte Datenkategorie zu erkennen.

In der Praxis besteht ein Großteil der Arbeit in ML-Projekten darin, die richtigen Daten zu beschaffen und aufzubereiten. Dieser Vorgang wird Preprocessing genannt und stellt sicher, dass die Daten genau die gewünschte Problemstellung repräsentieren, auf die das ML-Modell trainiert werden soll.

In der Forschung und Entwicklung stehen entsprechende Trainingsdaten als Dataset zur Verfügung. Hierbei handelt es sich meist um handgeschriebene Ziffern, Blumen oder Tiere, die zum Trainieren verschiedener ML-Modelle, wie zum Beispiel für die Mustererkennung verwendet werden können.

Validierungsdaten

Hierbei handelt es sich um Daten, die zum Feintuning von Parametern des ML-Modells verwendet werden. Das heißt, mit diesen Daten kann der Entwickler, das ML-Modell während des Trainings mit den Trainingsdaten validieren, um zu erkennen, ob ein Lernprozess stattfindet oder ob der Algorithmus im Modell die Daten nur auswendig lernt (siehe Kasten zu Overfitting).

Sehr häufig wird hier das Verfahren der Kreuzvalidierung eingesetzt. Die Kreuzvalidierung ist eine Trainings- und Validierungstechnik, die die Daten in mehreren Partitionen trainiert. Dieses Verfahren verbessert die Stabilität des Modells.

Testdaten

Die Testdaten werden verwendet, um die Leistung eines ML-Modells gegen unbekannte Daten zu prüfen bzw. zu bewerten. Um das ML-Modell richtig trainieren zu können, ist es sinnvoll vorher die vorhandenen Daten in Trainings- und Testdaten zu unterteilen. In der ML-Terminologie bezeichnet man diesen Schritt als Splitting.

Die erstellten Testdaten werden weder im Training noch zur manuellen Optimierung des Systems verwendet und dienen nur dazu, das fertige ML-Modell gegen unbekannte Daten zu testen, mit denen es nicht trainiert wurde. Es ist erforderlich die zur Verfügung stehenden Daten für das ML-Modell in mindestens zwei oder mehr Teile zu splitten.

Des Weiteren benötigen alle ML-Modelle eine sogenannte Fehlerfunktion, um trainiert werden zu können. Die Fehlerfunktion bestimmt die Differenz zwischen der Prognose, die das Modell liefert, und dem vorgegebenen Label. Sie muss für die Menge aller Daten berechnet werden und beschreibt, wie gut das ML-Modell die Daten abbildet. In der Regel versucht man den Verlust zu minimieren und einen Kompromiss zwischen dem minimalen Verlustwert und zu hoher Komplexität zu finden, um eine sogenannte Überanpassung (Overfitting) des ML-Modells zu vermeiden.



Overfitting

Von Overfitting spricht man, wenn ein ML-Modell nach dem Training nicht mehr in der Lage ist, gültige Ergebnisse zu erzielen. Man spricht dann von einer Überanpassung des ML-Modells. Bei neuronalen Netzen bedeutet das, dass ein Netz für die Inputs aus den Trainingsdaten sehr genau ist, für den Testsatz allerdings nicht. Overfitting tritt dann auf, wenn die Trainingsdatenmenge relativ gering und die ML-Modelle komplex sind. Dann erzeugt das ML-Modell keine allgemeine Funktion mehr, sondern verhält sich so, als hätte es die Daten auswendig gelernt. Im schlimmsten Fall werden dann nur noch korrekte Ergebnisse auf dem Trainings- satz erreicht.



Label

Label ist die Bezeichnung für eine Gruppe von Elementen. Bevor Sie ein Modell für überwachtes Lernen trainieren, müssen die Rohdaten mit den richtigen Annotationen versehen werden.

Bei einer Bildanalyse kann es vorkommen, dass man einem Bild mehrere Labels zuordnen möchte. So zum Beispiel ist in einem Bild das Objekt Schrank und das Objekt Stuhl als jeweils ein Label zu kennzeichnen. Für sehr große Datensets, kann es sinnvoll sein, ein entsprechendes Data-Labeling-Tool einzusetzen. Diese Tools nutzen bereits vortrainierte neuronale Netze mit dem Vorteil, dass wesentlich weniger Daten benötigt werden, um Merkmale zu erkennen.

■ 2.4 Machine-Learning-Verfahren

Nachdem man also für die Datenaufbereitung gesorgt hat, geht es um die Auswahl des passenden Algorithmus und der entsprechenden Lernform. Die ML-Modelltypen und Algorithmen unterscheiden sich in unterschiedlichen Aspekten voneinander. In der ML-Terminologie spricht man daher auch von den sogenannten Machine-Learning-Verfahren.

Welchen Algorithmus Sie für Ihr ML-Projekt wählen, hängt in erster Linie von den folgenden Fragestellungen ab:

- Was möchten Sie mit Ihren Daten machen? Wie genau lautet die Problemstellung, die Sie beantworten möchten?
- Welche Anforderungen sind zu erfüllen? Welche Genauigkeit, Trainingszeit und Linearität werden benötigt?
- Wie hoch ist die Anzahl der Parameter, die Ihre Lösung unterstützen soll?

Die wichtigsten Verfahren und deren Anwendungsfälle werden nachfolgend kurz vorgestellt.

2.4.1 Klassifikation

Das Klassifikationsverfahren teilt Objekte nach ihren Merkmalen mithilfe eines Klassifizierungsmodell (Klassifikator) in vordefinierte Kategorien ein. Der Klassifikator (das ML-Modell) ist eine mathematische Funktion, die einen Input auf eine Klasse abbildet.

Das heißt, der unbekannte Input soll einer Klasse, welche die Kategorie darstellt, zugeordnet werden. Somit besteht das Ziel darin, passende Regeln zu finden, nach denen sich die Daten den jeweiligen Klassen zuordnen lassen. Bei einer Klassifikation steht schon im Vorfeld fest, in welche Gruppen ein Objekt eingeordnet werden kann. Hier geht es hauptsächlich darum, die Merkmale zu identifizieren, die für die Zuordnung am erkennbarsten sind. Ein Beispiel für Klassifikation ist die Erkennung von Spam-Mails, bei der das ML-Modell entscheidet, ob eine Mail als Spam markiert wird oder nicht. Auch das Erkennen von Fotomotiven, ob zum Beispiel ein Hund, eine Katze oder eine Maus abgebildet ist, wird über das Klassifikationsverfahren gelöst. Liegen nur zwei unterschiedliche Ergebniskategorien vor, spricht man von einer Binär-, ansonsten von Mehrklassen-Klassifizierung.

2.4.2 Regression

Regression hat das Ziel, einen Wert vorherzusagen, sofern eine Kontinuität bei der Ausgabe vorliegt. Das Ziel besteht also darin, eine Funktion zu beschreiben, die den Zusammenhang zwischen abhängigen und unabhängigen Variablen eines Datensatzes darstellt. So ist zum Beispiel bei einer linearen Regression das ML-Modell eine lineare Funktion. So wird als Antwort auf neue Daten der Funktionswert des ML-Modells zurückgegeben.

Bei der logistischen Regression stellt das ML-Modell eine Trennungslinie, zwischen zwei Klassen dar. Hierbei liegen die Ergebniswerte in einer kontinuierlichen Verteilung vor, so dass eine Vorhersage numerischer Werte möglich ist. Beispiele für die Regression sind die Vorhersage von Immobilienpreisen, Prognosen über Aktienkurse, Temperaturen, Umsatz-Forecasts für Unternehmen und Banken oder auch die Lebensdauer von Bauteilen.

2.4.3 Clustering

Clustering wird für die Datenanalyse verwendet, um verborgene Muster oder Gruppierungen in Daten zu finden. Hierbei wird automatisch eine Kategorisierung der Daten vorgenommen und diese werden in zusammenhängende Gruppen, die sogenannten Cluster, sortiert, ohne diese Gruppen vorher zu definieren. Beim Clustering soll die Datenmenge so in Gruppen (Cluster) aufgeteilt werden, dass die Daten in einem Cluster möglichst ähnlich und die Daten aus verschiedenen Clustern möglichst unähnlich sind. Die Ähnlichkeit wird durch eine Distanzfunktion auf der Datenmenge ausgedrückt.

Die Anwendung von Clustering umfasst zum Beispiel die Gensequenzanalyse, wiederkehrende Nutzungsmuster und die Objekterkennung.

2.4.4 Bayes-Klassifikation

Der Bayes-Klassifikator ordnet Objekte einer Klasse zu, zu der sie mit größter Wahrscheinlichkeit gehören. Grundlage für die Berechnung der Wahrscheinlichkeit ist dabei eine Kostenfunktion. Diese stellt die Objekte als Vektor dar, bei dem jede Eigenschaft des Objekts auf eine Dimension abgebildet wird.

Ein Bayessches Netz besteht aus Knoten für Zufallsvariablen und Pfeilen, die Abhängigkeiten zwischen den Variablen darstellen. Zu jedem Knoten gibt es Tabellen mit bedingten Wahrscheinlichkeiten. Diese Tabellen können wiederum von Parametern abhängen, die von Benutzern aus den vorliegenden Daten spezifiziert werden. Das heißt, der Anwender erstellt die Parameter in der Tabelle aus den vorhandenen Daten und nimmt damit Einfluss auf die Wahrscheinlichkeiten.

Bestimmte ML-Algorithmen wie Naive Bayes, Gaussian Naive Bayes oder Bayesian Belief Network (BBN) verwenden für die Klassifikation und Regression den Satz von Bayes. Dieser besteht aus entsprechenden Entscheidungsknoten, in denen Tests für bestimmte Attribute durchgeführt werden [6].

2.4.5 Künstliche neuronale Netze

Natürlich gehört auch das Hauptthema dieses Buches, die künstlichen neuronalen Netze (KNN) zu den Machine-Learning-Verfahren. Mit neuronalen Netzen ist es möglich, Objekte zu klassifizieren. Die Verbindungen zwischen den Neuronen sind, wie schon in Kapitel 1 erläutert, als Gewichte dargestellt und werden beim Training des neuronalen Netzes so lange verändert, bis die Ausgabe eine entsprechende Qualität erreicht hat. Das heißt, die numerischen Eingaben werden als Signale über mehrere Schichten in Ausgaben umgewandelt. Neue Eingaben werden durch das Netz propagiert, indem sie mit den Gewichten multipliziert werden und vor jedem Knoten aufsummiert an die jeweilige Aktivierungsfunktion übergeben werden. Sind genügend Trainingsdaten vorhanden, erreichen neuronale Netze in der Regel sehr gute Ergebnisse im Vergleich zu anderen Verfahren.

Bild 2.3 zeigt die zentralen ML-Verfahren für die Verwendung von Klassifikation, Regression und Clustering.

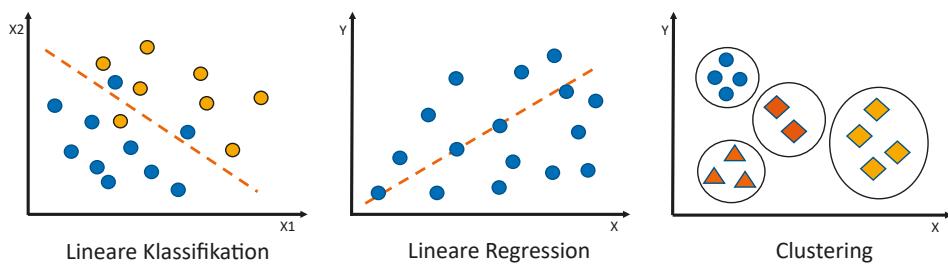


Bild 2.3 Die wichtigsten Machine-Learning-Verfahren

■ 2.5 Lernformen

Die Auswahl der Lernform entscheidet, mit welchem Ansatz ein ML-Modell trainiert bzw. angelernt werden soll. Die nachfolgende Auflistung enthält eine kurze Beschreibung jeder Lernform und zeigt, welche Aufgaben mit dieser Form des Machine Learning vorzugsweise bearbeitet werden können.

2.5.1 Überwachtes Lernen

Beim überwachten Lernen (Supervised Learning) lernt der Algorithmus die Muster und Zusammenhänge anhand eines Trainingsdatensatzes der gelabelt (siehe Label-Definition im Kasten) ist. Das heißt, die Daten müssen zuvor von einem Anwender als korrekt bewertet werden.

Beim überwachten Lernen basiert das ML-System auf einer Anzahl von vordefinierten Trainingsdaten und es erlernt immer den Zusammenhang von relevanten Kategorien (Klassen), die als Label gekennzeichnet wurden, zu einer Zielvariable und versucht diese richtig vorherzusagen. Die Zielvariable kann zum Beispiel ein einfaches Ja oder Nein sein, ein Gegenstand oder ein Sachverhalt in einem Bild oder ein numerischer Wert, wie zum Beispiel der Umsatz für den nächsten Monat. Das heißt, der Algorithmus lernt, indem er die Zuordnung bekannter gelabelter Daten auf bisher unbekannte Daten überträgt. Anhand der gelabelten Trainingsdaten kann das ML-System einen entsprechenden Vorhersage-Algorithmus erstellen. Das Ergebnis ist ein Machine-Learning-Modell. Wie gut oder schlecht das ML-Modell dann funktioniert, hängt besonders von Menge und Qualität der Trainingsdaten ab.

Überwachte Lernansätze werden oft verwendet, um zum Beispiel bestimmte Kategorien oder Klassen einzuführen. Hier kommen die in Abschnitt 2.4.1 und Abschnitt 2.4.2 beschriebenen Verfahren für Klassifizierung und Regression zum Zuge.

Die ML-Verfahren im überwachten Lernen lassen sich aufgrund ihrer Trainingsdatenstruktur gut nachvollziehen. Es besteht die Möglichkeit verschiedene ML-Algorithmen einander gegenüberzustellen, zu parametrisieren und dadurch eine für den Anwendungsfall optimierte Lösung zu finden.

So kann man z. B. im Bereich Industrie 4.0 das überwachte Lernen für eine Störungsanalyse einsetzen. Hierfür würde man die Sensordaten einer Produktionsmaschine zusammen mit Informationen über vergangene Störungen in das ML-System einspielen. Das ML-Modell lernt dann den Zusammenhang zwischen Sensordaten und Störung und kann daraus Vorhersagen über zukünftige Störungen ableiten.

2.5.2 Unüberwachtes Lernen

Beim unüberwachten Lernen (Unsupervised Learning) ist die Ziel- bzw. Ausgabevariable nicht bekannt. Demzufolge verfügt der Algorithmus für das ML-Modell beim unüberwachten Lernen über keine gelabelten Trainingsdaten.

Das ML-System lernt infolgedessen ohne Anleitung, da die Trainingsdaten nicht gekennzeichnet sind oder eine unbekannte Struktur besitzen. Dieses ML-Verfahren kommt sehr häufig bei unstrukturierten Datenmengen zum Einsatz. Das ML-System muss ohne definierte Ausgabewerte die Trainingsdaten strukturieren und vorhandene Muster erkennen, um diese dann in Cluster (Gruppen) einzufügen.

Das ML-System versucht herauszufinden, welche Datenpunkte ähnlich und welche Ausreißer sind. Daher spricht man beim unüberwachten Lernen auch vom Clustering-Verfahren, da man die Datensätze in mehrere, möglichst ähnliche Gruppen (Cluster) aufteilen möchte. Hauptanwendungsgebiete sind für das hieraus resultierende ML-Modell die Sequenzanalyse, die Objekterkennung oder auch das Marketing. Im Marketing ist es für Werbekampagnen, Kundenbewertungen und Produktempfehlungen sehr nützlich, über ein ML-System zu verfügen, das Trends und Aktionen schnell und in Echtzeit vorhersagen und dann ohne menschliches Zutun entsprechend reagieren kann.

Die Vorteile des unüberwachten Lernens bestehen in der teilweise vollautomatisierten Erstellung von ML-Modellen. Das ML-Modell lernt mit jedem neuen Datensatz dazu und verfeinert wiederum seine Berechnungen im ML-Modell.

Auch für das unüberwachte Lernen kann man die Sensordaten einer Produktionsmaschine nutzen. Hier würde das ML-Modell mit typischen Sensordaten der Maschine trainiert werden. Bei Abweichungen von diesen Daten würde es auf Fehlerzustände schließen. So könnte man z. B. die entsprechenden Daten-Cluster in die Gruppen „Produktion“, „Störung“ und „Stillstand“ einteilen.

2.5.3 Semi-überwachtes Lernen

Einen Kompromiss zwischen überwachtem (supervised) und unüberwachtem (unsupervised) Lernen bildet das semi-überwachte Lernen (Semi-Supervised Learning). Es handelt sich um eine spezifische Klasse des überwachten Lernens, die sowohl markierte (gelabelte) als auch unmarkierte Eingangsdaten in den Trainingsdaten enthält.

Das ML-System muss aufgrund der vorhandenen Labels entscheiden welche Merkmale für die Gruppierung der Eingabedaten benutzt werden. Beim Semi-Supervised Learning wird sehr häufig eine Kombination von Clustering und Klassifikation verwendet, indem man erst einen Clustering-Algorithmus anwendet und anschließend die wenigen vorhandenen Trainingsdaten mit Label nutzt, um dem Cluster eine Klasse zuzuweisen. Ein häufiger Anwendungsfall ist die Identifikation von Gesichtern in Webcam- und Videoaufnahmen.

2.5.4 Verstärkendes Lernen

Eine weitere Lernmethode ist das verstärkende Lernen (Reinforcement Learning). Hier besteht die Zielsetzung darin, ein ML-System, einen sogenannten Agenten, zu entwickeln, der seine Leistung durch Interaktion mit seiner Umgebung verbessert.

Das Reinforcement Learning generiert Strategien und Lösungen auf Basis von erhaltenen Belohnungen im Trial-and-Error-Verfahren. Auch können für das Reinforcement Learning unterschiedliche ML-Algorithmen zum Einsatz kommen. Grundsätzlich arbeiten die verwen-

deten ML-Algorithmen aber so, dass die Aktionen eines Agenten (ML-System) die Systemumgebung verändern. Abhängig von erhaltenem Feedback führt der Agent anschließend die nächste Aktion aus. Ziel der Algorithmen ist es, die erhaltenen Belohnungen innerhalb des simulierten Systems zu maximieren.

Typische Anwendungsbereiche für das Reinforcement Learning sind Problemstellungen, die folgende Eigenschaften aufweisen:

- Das Trial-and-Error-Prinzip ist anwendbar.
- Klassische Programmierung führt nicht zum Ziel.
- Die Aufgabe kann simuliert werden.
- Das System soll eigene Strategien zur Lösungsfindung entwickeln.
- Komplexe Lösungsschritte sollen gefunden und optimiert werden.

Praktische Anwendungsfälle von Reinforcement Learning sind das Erlernen des Brettspiels Go durch AlphaGo bzw. AlphaGo Zero, wie auch eine Ampelsteuerung zur Minimierung von Stausituationen oder die Optimierung von Logistikprozessen.

Neben den oben aufgeführten Lernformen gibt es auch noch das sogenannte aktive Lernen, das auch als interaktives semi-überwachtes Lernen bezeichnet werden kann. Hierbei werden für die Klassifizierung großer Datenmengen einfach entsprechende Benutzer bzw. Anwender eingesetzt, die die Reaktion des Agenten prüfen bzw. beurteilen.

Inzwischen gibt es eine Vielzahl von ML-Diensten, -Tools und -Frameworks, die genau für die unterschiedlichen Lernformen und ML-Verfahren entwickelt wurden. Allerdings muss man heute bei der immer schneller werdenden Entwicklung feststellen, dass die Grenzen zwischen überwachtem, semi-überwachtem und unüberwachtem Lernen mehr und mehr verschwimmen.

■ 2.6 Machine-Learning-Algorithmen

Die Auswahl des richtigen Algorithmus ist unter Umständen nicht ganz so einfach, denn es gibt eine ganze Reihe von überwachten und unüberwachten ML-Algorithmen und somit unterschiedliche Lernansätze.

Es gibt keine beste Methode und auch keine, die sich immer eignet. Die Auswahl des richtigen Algorithmus besteht zum Teil einfach auf Versuch und Irrtum (trial and error). Mit einem guten logischen Verständnis, sowohl für die Daten als auch für ML-Algorithmen kann man schon ein sehr gutes ML-Modell erstellen. Daraus ergibt sich, dass die Auswahl eines Algorithmus auf jeden Fall auch vom Umfang und Art der Daten abhängig ist, mit denen gearbeitet werden soll.

Als Erstes entscheidet man sich für die Lernmethode:

- Wählen Sie überwachtes (supervised) Lernen aus, wenn Sie ein ML-Modell trainieren möchten, damit es eine Prognose abgibt. Wie zum Beispiel die Sensorwerte von Betriebszuständen oder die Identifizierung von Objekten in Bilddateien.

- Wählen Sie nicht überwachtes (unsupervised) Lernen aus, wenn Sie Ihre Trainingsdaten nach Strukturen unter- bzw. durchsuchen möchten, um ein ML-Modell zu trainieren. Das nicht überwachte Lernen wird also verwendet, um Rückschlüsse aus den Datenmengen zu ziehen, die aus Eingabedaten ohne klassifizierte Ausgangsdaten bestehen.

Ein strukturiertes Vorgehen kann bei der Suche nach dem richtigen Algorithmus sehr hilfreich sein. Es ist vielleicht auch sinnvoll, einen sogenannten Machine-Learning-Katalog aufzubauen, der die wichtigsten Begriffe aufführt und definiert, wie auch die wesentlichen Algorithmen und Techniken auflistet und sie übersichtlich in Funktionsblöcke, Input- und Output-Dateitypen sowie Lernformen einteilt. Auf diese Weise können Sie als Entwickler schnell und strukturiert auf entsprechende Lösungsansätze zurückgreifen.

Sie müssen als Entwickler also beachten, dass jeder Algorithmus für das maschinelle Lernen seine eigene Ausprägung oder auch seinen eigenen induktiven Bias (siehe Abschnitt 1.5, „Grundbaustein: Neuron“) besitzt. Für ein bestimmtes Problem können auch verschiedene Algorithmen geeignet sein, aber ein Algorithmus passt möglicherweise besser als andere. Das heißt, Sie müssen, einen Algorithmus testen und bei einem nicht zufriedenstellenden Ergebnis einen anderen versuchen.

Alle später in diesem Buch beschriebenen ML-Frameworks wie TensorFlow, Microsoft Cognitive Services oder auch Amazon Web Services verfügen über einen sogenannten Spickzettel für ML-Algorithmen. Dieser Spickzettel hilft Ihnen bei den ersten Überlegungen zum ML-Modell: Welche Daten stehen zur Verfügung und was und wie sollen diese ausgewertet werden? Über den Spickzettel wählen Sie dann den Algorithmus nach Art der Aufgabe aus, die Sie erledigen möchten.

Soll mit den Trainingsdaten eine Prognose erstellt werden und das überwachte Lernen im Vordergrund stehen, so werden Klassifikations- oder Regression-Techniken angewandt, und entsprechende ML-Modelle darauf entwickelt. Beachten Sie, dass manche ML-Algorithmen als Eingabedaten nur numerische Werte entgegennehmen. So kann es erforderlich sein, die Kennzeichendaten in Zahlen umzuwandeln, beispielsweise in Werte wie 0 oder 1, um eine Kennzeichnung zu repräsentieren.

Häufige Algorithmen für die Durchführung der Klassifikation sind k -Nearest-Neighbour, Support Vector Machine (SVM), Naive Bayes, Decision Tree, Random-Forest bzw. Gradient Boosting Trees und neuronale Netze. Für Regressionsalgorithmen werden lineare Modelle, multivariate lineare Modelle, Logistic, Cox und neuronale Netze eingesetzt.

Beim Einsatz von unüberwachtem (unsupervised) Lernen, ist die am häufigsten verwendete Technik das Clustering. Die Algorithmen für das Clustering sind K-Means und K-Medians, Expectation maximization, hierarchische Cluster, selbstorganisierende Karten (Self Organizing Maps) und neuronale Netze. Nachfolgend werden die meist genutzten ML-Algorithmen in einem kurzen Überblick vorgestellt.

2.6.1 k -Nearest-Neighbour

Der k -Nearest-Neighbour-Algorithmus ist ein Verfahren, mit dem neue Daten auf Basis von vorhandenen Daten klassifiziert werden können.

k-Nearest-Neighbour ist einer der einfachsten Algorithmen zur Klassifizierung. Es werden zu einem neuen Datenpunkt die *k* nächsten Nachbarn bestimmt, wobei *k* eine beliebige Zahl darstellt. Der Algorithmus versucht, naheliegende Datenpunkte heranzuziehen, um zu entscheiden, welche Klasse einen Datenpunkt bekommt. Die Klasse, die unter *k*-Nachbarn am häufigsten vorkommt, wird die Klasse, zu der der neue Datenpunkt gehört. *k* stellt bei diesem Algorithmus den Tuningparameter dar. Für das Bestimmen von *k* gibt es aber keine allgemeingültige Regel. Beachten Sie bei diesem Algorithmus, dass für zu große Werte von *k* das ML-Modell seine sogenannte Lokalität verliert. Das heißt, es werden irrelevante Datenpunkte mit einbezogen, die sehr weit entfernt liegen.

Der optimale Wert für *k* kann wie folgt bestimmt werden, dazu benötigt man einen Trainingsdatensatz sowie einen Testdatensatz. Für verschiedene Werte von *k* werden dann die Testdaten anhand der Trainingsdaten klassifiziert.

Der *k*-Nearest-Neighbour funktioniert bei einem Datensatz mit zwei Features sehr gut. Sobald aber mehrere Features (in Spalten gespeicherte Variablen oder Felder) vorhanden sind, kann der Algorithmus nicht mehr korrekt arbeiten. Die Daten liegen dann nicht mehr auf einer Ebene der Dimension 2. Um mit mehrdimensionalen (größer als zwei Dimensionen) Daten zurechtzukommen, empfehlen sich Entscheidungsbaum-Algorithmen wie Decision Tree oder Random-Forest. Bild 2.4 zeigt den Einsatz von *k* zum nächsten Nachbarn.

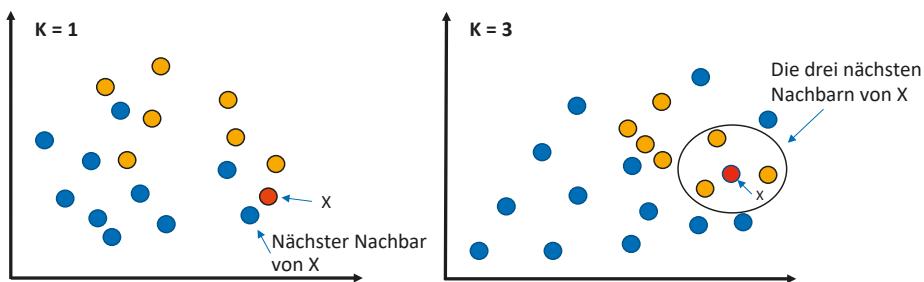


Bild 2.4 *k*-Nearest Neighbour mit Tuningparameter *k*

2.6.2 Support Vector Machine

Die Support Vector Machine, kurz SVM genannt, ist einer der bekanntesten ML-Algorithmen und wurde schon 1963 von Wladimir Wapnik und Alexei Tscherwonenski eingeführt. Bei diesem Algorithmus handelt es sich um eine mathematische Methode und einen statistischen Ansatz zur Klassifizierung von Objekten.

Die SVM unterstützt sowohl die lineare/nicht lineare Klassifizierung wie auch die lineare/logistische Regression. Bei der Klassifizierung kann die Support Vector Machine die Objektklassen mithilfe von Trennungsebenen einteilen. Diese Ebenen werden so gewählt, dass zwischen verschiedenen Klassen ein möglichst großer Bereich frei von Objekten bleibt. Die Trennungsfläche mit dem größten objektfreien Bereich gilt als optimale Lösung. Die von der SVM ermittelten Klassengrenzen sind sogenannte Large Margin Classifier und lassen um die Klassengrenzen herum einen breiten Bereich frei von Objekten (Bild 2.5).

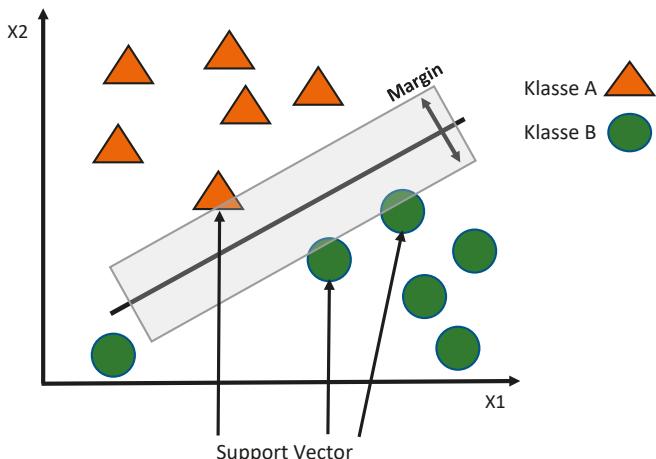


Bild 2.5 Arbeitsweise der Support Vector Machine

In der Praxis hat sich allerdings gezeigt, dass die Datenobjekte in den meisten Fällen nicht durch rein lineare Grenzen zu klassifizieren sind. Das heißt, bei nicht linearen Klassengrenzen verwendet die SVM den sogenannten Kernel-Trick (siehe Kasten). Hierbei löst die SVM das Problem durch die Ermittlung einer Entscheidungsgrenze (Grenze, die die verschiedenen Klassen voneinander trennt), so entsteht eine Hyperfläche (siehe Kasten) die den Vektorraum auf zwei Sätze aufteilt die mit positiver Kennung und die mit negativer Kennung.

Die Support Vector Machine besitzt folgende Eigenschaften:

- Sie ist ein sehr guter und effektiver Algorithmus zur Klassifizierung von Objekten.
- Durch den Einsatz von Hyperebenen ist auch eine sehr komplexe nicht lineare Trennfläche abbildbar.

Bei zweidimensionalen linearen Merkmalsvektoren kann die Lösung des SVM-Algorithmus wie in Bild 2.5 dargestellt werden. Sollte die Support Vector Machine für die Regression benutzt werden, wird nicht die optimale Trennfläche gesucht, sondern die Ebene, die die Ausprägungen der Zielvariablen am besten beschreibt.

Der SVM-Algorithmus liefert sehr gute Ergebnisse bei einfachen Klassifikationsproblemen, bei sehr großen Datenmengen lässt er sich aber schlecht skalieren und liefert auch keine überzeugenden Ergebnisse mehr.



Kernel-Trick

Die Support Vector Machine überträgt die Objekte einer mehrdimensionalen Funktion erst in einen zwei- oder dreidimensionalen Datenraum. So können auf jeden Fall die Objekte nach einer nicht linearen Vorschrift in einem höherdimensionalen Datenraum positioniert werden.

Nach der Transformation wird die Trennfläche wieder als linear dargestellt. So wird mit einer einfachen Transformation von Zahlen implizit eine hochkomplexe Vektortransformation durchgeführt. Dementsprechend lassen sich die zwei- oder dreidimensionalen neuen Objekte, welche repräsentativ für eine mehrdimensionale Funktion stehen, einfach in zwei Klassen aufteilen.



Hyperfläche

Wenn man bei SVM von der Entscheidungsgrenze spricht, beschreibt man damit eine sogenannte Hyperfläche bzw. Hyperebene. Hierbei handelt es sich um einen Unterraum, dessen Dimension um 1 kleiner ist als seine Umgebung. So stellt bei einem 3D-Vektor-Datenraum, die Hyperfläche eine zweidimensionale Ebene dar. Daraus folgt, dass in einem zweidimensionalen Datenraum die Hyperfläche einfach eine gerade Linie repräsentiert (1D-Raum). Dieses Verfahren macht somit aus einem 4D-Vektor-Raum eine Hyperfläche, die ein 3D-Objekt darstellt und immer so fort.

2.6.3 Entscheidungsbäume

Ein Entscheidungsbaum dient der Aufteilung von Objekten unter Zuhilfenahme geeigneter Merkmale in Gruppen hinsichtlich einer vorgegebenen Zielgröße. Das ML-Modell ist somit immer ein Entscheidungsbaum, der in seiner Baumstruktur die Entscheidungskriterien und die möglichen Ergebnisse darstellt. Bild 2.6 zeigt die Baumstruktur in einem ML-Modell.

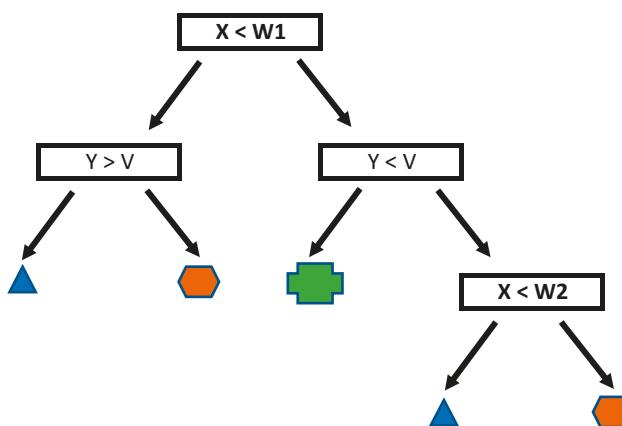


Bild 2.6
ML-Modell-Darstellung
eines Entscheidungsbaums

Darüber hinaus besteht der Entscheidungsbaum aus mindestens zwei Blättern, die das Ergebnis einer Entscheidung repräsentieren. Um einen Fall zu klassifizieren, müssen Sie den Baum bis zum Blatt hinuntergehen und den entsprechenden Wert angeben. Ziel ist es, ein ML-Modell zu erstellen, das den Wert der Zielvariablen auf der Grundlage mehrerer EingabevARIABLEN vorhersagt.

Bei Entscheidungsbäumen im Machine Learning repräsentieren die Knoten keine Entscheidungen, sondern Daten. Jede Verzweigung enthält eine Reihe von Attributen beziehungsweise Klassifizierungsregeln wie z. B. $X < W_1$, die mit einer bestimmten Klassenbezeichnung (Kategorie) am Ende der Verzweigung zusammenhängt. Man geht daher vom Wurzelknoten aus abwärts, während jeder Knoten dabei ein Attribut abfragt. Ist ein passender Endknoten (Blatt) im Baum erreicht worden, stellt dieser den Klassifizierungsfall dar. Handelt es sich bei dem ML-Modell um einen Entscheidungsbaum, der jedem Endpunkt einen numerischen Wert zuweist, so spricht man von Klassifikations- bzw. Regressionsbäumen.

2.6.4 Decision Tree und Random-Forest

Der Decision-Tree (Entscheidungsbaum)-Algorithmus ist einer der am häufigsten verwendeten Algorithmen im Machine Learning.

Im Allgemeinen erstellt der Decision-Tree-Algorithmus eine Reihe von Teilmengen in der Entscheidungsstruktur, die zusammen ein ML-Modell bilden. Diese Teilmengen werden als Knoten dargestellt. Der Algorithmus fügt dem Modell jedes Mal einen Knoten hinzu, wenn eine Eingabespalte in erheblichem Ausmaß von der vorhersagbaren Spalte abweicht.

Durch die Veränderung durch neue Daten, wird der Baum sehr schnell instabil und das Ergebnis ist ein aus den neuen Daten resultierender neuer Baum, der ganz anders aussehen kann. Abhilfe schafft hier der Random-Forest-Algorithmus. Dieser Algorithmus besteht aus einer Vielzahl von einzelnen Bäumen, die jeweils auf eine Zufallsauswahl der Trainingsdaten zugreifen und dort jeweils nur mit einer Zufallsauswahl der Daten arbeiten. Man erhält dann am Ende ein ML-Modell mit einer Menge an einzelnen Bäumen, die jeweils eine Vorhersage für neue Daten machen können.

Somit ist Random-Forest ein einfaches, aber gleichzeitig sehr mächtige ML-Modell. Dessen Weiterentwicklung ist der Gradient-Boosting Tree-Algorithmus. Dieser erlaubt eine iterative Verbesserung des Lernmodells, weil die einzelnen Modelle nicht wie bisher beim Random-Forest separat betrachtet und angepasst, sondern zu einem additiven Gesamtmodell zusammengefasst und so die Schwachstellen der Vorgängermodelle aufgegriffen und ausgebessert werden.

2.6.5 Clustering

Zum Aufgabengebiet des unüberwachten (unsupervised) Lernens gehört das Clustering bzw. die Clusteranalyse. Der Hauptunterschied zwischen Clustering und Klassifikation besteht darin, dass bei der Klassifikation vordefinierte Klassen verwendet werden, in denen die Objekte zugeordnet werden, während das Clustering Ähnlichkeiten zwischen Objekten identifiziert, die es anhand gemeinsamer Merkmale gruppiert und von anderen Objektgruppen unterscheidet. Diese Gruppen werden als Cluster bezeichnet. Im Machine Learning werden die nachfolgenden Methoden für das Clustering eingesetzt.

2.6.5.1 K-Means Clustering

Der K-Means-Algorithmus stellt die einfachste Clustering-Methode zur Verfügung. Er spaltet die Menge der Elemente eines Vektorraumes in eine vorher bekannte Anzahl von Clustern (K) auf (Bild 2.7).

Ziel ist es, dass die verschiedenen Objekte innerhalb einer Gruppe sich nach der erfolgten Einteilung möglichst ähnlich sind.

Der K-Means-Algorithmus lässt sich für mehrdimensionale Objekte anwenden und nähert sich durch sich ständig wiederholende Neuberechnungen den jeweiligen Clusterzentren an bis keine signifikanten Veränderungen mehr stattfinden.

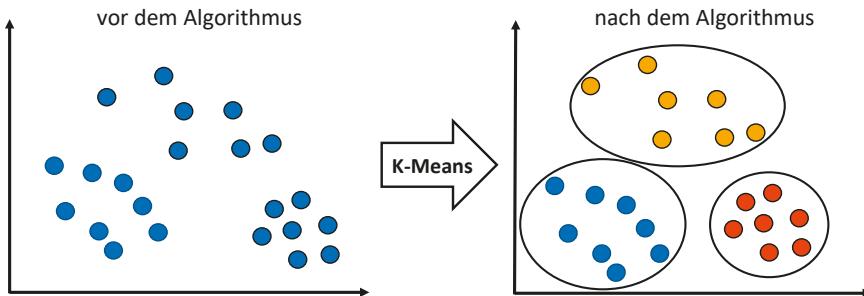


Bild 2.7 Objekte vor und nach dem Einsatz des K-Means-Algorithmus

2.6.5.2 EM-Clustering

Beim Expectation-Maximization-Clustering optimiert der Algorithmus iterativ ein Clusteranfangsmodell, um die Daten anzupassen, und bestimmt die Wahrscheinlichkeit, mit der ein Datenpunkt in einem Cluster vorhanden ist. Der Algorithmus beendet den Prozess, wenn das probabilistische ML-Modell den Daten entspricht.

Probabilistisch bedeutet, dass jeder Datenpunkt zu allen Clustern gehört, jede Zuweisung eines Datenpunktes zu einem Cluster jedoch eine andere Wahrscheinlichkeit hat.

2.6.5.3 Hierarchische Clusteranalyse

Für kleinere Datenmengen kann ein hierarchischer Cluster-Algorithmus verwendet werden, der Schritt für Schritt aus den Fällen ein Cluster formt. Diese Methode wird immer dann angewendet, wenn die Datenmenge klein ist oder es keine globalen Kategorien gibt, nach denen sich Daten clustern lassen.

Beim hierarchischen Cluster-Algorithmus müssen sowohl das Distanzmaß als auch eine Berechnungsregel für den Abstand zweier Cluster vorgegeben werden. Als Distanzmaß wird sehr häufig der euklidische Abstand verwendet. Hierbei wird der Abstand zweier Vektoren a und b ermittelt, indem man die Differenz zwischen den beiden Vektoren bildet und anschließend die Länge berechnet.

Beim hierarchischen Cluster unterscheidet man zwei grundsätzliche Verfahren:

- **Agglomerativ:** anfangs so viele Gruppen wie Fälle, sukzessives Zusammenfassen der Gruppen.
- **Divisiv:** anfangs alle Fälle in einer Gruppe, sukzessives Aufteilen der Fälle in Gruppen.

Für beide Verfahren gilt, dass einmal gebildete Cluster nicht mehr verändert werden können.

Wie Sie an den aufgezeigten Algorithmen sehen, gibt es nicht wirklich nur einen richtigen Algorithmus für eine Problemstellung, sondern verschiedene Ansätze, um den richtigen oder den am besten passenden Algorithmus zu finden, der zu den Daten und der zugrundeliegenden Problemstellung passt.

■ 2.7 Training und Validierung des ML-Modells

Um ein ML-Modell mit dem passenden Algorithmus trainieren zu können, müssen Sie sicherstellen, dass sowohl Trainings- als auch Testdaten vorliegen. Um während des Trainings- und Optimierungsprozesses prüfen zu können, ob sich die Qualität der Ergebnisse verbessert, kann es sinnvoll sein, einen weiteren Teil der Daten als Validierungsdaten zu reservieren.

Sie müssen in dieser Phase evaluieren, ob das ML-Modell mit angemessenem Aufwand ausreichend gute Ergebnisse liefert. So ist es durchaus möglich, dass ein einfacher Algorithmus schon sehr gute Ergebnisse liefern kann und der Aufwand, um die Ergebnisse zu verbessern, oft deutlich höher ist als der erreichte Nutzen.

Daher ist es sinnvoll, die Qualität des Modells anhand der Testdaten zu messen, indem der vorhergesagte Wert der Zielvariablen mit dem vorhandenen Wert der Zielvariablen verglichen wird. Nachdem man so ein funktionierendes ML-Modell oder besser gesagt einen Algorithmus mit ML realisiert hat, müssen Sie diesen noch für Ihr System so anpassen, dass er auch im Dauereinsatz verwendet werden kann.

Sie fragen sich sicher, warum man nicht sofort ein künstliches neuronales Netz (KNN) für eine Aufgabenstellung entwickelt. Da schon ein einfacher ML-Algorithmus sehr gute Ergebniswerte für ein Problem liefern kann, wäre es vollkommen übertrieben, ein neuronales Netz aufzubauen zu wollen. Das beste Beispiel liefert der Gradient-Boosting Tree. Dieser Algorithmus ist eigentlich der beste zu verwendende ML-Algorithmus, wenn es um die Auswertung von strukturierten Daten geht. Durch die vorgegebenen Implementierungen in den ML-Frameworks erreichen Sie mit diesem Algorithmus sehr viel schneller und effektiver eine passende Lösung als über den Aufbau eines KNN. Daher ist es wichtig, dass man zumindest die ML-Algorithmen konzeptionell kennt, damit man bei Bedarf auf sie zurückgreifen kann. Bild 2.8 zeigt noch einmal den strukturierten Prozess für die Datenaufbereitung im Machine Learning.

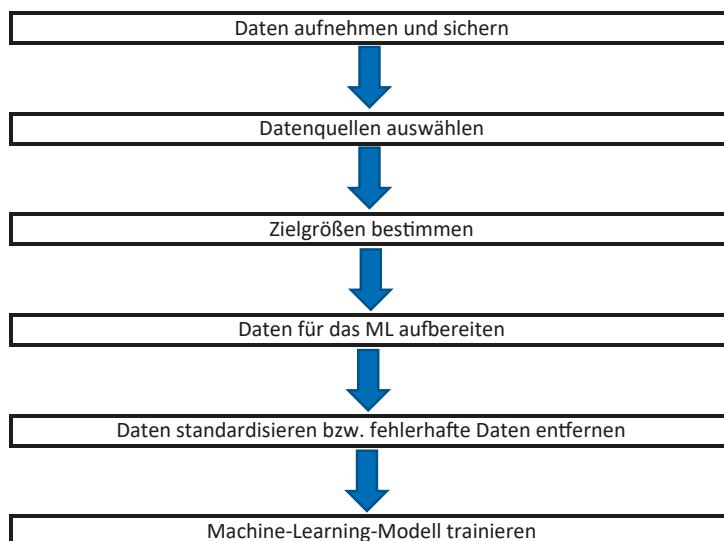


Bild 2.8 Prozess der Datenaufbereitung im Machine Learning

Damit jetzt aber zurück zum Hauptthema dieses Buches – den KNNs. Es gibt gute Gründe ein eigenes künstliches neuronales Netz zu erstellen. Zum einen hat man die vollständige Kontrolle über das System, welches dadurch auch an spezifische Probleme anpassbar ist. Zum anderen sollen Sie lernen, wie man ein neuronales Netzwerk von Grund auf neu erstellt, denn nur so entwickeln Sie ein umfassendes Verständnis für die Funktionsweise neuronaler Netze und werden so in die Lage versetzt, schon vorhandene KNNs besser und effektiver zu nutzen. Auch können Sie die gezeigten Programmiertechniken in anderen Programmen einsetzen und weiterverwenden bzw. ausbauen. Los geht es mit dem „Hallo Welt“-Beispiel für künstliche neuronale Netze – dem Perzeptron (engl. perceptron).

■ 2.8 Das einfache neuronale Netz

Alle ML-Algorithmen lassen sich auch mithilfe eines künstlichen neuronalen Netzes (KNN) abbilden. Die einfachste Form ist das Perzeptron, das einen linearen Klassifizierer bildet. Das Perzeptron ist das einfachste mathematische Modell eines künstlichen neuronalen Netzes. Erstmals wurde es schon 1958 von Frank Rosenblatt publiziert und es stellt bis heute die Grundlage von KNNs dar. Rosenblatt definierte ein Perzeptron als Schicht von Neuronen. Danach besteht das Perzeptron in der einfachsten Form aus einem einzigen Neuron mit einem Ausgang und mehreren Eingängen (mindestens zwei) (siehe Bild 2.9). Das Neuron selbst bildet hier schon den Ausgabevektor.

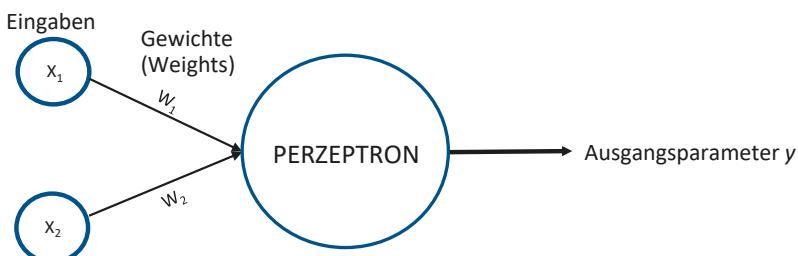


Bild 2.9 Das einstufige Perzeptron

Die Inputgewichte waren bereits anpassbar, wodurch das Perzeptron in der Lage ist, Inputs, die leicht vom ursprünglich gelernten Vektor abweichen, zu klassifizieren. Dementsprechend ist es dem Perzeptron möglich, mit zwei Eingabewerten und einem einzigen Ausgabeparameter die logischen Operanden AND(UND), OR(ODER), NOT(NICHT), NAND(NICHT-UND) und NOR(NICHT-ODER) zu erlernen, da diese linear separierbar sind.

Das Perzeptron ist eine Lernmaschine, die für eine Eingabe x eine binäre Ausgabe y liefert, und zwar durch Auswerten einer Funktion $y = f(x)$.

Das heißt, für das Erlernen von linear separierbaren Klassifikationen werden die Eingabewerte $[X_1, X_2, \dots, X_n]$ in einen binären Ausgabeparameter Y_i umgerechnet. Die Gewichte $[W_1, W_2, \dots, W_n]$ beziffern die Wichtigkeit der jeweiligen Eingabe für den Ausgangsparameter.

Der Ausgangsparameter errechnet sich somit als Summe der Gewichte über die Eingabewerte: $y_i = \sum_i W_i X_i$ (siehe auch Abschnitt 1.5, „Grundbaustein Neuron“).

AND-Funktion

Die logische AND-Funktion sagt aus, dass zwei oder mehr Ereignisse gleichzeitig auftreten müssen, damit eine Ausgabeaktion (Ausgangsparameter/Output) stattfindet. Der Ausgangsparameter der AND-Funktion ist WAHR (1), wenn alle ihre Eingaben wahr sind, ansonsten ist der Ausgangsparameter FALSCH bzw. 0.

OR-Funktion

Die logische OR-Funktion gibt an, dass eine Ausgabe WAHR (1) wird, wenn entweder ein ODER bzw. mehrere Ereignisse WAHR (1) sind, aber die Reihenfolge, in der diese auftreten, ist unwichtig, da dies das Ergebnis nicht beeinflusst. Der Ausgangsparameter der OR-Funktion ist also wahr, wenn eine oder mehrere Eingaben wahr sind andernfalls ist der Ausgangsparameter FALSCH (0).

NOT-Funktion

Die logische NOT-Funktion wird so genannt, weil ihr Ausgangszustand nicht derselbe ist wie ihr Eingangszustand. Der Ausgangsparameter der NOT-Funktion ist WAHR (1), wenn ihre einzelne Eingabe falsch ist, und falsch, wenn ihre einzelne Eingabe wahr ist.

NOR-Funktion

Die logische NOR-Funktion hat einen Ausgangsparameter, der sich normalerweise auf logisch 1 befindet und nur dann auf logisch 0 geht, wenn jeder seiner Eingänge auf logisch 1 gesetzt ist. Das logische NOR ist die umgekehrte Funktion von OR(ODER).

NAND-Funktion

Die Funktion NAND bzw. Not AND ist eine Kombination der logischen AND- und der logischen NOT-Funktion. Der Ausgabeparameter der logischen NAND-Funktion ist nur dann falsch (0), wenn alle ihre Eingaben wahr sind, andernfalls ist der Ausgabeparameter immer wahr. Logische NAND-Operatoren werden sehr häufig als Basisbausteine für den Aufbau weiterer Logik-Funktionen verwendet.

Trägt man für die logischen Operatoren die beiden Eingangswerte $X_1 (= X)$ und $X_2 (= Y)$ in ein Koordinatensystem ein, lässt sich die Lösung ganz einfach mit einer Geraden teilen. Bild 2.10 zeigt das entsprechende Diagramm für die jeweilige Funktion.

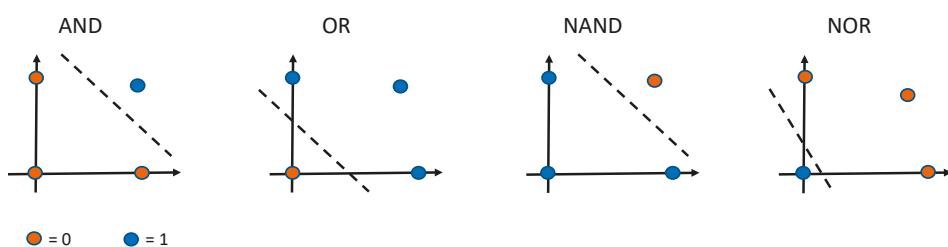


Bild 2.10 Logische Operatoren für das Perzeptron

Das NotAnd-Perceptron-Beispiel

Alle im Bild 2.10 aufgeführten logischen Operatoren lassen sich in C# als Perzepron abbilden. Zur besseren Veranschaulichung implementieren wir den *NotAnd*-Operator als Perzepron mithilfe von Visual Studio 2019. Das Programmbeispiel verzichtet hierbei auf entsprechende Kommentare im Code sowie auf den normalen Fehlerprüfcode, um das Perzepron so einfach wie möglich zu halten.

Da das Beispiel auf keine signifikanten .NET-Framework-Abhängigkeiten zugreift, können Sie es auch mit jeder anderen Version von Visual Studio erstellen. Alternativ ist natürlich auch die Verwendung von Visual-Studio-Code mit C# möglich.

Beginnen Sie also mit der Programmerstellung. Starten Sie Visual Studio und wählen Sie eine neue *Console App* (Konsolen-Applikation) über die Projektvorlage aus. Als *Project name* wurde für das Beispiel *NotAndPerceptron* gewählt.

Den benötigten Wertebereich für die Ein- und Ausgabewerte beschränken wir auf 1 (*true*) und 0 (*false*). Der Bias-Wert ist bei den oben genannten Operatoren nicht von Relevanz und geht daher auch nicht in die Berechnung des Ausgangswertes ein. Als Trainingsdaten benutzen Sie die korrekte Wahrheitstabelle für den booleschen *NotAnd*-Operator (Tabelle 2.1). Der Code zur Erzeugung des *NotAnd*-Perzepron ist exemplarisch durch die C#-Klasse in Listing 2.1 implementiert.

Tabelle 2.1 Wahrheitstabelle für das NotAnd-Perzepron

X_1	X_2	Output
0	0	1
0	1	1
1	0	1
1	1	0

Die Klasse *Perceptron* stellt drei öffentliche Methoden vor: einen Klassen Konstruktor, der Methode *Coaching* und der Methode *GetResult*. In diesem Fall ist der Konstruktor eine Methode, dessen Name derselbe ist wie der seines Typs.

Listing 2.1 Die Klasse Perceptron

```
using System;
using System.Linq;

namespace NotAndPerceptron
{
    public class Perceptron
    {
        public double LearningRate { set; get; }
        public double Threshold { set; get; }
        public double[] Weights { set; get; }

        public Perceptron(int inputCount, double learningRate = 0.2,
                          double threshold = 0.5)
```

```

{
    Weights = new double[inputCount];
    LearningRate = learningRate;
    Threshold = threshold;
}

public bool Coaching(bool expectedResult, params double[] inputs)
{
    bool result = GetResult(inputs);

    if (result != expectedResult)
    {
        double error = (expectedResult ? 1 : 0) - (result ? 1 : 0);
        for (int i = 0; i < Weights.Length; i++)
        {
            Weights[i] += LearningRate * error * inputs[i];
        }
    }

    return result;
}

public bool GetResult(params double[] inputs)
{
    if (inputs.Length != Weights.Length)
        throw new ArgumentException("Ungültige Anzahl von Eingaben.
                                         Erwartet werden: " + Weights.Length);

    return DotProduct(inputs, Weights) > Threshold;
}

private double DotProduct(double[] inputs, double[] weights)
{
    return inputs.Zip(weights, (value, weight) => value * weight).Sum();
}
}
}

```

Der Konstruktor der *Perceptron*-Klasse erwartet einen Parameter vom Typ *int*, der die Anzahl der Gewichte festlegt. Des Weiteren werden hier auch schon einzeln die Lernrate und die Schwellwertfunktion (siehe auch Abschnitt 1.5, „Grundbaustein Neuron“) festgelegt. Um die Ausgabe zu errechnen, müssen sämtliche Eingabewerte mit den entsprechenden Gewichten multipliziert und aufaddiert werden. Der Quellcode der Klasse *Perceptron* zeigt für die benötigte Berechnung die Methode *DotProduct*.



Lernrate

Die Geschwindigkeit und Genauigkeit eines Lernverfahrens kann immer von einer Lernrate gesteuert werden. Diese gibt an, wie stark das Eingangsgewicht im Neuron verändert wird, bei einer Lernrate = 0 findet keine Veränderung statt.

In der Methode *DotProduct* nutzen Sie die *Enumerable.Zip* Methode, die jedes Element der ersten Sequenz mit einem Element zusammenführt, das in der zweiten Sequenz denselben Index aufweist. Der Code in Listing 2.2 zeigt die Main-Methode für den Programmstart und, wie Trainingsdatenelemente festgelegt werden.

Listing 2.2 Die Klasse Program

```
using System;

namespace NotAndPerceptron
{
    class Program
    {
        static void Main(string[] args)
        {
            CoachingItem[] trainingSet =
            {
                new CoachingItem(true, 1, 0, 0),
                new CoachingItem(true, 1, 0, 1),
                new CoachingItem(true, 1, 1, 0),
                new CoachingItem(false, 1, 1, 1)
            };

            Perceptron perceptron = new Perceptron(3);

            int attemptCount = 0;

            while (true)
            {
                Console.WriteLine("---- Versuch: " + (++attemptCount) + " ----");

                int errorCount = 0;
                foreach (var item in trainingSet)
                {
                    var output = perceptron.Coaching(item.Output, item.Inputs);

                    if (output != item.Output)
                    {
                        Console.WriteLine("Durchgefallen\t {0} & {1} & {2} != {3}",
                            item.Inputs[0], item.Inputs[1], item.Inputs[2], output);
                        errorCount++;
                    }
                    else
                    {
                        Console.WriteLine("Richtig\t {0} & {1} & {2} = {3}",
                            item.Inputs[0], item.Inputs[1], item.Inputs[2], output);
                    }
                }

                if (errorCount == 0)
                    break;
            }
        }
    }
}
```

Die Datenelemente *CoachingItem* werden in der Klasse *CoachingItem* für das Perzeptron aufbereitet. Listing 2.3 zeigt den einfachen Aufbau der Klasse. Fügen Sie hierfür in Visual Studio über das Kontextmenü *Add/New Item...* der Projektsolution eine neue Klasse hinzu.

Listing 2.3 Die Klasse CoachingItem

```
namespace NotAndPerceptron
{
    public class CoachingItem
    {
        public double[] Inputs { get; private set; }
        public bool Output { get; private set; }

        public CoachingItem(bool expectedOutput, params double[] inputs)
        {
            Output = expectedOutput;
            Inputs = inputs;
        }
    }
}
```

Über den Aufruf *perceptron.Coaching* in Listing 2.2 verwendet das Perzeptron hinter den Kulissen die Trainingsdaten, um zu lernen wie man klassifiziert. So wird durch die Methode *Coaching* dann ein fertig gelerntes und einsatzbereites Perzeptron zurückgegeben. Das *NotAnd-Perceptron*-Beispiel zeigt also sehr vereinfacht, wie Sie mit Hilfe eines Perzeptron einfache logische Funktionen berechnen können. Des Weiteren verdeutlicht dieses Beispiel die wesentliche Architektur von ML-Systemen. Ein gelerntes ML-Modell, hier die Klasse *Perceptron*, ist durch Parameter (Gewichte) beschrieben, deren Werte durch den Lernalgorithmus der Methode *Coaching* auf Basis eines Datensatzes bestimmt werden.

Lernalgorithmus

Das Lernen in einem KNN erfolgt in der Regel durch die Veränderung der Gewichte zwischen den Neuronen. Die Lernregeln geben dabei an, wie das Netz lernen soll, für eine vorgegebene Eingabe eine gewünschte Ausgabe zu produzieren.

Die Lernregel für das einstufige Perzeptron funktioniert nur, wenn der Trainingsdatensatz linear separierbar ist. So entsteht folgende Lernregel:

1. Wenn die Ausgabe eines Neurons 1 (bzw. 0) ist und den Wert 1 (bzw. 0) annehmen soll, dann werden die Gewichtungen nicht geändert.
2. Ist die Ausgabe 0, soll aber den Wert 1 annehmen, so erfolgt eine schrittweise Erhöhung des Gewichtes.
3. Ist die Ausgabe 1, soll aber den Wert 0 annehmen, so erfolgt eine schrittweise Verminderung des Gewichtes.

Das heißt, dieser Algorithmus korrigiert immer genau dann den Gewichtsvektor, wenn das Perzeptron einen Punkt falsch klassifiziert hat. Daraus ergibt sich, dass nur eine endliche Anzahl an Korrekturen vorgenommen wird, wenn die Werte sich linear aussortieren lassen. Das Lernen ist beendet, sobald alle Punkte richtig eingeteilt wurden. Das Beispielprogramm schafft dies bei einer Lernrate von 0,2 und einem Schwellenwert von 0,5 nach sieben Durchläufen (Bild 2.11).

```
C:\Beispiele_Kapitel_2\NotAndPerceptron\NotAndPerceptron\bin\Debug\NotAndPerceptron.exe
---- Versuch: 1 ----
Durchgefallen 1 & 0 & 0 != False
Durchgefallen 1 & 0 & 1 != False
Durchgefallen 1 & 1 & 0 != False
Durchgefallen 1 & 1 & 1 != True
---- Versuch: 2 ----
Durchgefallen 1 & 0 & 0 != False
Richtig 1 & 0 & 1 = True
Richtig 1 & 1 & 0 = True
Durchgefallen 1 & 1 & 1 != True
---- Versuch: 3 ----
Durchgefallen 1 & 0 & 0 != False
Durchgefallen 1 & 0 & 1 != False
Richtig 1 & 1 & 0 = True
Durchgefallen 1 & 1 & 1 != True
---- Versuch: 4 ----
Richtig 1 & 0 & 0 = True
Durchgefallen 1 & 0 & 1 != False
Durchgefallen 1 & 1 & 0 != False
Durchgefallen 1 & 1 & 1 != True
---- Versuch: 5 ----
Richtig 1 & 0 & 0 = True
Richtig 1 & 0 & 1 = True
Durchgefallen 1 & 1 & 0 != False
Durchgefallen 1 & 1 & 1 != True
---- Versuch: 6 ----
Richtig 1 & 0 & 0 = True
Durchgefallen 1 & 0 & 1 != False
Richtig 1 & 1 & 0 = True
Richtig 1 & 1 & 1 = False
---- Versuch: 7 ----
Richtig 1 & 0 & 0 = True
Richtig 1 & 0 & 1 = True
Richtig 1 & 1 & 0 = True
Richtig 1 & 1 & 1 = False
```

Bild 2.11 Ergebnis des Lernalgorithmus

Das einstufige Perzeptron und sein Lernalgorithmus stellen nur das einfachste KNN dar. Mit diesem Perzeptron kann man, wie aufgezeigt, nur zwei Punktmengen linear gliedern. Leider sind in der Praxis die Punkte bzw. die Datenmengen nicht immer eindeutig und daher linear nicht separierbar und somit durch ein einstufiges Perzeptron nicht mehr darstellbar. Dies ist zum Beispiel bei einem XOR-Operator der Fall. Unter der XOR-Funktion (exklusives ODER) versteht man die boolesche Funktion, welche genau dann 1 ist, wenn genau eine ihrer beiden Eingaben 1 ist. Andernfalls ist sie 0. Diese Funktion lässt sich durch das einstufige Perzeptron nicht darstellen.

Abhilfe schaffen hier nur weitere Schichten im neuronalen Netz. So entstehen die sogenannten *Multilayer Perceptrons* (MLP). Sie ermöglichen es, beliebige Bereiche zu klassifizieren. Das einlagige Perzeptron und sein einfacher Lernalgorithmus bieten lediglich einen ersten Einblick in die Funktionsweise künstlicher neuronaler Netze. In Kapitel 3 werden Sie einige praktische Beispiele finden und in Kapitel 7 werden Sie die Leistungsfähigkeit der entsprechenden Machine-Learning-Frameworks kennenlernen.

■ 2.9 Deep Learning

Deep Learning (kurz DL) ist ein Optimierungsverfahren für künstliche neuronale Netze und eine weitere Teildisziplin des Machine Learning. Beim Deep Learning verwenden die neuronalen Netze eine umfangreiche innere Struktur, die durch eine Vielzahl von Zwischen-schichten, den Hidden Layers, zwischen Eingabeschicht (Input Layer) und Ausgabeschicht (Output Layer) aufgebaut wird. Bild 2.12 zeigt die Grundstruktur eines künstlichen neuro-nalen Netzes (KNN) für Deep Learning.

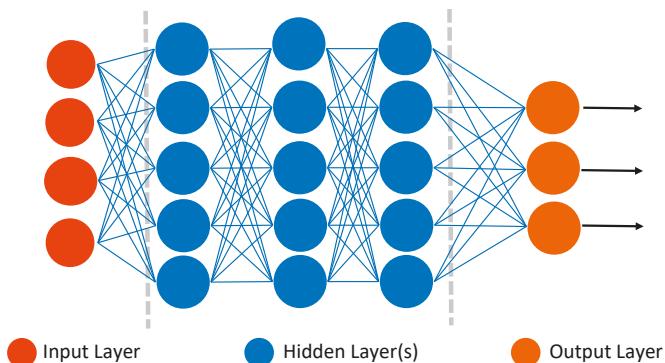


Bild 2.12

Darstellung eines künstlichen neuronalen Netzes

Wie Sie in Bild 2.12 sehen, kann ein neuronales Netz beliebig komplex aufgebaut werden. Einfache KNNs können wie hier abgebildet nur drei Hidden Layer haben, oder aber auch in ihrer komplexen Form eine Vielzahl (> 100) von Hidden Layer besitzen. Durch die Möglichkeit der tiefgehenden Abstraktion benötigt man aber für die Herstellung von Trainingsmethoden eine sehr große Datenmenge, aus der sich Muster und Modelle ableiten lassen.

Des Weiteren versucht man bei Deep Learning, das System in die Lage zu versetzen, selbstständig und ohne menschliche Hilfe seine Fähigkeiten zu verbessern. Hierzu nutzt man eine große Anzahl von Daten für die Klassifizierung und verknüpft während des Lernvorgangs das künstliche neuronale Netz immer wieder neu. Das heißt, der entscheidende Unterschied zwischen Machine Learning und Deep Learning besteht darin, dass bei ML der Mensch in die Analyse der Daten und in den Entscheidungsprozess eingreift. Bei DL werden lediglich die Informationen für das Lernen bereitgestellt. Die Analyse und das Ableiten von Prognosen oder Entscheidungen nimmt das DL-System selbst vor.

Durch ihre Leistungsfähigkeit sind Deep-Learning-Systeme den Machine-Learning-Systemen bei vielen Aufgaben im Bereich Sprach- und Bilderkennung sowie in der Robotik weit überlegen. So wird Deep Learning heute hauptsächlich in folgenden Bereichen erfolgreich eingesetzt:

- Bildklassifizierung
- Spracherkennung
- Handschrifterkennung
- Übersetzen von Fremdsprachen
- Digitale Assistenten in Form von Amazon Alexa, Apple Siri oder auch Google Now
- Autonomes Fahren
- Verbesserung der Suchergebnisse im Web

Die Vielzahl von Aufgaben, die Deep Learning lösen kann, wächst mit jedem Tag. Deep Learning bietet gegenüber Machine Learning die folgenden Vorteile:

- Bessere Ergebnisse als mit ML-Methoden
- Keine Feature-Entwicklung und kein Datenlabeling notwendig
- Effiziente Erledigung von Routinearbeiten ohne Qualitätsschwankungen
- Umgang mit unstrukturierten Daten
- Services zur Nutzung von künstlichen neuronalen Netzen

Man sollte aber auf jeden Fall beachten, dass für Deep Learning sehr große Sätze von Trainingsdaten benötigt werden. Stehen diese nicht zur Verfügung, so liefert Deep Learning leider keine wirklich guten Ergebnisse. Daher hat Deep Learning folgende Nachteile:

- Es erfordert eine hohe Rechenleistung.
- Die Entwicklung von Lernalgorithmen braucht viel Zeit.
- Es ist zwingend eine große Datenbasis erforderlich.

■ 2.10 Einsatzgebiete und Anwendungen

Bekannte Anwendungen von Machine Learning und Deep Learning findet man z. B. bei den Empfehlungsdiensten von Amazon und Netflix. Auch die künstlichen Assistenten von Apple, Google und Co. sowie die Gesichtserkennung von Facebook, die die Möglichkeit bietet, einzelne Mitglieder mit ihrem Namen auf Bildern zu markieren, sind vielen ein Begriff.

Darüber hinaus sind Machine Learning und Deep Learning inzwischen Schlüsseltechnologien, um Systeme und Maschinen mit neuen Fähigkeiten auszustatten, die in den unterschiedlichsten Anwendungsbereichen zum Einsatz kommen.

Verkehr, Mobilität und Transportfahrzeuge

ML- und DL-Algorithmen übernehmen nicht nur verschiedene Aufgaben in autonom fahrenden Autos. Auch Navigationsgeräte versuchen die Gewohnheiten ihres Fahrers zu analysieren und aus der Routine des Fahrers zu lernen um, entsprechende Routen im Vorfeld besser anzupassen.

Es werden auch Daten über Straßenzustände, Wetter und Verkehrsaufkommen erfasst, um mithilfe von ML z. B. auf ein frühzeitiges Ausbessern der Straße hinzuweisen. Auch fahrerlose Transportfahrzeuge werden heute autonom durch Fabriken gelenkt. Sie bilden einen wichtigen Baustein auf dem Weg zum vollautomatischen Lager.

Gesundheitswesen

Mit ML ist es möglich geworden, täglich anfallende Daten aus dem medizinischen Bereich zu analysieren und Ärzte bei Untersuchungen wie Bluttests, Röntgenaufnahmen und in der Krebsvorsorge bzw. Krebserkennung mit entsprechender Software zu unterstützen.

So versucht man z. B., ML-Algorithmen so zu trainieren, dass sie erkennen können, ob es sich bei der Abbildung einer Lunge im hochauflösenden CT-Scan um eine Läsion handelt, die krebshaltig ist oder nicht.

Marketing und Vertrieb/Verkauf

In diesem Bereich versucht man beispielsweise mithilfe von ML die Automation in der Kundenkommunikation zu steigern. Mit ML kann das Kaufverhalten über große Zeiträume hinweg analysiert werden, um Produktplatzierungen in Web-Shops vorzunehmen. Sehr häufig kommt es in diesem Bereich auch zum Einsatz von Service-Chatbots. Diese versuchen auf Firmenwebseiten oder am Telefon, einen Mitarbeiter bestmöglich zu imitieren, um lange Warteschleifen zu unterbinden oder triviale Aufgaben zu lösen.

Produktion und Logistik

Im Bereich Produktion und Logistik versucht man mithilfe von ML eine bessere Vorhersage von Bedarfs- und Bestandssituationen zu treffen, um die Prozessplanung zu optimieren. Auch im Bereich Industrie 4.0 kommt sehr häufig ML zum Einsatz. Vorreiter sind hier die vorausschauende und präventive Wartung, die Bild- und Objekterkennung bei der Qualitätskontrolle sowie die robotergesteuerte Prozess-Automatisierung bei der wiederkehrende, manuelle Routinetätigkeiten durch sogenannte Softwareroboter erlernt und automatisiert ausgeführt werden.

Ein weiteres ML-Einsatzgebiet ist die Maschinenbedienung durch ein Expertensystem. Hierdurch reduzieren sich Einarbeitungszeit, Schulungsaufwand und Rüstzeiten. ML ermöglicht die Prozesse im gesamten Kontext zu optimieren.

Finanz- und Versicherungswesen

In diesem Bereich wird ML genutzt um, in den komplexen Regelwerken der Finanz- und Versicherungsbranche einzigartige Aktivitäten oder auffällige Verhaltensweisen zu entdecken. Auch Regeln für die Vergabe von Krediten werden immer öfter über ML-Algorithmen definiert. Im Kontext der Kundenanfragen werden auch hier schriftliche- oder sprachbasierte Chatbots eingesetzt, um automatisierte und standardisierte Antworten zu geben.

IT Security

Der Security Sektor setzt schon lange auf verfeinerte ML-Algorithmen. Diese helfen bei der automatisierten Erkennung externer und interner Risiken und Angreifer. Auch für das Erkennen von Anomalien in Netzwerken werden sie eingesetzt und leiten die Erkenntnisse an entsprechende Sicherheitsteams weiter.

Dies ist nur ein kleiner Ausschnitt für die Einsatzmöglichkeiten von Machine Learning und Deep Learning. Szenarien im Katastrophenmanagement, im Zusammenhang mit intelligenten Stromnetzen, bei der vorausschauenden Polizeiarbeit und das Erkennen von Mineralvorkommen sind weitere mögliche Anwendungsgebiete, die auch heute schon genutzt werden. Da sich Machine Learning und Deep Learning rasant weiterentwickeln, kommen praktisch jeden Tag neue Anwendungsfälle hinzu.

3

Neuronale Netze

Nachdem Sie in den Kapiteln 1 und 2 die zahlreichen Facetten von Machine Learning und Deep Learning kennengelernt und sich mit den Begriffen KI, AI, neuronale Netze und Cognitive Computing vertraut gemacht haben, geht es nun um die Programmierung von künstlichen neuronalen Netzen mit C#.

Warum sollten Sie selber ein neuronales Netz programmieren? Es gibt doch inzwischen eine Vielzahl von Frameworks und Softwarebibliotheken, die Sie dabei unterstützen. Diese Frameworks haben aber auch einen gewissen Overhead an Funktionalität und Umfang, sodass Sie als Entwickler gezwungen sind, eventuell nicht nur das gewünschte Datenmuster zu erkennen bzw. zur Verfügung zu stellen, sondern auch noch die Besonderheiten des gewählten Frameworks erforschen müssen. Der Umgang mit einem solchen Framework ist ohne eine entsprechende Lernkurve nicht zu bewältigen.

Das *NotAndPerceptron*-Programmbeispiel in Kapitel 2 zeigte ja schon die Funktionsweise eines einlagigen Perzeptron bzw. dem *Singlelayerperceptron* (SLP). Allerdings wurde dort mit dem neuronalen Netz noch nicht wirklich experimentiert, sondern nur dessen Implementierung auf einfache Art und Weise aufgezeigt.

Dieses Kapitel zeigt, wie ein neuronales Netz funktioniert und wie Sie ein solches in C# programmieren können. Es werden in diesem Buch die Grundlagen und Konzepte beschrieben, um einige in der Praxis vorkommende Aufgaben mit neuronalen Netzen und Machine-Learning-Algorithmen zu lösen, die in C# programmiert wurden. Die gezeigten Programmbeispiele sollen Sie anregen eigene Lösungen für neuronale Netze und Machine-Learning-Modelle zu entwickeln.

■ 3.1 Vom Problem zum KNN

Ein künstliches neuronales Netz, im Folgenden kurz als KNN bezeichnet, sieht von außen betrachtet wie ein Black-Box-Modell aus. Es bekommt Daten als Eingaben (Input) und generiert daraufhin Ausgaben, den sogenannten Output.

Hierbei bleibt die innere Struktur für den Anwender verborgen und die Parameter des KNN werden so angepasst, dass ein Verhalten zwischen Input und Output gelernt und abgebildet wird. Beachten Sie daher als Entwickler und Anwender, dass ein KNN nur als Modell zur Annäherung an beliebige funktionale Zusammenhänge dient. Daher ist auch hier ein

strukturiertes Vorgehen bei der Suche nach dem richtigen Netztyp sehr hilfreich (siehe Abschnitt 2.5, „Machine-Learning-Algorithmen“).

Folglich lassen sich auch die Aufgaben wie beim Machine Learning für ein KNN grob in vier Kategorien einteilen. Als Erstes trifft man auch hier auf die Klassifikation und damit auf die Aufteilung der Daten bzw. Objekte in vordefinierte Klassen. Auch die Assoziation, die Suche nach Abhängigkeiten zwischen den Objekten, gehört zu den Lösungsmöglichkeiten von KNNs. Bei der Assoziation sind in den meisten Fällen immer lineare Zusammenhänge zwischen dem gemeinsamen häufigen Auftreten von zwei oder mehreren Ereignissen gemeint.

Des Weiteren lassen sich mit KNNs auch Clustering-Verfahren gut umsetzen. So lassen sich ähnliche Objekte in möglichst homogene Teilmengen einsortieren. Als vierte große Kategorie kommt jetzt noch die Prognose hinzu. Hier versucht man mithilfe des KNN Trends bzw. Vorhersagen im sogenannten Datenzeitbezug zu identifizieren.

Zeitreihen wie Aktienkurse an der Börse sind besonders gut für diese datenorientierte Modellierung geeignet. Die Kursentwicklungen werden transparent veröffentlicht und somit steht eine große Menge von Trainingsdaten zur Verfügung.

■ 3.2 KNN-Modelle

Einen vollständigen Überblick über alle existierenden KNNs zu behalten, ist angesichts der unüberschaubaren Anzahl bekannter, modifizierter und immer wieder neu hinzukommender Typen kaum mehr möglich. Aufgrund dessen sollen hier einige bekannte Modelle in Ergänzung zu Kapitel 1 nur kurz vorgestellt werden, da sie Grundlage der heute bekannten neuronalen Netze sind. Eine ausführliche Beschreibung würde den Rahmen des Buches sprengen und könnte auch nur theoretisches Wissen vermitteln. Wenn Sie nach den entsprechenden Namen online suchen, finden Sie weitere informative Quellen zu diesen Netzen. Dort können Sie sich über den Aufbau, das Training und die Verwendung ausführlich informieren.

Adaline

Adaline (Adaptive Linear Neuron) ist ein einzelnes Neuron mit Schwellenwertlogik und einer Ausgabe mit den Werten +1 oder -1.

Die Grundlagen von Adaline wurden schon 1959 von Bernard Widrow und Marcian Edward Hoff entwickelt und es dient zur Mustererkennung und als Assoziativspeicher. Das KNN von Adaline mit n Eingabeeinheiten (Inputs) kann unter Verwendung der Delta-Regel maximal n linear unabhängige Muster fehlerfrei speichern. Die Delta-Regel (siehe Kasten) lässt sich auch auf ein *Singlelayerperceptron* anwenden.

Madaline

Madaline (Multiple Adaptive Linear Neuron) besitzt im Vergleich zum Adaline zusätzlich eine versteckte Schicht. Die Gewichte der versteckten Schicht sind wie bei der Eingabeschicht konstant. Eine Besonderheit ist, dass es von jedem Neuron der zweiten, verborgenen Schicht genau eine Verbindung mit dem Gewicht 1 zum Ausgabeelement gibt.



Delta-Regel

Die Delta-Regel, auch Widrow-Hoff-Regel genannt, beruht auf einem Vergleich zwischen dem gewünschten und dem tatsächlich beobachteten Output des Ausgangsneuronen.

Hierbei kann man drei Möglichkeiten unterscheiden:

- Die beobachtete Aktivität ist zu niedrig. Um die Aktivität zu steigern, werden die Gewichte zwischen den sendenden und den empfangenden Neuronen erhöht, sofern von den sendenden Neuronen ein positiver Input ausgeht. Die Gewichte zu den sendenden Neuronen mit negativem Input werden hingegen gesenkt.
- Die beobachtete Aktivität ist zu groß. Da die Aktivität hier reduziert werden soll, schwächt man alle Verbindungen, bei denen der Input positiv ist, und stärkt die Verbindungen mit negativem Input.
- Die beobachtete und die gewünschte Aktivität sind gleich groß, stellt also das gewünschte Resultat dar. In diesem Fall erfolgen keinerlei Änderungen.

Diese Möglichkeiten lassen sich mit der Formel zur Delta-Regel ableiten:

$$\Delta W_{ij} = \eta O_i (P_j - O_j)$$

- ΔW_{ij} : Gewichtsänderung von i nach j .
- η : Gibt die Lernrate an.
- O_i : Ausgabe des Neurons i
- P_j : Soll-Ausgabe des Ausgabeneurons j .
- O_j : Ist-Ausgabe des Ausgabeneurons j .

Die Delta-Regel versteht sich als Fehlerkorrektur, die künftiges, nicht erwünschtes Verhalten in neuronalen Netzen unwahrscheinlich machen soll. Die Delta-Regel ermöglicht es, das abstrakte Neuronen-Modell weiter zu verfeinern. Dass liegt daran, dass die Delta-Regel einen Spezialfall des bekannten Backpropagation-Algorithmus abbildet, da sie nur bei linearen Aktivierungsfunktionen bei einer einzigen Schicht trainierbarer Gewichte verwendet wird. Der Backpropagation-Algorithmus wird in Abschnitt 3.7.3 noch ausführlich erklärt und als Codebasis implementiert.

Hopfield-Netz (HN)

Hopfield-Netze sind einschichtige, rekursive (rückgekoppelte) Netze, die als Input- und Output-Schicht dienen. Das Netz wurde 1982 von John Hopfield vorgestellt und hat die Besonderheit, dass jedes der Neuronen mit jedem anderen Neuron verbunden ist, außer mit sich selbst.

Auch im Hopfield-Netz können die Outputs der Neuronen die Werte +1 oder -1 annehmen. Gelernt wird einfach durch die Anpassung der Gewichte für alle Trainingsbeispiele. Bei rückgekoppelten Netzen müssen die Zustände der Neuronen solange neu berechnet werden, bis das Netz sich in einen sogenannten Ruhezustand begeben hat, das heißt, bis sich keine Änderung der Aktivierungszustände mehr ergeben.

Boltzmann-Maschine (BM)

Die Boltzmann-Maschine wurde 1985 vorgestellt. Sie ist ein rekursives, stochastisches Netz. Die Statuszustände, die das Netz annehmen kann, werden durch die sogenannte Boltzmann-Verteilung bestimmt, einer exponentiellen Form der Wahrscheinlichkeitsverteilung, die für die Modellierung der Statuszustände eines physikalischen Systems bei thermischem Gleichgewicht genutzt wird. Die Boltzmann-Maschine kann als Assoziativspeicher (siehe Kasten) oder zur Lösung von Optimierungsproblemen verwendet werden.



Assoziativspeicher

Ein Assoziativspeicher ist ein Speicher, auf den man nicht wie üblich per Adresse, sondern über seinen Inhalt zugreifen kann. In einem Assoziativspeicher werden die Informationen somit überlagernd abgelegt und nicht an einem bestimmten Ort.

Daher eignet sich ein neuronales Netz, wie das Hopfield-Netz, auch zum Realisieren von Assoziativspeichern. Beachten Sie hierbei, dass sich der Speicher über sämtlichen Gewichten im neuronalen Netz verteilt. Die Anzahl der Daten eines Assoziativspeichers ist kleiner oder höchstens gleich der Anzahl der Neuronen in der kleinsten Schicht.

Neural Turing Machine (NTM)

Die Neural Turing Machine, kurz NTM, erweitert rekurrende neuronale Netze mit einem adressierbaren externen Speicher. Besondere Aufmerksamkeit erhielt das NTM durch Google DeepMind, die Google Entwickler nennen ihr System Neural Turing Machine, benannt nach dem KI-Pionier Alan Turing.

Die Besonderheit ist die NTM-Architektur. Diese enthält zwei grundlegende Komponenten: einen neuronalen Netz-Controller und eine Memory-Matrix, die als Speicherbank funktioniert. Bild 3.1 zeigt die NTM-Architektur. Der Controller, der ein wiederkehrendes oder vorwärts gerichtetes neuronales Netz sein kann, verarbeitet die Eingangs- und Ausgangsvektoren sowie die Übergabe von Schreib- und Lesebefehlen an den Speicher, in diesem Fall also der Memory-Matrix.

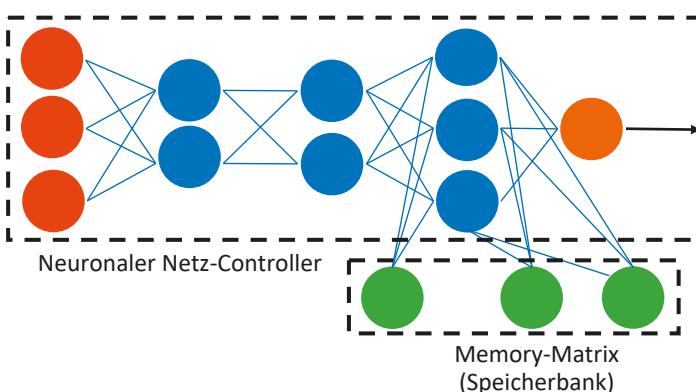


Bild 3.1
NTM-Architektur

Der Controller interagiert somit auch mit seiner Memory-Matrix durch die selektiven Lese- und Schreiboperationen. Durch dieses Vorgehen versucht man ein Kurzzeitgedächtnis zu schaffen, um aktuelle Informationen mit bereits gespeicherten Informationen zu vergleichen und Zusammenhänge herauszustellen.

Die hier kurz beschriebenen Netze bilden die historische Grundlage für die heute eingesetzten neuronalen Netze. Im folgenden Abschnitt beschäftigen wir uns nun mit den mathematischen Grundlagen neuronaler Netze.

■ 3.3 Mathematik neuronaler Netze

Bevor Sie mit der Programmierung von neuronalen Netzen beginnen, geht es in den folgenden Abschnitten noch um einige wichtige Begriffe aus der linearen Algebra, die Sie sowohl bei der Entwicklung von neuronalen Netzen wie auch bei der Implementierung vorhandener Lernalgorithmen benötigen.

Neuronale Netze werden mathematisch durch Vektoren und Matrizen aus der linearen Algebra abgebildet. Daher benötigen Sie ein paar Grundkenntnisse in Mathematik und Statistik. Des Weiteren gehört auch der Umgang mit booleschen Logikfunktionen und Wahrheitswerten zu den Voraussetzungen, da man es bei der Entwicklung von neuronalen Netzen nicht immer nur mit einem einfachen Klassifizierer zu tun hat.

3.3.1 Lineare Algebra

Die lineare Algebra ist eine Form der kontinuierlichen Mathematik. In den Bereichen Machine Learning und insbesondere im Deep Learning nutzen Sie als Entwickler oder Anwender intensiv lineare Algebra-Operationen mit weiteren verwandten Algorithmen, geometrischer Transformation und numerischen Solvern.

Diese komplexen Funktionen arbeiten mit Matrizen und Vektoren, die eine große Anzahl von Additionen und Multiplikationen von Zahlen erfordern. Daher gibt es in der linearen Algebra vier Arten von mathematischen Objekten, die Sie unbedingt kennen sollten:

- **Skalare:** Ein Skalar ist nur eine einzige Zahl, wie zum Beispiel 6, im Gegensatz zu den meisten anderen Objekten, die in der linearen Algebra untersucht werden, bei denen es sich in der Regel immer um Arrays aus mehreren Zahlen handelt.
- **Vektoren:** Ein Vektor ist ein Array von Zahlen. Sie können jede einzelne Zahl durch ihren Index im Array identifizieren. Im Allgemeinen bezeichnet man einen Vektor als ein Element aus einem Vektorraum. Aufgrund dessen kann ein Vektor auch durch einen Pfeil beschrieben werden, der von einem Startpunkt zu einem Endpunkt führt. Die Zahlenpaare im Array beschreiben dann die Richtung des Vektors.
- **Matrizen:** Matrizen bestehen aus m Zeilen und n Spalten und bilden somit eine Tabelle. Die Dimension einer einzelnen Matrix mit m Zeilen und n Spalten ist $m \times n$. Die Elemente einer Matrix bezeichnet man auch als Koeffizienten.

- **Tensoren:** Tensoren sind Größen, mit deren Hilfe man Skalare, Vektoren sowie Matrizen in ein einheitliches Schema einordnen kann, um mathematische und physikalische Zusammenhänge zu beschreiben.

So unterschiedlich auch die Einsatzgebiete der verschiedenen neuronalen Netze sein mögen, im Kern werden im Machine Learning und Deep Learning zwei Schritte durchgeführt: Der erste Schritt ist, Objekte als Vektoren darzustellen. Der zweite Schritt ist dann, mit den gewonnenen Vektoren zu rechnen, etwa um die Abstände zu ermitteln oder Vektoren zu finden, die in eine ähnliche Richtung zeigen. Wird aufgrund dessen ein Problem auf Vektoren zurückgeführt, so können alle Methoden darauf angewendet werden, ohne auf eine konkrete Aufgabenstellung einzugehen.

3.3.2 Vektor

Wie gesagt, bezeichnet ein Vektor eine Verschiebung in der Ebene oder im Raum und wird durch einen Pfeil repräsentiert, dessen Länge und Richtung genau die Länge und Richtung der Verschiebung ist. Die Ein- und Ausgabe in einer Neuronenschicht lässt sich sehr einfach durch einen Vektor darstellen. Bild 3.2 zeigt die Vektoren im zwei- und dreidimensionalen Raum.

Durch eine Festlegung der Objekte als Vektor ist somit im Machine Learning und Deep Learning die Möglichkeit einer Klassifizierung gegeben. Sie können, wie in Bild 3.2 dargestellt, zu einem beliebigen Punkt in einem zwei- oder dreidimensionalen Raum ($X_1 | X_2 | X_3$), alternativ wird auch die Bezeichnung ($X | Y | Z$) gewählt, mit einer einfachen Angabe wie $P(2 | 3 | 5)$ gelangen, indem Sie vom Nullpunkt des Koordinatensystems 2 Einheiten in $X_1(X)$ -Richtung, 3 Einheiten in $X_2(Y)$ Richtung und dann einfach 5 Einheiten in $X_3(Z)$ -Richtung gehen. So einfach können Sie den Vektor in einem dreidimensionalen Raum bestimmen.

Daher gilt in einem zweidimensionalen Raum die Regel:

- Alle Punkte auf der y-Achse haben den x-Wert 0. Hier gilt $P_y(0 | Y)$.
- Alle Punkte auf der x-Achse haben den y-Wert 0. Hier gilt $P_x(X | 0)$.

Für einen dreidimensionalen Raum gilt:

- Alle Punkte in der X_1X_2 -Ebene haben den X_3 -Wert 0. Hier gilt $P(X_1 | X_2 | 0)$.
- Alle Punkte in der X_2X_3 -Ebene haben den X_1 -Wert 0. Hier gilt $P(0 | X_2 | X_3)$.
- Alle Punkte in der X_1X_3 -Ebene haben den X_2 -Wert 0. Hier gilt $P(X_1 | 0 | X_3)$.

Wie das Beispiel aus Bild 3.2 zeigt, wird ein Vektor grafisch durch einen Pfeil dargestellt, der von Punkt zu Punkt zeigt. So wird jeder Punkt im Vektor eindeutig durch seine Koordinaten festgelegt. Da der gezeigte Vektor immer vom Koordinatenursprung ausgeht, wird dieser als Ortsvektor bezeichnet. Die Koordinaten können bei einem Ortsvektor auch spaltenweise aufgeschrieben werden. Ein Ortsvektor, der zum Punkt P führt, kann demnach wie folgt dargestellt werden:

$$\vec{r} = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \text{ ist } P(X_1 | X_2 | X_3)$$

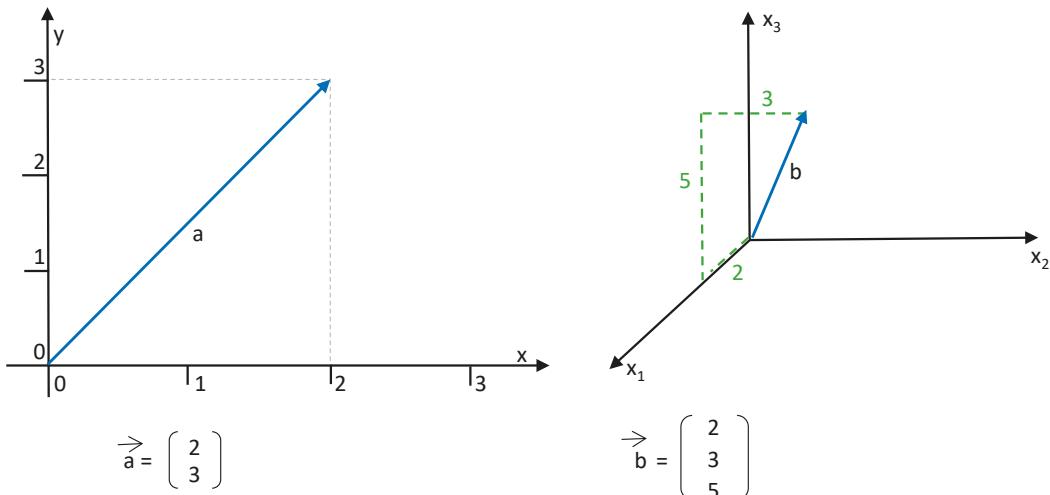


Bild 3.2 Vektor in den zwei- und dreidimensionalen Räumen

Insofern bezeichnet man einen Vektorraum V in der linearen Algebra als die Menge der Vektoren mit einer bestimmten Anzahl von Einträgen.

Auch das Rechnen mit Vektoren stellt kein Hexenwerk dar. So können Vektoren addiert, subtrahiert und auf mehrere Weisen multipliziert werden.

3.3.2.1 Rechnen mit Vektoren

Addition und Subtraktion lassen sich für Vektoren sehr einfach durchführen. Allerdings lassen sich Vektoren nur dann addieren bzw. subtrahieren, wenn sie die gleiche Dimension haben und von gleicher Art sind.

Bei der Addition und Subtraktion werden die einzelnen $X(X_1)$ -Werte und $Y(X_2)$ -Werte und, sofern vorhanden, auch die $Z(X_3)$ -Werte addiert bzw. voneinander abgezogen. Hier nun die allgemeine Formel und ein Beispiel für die Addition:

$$\vec{a} + \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

Rechnung:

$$\begin{pmatrix} 3 \\ -1 \\ 4 \end{pmatrix} + \begin{pmatrix} 4 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 3 & + & 4 \\ -1 & + & 1 \\ 4 & + & (-1) \end{pmatrix} = \begin{pmatrix} 7 \\ 0 \\ 3 \end{pmatrix}$$

Die Subtraktion von Vektoren funktioniert ähnlich, hier werden die Werte entsprechend voneinander abgezogen.

3.3.2.2 Skalarprodukt

Das Skalarprodukt, auch inneres Produkt genannt, ist eine mathematische Verknüpfung, die zwei Vektoren eine Zahl (Skalar) zuordnet. Wird ein Vektor \vec{a} mit der reellen Zahl \mathbb{R} multipliziert, wird jede Komponente des Vektors mit dieser Zahl multipliziert:

$$\mathbb{R} \cdot \vec{a} = \mathbb{R} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} \mathbb{R} \cdot X \\ \mathbb{R} \cdot Y \\ \mathbb{R} \cdot Z \end{pmatrix}$$

Das Skalarprodukt von zwei Vektoren A und B in einem dreidimensionalen Raum wird wie folgt abgebildet:

$$\vec{a} \cdot \vec{b} = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

Und als Zahlenbeispiel wie folgt:

$$\begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix} = 2 \cdot 1 + 1 \cdot 4 + 3 \cdot 1 = 9$$

$$\begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 2 \\ 2 \end{pmatrix} = -2 + 0 + 2 = 0$$

Ist das Ergebnis eines Skalarprodukts 0 (null), so stehen beide Vektoren senkrecht (orthogonal) aufeinander.

Somit können Sie mit Vektoren eine eindimensionale Ausgabeschicht Y mit m Ausgängen und eine eindimensionale Eingabeschicht X mit n Eingängen darstellen:

$$\vec{Y} = \begin{pmatrix} Y_0 \\ \vdots \\ Y_m \end{pmatrix} \quad \vec{X} = \begin{pmatrix} X_0 \\ \vdots \\ X_n \end{pmatrix}$$

3.3.3 Matrix

Eine Matrix ist lediglich eine Tabelle bzw. ein rechteckiges Schema, dessen Elemente in den meisten Fällen Zahlen sind. Alternativ können die Matrixelemente aber auch Variablen oder Funktionen abbilden.

Eine Matrix besteht aus m Zeilen und n Spalten und wird als (m, n) -Matrix bezeichnet. Die Dimension einer Matrix ergibt sich aus der Multiplikation von m Zeilen und n Spalten. Die Position eines Elements in der Matrix, zum Beispiel a_{ij} , wird immer mit einem Doppelindex verstehen. Der erste Index i gibt die Zeile und der zweite Index j die Spalte an, in der sich das Element in der Matrix befindet.

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{mn} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{mn} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}$$

Somit stellt man eine (3,2)-Matrix mit Werten wie folgt dar:

$$M = \begin{pmatrix} 1 & 2 \\ 5 & -1 \\ 7 & 8 \end{pmatrix}$$

Sie sollten also immer beachten, dass wirklich zuerst die Anzahl der Zeilen und dann die Anzahl der Spalten angegeben werden. Man kann da schon mal drüber stolpern. Das Element a_{21} wäre in diesem Beispiel also der Wert 5.

Bei neuronalen Netzen beschreibt eine Matrix die Verbindung zwischen zwei Schichten. Die Elemente der Matrix übernehmen dabei die Gewichtung der jeweiligen Verbindung (siehe Abschnitt 1.5.2, „Matrizedarstellung“). Auch Matrizen lassen sich addieren, subtrahieren und multiplizieren. Des Weiteren ist es möglich, Matrizen zu transponieren.

3.3.3.1 Rechnen mit Matrizen

Matrizen lassen sich nur addieren bzw. subtrahieren, wenn die beteiligten Matrizen jeweils die gleiche Anzahl von Zeilen und Spalten besitzen. Die Summe zweier Matrizen A und B erhalten Sie, indem Sie die Koeffizienten (a_{nm}) addieren:

$$M = (a_{ij}) + (b_{ij}) = (a_{ij} + b_{ij})$$

Das rechnerische Beispiel sieht dann wie folgt aus:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}$$

Das Ergebnis der Addition $C = A + B$ bezeichnet man als Summenmatrix oder alternativ auch als Matrixsumme. Hierbei besitzt die Summenmatrix C genauso viele Zeilen und Spalten wie die Matrizen A und B . Das Ergebnis einer Subtraktion von Matrizen, also der Fall $C = A - B$ wird als Differenzmatrix bezeichnet.

Beachten Sie, dass die Anzahl der Zeilen und Spalten von Matrix A und Matrix B übereinstimmen müssen, damit diese addiert bzw. subtrahiert werden können.

3.3.3.2 Matrizenmultiplikation

Die Matrizenmultiplikation lässt sich auf die Multiplikation von Vektoren zurückführen. Hierbei ist zu beachten, dass sich zwei Matrizen nur miteinander multiplizieren lassen, wenn die Spaltenanzahl der ersten Matrix mit der Zeilenanzahl der zweiten Matrix übereinstimmt.

Das Multiplizieren von A und B

$$A_{(2,2)} \cdot B_{(2,2)} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

ist möglich, da die Spaltenanzahl von A der Zeilenanzahl von B entspricht. Das Ergebnis der Multiplikation Matrix $C = A \cdot B$ wird als Matrixprodukt bezeichnet. Das Ergebnis Matrixprodukt C hat so viele Zeilen wie Matrix A und so viele Spalten wie Matrix B .

Im Beispiel besitzt Matrix A ebenso viele Spalten wie die Matrix B Zeilen hat. Zur Berechnung des Matrixproduktes wird das Schema Zeile mal Spalte angewandt:

$$\begin{pmatrix} 7 & 2 \\ 5 & 4 \end{pmatrix} \begin{pmatrix} 3 & 6 \\ 8 & 2 \end{pmatrix} = \begin{pmatrix} (7 \cdot 3) + (2 \cdot 8) & (7 \cdot 6) + (2 \cdot 2) \\ (5 \cdot 3) + (4 \cdot 8) & (5 \cdot 6) + (4 \cdot 2) \end{pmatrix} = \begin{pmatrix} 37 & 46 \\ 47 & 38 \end{pmatrix}$$

Wie Sie an diesem Rechenbeispiel erkennen können, werden zur Berechnung des ersten Matrixelements die Produkte der entsprechenden Einträge der ersten Zeile von A und der ersten Spalte von B aufsummiert. Für das folgende Element der Eingangsmatrix in der ersten Zeile und der zweiten Spalte wird dann die erste Zeile von A und die zweite Spalte von B verwendet. Dann wird das Rechenschema entsprechend der Zeilen und Spalten fortgesetzt.

Durch die Multiplikation der Matrix mit dem Eingabe-Vektor eines neuronalen Netzes ergibt sich der Ausgabe-Vektor der folgenden Neuronenschicht (Bild 3.3). Es ist eindeutig zu erkennen, dass der Vektor der Eingabewerte für die nächste Neuronenschicht sich aus der Summe der gewichteten Ausgabewerte zusammensetzt.

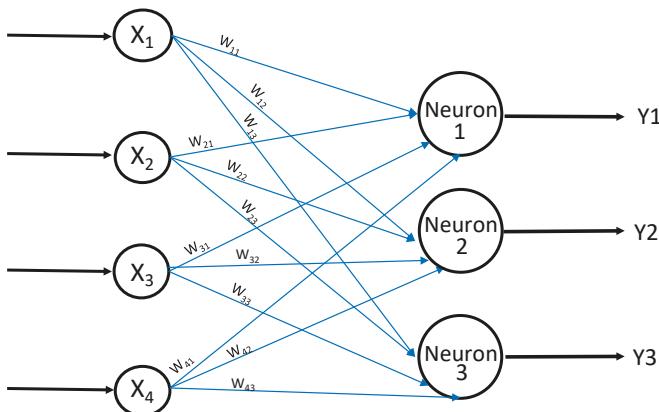


Bild 3.3

Berechnung im neuronalen Netz

Es erfolgt die Multiplikation einer Matrix mit einem Vektor in der Form Zeile mal Spalten. Das Ergebnis ist der Ausgabe-Vektor, dessen Koordinatenanzahl mit der Zeilenanzahl der Matrix übereinstimmt:

$$Y_1 = W_{11} \cdot X_1 + W_{21} \cdot X_2 + W_{31} \cdot X_3 + W_{41} \cdot X_4$$

$$Y_2 = W_{12} \cdot X_1 + W_{22} \cdot X_2 + W_{32} \cdot X_3 + W_{42} \cdot X_4$$

$$Y_3 = W_{13} \cdot X_1 + W_{23} \cdot X_2 + W_{33} \cdot X_3 + W_{43} \cdot X_4$$

Als Vektor und Matrix:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} W_{11} & W_{21} & W_{31} & W_{41} \\ W_{12} & W_{22} & W_{32} & W_{42} \\ W_{13} & W_{23} & W_{33} & W_{43} \end{pmatrix} \cdot \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{pmatrix}$$

3.3.3.3 Transponieren

Von einer transponierten Matrix A^T spricht man, wenn man die Zeilen und Spalten der Matrix vertauscht. Eine Matrix wird also transponiert, indem man aus den Zeilen Spalten macht:

$$A = \begin{pmatrix} 2 & 7 & 1 \\ 4 & 1 & 5 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 2 & 4 \\ 7 & 1 \\ 1 & 5 \end{pmatrix}$$

Durch das Transponieren einer Matrix kann man viele Operationen vertauschen. So zum Beispiel bei der Addition. Die Summe von zwei Matrizen A und B ist gleich der Summe der transponierten Matrix:

$$(A + B)^T = A^T + B^T$$

Die transponierte Matrix hilft uns später bei der Backpropagierung von Fehlern in einem neuronalen Netz.

3.3.4 Tensor

Die heute zur Verfügung stehenden Machine- und Deep-Learning-Frameworks, benötigen Input in Form von Tensoren. Wenn Sie also Daten für ML oder DL zur Verfügung stellen, müssen diese im Allgemeinen numerisch vorliegen. Speziell für die Datenrepräsentation bei neuronalen Netzen wird dies über einen Datenspeicher in Form eines Tensors erreicht. Ein Tensor repräsentiert in diesem Fall einen Container, der Daten in N-Dimensionen aufnehmen kann. Aus Sicht eines Entwicklers kann man Tensoren mit mehrdimensionalen Arrays von Fließkommawerten vergleichen. Somit ist der Tensor als Container in der Lage, einen Vektor als 1-D-Tensor, eine Matrix als 2-D-Tensor oder sogar ein RGB-Bild als 3-D-Tensor abzubilden bzw. aufzunehmen. Man kann daher Tensoren auch mit Datenstrukturen vergleichen.

3.3.5 Eigenwert- und Singulärwertzerlegung

Viele mathematische Objekte können besser analysiert werden, wenn man sie in ihre Bestandteile zerlegt. Des Weiteren kann es auch sinnvoll sein, Eigenschaften zu ermitteln, die universell sind. So kann man zum Beispiel ganze Zahlen in Primfaktoren zerlegen um, diese besser zu identifizieren. Die Art, wie man Zahlen darstellt, ob zur Basis zehn oder binär macht einen Unterschied und kann sich je nach Anwendungsfall als nützlich erweisen.

Singulärwertzerlegung

Die Singulärwertzerlegung [7] bietet eine weitere Möglichkeit, eine Matrix in Singulärvektoren und Singulärwerte zu faktorisieren. Die Zerlegung einer beliebigen Daten-Matrix erlaubt es, die Kleinstes-Quadrat-Schätzung zu finden. Die Singulärwertzerlegung ist auf Matrizen allgemein anwendbar.

Damit sollten die mathematischen Grundlagen wieder ein wenig aufgefrischt sein und Sie können beginnen, sich mit mehrschichtigen neuronalen Netzen zu beschäftigen.

■ 3.4 Mehrschichtige neuronale Netze

Wie Sie in Abschnitt 2.7.1 gelernt haben, ist es mit einem einstufigen Perzeptron nicht möglich, alle booleschen Funktionen der linearen Algebra abzubilden. Ein einstufiges Perzeptron kann lediglich eine lineare Klassifikation leisten. Oftmals ist es jedoch notwendig, weitaus komplexere Klassifikationen durchzuführen. Wie kann man also das XOR (exklusives ODER)-Problem lösen? Sie benötigen für ein richtiges Ergebnis sowohl die Grenzen des AND- als auch jene des OR-Operators.

Da ein Perzeptron, in Form eines Neurons, wie Sie wissen, nur binär klassifizierbar ist – das heißt, die Ausgabe beschränkt sich nur auf 1 oder -1 bzw. 0 oder 1 – ist es in der Praxis nur möglich, eine Klassifikation zu implementieren, in der die 1 für die Erkennung einer konkreten Klasse steht, während alle übrigen Klassen als negativ betrachtet werden. So kam man auf die Idee, wenn man alle Klassen erkennen möchte, dass man einen N-Klassifizierer benötigt. Dementsprechend stellt jeder Klassifizierer ein Neuron (Perzeptron) dar, das auf die Erkennung einer bestimmten Klasse trainiert wird.

Da ein neuronales Netz kein statisches Gebilde ist, sondern als ein dynamisches System aufgefasst werden kann, ist es möglich, mehrere Neuronen (Perzeptronen) miteinander zu verschachteln. So entsteht ein mehrschichtiges neuronales Netz, mit dem sich dann auch das XOR-Problem lösen lässt.

3.4.1 Multilayer Perceptron (MLP)

Das Multilayer Perceptron, kurz als MLP bezeichnet, gliedert sich in verschiedene Schichten, den sogenannten Layers, die wiederum aus einem oder mehreren künstlichen Neuronen bestehen (siehe Abschnitt 1.4.1, „Funktionsweise von KNNs“). Dort wurden die Schichten schon in die Kategorien Input Layer, Hidden Layer und Output Layer unterteilt.

Bild 3.4 zeigt schematisch ein einfaches mehrschichtiges neuronales Netz, das aus einer Eingabeschicht (X_1 bis X_4), einer verdeckten Schicht (Hidden Layer) und einer Ausgabeschicht mit den Knoten Y_1 und Y_2 besteht.

Die erste Schicht eines MLP bildet die Eingabe als Input Layer ab, hierbei handelt es sich in den meisten Fällen um einen Vektor. Diese Schicht ist nur dafür verantwortlich, die einzelnen Werte der Eingänge über die Eingabeneuronen an jedes Neuron der nächsten Schicht weiterzuleiten.

Nach der Eingabeschicht folgen ein oder mehrere verborgene bzw. verdeckte Schichten. In den verdeckten Schichten findet die eigentliche Klassifikation statt. Durch den Einsatz mehrerer Neuronen in den verdeckten Schichten ist es jetzt auch möglich, eine nichtlineare Klassifikation, so wie man sie beim XOR-Problem vorfindet, durchzuführen.

Dieses Vorgehen bietet die Möglichkeit, mathematische Funktionen, die eine oder mehrere numerische Eingaben akzeptieren und eine oder mehrere numerische Ausgaben erzeugen, für eine entsprechende Lösung zu nutzen. Die Funktionen können dann durch logische Verknüpfungen wie AND oder OR miteinander verbunden werden, um so ein Ergebnis zu erreichen. Nach den verdeckten Schichten folgt nur noch die Ausgabeschicht, die auch gleichzeitig die letzte Schicht eines MLP darstellt. Hierbei stellt auch die Ausgabe in den meisten Fällen einen Vektor bereit.

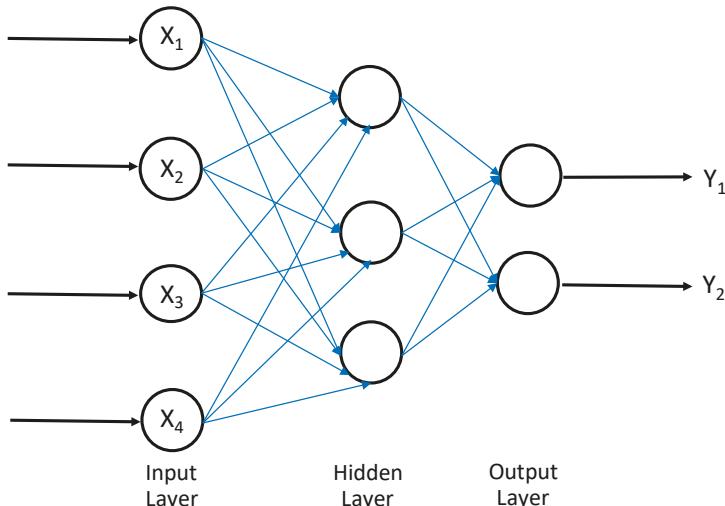


Bild 3.4 Ein einfaches MLP

Beim Multilayer Perceptron handelt es sich immer um ein sogenanntes Feedforward-Netz, das folgende Voraussetzungen erfüllt:

- Die Neuronen (Perzeptrons) einer Schicht sind nur mit den Neuronen der darauffolgenden Schicht verbunden. Es darf keine Schicht ausgelassen werden, und Verbindungen innerhalb einer Schicht oder mit zurückliegenden Schichten sind nicht erlaubt.
- Das Feedforward-Netz (FFN) ist immer fully connected, also ein vollverschachteltes neuronales Netz, da immer alle Neuronen einer Schicht mit allen Neuronen der darauffolgenden Schicht verbunden sind.

Bei einem FFN findet der Datenfluss von der Eingabe zur Ausgabe statt, daher auch die entsprechende Namensgebung. Das MLP stellt eine komplexe mathematische Funktion dar, die aus vielen einfacheren Funktionen zusammengesetzt wird.

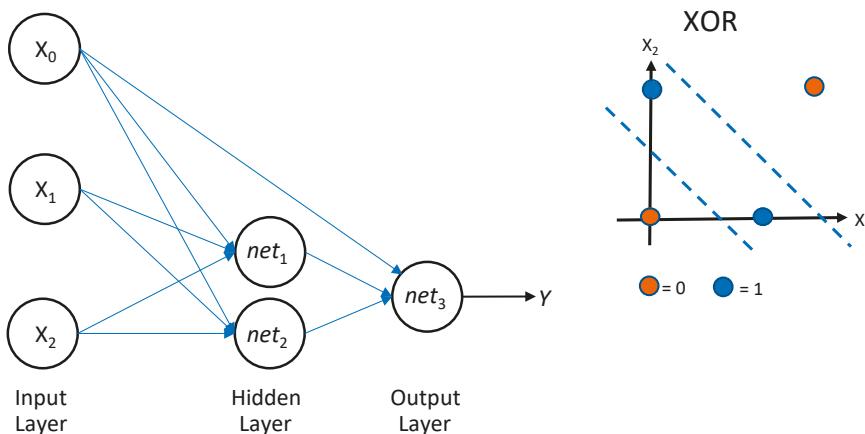


Bild 3.5 Lösung des XOR-Problems

In der Praxis wird das neuronale Netz in den versteckten Schichten mit Neuronen erweitert, die dann eine entsprechende erweiterte boolesche Funktion erfüllen. So lässt sich auch das XOR-Problem mittels eines mehrschichtigen neuronalen Netzes (MLP) wie in Bild 3.5 darstellen.

Hier werden zwei Perzeptrons in Form eines künstlichen Neurons verwendet, die von den gleichen Eingängen gespeist werden. Somit ist es möglich, das Neuron net_1 auf die boolesche Funktion OR und das Neuron net_2 auf AND abzubilden.

Fügt man dem Netz nun noch ein weiteres Neuron net_3 hinzu, lassen sich die Outputs von net_1 und net_2 zusammenführen. So gelingt es dem FNN innerhalb weniger Iterationen die richtige Kombination für das XOR-Problem zu finden und dies über die Ausgabe darzustellen. Tabelle 3.1 zeigt die Lösung des XOR-Problems.

Tabelle 3.1 Wahrheitstabelle für die XOR-Lösung

X_1	X_2	net_1	net_2	net_3
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	0

Bei mehrschichtigen neuronalen Netzen (MLP) ist noch Folgendes zu beachten. Die Eingabewerte sind immer mit Gewichten versehen. Diese Gewichte werden beim Lernen des neuronalen Netzes angepasst und sorgen so für den Lernfortschritt im neuronalen Netz.

Das Perzeptron stellt dadurch eine spezielle Art von künstlichem Neuron dar. Es kombiniert seine Eingaben linear und wendet dann eine Schrittfunktion als Aktivierungsfunktion an. Eine Sonderstellung bei den Eingaben kommt dem sogenannten Bias-Wert zu, dargestellt als Input Layer X_0 , der als zusätzliche Eingabe mit konstantem Wert interpretiert werden kann, im Beispiel für das XOR-Problem ist die Vorbelegung des Input Layer X_0 immer 1 (siehe Abschnitt 1.5, „Das Neuron“).

Die Aktivierungsfunktion

Wie Sie schon in Kapitel 1 erfahren haben, lassen sich die am häufigsten verwendeten Aktivierungsfunktionen in drei Hauptklassen unterteilen:

- Binäre Treppenfunktionen: $\alpha = f(s) = \begin{cases} 1, & \text{falls } s \geq 0 \\ 0, & \text{falls } s < 0 \end{cases}$

- Semilineare Funktionen: $\alpha = f(s) = \begin{cases} 1, & \text{falls } s \geq 1 \\ s, & \text{falls } 0 < s < 1 \\ 0, & \text{falls } s \leq 0 \end{cases}$

- Sigmoide Funktionen: $\alpha = f(s) = \frac{1}{1+e^{-s}}$

Durch die Aktivierungsfunktion lässt sich, durch die Summe der Eingaben multipliziert mit den Gewichten, ein Aktivierungspotenzial simulieren. Hierbei hat sich herausgestellt, dass sich für Neuronen der versteckten Schicht häufig der hyperbolische Tangens als geeignete Aktivierungsfunktion ergeben hat. In der Ausgabeschicht wird gerne auf die Sigmoid-Funktion zurückgegriffen.

Aktivierungsfunktionen sind in künstlichen neuronalen Netzen notwendig, um Nichtlinearität zu erzeugen. Ohne Aktivierungsfunktion würden die Gewichte nur mit den Outputs des Vorgängerneurons multipliziert werden und somit nur lineare Operationen durchführen können. Des Weiteren ist es durch die Aktivierungsfunktion möglich, irrelevante Informationen, die keinen Einfluss auf das Endergebnis haben, auszusortieren.

Da MLPs sich vielseitig einsetzen lassen, kommen sie in zahlreichen Anwendungen zum Einsatz, hierzu zählen vor allem Mustererkennung, Klassifizierung, Prognosen, Diagnosen, Steuerungen sowie die Optimierung von Prozessen.

So, jetzt aber genug der Theorie. Am besten lässt sich der Feedforward-Mechanismus und die Verwendung der Aktivierungsfunktion an einer konkreten Aufgabe erläutern. Im folgenden Abschnitt programmieren Sie ein Modell für die Prognose einer Störung.

■ 3.5 Predictive Maintenance

Machine Learning und Deep Learning werden in der Automatisierungstechnik in den verschiedenen Produktionsphasen für die unterschiedlichsten Zwecke eingesetzt, dazu zählen zum Beispiel die Fehleranalyse, Ausfallvorhersage oder auch die Verbrauchsoptimierung von Rohstoffen und Werkzeugteilen.

Durch die Flexibilität von neuronalen Netzen können diese auch sehr gut zur Prozesskontrolle verwendet werden. So erfordert zum Beispiel die Automatisierung von Fertigungsprozessen eine fortlaufende Erfassung von Produktions- oder Maschinendaten.

Für diesen Vorgang nutzt man sogenannte Predictive Maintenance-Techniken (deutsch: vorausschauende Wartungstechniken), die es ermöglichen, den Zustand der genutzten Maschine online, in periodisch definierten Abständen oder fortlaufend zu bestimmen, um dann auf Basis der Zustandsdaten Vorhersagen zu treffen, wann zum Beispiel eine Wartung durchgeführt werden soll.



Predictive Maintenance, also die vorausschauende Wartung, stellt eine Instandhaltungsstrategie dar. Durch diese Technik ist es möglich, den Zustand von Geräten/Maschinen während des Betriebs zu überwachen, um festzustellen, wann sie gewartet werden müssen.

Durch die Vernetzung der Maschinen können gewonnene Zustandsdaten und Informationen vielfach per Sensor erfasst und so in Echtzeit zur Analyse oder als Wartungsplan genutzt werden. Daraus ergeben sich noch weitere Vorteile:

- Probleme bei der Maschine können in der Entstehung erkannt und behoben werden.
- Die Lebensdauer kann durch individuelle Wartungspläne verlängert werden.
- Wartungskosten können evtl. gesenkt werden.

Die Maschinesensoren messen während der Produktion verschiedene Parameter. Durch diese softwaregestützte Überwachung der Maschine können in den meisten Fällen folgende Daten erfasst werden:

- Druck
- Feuchtigkeit
- Geräuschlevel
- Durchlaufzeiten, Geschwindigkeit, Drehzahl
- Energieverbrauch
- Abgase
- Temperatur
- Spannung

Diese und viele weitere Daten werden dann über die Sensoren oder Messvorrichtung laufend kontrolliert und protokolliert.

Infolgedessen prognostiziert die Maschine aus den erfassten Daten mögliche Störungen und Ausfälle. Die hierdurch vorgenommene Auswertung zeigt rechtzeitig den zukünftigen Wartungsbedarf der Maschine und verhindert so einen ungeplanten Stillstand.

Daher erfolgt die Predictive Maintenance in den meisten Fällen in drei vorgegebenen Schritten:

- Digitale Erfassung von Daten
- Analyse der Daten
- Vorhersage/Prognose wahrscheinlicher Ergebnisse

Aus der Datenanalyse erhalten Sie Hinweise auf den Zustand der Maschine, können daraus die relevanten Merkmale extrahieren und einem Mustererkennungsprozess unterziehen.

Die größte Herausforderung in der Praxis stellt die Verarbeitung der riesigen Datenmengen dar, die nötig sind, um verlässliche Aussagen über den Zustand von Maschinen und Anlagen zu treffen. Stehen aber die relevanten Merkmale für den Mustererkennungsprozess zur Verfügung, können selbst komplexe Prozesse dynamisch analysiert werden.

So ist es möglich, komplexe Datenmuster aufzudecken, die in ähnlicher oder veränderter Art und Weise entstehen, bevor die Maschine ineffizient arbeitet oder sogar ganz ausfällt. Diese sich ändernden Muster können sich über Wochen oder Monate entwickeln. Daher sind eine entsprechende Bewertung und Analyse der Daten ungemein wichtig.

Predictive Maintenance bietet erhebliche Vorteile, da Wartungszyklen optimiert und Ausfallzeiten minimiert werden können. Stehen hinreichend viele und brauchbare Daten für die Extraktion von Merkmalsvektoren zur Verfügung, so lässt sich damit auch ein neuronales Netz als Multilayer Perceptron erstellen und trainieren. Wir beginnen jetzt mit der Datenanalyse unserer Beispieldaten für die vorausschauende Wartung.

■ 3.6 Maschinensimulation mit MLP

Das nachfolgende Beispiel soll zur Prognose der vorausschauenden Wartung mit einem Feedforward-Netz auf Basis eines Multilayer Perceptron dienen. Zur Prognose wird die Ausgabegeschwindigkeit einer Maschine mit drei Sensoren herangezogen bzw. simuliert. Die Trainingsaufgabe des Netzes besteht darin, aus n Werten in der Input-Schicht auf den nächsten Wert zu schließen. Das Training erfolgt anhand der bekannten Werte. So soll es möglich sein, die Wartungsintervalle in Kategorien einzuteilen und zu prognostizieren. Die Trainingsdaten über die Sensoren liegen als Geschwindigkeit in mm/s in einer entsprechenden Maschinendaten-Tabelle vor.

3.6.1 Datenmodellierung

Eine effiziente Vorverarbeitung der anfallenden Daten ist notwendig, um sie dem MLP eingeben zu können. Insbesondere sollte aufgrund der Aktivierungsfunktion eine Skalierung der unterschiedlichen Inputwerte erfolgen. Dies ist auch notwendig, um die qualitativen und quantitativen Einflussgrößen nach Möglichkeit im Gleichgewicht zu halten.

Da das neuronale Netz aufgrund von mathematischen Funktionen agiert, müssen alle Daten aus der Vorverarbeitung als Zahlen zur Verfügung gestellt werden. Der erste Schritt besteht also darin, alle vorhandenen nicht numerischen Daten in numerische Daten zu kodieren.



Normierung

In der Mathematik und in der Statistik bedeutet Normierung immer die Skalierung des Wertebereichs einer Variablen auf einen bestimmten Bereich, üblicherweise zwischen 0 und 1.

Des Weiteren sollten Sie numerische Daten so normieren, dass die Größen der Werte alle ungefähr im gleichen Bereich liegen. Bei der Vorverarbeitung von Zeitreihendaten werden neben der notwendigen Normierung vielfach noch weitere Techniken angewandt, um so das notwendige Differenzieren der Werte zu gewährleisten. In der Praxis haben sich folgende Normalisierungsverfahren als sehr nützlich erwiesen:

- **Min-Max-Normalisierung:** Hierunter versteht man die lineare Transformation der Daten in einem Bereich, zum Beispiel zwischen 0 und 1. Der Mindestwert wird auf 0 skaliert und der Höchstwert auf 1. Als allgemeine Formel ist gegeben:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- **Z-Wert-Normalisierung:** Hier werden die Daten basierend auf dem Mittelwert und der Standardabweichung skaliert. Das heißt, es werden durch das Teilen der Differenz aus Daten und Mittelwert durch die Standardabweichung die Daten entsprechend skaliert. Bei der Transformation der Daten geben Sie eine Zahl für den endgültigen Mittelwert und eine positive Zahl für die endgültige Standardabweichung an. Die Standardwerte sind entsprechend der Neuskalierung 0 bzw. 1.

- **Dezimalkalierung:** Hier wird das Skalieren der Daten durch das Verschieben des Dezimaltrennzeichens des Attributwertes erreicht.

Möchte man aus einer großen Datenmenge eine diskrete Teilmenge gewinnen, so können die durch die Konvertierung kontinuierlichen Werte in nominale Attribute oder Intervalle diskretisiert werden. Beim Machine Learning unterscheidet man hauptsächlich die zwei folgenden Gruppierungen:

- **Festbreitengruppierung:** Hierbei handelt es sich um das Aufteilen des Datenbereichs aller möglichen Werte eines Attributs in N Gruppen derselben Größe und dem Zuweisen einer Gruppennummer zu den Werten einer Gruppe.
- **Festhöchengruppierung:** Aufteilen des Datenbereichs aller möglichen Werte eines Attributs in N Gruppen mit derselben Anzahl von Instanzen und Zuweisen der Gruppennummer zu den Werten in einer Gruppe.

Tabelle 3.2 enthält die relevanten Maschinen-Trainingsdaten. Die Aufbereitung der Daten wurde als Dezimalkalierung gewählt. Durch das Teilen der Ausgangsgeschwindigkeit durch 1000 entsprechen die Daten dem Aufbau der Beispielgewichte für das Multilayer Perceptron.

Tabelle 3.2 Maschinen-Sensordaten als Trainingsdaten

Motorsensor	T1	T2	T3
Geschwindigkeit in mm/s	126	115	135
	127	116	136
	128	117	137
	129	118	138

3.6.1.1 Ziel des Feedforward-Netzes

Das Ziel des Beispiel-Multilayer-Perceptron ist die reine Klassifizierung mit Neuronen. Es wird ein FNN erzeugt, bei dem sich jedes Neuron mit allen Neuronen der davor liegenden Schicht verbindet. Allerdings besitzt das FNN als Erweiterung diesmal zwei Ausgabewerte Y_1 und Y_2 , um eine Mehrklassen-Klassifizierung zu simulieren.

Bisher haben Sie nur die einfache binäre Klassifikation mit Zwei-Klassenproblem $Y \in \{0, 1\}$ betrachtet. Daraus folgte immer:

- $Y = 1$: positive Klasse, zum Beispiel E-Mail ist Spam
- $Y = 0$: negative Klasse, zum Beispiel E-Mail kein Spam

Ziel der Klassifikation ist es, die Daten automatisch in vordefinierte Klassen einzuteilen. In der Praxis, vor allem bei Mustererkennung, Textanalyse und Prognosen ist oftmals eine Unterscheidung in mehr als zwei Klassen erforderlich. In diesem Fall spricht man von einem Mehrklassen-Klassifikationsproblem.

3.6.1.2 Mehrklassen-Klassifikation

Auch bei der Wartung oder Steuerung von Maschinen und Anlagen reichen oft zwei Klassen für eine Einteilung der Daten nicht aus. Es gibt verschiedene Ansätze zur Erweiterung der binären Klassifikation. Die drei wichtigsten sind:

- Mehrklassen-Feedforward-Netze. Die Objekte werden mithilfe einer einzigen Entscheidungsfunktion klassifiziert.
- Einer gegen den Rest (bzw. alle). Hierbei wird für jede Klasse eine binäre Support Vector Maschine (SVM) verwendet. Die Elemente dieser Klasse bilden die eine, die Elemente aller anderen Klassen bilden die zweite Klasse (siehe Abschnitt 2.6.2, „Support Vector Machine“).
- Paarweise Klassifikation. Bei diesem Ansatz wird für jedes Paar von Klassen eine Support Vector Machine genutzt, die jeweils die Mitglieder der zwei Klassen voneinander trennt.

Das heißt, mit der Mehrklassen-Klassifikation können Sie Prognosen für mehrere Klassen generieren, die aus mehr als zwei Ergebnissen bestehen, wie zum Beispiel:

- Ist das Werkzeug ein Bohrer, eine Fräse oder eine Säge?
- Liegt eine Störung oder eine Wartung vor?
- Gehört der Text in die Sparte Politik, Sport, Kultur oder Reise?
- Ist das Produkt ein Buch, ein Film oder ein Kleidungsstück?
- Ist die Farbe rot, grün oder blau?

Das Mehrklassen-Modell im Machine Learning besteht also darin, Werte vorherzusagen, die zu einem begrenzten und vordefinierten Satz an zulässigen Werten gehören.

So könnte man bei einer Maschine die Benutzung der einzelnen Werkzeuge mithilfe einer Konfusionsmatrix auf Mehrklassen-Probleme prüfen. Bild 3.6 zeigt die Konfusionsmatrix für ein Mehrklassen-Klassifizierungsmodell.

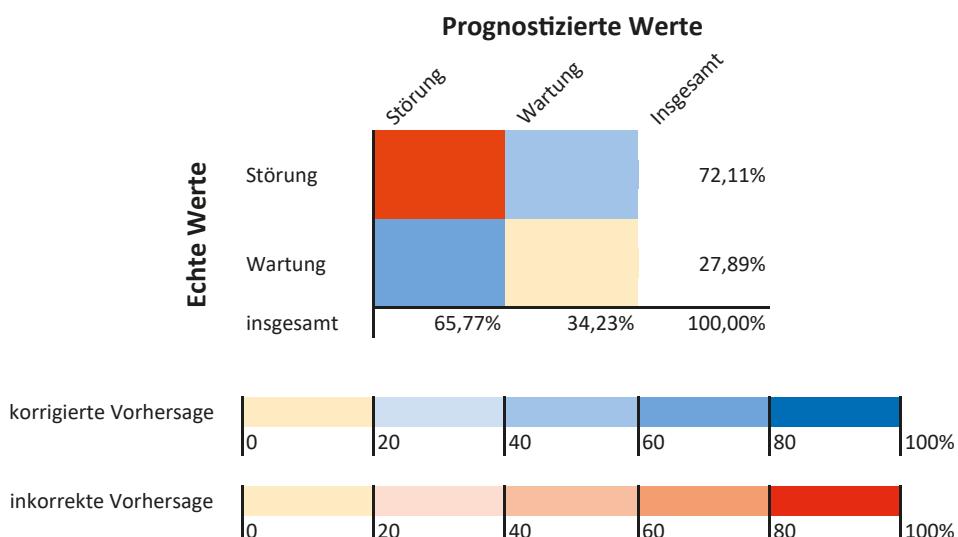


Bild 3.6 Konfusionsmatrix für die Prognose

Eine Konfusionsmatrix (Wahrheitsmatrix) dient zur Beurteilung eines Klassifikators. Die Spalten der Matrix repräsentieren die Ausgabe des Klassifikators und die Zeilen stellen die tatsächliche Klasse dar. Die diagonalen Elemente der Matrix geben die Anzahl der korrekten Klassifikation an. Die nicht-diagonalen Elemente zeigen an, wie viele Probleme falsch klassifiziert wurden.

Die folgenden Informationen fließen in eine Konfusionsmatrix:

- Anzahl richtiger und falscher Voraussagen für jede Klasse: Jede Zeile in der Konfusionsmatrix entspricht den Metriken für eine der tatsächlichen Klassen.
- Echte Klassenfrequenzen in den Auswertungsdaten: In der letzten Spalte wird angezeigt, dass im Auswertungsdatensatz 72,11 % der Beobachtungen in den Auswertungsdaten unter Störung fallen.
- Vorhergesagte Klassenfrequenzen für die Auswertungsdaten: In der letzten Zeile wird die Häufigkeit jeder Klasse in den Voraussagen angezeigt.

Über ein MLP können Sie das Mehrklassen-Klassifizierungsproblem lösen. Anstelle von nur einem Neuron in der Ausgabeschicht ist es möglich, mit n binären Neuronen die Klassifizierung von Mehrklassen durchzuführen. In der Praxis hat sich für das Lösen des Mehrklassenproblems für die letzte Schicht im neuronalen Netz die logistische Regression per Softmax-Funktion durchgesetzt. Dieser Lösungsweg wird auch in dem Beispiel-FNN (Abschnitt 3.6.7) eingesetzt.

3.6.2 Entwurf

Aufgrund der zur Verfügung stehenden Daten aus Tabelle 3.2 wird ein Feedforward-Netz aufgebaut. Das Beispiel soll zeigen, wie schnell und effektiv mit ein wenig Programmiererfahrung ein einfaches mehrschichtiges neuronales Netz mit drei Eingängen, stellvertretend für die drei Sensoren, vier verborgenen Neuronen und zwei Ausgangswerten für eine Mehrklassen-Klassifikation erstellt werden kann.

Einfach heißt in diesem Fall ein gutes Gleichgewicht zwischen Funktionalität und Einfachheit. Die erstellte WPF App soll zur Kontrolle Zwischenwerte der Hidden Layer sowie die Ausgabewerte Y_1 und Y_2 zur Bewertung in einer *ListBox* ausgeben. Hierfür benötigen Sie in dem Projekt nur einige Steuerelemente wie die *ListBox* zum Anzeigen, einen Bestätigungs-Schalter (Button) für das Starten der Berechnung und einen Schalter zum Beenden der WPF-Applikation.

Mit den Bedienelementen von Visual Studio lässt sich eine WPF-App sehr einfach gestalten. Sie können die Oberfläche mithilfe des *XAML-Editors* erstellen, indem Sie in der Toolbox das entsprechende Steuerelement auswählen und es auf der Entwurfsoberfläche der Datei *MainWindows.xaml* ablegen. Bild 3.7 zeigt die erstellte Oberfläche mit den benötigten Steuer-elementen in der XAML-Editor-Ansicht von Visual Studio.

Im Laufe der Programmierung ändern Sie einige Steuerelementeigenschaften und implementieren den Code, um auf entsprechende Ereignisse zu reagieren.

Für die Programmierung der Beispiele wird der Code so einfach wie möglich gehalten. Daher wird bei der Implementierung wie folgt vorgegangen:

- Methoden und Funktionen werden genutzt, um Komplexität durch Abstraktion zu reduzieren.
- Methoden und Funktionen sollten nur eine einzige Aufgabe erledigen,
- Kleine übersichtliche Code-Segmente verwenden und darstellen.
- Die Namen der Variablen sollten den Zweck anzeigen.

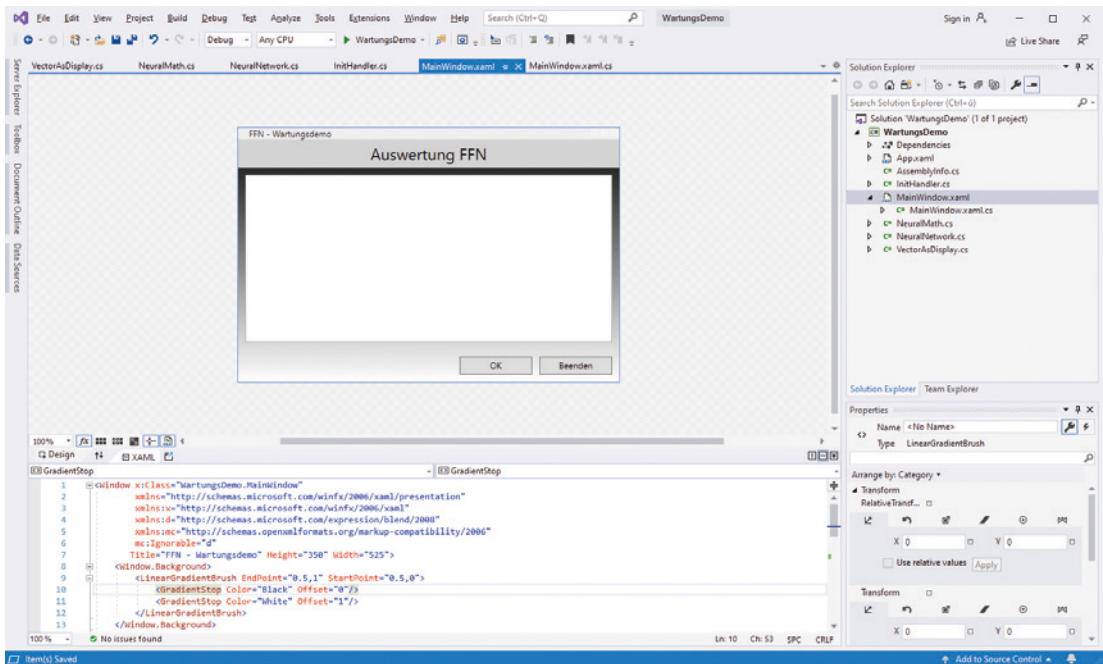


Bild 3.7 Entwurfsoberfläche im XAML-Editor

Mithilfe der in den nachfolgenden Anwendungsbeispielen verwendeten Methoden soll der Code nach Möglichkeit vereinfacht, übersichtlich und nachvollziehbar gestaltet werden.

3.6.3 Projekt anlegen

Die benötigte Projektvorlage für die WPF-Applikation finden Sie auf der Startseite von Visual Studio 2019 unter *Create a new project*. Mit der Auswahl von *C#* und der Plattform *Windows* steht Ihnen die Projektvorlage *WPF App (.NET Core)* bzw. alternativ *WPF App (.NET FRAMEWORK)* in der Liste zur Verfügung.

Wählen Sie die Projektvorlage aus indem Sie diese markieren und auf *Next* klicken. Vergeben Sie als Vorbereitung für unser Beispiel den Namen (*Project name*) *WartungsDemo*. Legen Sie noch den Speicherort für das Projekt fest und schließen Sie das Dialogfeld über *Create* ab. Visual Studio erstellt anschließend automatisch aus der Vorlage eine bereits lauffähige WPF-Applikation mit einem leeren *MainWindow*.

Wechseln Sie jetzt in den XAML-Editor des *MainWindow* und erweitern Sie die *XAML*-Datei wie in Listing 3.1 vorgeschlagen. Alternativ können Sie aber auch die GUI nach eigenen Vorstellungen erstellen bzw. anpassen.

Listing 3.1 Anpassung für das GUI-Layout

```

<Window x:Class="WartungsDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="FFN - Wartungsdemo" Height="350" Width="525">
<Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="Black" Offset="0"/>
        <GradientStop Color="White" Offset="1"/>
    </LinearGradientBrush>
</Window.Background>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <StackPanel Background="LightGray">
        <Label FontSize="22" HorizontalAlignment="Center">Auswertung FNN</Label>
    </StackPanel>
    <Grid Grid.Row="1">
        <ListBox Name="ListVectorValues" Margin="10"></ListBox>
    </Grid>
    <StackPanel Grid.Row="2" Orientation="Horizontal" HorizontalAlignment="Right">
        <Button Name="ButtonOk" Height="25" Width="100" Margin="10"
            Click="ButtonOk_Click">OK</Button>
        <Button Height="25" Width="100" Margin="0 10 10 10"
            Click="Button_Cancel">Beenden</Button>
    </StackPanel>
</Grid>
</Window>

```

Listing 3.1 erweitert die *XAML*-Datei um einen *LinearGradientBrush*-Farbverlauf für den Hintergrund. Das Tag *<Grid.RowDefinitions>* teilt das *WPF Window* in das gewünschte Layout auf, damit die *Listbox* und die beiden benötigten *Buttons* an der richtigen Stelle platziert werden können. Die *Listbox* verfügt in der XAML-Datei über kein spezielles *ItemTemplate* oder *DataTemplate*. Die Zuweisung der Daten erfolgt später im Code-Behind der *MainWindows.xaml.cs*-Datei.

Fügen Sie bei der Erweiterung im XAML-Editor auch schon über das Tag *Click* die benötigte Event-Handler-Methode über *New Event Handler* für den *Ok*- und den *Beenden*-Button hinzu. Über die Taste **F5** oder das grüne Startsymbol lässt sich die Applikation jetzt auch schon ausführen. Somit ist das Grundgerüst für die Anzeige der Werte für unser neuronales Netz schon erstellt und lauffähig. Jetzt können Sie das neuronale Netz mit Leben füllen. Legen Sie hierfür über den Solution Explorer in Visual Studio vier weitere Klassen in Ihrem Projekt an. Bezeichnen Sie diese wie folgt:

- InitHandler.cs
- NeuralMath.cs
- NeuralNetwork.cs
- VectorAsDisplay.cs

Sind die Klassen erstellt, so können Sie mit der Code-Umsetzung beginnen. Bild 3.8 zeigt die neue Projektstruktur der WPF-Applikation mit den hinzugefügten Klassen.

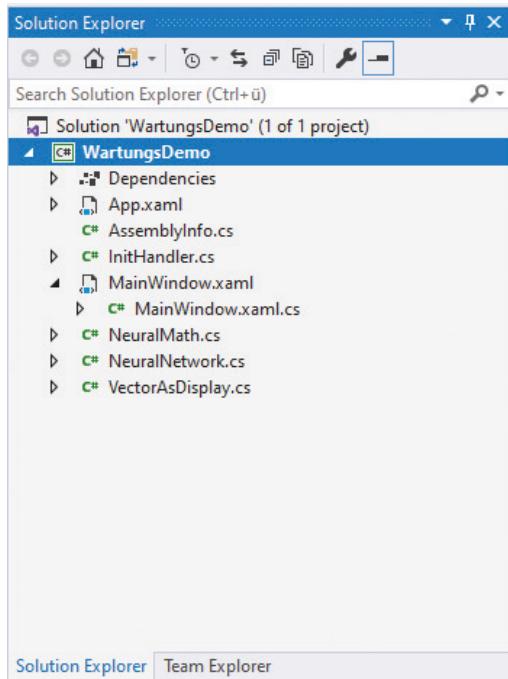


Bild 3.8

Die vollständige Projektstruktur

3.6.4 Erfassung und Berechnung der Daten

Nachdem Sie nun mit der grundsätzlichen Funktionsweise mehrschichtiger neuronaler Netze vertraut sind, sehen wir uns den Feedforward-Mechanismus und damit den eigentlichen Datenfluss von der Eingabe zur Ausgabe anhand der Daten aus Tabelle 3.2 mit den konkreten angepassten Zahlen als Gewichte an.

Als Vorlage für die Implementierung in C# dient Bild 3.9. T_1 , T_2 und T_3 bezeichnen hier die drei Input Layer, des Weiteren finden Sie in der Skizze vier Hidden- und zwei Output Layer die als Y_1 und Y_2 gekennzeichnet sind. W_{IH} sind die Gewichte ($W_{_}$) vom Input Layer (I) zum Hidden Layer (H). Diese Gewichte entsprechen den angepassten Daten aus Abschnitt 3.6.2. W_{HO} steht für die Gewichte vom Hidden Layer zum Output Layer und diese sind für das Beispiel rein fiktiv. Das heißt, jede Linie, die einen Knoten mit einem anderen verbindet, stellt eine Gewichtskonstante dar.

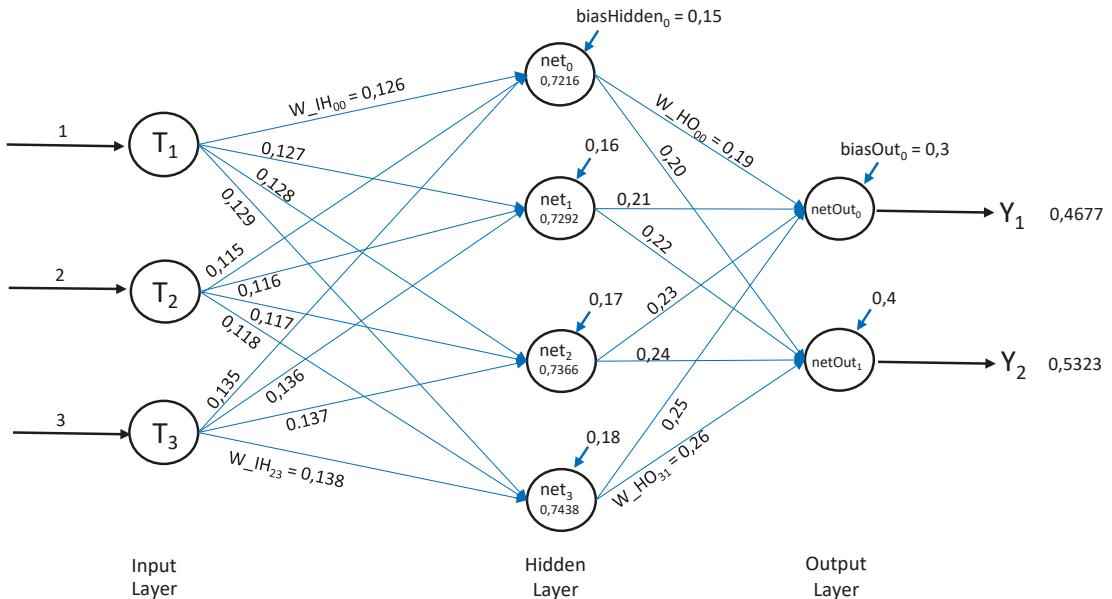


Bild 3.9 Aufbau eines Feedforward-Netzes

Über die Angabe *biasHidden* und *biasOut* werden die Bias-Werte für die Hidden- und Output-Neuronen festgelegt. Der Pfeil zeigt die entsprechenden Werte an. Daher bezeichnet die Angabe *biasHidden₀* = 0,15 den Bias-Wert für das verborgene Neuron net_0 . Die tiefgestellten Zahlen wie zum Beispiel W_{IH00} bezeichnen die Indizes für die späteren Rechenoperationen.

Die Darstellung der Pfeilrichtung von Input Layer nach Hidden Layer besagt, dass ein Gewicht für eine Verbindung gilt, die von einem Neuron zu einem Neuron führt. Somit ist der Feedforward-Mechanismus auch wirklich abgebildet.

Das Gewicht W_{IH23} bezeichnet das Gewicht der Verbindung von Neuron T_3 nach Neuron net_4 im Hidden Layer.

Für die Umsetzung des aufgeführten Modells muss für den Feedforward-Mechanismus im ersten Schritt die Berechnung der Werte vorgenommen werden, die auf die verborgenen Neuronen der verdeckten Sicht (Hidden Layer) treffen. Dazu betrachten Sie das erste Neuron net_0 in der verdeckten Schicht. Als Zwischenergebnis (Produkt) der gewichteten Eingabewerte ergibt sich:

$$\begin{aligned} net''_0 &= T_1 \cdot W_{IH00} + T_2 \cdot W_{IH10} + T_3 \cdot W_{IH20} \\ net''_0 &= (1,0)(0,126) + (2,0)(0,115) + (3,0)(0,135) = 0,761 \end{aligned}$$

Addieren Sie jetzt noch den Bias-Wert hinzu, erhalten Sie:

$$\begin{aligned} net'_0 &= net''_0 + biasHidden_0 \\ net'_0 &= 0,761 + 0,15 = 0,911 \end{aligned}$$

Der Wert von net' dient als Summe vor der Aktivierungsfunktion. Nach Anwendung des hyperbolischen Tangens (\tanh) bekommen Sie den endgültigen Wert für das Neuron aus dem Hidden Layer:

$$\text{net}_0 = \tanh(\text{net}'_0)$$

Somit ist der Ausgangswert von net_0 :

$$\text{output_net}_0 = \tanh(0,911) = 0,7216$$

Die drei weiteren Neuronen in den Hidden Layern werden auf die gleiche Art und Weise berechnet.

Das fast gleiche Berechnungsschema lässt sich auch auf die Neuronen der Ausgangsschicht anwenden. Allein anstelle der Eingabewerte von T_1 bis T_3 werden hier die Zwischenergebnisse der versteckten Schicht ($\text{net}_0 \dots \text{net}_3$) verwendet, und der hyperbolische Tangens wird durch die Softmax-Funktion ersetzt. Diese wird bei der Implementierung noch genauer erklärt (siehe Abschnitt 3.6.7). Das heißt, der Wert des Ausgabeknotens Y_1 ergibt sich dann nach der Softmax-Aktivierungsfunktion.

Es ergibt sich die vorläufige Eingabesumme vor der Aktivierung für den Ausgabeknoten netOut_0 zusammen mit dem $biasOut_0$ wie folgt:

$$\begin{aligned} oSums_0 &= (0,7216)(0,19) + (0,7292)(0,21) + (0,7366)(0,23) + (0,7438)(0,25) + 0,3 \\ &= 0,9456 \end{aligned}$$

Die Werte 0,7216, 0,7292, 0,7366 und 0,7438 stellen jeweils die Ausgangswerte der Neuronen net_0 bis net_3 dar.

3.6.5 Bias-Neuron

Das Beispiel zeigt nochmals, dass die Gewichte die Intensität des Informationsflusses entlang einer Verbindung von einem Neuron zu einem anderen beschreiben. Jedes Neuron vergibt dazu ein Gewicht für die durchfließende Information und gibt diese dann nach der Addition des Bias-Wertes an die Neuronen der nächsten Schicht weiter.

Der Bias hat die Aufgabe, den Schwellenwert der nachfolgenden Aktivierungsfunktion festzulegen. Somit stellt er die Neuronen-spezifische Verzerrung dar und wird üblicherweise zu Beginn des Trainings im Wertebereich zwischen -1 und 1 initialisiert. Sie müssen als Entwickler sehr sorgfältig darauf achten, wie mit den Verzerrungswerten umgegangen wird, da das Ergebnis der Gewichte und des Bias-Wertes durch die Aktivierungsfunktion geleitet wird, bevor es an die Neuronen der nächsten Schicht weitergeleitet wird.

In der Forschung und in vielen Lehrbüchern behandelt man den Bias-Wert und damit die Verzerrung als rein spezielles Gewicht, die in diesem Fall immer mit einem Dummy-Eingabewert von 1,0 verbunden wird. Dieser Bias-Wert wird dann als Bias-Neuron bezeichnet.

Bei der Entwicklung von neuronalen Netzen sollte man den Bias-Wert als echte Verzerrung für die Aktivierungsfunktion nutzen, da die Umsetzung und Verwendung als Gewicht konzeptionell etwas umständlich ist.

3.6.6 Die Programmierung

Das Modell aus Bild 3.9 zeigt den benötigten Bauplan für die C#-Implementierung des Feed-forward-Algorithmus. Als Erstes wird die *MainWindow.xaml.cs* Datei beschrieben. Sie bildet gewissermaßen das Rahmenprogramm, das benötigt wird, um den Algorithmus auszuführen.

Listing 3.2 Code-Behind der Klasse MainWindow

```
using System.Collections.Generic;
using System.Windows;

namespace WartungsDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        InitHandler initHandler = new InitHandler();
        List<string> vectorValues = new List<string>();

        public MainWindow()
        {
            InitializeComponent();
        }

        private void Button_Cancel(object sender, RoutedEventArgs e)
        {
            Application.Current.Shutdown();
        }

        private void ButtonOk_Click(object sender, RoutedEventArgs e)
        {
            ListVectorValues.ItemsSource = null;
            vectorValues.Clear();

            vectorValues = initHandler.setInitNeuralNetwork();
            ListVectorValues.ItemsSource = vectorValues;
            ButtonOk.IsEnabled = false;
        }
    }
}
```

Die *MainWindow.xaml.cs* enthält die Instanz der *InitHandler*-Klasse, eine Liste vom Typ *string* zur Aufnahme der Werte und Texte für die *Listbox* sowie die Methode *initHandler.setInitNeuralNetwork* für das Initialisieren des Algorithmus. Über den *EventHandler ButtonOk_Click* lässt sich das FNN dann programmdefiniert ausführen.

Der Konstruktor der Klasse InitHandler

Die Klasse *InitHandler* verfügt über einen gleichlautenden Konstruktor. Neben der Initialisierung der Gewichte und Bias-Werte als eindimensionales Array vom Typ *double* wird auch über die Hauptklasse *NeuralNetwork* die Vorgabe der Netz-Architektur mit drei Input Layern, vier Hidden Layern und zwei Output Layern vorgenommen.

Listing 3.3 Die Klasse InitHandler

```
using System.Collections.Generic;

namespace WartungsDemo
{
    public class InitHandler
    {
        NeuralNetwork neuralNetwork = null;
        private VectorAsDisplay vectorDisplay = new VectorAsDisplay();
        List<string> vectorShowValues = new List<string>();

        public List<string> setInitNeuralNetwork()
        {
            vectorShowValues.Clear();
            neuralNetwork = new NeuralNetwork(3, 4, 2);

            double[] weights = new double[] { 0.126, 0.127, 0.128, 0.129,
                0.115, 0.116, 0.117, 0.118,
                0.135, 0.136, 0.137, 0.138,
                0.15, 0.16, 0.17, 0.18,
                0.19, 0.20, 0.21, 0.22,
                0.23, 0.24, 0.25, 0.26,
                0.3, 0.4 };

            neuralNetwork.SetWeights(weights);

            double[] xValues = new double[] { 1.0, 2.0, 3.0 };

            double[] yValues = neuralNetwork.Feedforward(xValues);

            vectorShowValues = neuralNetwork.GetVectorValueList();

            vectorShowValues.Add("Ausgabewerte Y1 und Y2");
            List<string> rightOutput = vectorDisplay.ShowVector(yValues, 4);
            foreach(string element in rightOutput)
            {
                vectorShowValues.Add(element);
            }

            return vectorShowValues;
        }
    }
}
```

Die Variable *xValues* definiert die Werte der Eingangsneuronen als 1.0, 2.0, 3.0. Die Zuweisung der Ausgangsneuronen Y_1 und Y_2 erfolgt über die Methode *ComputeOutputs(xValues)* aus der Klasse *NeuralNetwork* und wird in der Variablen *yValues* gespeichert.

In der Liste *vectorShowValues* werden dann alle Werte für die Elemente der *Listbox* in der GUI-Ansicht zusammengefasst und die Liste an den Aufrufer über *return* zurückgegeben.

Gestaltung der Klasse NeuralMath

Die Klasse *NeuralMath* stellt alle benötigten Methoden für die Erstellung der Rechenmatrix, die Aktivierungsfunktion *HyperTan* für die Hidden Layer und die Aktivierungsfunktion *Softmax* für die Output Layer zur Verfügung.

Listing 3.4 Die Klasse NeuralMath

```
using System;

namespace WartungsDemo
{
    public static class NeuralMath
    {
        public static double[][] MakeMatrix(int rows, int cols)
        {
            double[][] result = new double[rows][];

            for (int i = 0; i < rows; ++i)
            {
                result[i] = new double[cols];
            }

            return result;
        }

        public static double HyperTan(double v)
        {
            return Math.Tanh(v);
        }

        public static double[] Softmax(double[] oSums)
        {
            double max = oSums[0];

            for (int i = 0; i < oSums.Length; ++i)
            {
                if (oSums[i] > max) max = oSums[i];
            }

            double scale = 0.0;

            for (int i = 0; i < oSums.Length; ++i)
            {
                scale += Math.Exp(oSums[i] - max);
            }
        }
    }
}
```

```

        double[] result = new double[oSums.Length];

        for (int i = 0; i < oSums.Length; ++i)
        {
            result[i] = Math.Exp(oSums[i] - max) / scale;
        }

        return result;
    }

    public static double[] SoftmaxNaive(double[] oSums)
    {
        double denom = 0.0;

        for (int i = 0; i < oSums.Length; ++i)
        {
            denom += Math.Exp(oSums[i]);
        }

        double[] result = new double[oSums.Length];

        for (int i = 0; i < oSums.Length; ++i)
        {
            result[i] = Math.Exp(oSums[i]) / denom;
        }

        return result;
    }
}
}

```

Die Funktionsweise der einzelnen Methoden wird im weiteren Verlauf in Abschnitt 3.6.7 beschrieben.

Die Klasse NeuralNetwork

Die Klasse *NeuralNetwork* bildet das Hauptprogramm und führt den Feedforward-Algorithmus aus. Auch diese Klasse verfügt über einen gleichlautenden Konstruktor.

Listing 3.5 Die Klasse NeuralNetwork

```

using System;
using System.Collections.Generic;

namespace WartungsDemo
{
    public class NeuralNetwork
    {
        private int nInput;
        private int nHidden;
        private int nOutput;
    }
}

```

```

private double[] inputs;
private double[][] W_IH;

private double[] biasHidden;
private double[] biasOut;

private double[] outputs_h;
private double[][] W_HO;

private double[] outputs;

private List<string> vectorDisplayValue = null;
private VectorAsDisplay vectorDisplay = new VectorAsDisplay();
public NeuralNetwork(int nInput, int nHidden, int nOutput)
{
    vectorDisplayValue = new List<string>();
    vectorDisplayValue.Add("Starte Initialisierung des neuronalen Netzes...");

    this.nInput = nInput;
    this.nHidden = nHidden;
    this.nOutput = nOutput;

    inputs = new double[nInput];
    biasHidden = new double[nHidden];
    outputs_h = new double[nHidden];

    biasOut = new double[nOutput];
    outputs = new double[nOutput];

    W_IH = NeuralMath.MakeMatrix(nInput, nHidden);
    W_HO = NeuralMath.MakeMatrix(nHidden, nOutput);
}

public void SetWeights(double[] weights)
{
    int numWeights = (nInput * nHidden) + nHidden +
        (nHidden * nOutput) + nOutput;

    if (weights.Length != numWeights)
    {
        throw new Exception("Ungültiges Gewicht");
    }

    int k = 0;

    for (int i = 0; i < nInput; ++i)
    {
        for (int j = 0; j < nHidden; ++j)
        {
            W_IH[i][j] = weights[k++];
        }
    }

    for (int i = 0; i < nHidden; ++i)

```

```

    {
        biasHidden[i] = weights[k++];
    }

    for (int i = 0; i < nHidden; ++i)
    {
        for (int j = 0; j < nOutput; ++j)
        {
            W_HO[i][j] = weights[k++];
        }
    }

    for (int i = 0; i < nOutput; ++i)
    {
        biasOut[i] = weights[k++];
    }
}

public double[] Feedforward(double[] xValues)
{
    double[] hSums = new double[nHidden];

    double[] oSums = new double[nOutput];

    for (int i = 0; i < xValues.Length; ++i)
    {
        inputs[i] = xValues[i];
    }

    for (int j = 0; j < nHidden; ++j)
    {
        for (int i = 0; i < nInput; ++i)
        {
            hSums[j] += inputs[i] * W_IH[i][j];
        }
    }

    for (int i = 0; i < nHidden; ++i)
    {
        hSums[i] += biasHidden[i];
    }

    for (int i = 0; i < nHidden; ++i)
    {
        outputs_h[i] = NeuralMath.HyperTan(hSums[i]);
    }

    vectorDisplayValue.Add("Werte im Hidden-Layer:");
    List<string> hOutputsList = vectorDisplay.ShowVector(outputs_h, 4);
    foreach (string element in hOutputsList)
    {
        vectorDisplayValue.Add(element);
    }
}

```

```

        for (int j = 0; j < nOutput; ++j)
        {
            for (int i = 0; i < nHidden; ++i)
            {
                oSums[j] += outputs_h[i] * W_HO[i][j];
            }
        }

        for (int i = 0; i < nOutput; ++i)
        {
            oSums[i] += biasOut[i];
        }

        double[] softOut = NeuralMath.Softmax(oSums);

        for (int i = 0; i < outputs.Length; ++i)
        {
            outputs[i] = softOut[i];
        }

        double[] result = new double[nOutput];

        for (int i = 0; i < outputs.Length; ++i)
        {
            result[i] = outputs[i];
        }

        return result;
    }

    public List<string> GetVectorValueList()
    {
        if(vectorDisplayValue.Count > 0)
        {
            return vectorDisplayValue;
        }

        return null;
    }
}

```

Als Erstes speichern die Member-Felder *nInput*, *nHidden* und *nOutput* die Anzahl der Neuronen im Input Layer, im Hidden Layer und im Output Layer. Das Array *inputs* speichert die numerischen Eingaben in das FNN.

Das Array *biasHidden* enthält die Bias-Werte für die verborgenen Neuronen und *outputs_h* speichert die Ausgänge der verborgenen Knoten nach der Summierung der Produkte aus Gewichten und Eingaben, der Addition des Bias-Wertes und der Anwendung einer Aktivierungsfunktion während der Berechnung der Ausgangswerte. Das Array *biasOut* enthält die Bias-Werte der Ausgangsknoten und das Array *outputs* enthält die endgültigen berechneten Gesamtausgabewerte des neuronalen Netzes.

Dann weisen Sie dem Matrixelement W_{IH} über die Methode `MakeMatrix(nInput, nHidden)` aus der Klasse `NeuralMath` die Gewichte von den Eingangsknoten zu den verborgenen Knoten zu. Es stellt somit die Input-to-hidden-Gewichtsmatrix dar. Hierbei entspricht der Zeilenindex dem Index eines Eingangsknotens und der Spaltenindex dem Index eines verborgenen Knotens.

In der Methode `MakeMatrix` nutzen Sie die Möglichkeit der verzweigten Arrays in C#. Ein verzweigtes Array ist ein Array, dessen Elemente wiederum Arrays sind. Die Elemente eines verzweigten Arrays können eine unterschiedliche Dimension und Größe besitzen. Es ist auch möglich, verzweigte und mehrdimensionale Arrays zu mischen. Dies erleichtert in C# den effektiven Umgang mit Matrixelementen ungemein.

Die Methode `SetWeights`

Die Methode `SetWeights` füllt die Gewichtsmatrizen W_{IH} und W_{HO} . Über `biasHidden` und `biasOut` werden auch diese Arrays mit Werten gefüllt. Die Methode setzt allerdings voraus, dass die Gewichte als Parameter in der richtigen Reihenfolge übergeben werden.

Die Übergabestruktur sieht hier genauso aus wie im Modell in Bild 3.9. Als Erstes kommen alle Input-to-hidden-Gewichte, dann folgen die verborgenen Bias-Werte, gefolgt von den Hidden-to-output-Gewichten und ganz zum Schluss folgt dann noch der Output-Bias. Die Werte müssen in dem Beispiel auch von links nach rechts und von oben nach unten geordnet sein.

Der Feedforward-Algoritmus

Die Methode `Feedforward` implementiert den benötigten Feedforward-Algoritmus und stellt somit das Herz der Klasse dar.

Als Erstes werden die Summe der Produkte aus Eingaben und Gewichten sowie die Bias-Werte den Arrays `hSums` und `oSums` vor der eigentlichen Aktivierungsfunktion zugewiesen. Als Nächstes werden die Eingabewerte aus dem Array-Parameter `xValues` in das Array der Eingabeelemente `inputs[i]` kopiert. Nach dem Kopieren der Werte akkumuliert der Feedforward-Mechanismus die Summen der Produkte aus Input-to-hidden-Gewichten und den Inputs. Dann werden die Bias-Werte zu den kumulierten Summen der verborgenen Knoten hinzugefügt.

Jetzt wird durch den Algoritmus die hyperbolische Tangenten-Aktivierungsfunktion auf jede Summe der einzelnen Neuronen angewendet, um so die Ausgabewerte für die verborgenen Knoten zu errechnen. Die errechneten Werte der Neuronen im Hidden Layer werden dann für die Ausgabe in der `Listbox` in einer `List<T>` vom Typ `String` gespeichert, um diese mit den Werten aus dem Modell zu vergleichen. Sie können hier aber auch die Ausgabe für die `Listbox` nach Wunsch selbst erweitern.

Nach der Übergabe der Werte im Hidden Layer an die Liste von Strings werden die Voraktivierungssummen für die Knoten der Ausgabeschicht errechnet. Danach wird die Softmax-Aktivierung auf die Summe angewendet, und die Endergebnisse werden in dem Array `outputs[i]` gespeichert. Dann werden die Ergebnisse in das `Result`-Array kopiert und über `return` zurückgegeben. Fertig ist der Feedforward-Algoritmus.

Die Klasse `VectorAsDisplay`

Die Klasse `VectorAsDisplay` enthält nur eine kleine Hilfsmethode zur Aufbereitung und Formatierung der Werte und Angaben für die `Listbox` in der GUI.

Listing 3.6 Die Klasse VectorAsDisplay

```
using System.Collections.Generic;

namespace WartungsDemo
{
    public class VectorAsDisplay
    {
        List<string> vectorValue = new List<string>();
        public List<string> ShowVector(double[] vector, int decimals)
        {

            for (int i = 0; i < vector.Length; ++i)
            {
                vectorValue.Add(vector[i].ToString("F" + decimals).PadLeft
                               (decimals + 4));
            }

            return vectorValue;
        }
    }
}
```

Ergebnisanzeige

Die Rückgabe der Ausgaben des neuronalen Netzes, und damit auch das Starten des Algorithmus, ermöglicht der Aufruf der Methode *Feedforward(xValues)*. Über die Methode *GetVectorListe* wird die Liste der String-Objekte für die *Listbox* in der Oberfläche abgerufen und die Werte können entsprechend dargestellt werden. Bild 3.10 zeigt die Auswertung und das Ergebnis für die Ausgaben Y_1 und Y_2 , die dem Modell aus Bild 3.9 mit allen angegebenen Werten entspricht.

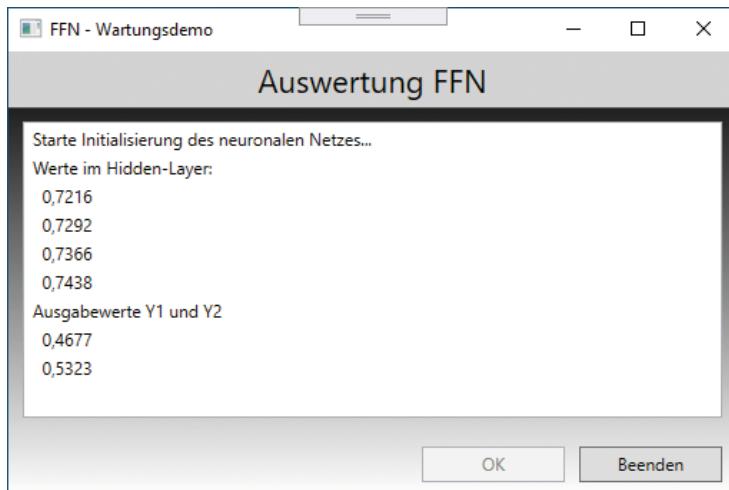


Bild 3.10 Das Ergebnis des FNN

3.6.7 Aktivierungsfunktionen implementieren

Die Aktivierungsfunktion ermöglicht es dem neuronalen Netz, Grenzwerte seiner Entscheidungsfunktion zu erlernen. Hierfür wird bei komplexen Entscheidungsgrenzen eine nichtlineare Aktivierungsfunktion auf Hidden- und Output-Layer-Neuronen angewendet. Die Aktivierungsfunktionen können grundsätzlich in zwei Typen unterteilt werden:

- Lineare Aktivierungsfunktionen
- Nicht-Lineare Aktivierungsfunktionen

Die Aktivierungsfunktion sorgt für das nichtlineare funktionale Mapping zwischen den Eingangsdaten und den abhängigen Ergebnissen, um das Lernen zu ermöglichen. Ohne die Anwendung einer Aktivierungsfunktion wäre das Ausgangssignal immer nur eine einfache lineare Funktion. Ein neuronales Netz ohne Aktivierungsfunktion ist somit immer nur ein einfaches lineares Regressionsmodell und wäre damit auch nicht in der Lage, Datenarten wie Bilder, Videos, Audio und Sprachen zu verarbeiten und zu modellieren.

Das Feedforward-Netz in unserem Beispiel verwendet die hyperbolische Tangente für die Aktivierung der verborgenen Layer-Knoten und die Softmax-Funktion für die Aktivierung der Output Layer. In der Praxis haben sich für die Aktivierung im Hidden Layer die hyperbolische Tangente oder auch die Rectified Linear Units (*ReLU*)-Funktion durchgesetzt. Da C# schon eine eingebaute .NET *Tanh()*-Methode besitzt, ist die Implementierung im Beispiel einfach über

```
return Math.Tanh(v);
```

realisiert. Die *Tanh*-Funktion akzeptiert jeden numerischen Wert von negativ unendlich bis positiv unendlich und gibt einen Wert zwischen -1.0 und +1.0 zurück.

Das heißt, bei den Hidden Layern werden die Vektoren der Eingangsschicht mit den Neuronen im Hidden Layer verknüpft und anschließend mit der *Tanh*-Methode aktiviert.

Die *Softmax*-Aktivierungsfunktion kommt in der Praxis eigentlich nur im Output Layer zum Einsatz. Softmax wird immer dann verwendet, wenn es sich bei der Klassifizierung um Wahrscheinlichkeiten handelt, so dass ein Objekt einer bestimmten Klasse zugewiesen wird. Die Summe aller Output Neuronen ist immer 1. Wenn also Ausgabedaten, wie die Vorsagewerte für Wartung oder Störung erwartet werden, so möchten Sie, dass das neuronale Netz zwei numerische Werte ausgibt, damit Sie beim Trainieren des Netzes einen Fehler ermitteln können.

Im Beispielfall gibt das neuronale Netz zwei numerische Werte aus, die zwischen 0.0 und 1.0 liegen und die sich zu 1.0 summieren. Dann können die angegebenen Werte als Wahrscheinlichkeiten interpretiert werden. Somit ist die Softmax-Funktion die erste Wahl, wenn im Output Layer eine Klassifizierung durchgeführt werden soll.

Die Softmax-Aktivierungsfunktion lässt sich in C# über die *Math.Exp*-Methode implementieren, wobei die Methode die angegebene Potenz von e zurückgibt. Die Softmax-Funktion berechnet im Prinzip einen Skalierungsfaktor, in dem die *Exp* jeder Output Layer Summe genommen und summiert werden. Danach wird die *Exp* jedes Wertes durch den Skalierungsfaktor dividiert. Beachten Sie, dass die endgültigen Ausgaben immer zwischen 0.0 und 1.0 liegen.

Bild 3.11 stellt die beiden Aktivierungsfunktionen als Graph nebeneinander dar. Als Entwickler sollten Sie auf jeden Fall beachten, dass die Aktivierungsfunktion die finale Ausgabe des Neurons bestimmt.

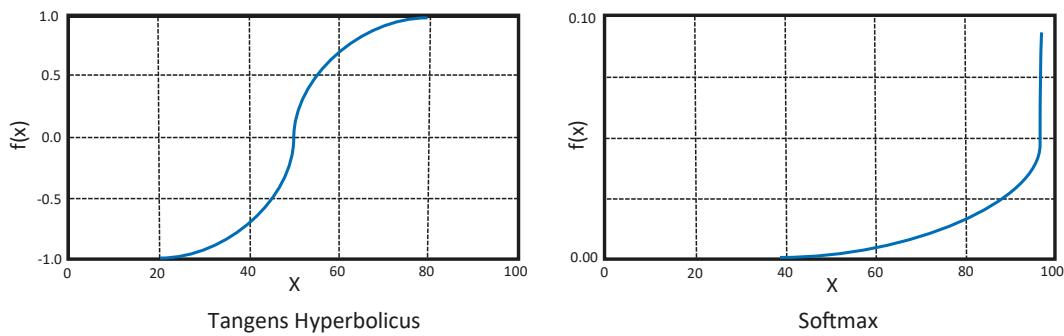


Bild 3.11 Vergleich der Aktivierungsfunktionen \tanh (links) und Softmax (rechts)

3.6.8 Fazit

Das Beispiel hat gezeigt, wie man einen Feedforward-Algorithmus erstellt und wie man ein neuronales Netz vom Input Layer über den Hidden Layer bis zum Output Layer in C# erstellt und berechnen kann. Sie sind mit dem Modell aus Bild 3.9 gestartet und haben dann in einzelnen Schritten über die Aktivierungsfunktionen ein Feedforward-Netz auf Basis eines Multilayer Perceptron für die Mehrklassen-Klassifizierung entwickelt.

Das Beispiel bietet viele Möglichkeiten, die vorgestellten Klassen und Methoden des neuronalen Netzes zu modifizieren, um beispielsweise parametrisierte Aktivierungsfunktionen zu verwenden oder mit anderen Beispieldaten zu experimentieren.

Es gibt bisher auch noch keine allgemein gültige neuronale Netzarchitektur, die den vielfältigen Aufgaben rund um das Erkennen, Verarbeiten und der Problemlösung gewachsen ist. Generell lässt sich aber feststellen: Je komplexer eine Aufgabe ist, umso mehr Hidden Layer (tiefe Netzstruktur) müssen in dem Netz vorhanden sein, um die benötigten Funktionen abzubilden.

Nachdem wir nun ein einfaches neuronales Netz erstellt haben, müssen jetzt die benötigten Werte ermittelt und das neuronale Netz trainiert werden. Die bekannteste Vorgehensweise hierfür ist das Backpropagation-Verfahren (siehe Abschnitt 3.7.3), das bereits 1974 von Paul Werbos [8] entwickelt wurde.

■ 3.7 Lernalgorithmus für Neuronen

Die bisherigen Beispiele, das Perzeptron aus Kapitel 2 und das FNN in Abschnitt 3.6, das gleich mit konkreten Zahlen für die Gewichte und den Bias-Werten initialisiert wurde, können nur begrenzte mathematische Funktionen umsetzen, sie können aber keine mathematischen Funktionen zur Laufzeit lernen. Die Netztopologie war hierfür zu einfach gestaltet. Die Form und Größe eines neuronalen Netzes entscheidet maßgeblich darüber, ob eine Problemstellung überhaupt erlernt werden kann und wie gut dann das Ergebnis ausfällt.

Leider stehen Ihnen als Entwickler eines neuronalen Netzes in der Praxis nicht die konkreten Gewichte und Bias-Werte wie beim FNN-Modell zur Verfügung, sondern anhand des Lernmaterials (Trainingsdaten bzw. Ergebniswerte) nur der entsprechende Output für das Netz. Dementsprechend werden in der Regel durch das Training des neuronalen Netzes diese Gewichte erst ermittelt, indem man die Gewichte zwischen den einzelnen Neuronen modifiziert. Das heißt, beim Supervised Learning, dem überwachten Lernen, wird versucht, durch den korrekt vorgegebenen Output die Gewichte zu optimieren. Es hat sich gezeigt, dass es bei großen neuronalen Netzen nicht mehr möglich ist, alle Parameter und Gewichte des Netzes manuell korrekt herzuleiten.

Einfach dargestellt lernen neuronale Netze, indem sie iterativ die korrekten vorgegebenen Output-Daten mit den tatsächlich errechneten Daten vergleichen und die Gewichte im Netz so anpassen, dass in jeder Iteration der Fehler zwischen Vorgabe und Ist-Daten im Netz reduziert wird.

Die schon vorgestellte Delta-Regel für das überwachte Lernen beschreibt, wie die Verbindungsge wichte verändert werden können, um ein Minimum der Kostenfunktion (häufig auch als Fehlerfunktion bezeichnet) zu erreichen. Die Delta-Regel bildet die Grundlage für den Backpropagation-Algorithmus, welcher auf Feedforward-Netze anwendbar ist. Auch hier wird versucht, den Fehler im Netz zu minimieren, wobei eine Kostenfunktion zur Berechnung eingesetzt wird, welche die Abweichung der Netzausgabe um den tatsächlichen Wert bewertet.

3.7.1 Kostenfunktion

Hinter der Kostenfunktion in Bild 3.12 steckt folgende Idee: Sie dient zum einen als Maß für die entsprechende Beschaffenheit eines ML-Modells und zum anderen bildet sie eine Fehlerlandschaft, auf deren Basis das ML-Modell immer weiter optimiert wird.

Bei der Kostenfunktion werden zunächst die Differenzen zwischen der Netzausgabe und den tatsächlichen Werten, die über die Trainingsdaten bekannt sind, quadriert und summiert. Die Kostenfunktion (Fehlerfunktion) wird in der Mathematik auch als Loss-Funktion bezeichnet.

Das heißt, beim Lernen im neuronalen Netz werden die Gewichte des Netzes schrittweise so verändert, dass der errechnete Netz-Output dem gewünschten bzw. festgelegten Ausgangswert immer näherkommt. Dieser Vorgang wird so lange wiederholt, bis der Fehler, den das Netz macht, auf ein Minimum gesunken ist.

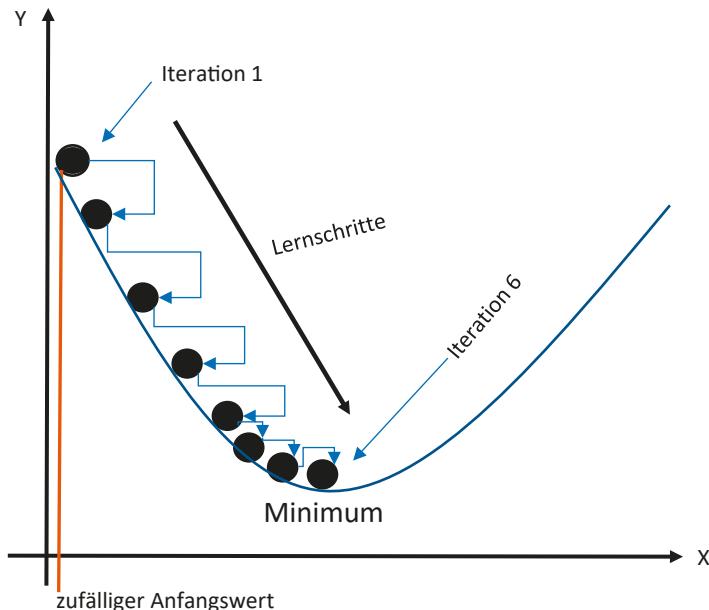


Bild 3.12 Die Kostenfunktion mit sieben Iterationsschritten

Die Differenz zwischen vorgegebenem und errechnetem Netz-Output wird quadriert, um ausschließen zu können, dass sich Fehler der Form 1 und -1 gegenseitig aufheben. Die aufgezeigte Fehlerkurve in Bild 3.12 hat nach sieben Iterationsschritten ihr Minimum erreicht. Wenn jetzt also das Gewicht diesen Wert hätte, würde das Netz einen sehr geringen Fehler aufweisen. Allerdings hat aber jedes Gewicht im Netz eine hiervon abweichende Fehlerkurve. Somit muss man für alle Gewichte, den jeweiligen Punkt, an dem die Fehlerkurve minimal ist, errechnen. Gelingt dies, so ist auch der Gesamtfehler des Netzes minimal.

Die Anpassung der Gewichte von Iteration zu Iteration, um die Kostenfunktion zu minimieren, hängt also lediglich vom Wert des jeweiligen Gewichtes aus der vorhergehenden Iteration sowie der gewünschten Lernrate ab. Die Lernrate steuert hierbei, wie groß die Schritte in Richtung der Fehlerminimierung ausfallen. Dieses Verfahren der Fehlerreduzierung bei der Kostenfunktion nennt sich Stochastic Gradient Descent (SGD), bzw. Gradientenabstieg, und ist ein Optimierungsalgorithmus im Machine Learning.

3.7.2 Gradientenabstiegsverfahren

Im Gradientenabstieg tasten Sie sich schrittweise an das Funktionsminimum heran. Hierbei wird mit einem zufälligen Initialisierungswert gestartet. Dann berechnen Sie über den Feedforward-Mechanismus die Ausgabe und vergleichen diese über den Fehlerwert der Kostenfunktion mit dem tatsächlichen korrekten Wert. Die zufällige Initialisierung führt in den meisten Fällen zu einem garantierten fehlerhaften Ergebnis und somit zu einem Fehlerwert. Somit ist der Gradient einfach die Ableitung, die angibt, wie sich der Output verändert, wenn die Gewichte verändert werden.

Das heißt, Sie können mit dem Gradienten jede Parametereinstellung der Gewichte ausrechnen, um die Richtung zu bestimmen, in der sich der Wert ändern muss, um den Output kleiner zu machen. Die Eingabe in dem Neuron wird dann wieder mit dem neuen Gewicht multipliziert und erst dann in das Neuron eingespeist, in der dann die gewählte Aktivierungsfunktion den Output für das Neuron erzeugt.

Die Berechnung in einem komplexen neuronalen Netz gestaltet sich allerdings etwas aufwendiger als die Betrachtung von nur einem Neuron. Der Grund dafür ist, dass die Gradienten der einzelnen Neuronen voneinander abhängig sind. Somit besteht eine Verkettung der einzelnen Neuronen im Netz und deren Wirkungen. Dieses Problem löst der Backpropagation-Algorithmus, der es erlaubt, die Gradienten in jeder Iteration rekursiv zu berechnen.

3.7.3 Backpropagation-Algorithmus

Neben dem Gradientenverfahren ist der Backpropagation-Algorithmus das wichtigste Prinzip eines neuronalen Netzes. Der Algorithmus ermöglicht die Ableitungen der Kostenfunktion nach den Gewichten zu berechnen. Die Ableitungen sind für das neuronale Netz zwingend erforderlich, um bestimmen zu können, in welche Richtung die Gewichte im Rahmen des Gradientenabstiegsverfahren zu verschieben sind, um das Minimum der Kostenfunktion zu erreichen.

Bild 3.13 zeigt ein FNN, bei dem Sie die Auswirkungen der Gewichtsänderung (roter Pfeil) auf die jeweiligen Neuronen sehen können. Wird das Gewicht eines einzelnen Neurons verändert, so wirkt sich dies auch auf alle Neuronen der nächsten Schicht und somit auch auf die Aktivierungsfunktion aller Neuronen und somit schließlich auch auf die Kostenfunktion aus.

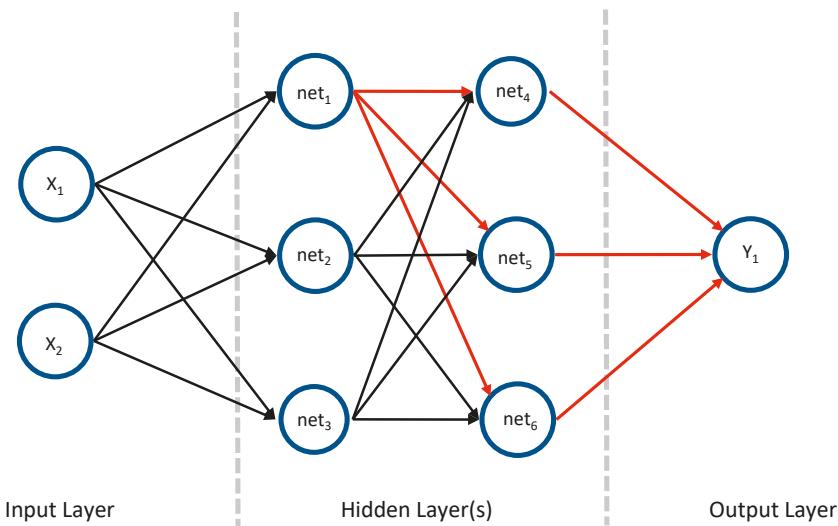


Bild 3.13 FNN mit Gewichten in Pfeilrichtung (rot)

Somit kann im Grundsatz der Backpropagation-Algorithmus wie folgt beschrieben werden:
Als Erstes werden zu Beginn des Lernvorgangs die Gewichte der einzelnen Neuronen mit zufälligen Werten initialisiert. Dann wird während des Lernvorgangs die Ausgabe, die das Neuron im Feedforward-Teil (Bild 3.14) erzeugt, mit der Lernrate verglichen.

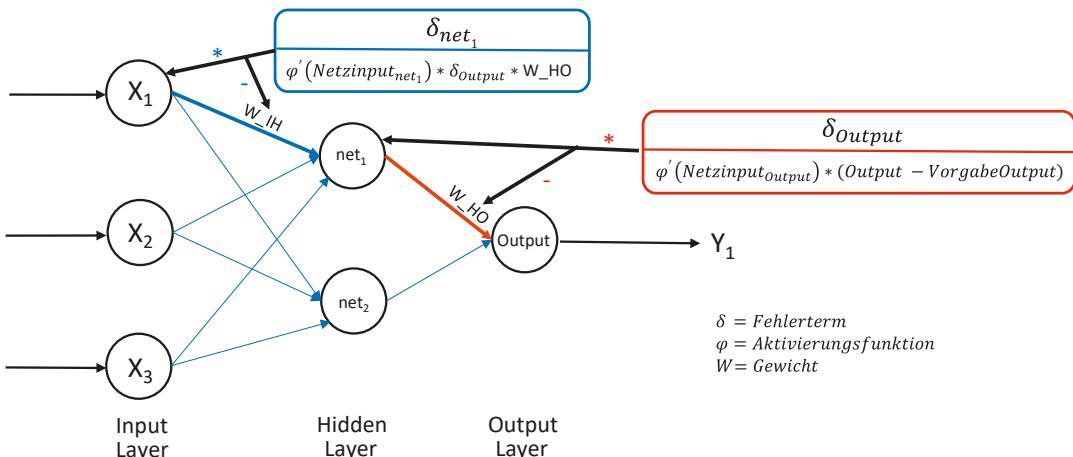


Bild 3.14 Backpropagation im Forward-Pass

Die Differenz der Ausgabe zum gewünschten Wert wird dann in Form einer Fehlerfunktion rückwärts durch die einzelnen Schichten weitergeleitet (Bild 3.15) bzw. propagiert (Backpropagation-Abschnitt), mit dem Ziel, dass die Gewichte der einzelnen Neuronen abgeändert werden. Dieses Verfahren wird so lange durchgeführt, bis jedes Gewicht im Netz bis hinab zur Eingangsschicht einen solchen Wert erhalten hat. Erst dieses Fehler-Verteilungsverfahren ermöglicht es, die zahlreichen Gewichte zur Fehlerminimierung einzusetzen.

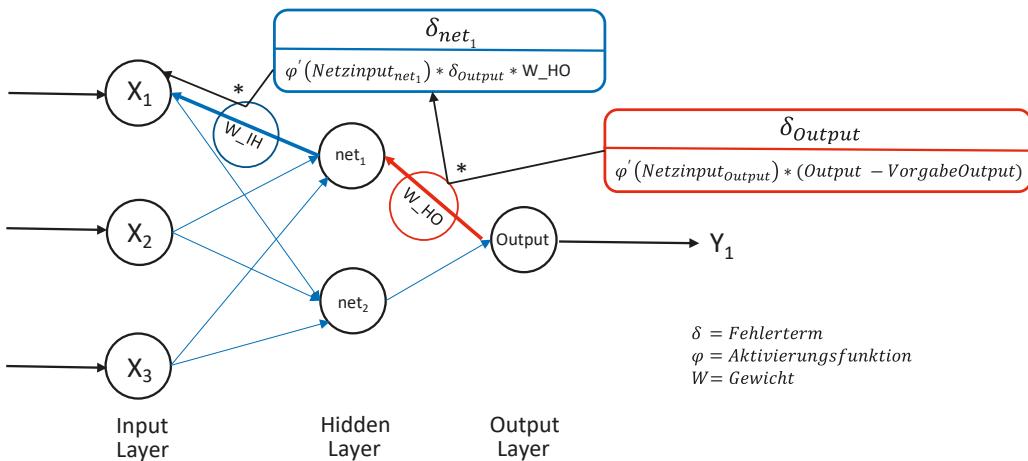


Bild 3.15 Backpropagation im Backward-Pass

Das heißt, um eine Veränderung der Gewichte und Bias-Werte über die entsprechenden Fehlerterme vornehmen zu können, muss der Backpropagation-Algorithmus jede Gewichtsänderung in drei Schritte aufteilen:

1. Forward-Pass: Der Output wird nach dem Feedforward-Mechanismus berechnet.
2. Im zweiten Schritt erfolgt die Fehlerbestimmung der Output-Neuronen. Wird festgestellt, dass die Fehler gering sind, wird die Trainingsphase beendet.
3. Die ermittelten Fehlerterme werden in entgegengesetzte Richtung bis zum Input Layer zurückgereicht und die Gewichte entsprechend modifiziert, so dass die Fehlerterme kleiner werden.

Für den Backpropagation-Algorithmus gibt es auch einen zugrunde liegenden Pseudocode, den jeder Entwickler in seine eigene Programmiersprache umsetzen kann:

```
REPEAT
    Ausgabe aus Eingabe berechnen
    Gradienten der Ausgabe-Neuronen ermitteln
    Gradienten der versteckten Neuronen ermitteln
    Gewichtungen und Bias-Werte anpassen
UNTIL Optimum erreicht
```

Daher kann man den Backpropagation-Algorithmus auch wie folgt beschreiben:

- Der Backpropagation-Algorithmus stellt eine Art modifiziertes Gradientenverfahren für neuronale Netze zur Verfügung.
- In seinen Optimierungsschritten wird der Fehler am Ausgang nach bestimmten Regeln anteilig Schicht für Schicht negativ auf die Gewichte verteilt.
- Es kann sich hierbei um den Fehler handeln, der aus der Summe aller Testfälle oder nur aus einem einzelnen Testfall resultiert.
- Man unterscheidet, ob ein Gewicht direkt zu einem Output-Neuron führt oder zu einem Neuron im Hidden Layer.

Bei dem hier vorgestellten Algorithmus handelt es sich um ein reines numerisches Lernverfahren zum überwachten Lernen vorwärts gerichteter mehrschichtiger neuronaler Netze. Inzwischen wurde der Backpropagation-Algorithmus mit verschiedensten Techniken weiterentwickelt und angepasst, um das Lernverhalten für unterschiedliche Netze zu implementieren. Die Netze unterscheiden sich im Wesentlichen nur durch die zum Lernen verwendete Variante des Backpropagation-Algorithmus (siehe Abschnitt 4.3, „Lernprozess beim Backpropagation-Algorithmus“).

Beim Einsatz einer konstanten Schrittweite sowie der ersten Ableitung der Fehlerfunktion gibt es einige Spezialfälle, bei denen der Backpropagation-Algorithmus nicht zu einem automatisierten Erkennen der Lösung verwendet werden kann und es treten dann folgende Fehler im Netz auf:

- Symmetry Breaking
- Falsches Minimum
- Oszillation
- Überspringen von Minima

Unter Symmetry Breaking, also die Symmetrie brechen, versteht man einen Fehlerfall bei der Initialisierung der Startgewichte. Diese dürfen zu Beginn des Trainings nicht alle gleich groß sein, da dann der Algorithmus ab der ersten Schicht keine unterschiedlichen Gewichte mehr ausprägen kann. Daher sollten die Gewichte immer mit zufälligen Werten initialisiert werden.

Bleibt der Backpropagation-Algorithmus bei der Ermittlung seines Minimums in einer Endlosschleife hängen, so spricht man von einer Oszillation in der Fehlerklasse der Kostenfunktion. Ist die Schrittweite, entlang der Fehlerfunktion zu groß gewählt, kann es vorkommen, dass ganze Minima übersehen werden. In diesem Fall spricht man vom Überspringen von Minima.

Diese auftretenden Fehler können Sie in den meisten Fällen über folgende Methoden lösen:

- Veränderung und Art der Initialisierung der Gewichte.
- Veränderung der Lernrate: Eine Erhöhung der Lernrate kann allerdings zu einer Oszillation führen. Oder Sie reduzieren die Lernrate. Der Nachteil bei der Reduzierung ist, dass die Trainingszeit bis zum Erreichen eines Minimums extrem lang werden kann.

In der Praxis haben sich bei der Nutzung des Backpropagation-Algorithmus folgende Begriffe zur Konfiguration durchgesetzt:

- **Epoch:** Beschreibt einen Vorwärts- und Rückwärtslauf aller Trainingsbeispiele.
- **Batch Size:** Beschreibt die Anzahl der Trainingsbeispiele.
- **Iteration:** Beschreibt die Anzahl der Läufe.

Damit haben Sie den Backpropagation-Algorithmus in der Theorie kennengelernt. Das Verfahren ist prinzipiell einfach und leicht erklärt. Die Funktionsweise des Algorithmus versteht man aber am besten, wenn man selbst ein neuronales Netz mit Backpropagation-Algorithmus implementiert.

■ 3.8 Backpropagation programmieren

Das neuronale Netz, wie es für den Backpropagation-Algorithmus verwendet werden soll, sehen Sie in Bild 3.16. Es besteht erst einmal aus drei Neuronen-Schichten. Es gibt den Input Layer mit acht Neuronen als Eingangsschicht, den Hidden Layer mit zehn Neuronen für die verdeckte Schicht und den Output Layer mit einem Neuron als Ausgabeschicht und der Ausgabe Y_1 .

Beim überwachten Lernen mit dem Backpropagation-Algorithmus wird ein Modell trainiert, das aus einer Menge von geprüften Daten eine oder mehrere Zielvariablen angeleichen kann. Wenn die Zielvariable kontinuierlich ist, handelt es sich um eine Regression, im Fall von diskreten Zielwerten spricht man von Klassifikation. Wie schon bei der vorausschauenden Wartung erklärt wurde (siehe Abschnitt 3.5), benötigen Sie in neuronalen Netzen bei der Klassifikation mit mehr als zwei Klassen normalerweise so viele Neuronen im Output Layer wie Klassen vorhanden sind. Das Neuron, das für die gegebenen Eingangswerte die größte Aktivierung aufweist, ist dann diejenige Klasse, die das Netz für am wahrscheinlichsten hält. Das nachfolgende Beispiel beschränkt sich zum Nachvollziehen des Backpropagation-Algorithmus auf die Klassifizierung von zwei Klassen und liefert somit am Ausgang Y_1 entweder 0 oder 1.

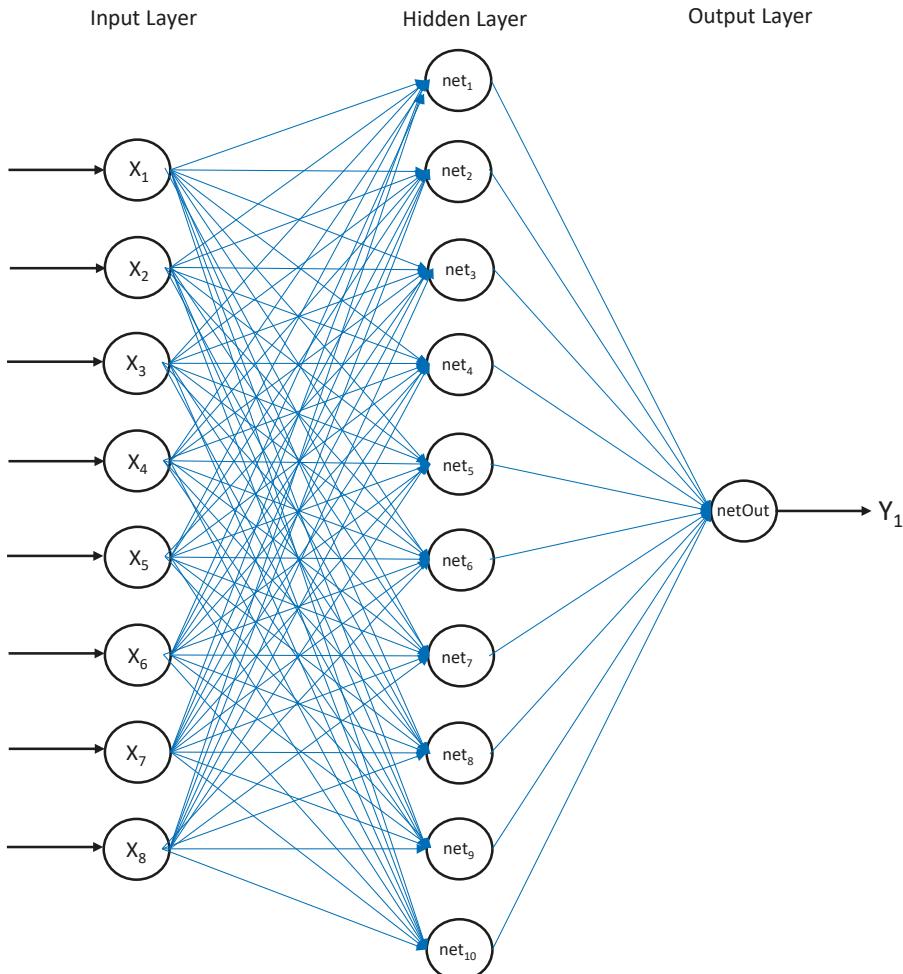


Bild 3.16 Das neuronale Netz für den Backpropagation-Algorithmus

Die Klassifizierung mit neuronalen Netzen ist ein wichtiges, aber auch komplexes Thema. Das hier vorgestellte Beispiel soll Ihnen eine solide Basis zum Experimentieren geben und bildet daher das Szenario mit numerischen Eingabedaten und einer kategorischen AusgabevARIABLEN am Ausgangspunkt ab. So sind alle benötigten Schritte, die im neuronalen Netz durchgeführt werden, für Sie als Entwickler in jedem einzelnen Schritt nachvollziehbar.

Damit auch dieses Netz seine Arbeit verrichten kann, muss es trainiert werden. Dazu werden Beispieldaten generiert, die Klassifizierungsergebnisse des Netzes darstellen. Beim Training versucht dann das Netz, die Beispiele zu reproduzieren. Beim Output-Neuron wird die angegebene Zahl als 0 für keine Primzahl und als 1 für eine Primzahl darstellt. Dazu passt das Netz die Gewichtung und den Bias-Wert immer wieder an. Als Lernschritt (Epoch, engl. Epoch) bezeichnet man dabei das Durcharbeiten aller Beispiele. Es müssen viele Lernschritte durchlaufen werden, bis das Netz nur noch geringe Fehler macht. Sie sollten daher im Beispiel auf jeden Fall mit der Anzahl der Lernschritte und der Lernrate experimentieren.

Nach dem Training kann man dem Netz einen Parameter in Form eines Zahlenwertes mitgeben, und es errechnet ein Ergebnis in Form von 0 und 1 für diesen Parameter.

Erfassung der Daten

Unser Beispiel fokussiert nicht auf die Verarbeitung großer Datenmengen, sondern es werden nur die erforderlichen Daten übertragen, um ein kleines neuronales Netz aufzubauen. Somit beschränken Sie sich auf die absolut nötigen Daten und etablieren quasi eine Mikrointelligenz auf Ihrem lokalen Computer, die schnell und effektiv die jeweilige Aufgabe lösen kann. Sie haben so eine gute Ausgangsbasis, um Ihr Netz selbst zu erweitern und mit verschiedenen Varianten zu experimentieren.

Das Beispiel für die Primzahlenanalyse verwendet auch keine fiktiven Trainingsdaten, sondern dient der Erkennung von Primzahlen zwischen 0 und 251. Sie können aber das Netz jederzeit um weitere Primzahlen erweitern. Sie finden im Internet unzählige Seiten und Tabellen mit Primzahlen, mit denen Sie Ihr neuronales Netz füttern können.

Die verwendeten Daten für das neuronale Netz werden in Binärform eingelesen. Jede Primzahl von 0 bis 251, also 54 Werte, werden in die Input-Neuronen eingespeist, wie auch alle Nicht-Primzahlen. Die Ausgabe Y_1 soll dann mit dem Wert 1 eine Primzahl und mit 0 keine Primzahl bestimmen.

Bei der Implementierung muss die Anzahl der Eingangsneuronen mit der Anzahl der Binärriffen der größten eingegebenen Zahl übereinstimmen. Da im Beispiel die gewählte höchste Zahl 251 ist, beträgt das binäre Äquivalent dazu $2^8 = 256$. Daher benötigen Sie im ersten Schritt für das Netz acht Eingangsneuronen. Möchten Sie später einen größeren Zahlenbereich verarbeiten, müssen Sie die Eingangsneuronen entsprechend anpassen. Mit 10 Neuronen im Input Layer könnten Sie also $2^{10} = 1024$ Primzahlen verarbeiten. Bei 12 Neuronen wären es schon 4096 und bei 16 Eingangsneuronen können Sie Ihr Netz auf 65.536 Primzahlen trainieren.

Der FNN-Mechanismus

Als Erstes bedienen Sie sich in diesem Beispiel wieder des FNN-Mechanismus. Die Werte werden an die Eingangsneuronen im Input Layer geleitet. Daraufhin werden diese Gewichte in allen Verbindungen zu den Knoten im Hidden Layer multipliziert und zu einem Aktivierungswert summiert. Anschließend greift die Delta-Funktion, die die errechnete Summe in einen nichtlinearen Wert umwandelt und diesen neuen gewichteten Wert weitergibt und den Output Layer erreicht und vom Output-Neuron gelesen wird.

Training

Als nächstes wird das Netz über die Backpropagation trainiert, indem der Wert von Y_1 mit dem gewünschten Ausgangswert verglichen wird und der Fehlerwert rückwärts durch das Netz geleitet zur Anpassung der Gewichte auf den neuronalen Konten verwendet wird.

Durch entsprechend viele Iterationen rückwärts durch die Knoten der Neuronen werden die Gewichte zu entsprechenden Werten konvergiert, die der gewünschten Ausgabe entsprechen. Es besteht aber immer die Möglichkeit, dass die Gewichte nicht übereinstimmen oder sich auf ein lokales Minimum annähern, das dann keine optimale Lösung bereitstellt. Hier können Sie als Entwickler Einfluss durch die Anpassung einiger Parameter, wie die

Erhöhung der Anzahl der Neuronen im Hidden Layer oder weiterer Schichten im Hidden Layer oder durch den Schwellenwert der Aktivierungsfunktion, der Lernrate oder der Anzahl der Iterationen, nehmen.

Auf diese Weise können Sie die Genauigkeit Ihres neuronalen Netzes verbessern. Beachten Sie beim Training eines neuronalen Netzes, dass Sie als Input immer Daten verwenden, die statistisch ein sehr breites Spektrum der Möglichkeiten abdecken, die Sie als Ergebnis erwarten. Nur so ist es auch möglich, kleine und sehr effektive neuronale Netze zum Beispiel für Services oder Microcontroller zu entwickeln.

Backpropagation

Nach dem Training des Netzes wird der Forward-Pass für die Ausführung im neuronalen Netz verwendet. Da im Beispiel das Netz nur über die drei Schichten Input Layer, Hidden Layer und Output Layer verfügt, wird zuerst die Aktivierungsebene im Hidden Layer berechnet. Dies wird durch die Berechnung der Eingabewerte multipliziert mit der Gewichtsmatrix zwischen dem Input Layer und dem Hidden Layer erreicht und dieses Ergebnis wird der Sigmoid-Aktivierungsfunktion übergeben. Als Formel lässt sich dies wie folgt darstellen:

$$\text{Hidden}_{\text{Layer}} = \text{sig}(\text{Input}_{\text{Layer}} * \text{Weights}_{\text{HiddenOutput}})$$

Die Sigmoid(x)-Funktion haben Sie ja schon kennengelernt, diese nähert sich für große Werte von x auf 1 und für sehr kleine Werte von x auf 0. Mithilfe der Sigmoid(x)-Funktion ist eine nichtlineare Aktivierung möglich. Die Formel für Sigmoid(x) lautet:

$$\text{sig}(x) = \frac{1}{1 + e^{-x}}$$

Als Nächstes folgt die Berechnung der Aktivierungen im Output Layer zusammen mit der Gewichtsmatrix aus dem Hidden Layer. Die hieraus resultierende Produktmatrix wird wiederum auf die Sigmoid-Aktivierungsfunktion ausgeführt. Die Formel lautet hierfür:

$$\text{Output}_{\text{Layer}} = \text{sig}\left(\text{Hidden}_{\text{Layer}} * \left[\text{Weights}_{\text{HiddenOutput}}\right]\right)$$

Somit ist der Forward-Pass für den Backpropagation-Algorithmus berechnet. Der $\text{Output}_{\text{Layer}}$ enthält die Ergebnisse dessen, was am Anfang in das neuronale Netz eingegeben wurde. Danach kann der Backward-Pass ausgeführt werden.

Backward-Pass

Sind die Aktivierungen für die Neuronen im Netz berechnet, können Sie die resultierenden Fehlerwerte zwischen dem tatsächlichen Output und dem gewünschten Output berechnen. Der errechnete Fehlerwert wird über das neuronale Netz zurückgekoppelt und zur Anpassung der Gewichte verwendet, um die Daten im Modell des neuronalen Netzes anzupassen.

Die Berechnung des Fehlerwertes im Output Layer erfolgt über:

$$\text{Error}_{\text{Output}} = \text{Output}_{\text{Layer}} * (1 - \text{Output}_{\text{Layer}}) * (\text{Output}_{\text{Layer}} - \text{Desired}_{\text{Layer}})$$

$\text{Desired}_{\text{Layer}}$ steht in der Formel für den gewünschten (Soll) Wert von Y_1 . Dieser errechnete Fehlerwert wird jetzt in den Hidden Layer zurückgespeist, um den Fehler in dieser Schicht zu berechnen:

$$\text{Error}_{\text{Hidden}} = \text{Hidden}_{\text{Layer}} * (1 - \text{Hidden}_{\text{Layer}}) * [\text{Weights}_{\text{HiddenOutput}}] * \text{Error}_{\text{Output}}$$

Jetzt muss noch die Gewichtsmatrix zwischen Hidden Layer und Output Layer angepasst werden, um dann wiederum das gesamte Netz anzupassen und den Fehlerwert auszugleichen. Um die Gewichte anpassen zu können, werden im Backpropagation-Algorithmus zwei konstante Faktoren genutzt. Bei dem ersten Faktor handelt es sich um die Lernrate, die anpasst, wie stark der berechnete Fehler die Gewichtsänderung in einer einzigen Abfolge durch das neuronale Netz beeinflusst.

Der zweite Faktor ist ein sogenannter *Momentum Factor*, der in Verbindung mit der vorherigen Gewichtsänderung verwendet wird. Als Entwickler haben Sie mit diesen beiden Faktoren einen erheblichen Einfluss auf die Genauigkeit Ihres neuronalen Netzes. Ein Anpassen dieser Faktoren ist für das Funktionieren fast unverzichtbar. Experimentieren Sie auf jeden Fall mit diesen Faktoren, es zeigt sich dann sehr schnell deren Zusammenspiel im Netz und die Auswirkungen der Änderungen auf das Lernverhalten des Netzes.

Die Berechnung mit den Faktoren erfolgt in der Form:

$$[\text{Weights}_{\text{HiddenOutput}}] = [\text{Weights}_{\text{HiddenOutput}}] + \Delta [\text{Weight}_{\text{HiddenOutput}}]$$

Das Δ (Delta) wird wie folgt berechnet und bindet hierbei auch die Lernrate L_R und den Momentum Factor M_F mit ein:

$$\begin{aligned} \Delta [\text{Weight}_{\text{HiddenOutput}}] &= (L_R) * \text{Hidden}_{\text{Layer}} * \text{Error}_{\text{Output}} \\ &\quad + (M_F) * \Delta [\text{Weight}_{\text{HiddenOutput(Vorherige)}}] \end{aligned}$$

Jetzt müssen nur noch die Gewichte für den Input Layer angepasst werden:

$$[\text{Weights}_{\text{InputHidden}}] = [\text{Weights}_{\text{InputHidden}}] + \Delta [\text{Weight}_{\text{InputHidden}}]$$

Und für die Berechnung des Δ (Delta):

$$\begin{aligned} \Delta [\text{Weight}_{\text{InputHidden}}] &= (L_R) * \text{Input}_{\text{Layer}} * \text{Error}_{\text{Hidden}} \\ &\quad + (M_F) * \Delta [\text{Weight}_{\text{InputHidden(Vorherige)}}] \end{aligned}$$

Fertig ist der Backward-Pass im Algorithmus. Diese Schritte werden jetzt für jeden nächsten Satz wiederholt. Sobald der gesamte Datensatz durchlaufen wurde, wird der Algorithmus für alle Daten erneut durchlaufen. Dieser Prozess wird solange fortgesetzt, bis das Netz die Annährung durchlaufen hat oder die angegebene Iterationsgröße erreicht wurde und der Prozess beendet wird.

Sie können nun beginnen, den Code für den Backpropagation-Algorithmus zu implementieren.

■ 3.9 Implementierung

Das zu implementierende neuronale Netz mit dem Backpropagation-Algorithmus besteht wiederum aus zwei Klassen: der Klasse *NeuralNetwork* und der Klasse *NetworkNode*. Auch dieses Beispiel basiert auf WPF als Oberfläche und bildet mit der *MainWindows.xaml*-Datei das Programm, über den der Algorithmus ausgeführt wird.

Als Projektname wird *BackpropagationDemo* verwendet. Bild 3.17 zeigt die WPF-Oberfläche mit dem entsprechenden neuronalen Netz, das als Image eingebunden wurde. Die benötigte Oberfläche wird über den XAML-Editor des *MainWindows* erstellt.

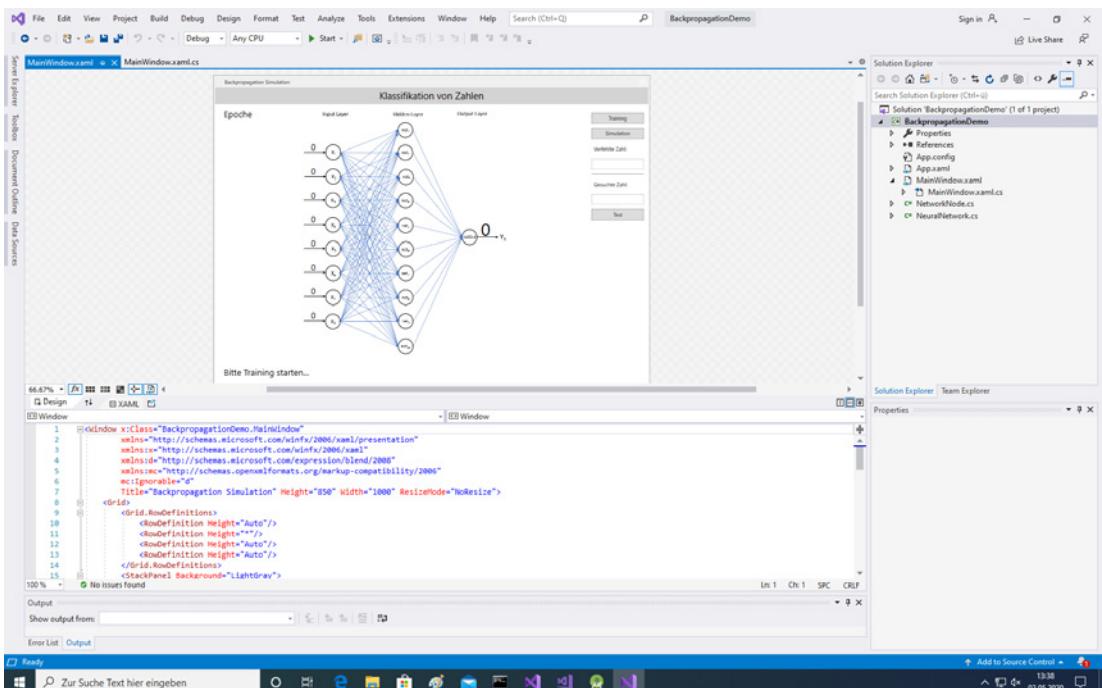


Bild 3.17 XAML-Editor mit dem neuronalen Netz als Oberfläche

Nutzen Sie hierfür einfach die XAML-Datei aus Listing 3.7. Alternativ können Sie die Oberfläche aber auch ganz nach Ihren eigenen Vorstellungen gestalten.

Listing 3.7 XAML-Code für die Oberfläche

```
<Window x:Class="BackpropagationDemo.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
       mc:Ignorable="d"
```

```

Title="Backpropagation Simulation" Height="850" Width="1000"
      ResizeMode="NoResize">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <StackPanel Background="LightGray">
        <Label FontSize="22" HorizontalAlignment="Center">Klassifikation von
Zahlen</Label>
    </StackPanel>
    <Grid Grid.Row="1">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition Width="140"/>
        </Grid.ColumnDefinitions>
        <Canvas>
            <Image Canvas.Left="15" Canvas.Top="15" Source = "c:\resource\BackPropagationNN.png" Height="577" Width="813"/>
            <TextBlock Name="StatusText" Height="35" Text="Bitte Training
starten..." Canvas.Top="600" Canvas.Left="20" FontSize="20"/>
            <TextBlock Name="EpochText" Height="25" Text="Epoch" Canvas.
Top="10" Canvas.Left="20" FontSize="20" />
            <TextBlock Name="NeuronX1" Text="0" Canvas.Top="85"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX2" Text="0" Canvas.Top="140"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX3" Text="0" Canvas.Top="195"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX4" Text="0" Canvas.Top="250"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX5" Text="0" Canvas.Top="305"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX6" Text="0" Canvas.Top="360"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX7" Text="0" Canvas.Top="415"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="NeuronX8" Text="0" Canvas.Top="470"
Canvas.Left="220" FontSize="20" />
            <TextBlock Name="OutputNeuron" Text="0" Canvas.Top="260"
Canvas.Left="610" FontSize="40" />
        </Canvas>
    </Grid>
    <StackPanel Grid.Column="1" Margin="10">
        <Button Name="bTraining" Height="25" Margin="0 10 0 0"
Click="BTraining_Click">Training</Button>
        <Button Name="bSimulation" Height="25" Margin="0 10 0 0"
Click="BSimulation_Click">Simulation</Button>
        <Label Content="Verfehlte Zahl:" Height="25" Margin="0 10 0 0" />
        <TextBox Name="teNoMatch" Height="25" Foreground="DarkRed"
Margin="0 10 0 0" FontSize="16" />
    </StackPanel>

```

```

        <Separator Margin="0 10 0 0"/>
        <Label Content="Gesuchte Zahl:" Height="25" Margin="0 10 0 0"/>
        <TextBox Name="teSearchedNumber" Height="25" Margin="0 10 0 0"/>
        <Button Name="bTest" Height="25" Margin="0 10 0 0"
               Click="BTest_Click">Test</Button>
    </StackPanel>
</Grid>
<StackPanel Grid.Row="3" Orientation="Horizontal" HorizontalAlignment="Right">
    <Button Name="bCancel" Height="25" Width="100" Margin="0 10 10 10"
           Click="BCancel_Click">Beenden</Button>
</StackPanel>
</Grid>
</Window>

```

Listing 3.8 zeigt die Code-Behind-Datei mit den Methoden zum Initialisieren des Netzes und den Start des Trainings für das neuronale Netz aus der *MainWindow.xaml.cs*-Datei. Auch hier enthält ein Knoten immer die Gewichtsmatrix des Neurons sowie den benötigten Aktivierungswert. Die Klasse *NeuralNetwork* führt alle Schritte des Backpropagation-Algorithmus aus, wobei die in jeder Schicht enthaltene Knotenänderung von Input Layer, Hidden Layer und Output Layer verwendet wird.

Listing 3.8 Code-Behind der MainWindow-Klasse

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Threading;

namespace BackpropagationDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        NeuralNetwork neuralNetwork = null;
        bool SimulationStarted = false;
        double[,] inputs = new double[1022, 8];
        double[,] outputs = new double[1022, 1];
        int timerCount = 0;
        string[] CurrentOutputValue;
        int CurrentCount = 0;
        DispatcherTimer timer = new DispatcherTimer();
        public MainWindow()
        {
            InitializeComponent();
            InitControlButton();
            InitializeNeuralNetwork();
            timer.Tick += timer_Tick;
        }
    }
}

```

```

int[] valueRange = new int[]{2,      3,      5,      7,      11,      13,      17,
                           19,      23,      29,      31,      37,      41,      43,
                           47,      53,      59,      61,      67,      71,      73,
                           79,      83,      89,      97,      101,     103,     107,
                           109,     113,     127,     131,     137,     139,     149,
                           151,     157,     163,     167,     173,     179,     181,
                           191,     193,     197,     199,     211,     223,     227,
                           229,     233,     239,     241,     251};

private void InitControlButton()
{
    bSimulation.IsEnabled = false;
    bTest.IsEnabled = false;
    teNoMatch.IsReadOnly = true;
    teSearchedNumber.IsEnabled = false;
}
private void InitializeNeuralNetwork()
{
    for (int i = 0; i < inputs.GetLength(0); i++)
    {
        int num = i;
        int mask = 0x200;
        for (int j = 0; j < 8; j++)
        {
            if ((num & mask) > 0)
                inputs[i, j] = 1;
            else
                inputs[i, j] = 0;

            mask = mask >> 1;
        }

        if (Array.BinarySearch(valueRange, i) >= 0)
        {
            outputs[i, 0] = 1;
        }
        else
        {
            outputs[i, 0] = 0;
        }
    }
}

private void BTraining_Click(object sender, RoutedEventArgs e)
{
    TraningStartNeuralNetwork();
    bSimulation.IsEnabled = true;
    bTest.IsEnabled = true;
    teSearchedNumber.IsEnabled = true;
    StatusText.Text = "Training beendet";
    bTraining.IsEnabled = false;
}

private void TraningStartNeuralNetwork()

```

```
{  
    int numberofHiddenNeurons = 10;  
    int numberofEpoches = 1022;  
    neuralNetwork = new NeuralNetwork(inputs.GetLength(1),  
                                      numberofHiddenNeurons, (int)outputs.GetLength(1));  
    neuralNetwork.FirstTimeSettings();  
  
    neuralNetwork.TrainingDataToNetwork(inputs, outputs, numberofEpoches);  
}  
  
private void BCancel_Click(object sender, RoutedEventArgs e)  
{  
    Application.Current.Shutdown();  
}  
  
private void BSimulation_Click(object sender, RoutedEventArgs e)  
{  
    StartTimedSimulation();  
}  
  
private void StartTimedSimulation()  
{  
    timerCount = 0;  
    teNoMatch.Text = "0";  
    timer.Interval = TimeSpan.FromMilliseconds(250);  
    timer.Start();  
    SimulationStarted = true;  
}  
private void timer_Tick(object sender, EventArgs e)  
{  
    CurrentCount = timerCount;  
    CurrentOutputValue = neuralNetwork.TestDrive(CurrentCount, inputs).  
        Split(new char[] { ' ' });  
  
    DoStats(CurrentCount, Convert.ToInt32(CurrentOutputValue[0]));  
  
    timerCount++;  
    if (CurrentCount >= inputs.GetUpperBound(0))  
    {  
        VisualizingValues();  
        SimulationStarted = false;  
        timer.Stop();  
    }  
    else  
    {  
        VisualizingValues();  
    }  
}  
  
private void VisualizingValues()  
{  
    EpochText.Text = "Epoch = " + CurrentCount.ToString();  
}
```

```

        NeuronX1.Text = inputs[CurrentCount, 0].ToString();
        NeuronX2.Text = inputs[CurrentCount, 1].ToString();
        NeuronX3.Text = inputs[CurrentCount, 2].ToString();
        NeuronX4.Text = inputs[CurrentCount, 3].ToString();
        NeuronX5.Text = inputs[CurrentCount, 4].ToString();
        NeuronX6.Text = inputs[CurrentCount, 5].ToString();
        NeuronX7.Text = inputs[CurrentCount, 6].ToString();
        NeuronX8.Text = inputs[CurrentCount, 7].ToString();

        for(int i = 0; i < 1; i++)
        {
            if (SimulationStarted)
            {
                OutputNeuron.Foreground = new SolidColorBrush(Colors.Green);
                OutputNeuron.Text = CurrentOutputValue[i].ToString();
            }
            else
            {
                OutputNeuron.Foreground = new SolidColorBrush(Colors.Black);
                OutputNeuron.Text = outputs[CurrentCount, i].ToString();
            }
        }

        private void DoStats(int currentCount, int v)
        {
            if (Array.BinarySearch(valueRange, currentCount) >= 0)
            {
                if (v == 0)
                {
                    int val = Convert.ToInt32(teNoMatch.Text);
                    teNoMatch.Text = (val + 1).ToString();
                }
            }
        }

        private void BTest_Click(object sender, RoutedEventArgs e)
        {
            CurrentCount = Convert.ToInt32(teSearchedNumber.Text);
            CurrentOutputValue = neuralNetwork.TestDrive(_CurrentCount, inputs).
                Split(new char[] { ' ' });
            VisualizingValues();
        }
    }
}

```

In Listing 3.7 wird über die Methode *InitializeNeuralNetwork* das neuronale Netz mit den binären-Eingabewerten vorinitialisiert. Über das *ButtonClick-Event BTraining_Click* wird das Training des Netzes aufgerufen und der Aufbau des Netzes festgelegt. Die Methode *TrainingDataToNetwork* in der Klasse *NeuralNetwork* nimmt die entsprechenden Daten entgegen. Listing 3.9 zeigt die Implementierung des kompletten Backpropagation-Algorithmus in der Klasse *NeuralNetwork*.

Listing 3.9 Die Klasse NeuralNetwork

```
using System;

namespace BackpropagationDemo
{
    public class NeuralNetwork
    {
        private int input;
        private int output;
        private int hidden;

        private double LearnHidden = 0.15f;
        private double LearnOutput = 0.2f;

        private double[] inputsToNetwork;
        private double[] desiredOutputs;

        private NetworkNode[] inputNN;
        private NetworkNode[] outputNN;
        private NetworkNode[] hiddenNN;

        public double error_compared_to_tolerance = 0;

        public NeuralNetwork(int i, int h, int o)
        {
            input = i;
            output = o;
            hidden = h;
            int ahm = 0;
            inputNN = new NetworkNode[input];
            outputNN = new NetworkNode[output];
            hiddenNN = new NetworkNode[hidden];
            Random rand = new Random(unchecked((int)DateTime.Now.Ticks));

            for (int x = 0; x < input; x++)
            {
                inputNN[x] = new NetworkNode();
                inputNN[x].weights = new double[hidden];
                for (int j = 0; j < hidden; j++)
                {
                    ahm = rand.Next() & 1;
                    inputNN[x].weights[j] = rand.NextDouble();
                    if (ahm == 0)
                        inputNN[x].weights[j] *= -1;
                }
            }

            for (int y = 0; y < hidden; y++)
            {
                hiddenNN[y] = new NetworkNode();
                hiddenNN[y].weights = new double[output];
                for (int j = 0; j < output; j++)
                {
                    hiddenNN[y].weights[j] = rand.NextDouble();
                }
            }
        }
    }
}
```

```
        }

    }

    for (int z = 0; z < output; z++)
    {
        outputNN[z] = new NetworkNode();
    }
}

public void FirstTimeSettings()
{
    Random x = new Random(unchecked((int)DateTime.Now.Ticks));
    for (int i = 0; i < hidden; i++)
    {
        hiddenNN[i].Threshold = x.NextDouble();
    }

    for (int i = 0; i < output; i++)
    {
        outputNN[i].Threshold = x.NextDouble();
    }
}

public void TrainingDataToNetwork(double[,] inputlist, double[,] outputlist,
                                   int iterations)
{
    inputsToNetwork = new double[input];
    desiredOutputs = new double[output];

    int outputlistSampleLength = outputlist.GetUpperBound(0) + 1;
    int outputlistLength = outputlist.GetUpperBound(1) + 1;
    int inputlistLength = inputlist.GetUpperBound(1) + 1;

    for (int i = 0; i < iterations; i++)
    {
        for (int sampleindex = 0; sampleindex < outputlistSampleLength;
             sampleindex++)
        {
            for (int j = 0; j < inputlistLength; j++)
            {
                inputsToNetwork[j] = inputlist[sampleindex, j];
            }

            for (int k = 0; k < outputlistLength; k++)
            {
                desiredOutputs[k] = outputlist[sampleindex, k];
            }

            TrainingPattern();
        }
    }
}
```

```

private void TrainingPattern()
{
    CalculateActivation();
    CalculateErrorOutput();
    CalculateErrorHidden();
    CalculateNewThresholds();
    CalculateNewWeightsInHidden();
    CalculateNewWeightsInInput();
}

private void CalculateActivation()
{
    int countHidden = 0;
    while (countHidden < hidden)
    {
        for (int ci = 0; ci < input; ci++)
        {
            hiddenNN[countHidden].Activation += inputsToNetwork[ci] *
                inputNN[ci].weights[countHidden];
        }

        countHidden++;
    }

    for (int x = 0; x < hidden; x++)
    {
        hiddenNN[x].Activation += hiddenNN[x].Threshold;
        hiddenNN[x].Activation = sigmoid(hiddenNN[x].Activation);
    }
}

int countOutput = 0;
while (countHidden < output)
{
    for (int chi = 0; chi < hidden; chi++)
    {
        outputNN[countOutput].Activation += hiddenNN[chi].Activation *
            hiddenNN[chi].weights[countOutput];
    }

    countOutput++;
}

for (int x = 0; x < output; x++)
{
    outputNN[x].Activation += outputNN[x].Threshold;
    outputNN[x].Activation = sigmoid(outputNN[x].Activation);
}
}

private double sigmoid(double activation)
{
    return 1 / (1 + Math.Exp(-activation));
}

```

```

private void CalculateErrorOutput()
{
    for (int x = 0; x < output; x++)
    {
        outputNN[x].error = outputNN[x].Activation *
            (1 - outputNN[x].Activation) *
            (desiredOutputs[x] - outputNN[x].Activation);
    }
}

private void CalculateErrorHidden()
{
    int y = 0;
    while (y < hidden)
    {
        for (int x = 0; x < output; x++)
        {
            hiddenNN[y].error += hiddenNN[y].weights[x] * outputNN[x].error;
        }
        hiddenNN[y].error *= hiddenNN[y].Activation *
            (1 - hiddenNN[y].Activation);
        y++;
    }
}

private void CalculateNewThresholds()
{
    for (int x = 0; x < hidden; x++)
    {
        hiddenNN[x].Threshold += hiddenNN[x].error * LearnHidden;
    }

    for (int y = 0; y < output; y++)
    {
        outputNN[y].Threshold += outputNN[y].error * LearnOutput;
    }
}

private void CalculateNewWeightsInHidden()
{
    int x = 0;
    double temp = 0.0f;
    while (x < hidden)
    {
        temp = hiddenNN[x].Activation * LearnOutput;
        for (int y = 0; y < output; y++)
        {
            hiddenNN[x].weights[y] += temp * outputNN[y].error;
        }

        x++;
    }
}

```

```

private void CalculateNewWeightsInInput()
{
    int x = 0;
    double temp = 0.0f;
    while (x < input)
    {
        temp = inputsToNetwork[x] * LearnHidden;
        for (int y = 0; y < hidden; y++)
        {
            inputNN[x].weights[y] += temp * hiddenNN[y].error;
        }

        x++;
    }
}

public string TestDrive(int index, double[,] inputlist)
{
    PopulateInputList(inputlist, index);
    CalculateActivation();
    string value = GetValueOutput();
    return value;
}

void PopulateInputList(double[,] inputlist, int index)
{
    for (int j = 0; j < inputlist.GetUpperBound(1) + 1; j++)
    {
        inputsToNetwork[j] = inputlist[index, j];
    }
}

public string GetValueOutput()
{
    string getOutput = "";

    for (int x = 0; x < output; x++)
    {
        if (outputNN[x].Activation > 0.5)
            getOutput += "1" + " ";
        else
            getOutput += "0" + " ";
    }
    return getOutput;
}
}
}

```

Des Weiteren benötigen Sie noch die Klasse *NetworkNode* (Listing 3.10). Diese Klasse setzt die Knotenpunkte mit dem Activation-Wert und der Fehlerwertangabe *error*. Die Klasse stellt somit die Variablen *Activation* und *Threshold* für den Schwellwert zur Verfügung.

Listing 3.10 Die Klasse NetworkNode

```

namespace BackpropagationDemo
{
    public class NetworkNode
    {
        public double[] weights;
        public double error;

        public double Activation { set; get; }
        public double Threshold { set; get; }

        public NetworkNode()
        {
            Activation = 0;
            error = 0;
        }
    }
}

```

Die zentrale Methode *TrainingPattern* in der Klasse *NeuralNetwork* liefert schrittweise die einzelnen Methoden-Aufrufe für den Backpropagation-Algorithmus. Die Methode *CalculateActivation* implementiert den Forward-Pass. Hier wird der Knoten im Input Layer mit den Gewichten zwischen Import Layer und Hidden Layer multipliziert und die Sigmoid-Funktion für den Aktivierungswert des Hidden-Layer-Neurons errechnet. Im zweiten Schritt wird jeder Knoten im Hidden Layer mit den Gewichten zwischen dem Hidden Layer und dem Output Layer multipliziert und wiederum auf die Sigmoid-Aktivierungsfunktion angewendet, um den Aktivierungswert des Output-Neurons zu erhalten.

Die Methoden *CalculateErrorOutput* und *CalculateErrorHidden* errechnen jeweils die benötigten Fehlerwerte. Die Methode *CalculateNewThresholds* ermittelt für die Neuronen im Hidden Layer und entsprechend im Output Layer die benötigten Schwellenwerte.

Hiernach berechnet die Methode *CalculateNewWeightsInHidden* die neuen Gewichte im Hidden Layer, die mit dem Fehlerwert des Output Layers angepasst wurden. Die Methode *CalculateNewWeightsInInput* berechnet die neuen Gewichte im Input Layer mit den Fehlerwerten, die im Hidden Layer angepasst wurden.

Die Anzahl der Lernschritte (Epochen) wird im Beispiel mit dem Wert 1022 vorbelegt und die Anzahl der Neuronen für Input und Output ist ebenfalls fest codiert. Die Lernrate für den Hidden Layer wird über die Variable *LearnHidden* und die für den Output Layer über die Variable *LearnOutput* festgelegt.

Zum Ausführen des Netzes wird beim Start das Training durchgeführt. Nachdem das Training abgeschlossen ist, können Sie über den Button *Simulation* die einzelnen Iterationsschritte durchlaufen lassen. Die Methode *DoStats* in der *MainWindows.xaml.cs*-Datei sammelt dann Statistiken, um angeben zu können, ob das Netz entsprechende Fehler enthält bzw. ausweist.

Ist die Simulation abgeschlossen, können Sie über die Zahleneingabe testen, ob es sich bei der eingegebenen Zahl 5 (Bild 3.18) um eine Primzahl handelt. Das Ergebnis bei Y_1 ist 1, somit hat das Netz die Anfrage über die gestellte Zahl richtig klassifiziert. Bitte beachten Sie hier, dass das Netz nur bis 251 rechnen kann.

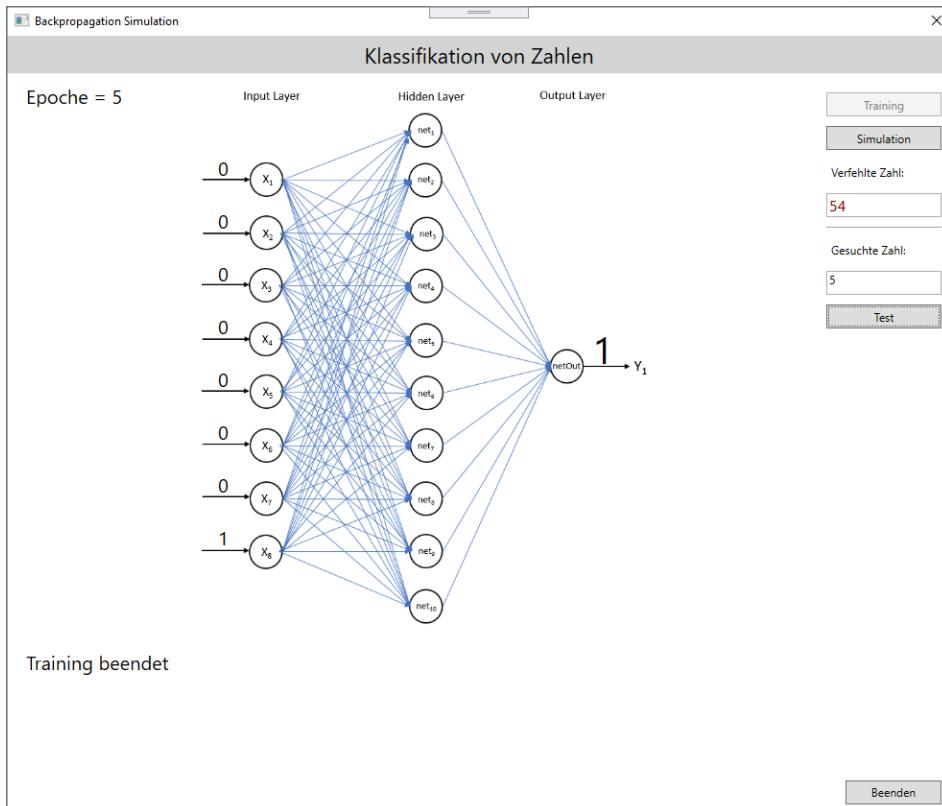


Bild 3.18 Die Klassifizierung einer Zahl

Experimentieren Sie jetzt mit dem neuronalen Netz und dem Backpropagation-Algorithmus. Verändern Sie die Parameter der Lernrate oder der Lernschritte, verwenden Sie eine andere Aktivierungsfunktion und erhöhen Sie die Anzahl der Input-Neuronen und der Hidden-Layer-Neuronen. Sie erhalten so weitere aufschlussreiche Einblicke in die Funktion eines neuronalen Netzes im Zusammenspiel mit dem Backpropagation-Algorithmus.

4

Training von neuronalen Netzen

Nachdem im vorherigen Kapitel an einem Beispiel die Kostenfunktion, das Gradientenabstiegsverfahren und der Backpropagation-Algorithmus dargestellt wurden, soll in diesem Abschnitt der allgemeine Ablauf des Trainingsprozesses und die Optimierung des Lernprozesses an einem ausführlichen Beispiel dargestellt werden.

Bei den bisher gezeigten Beispielen waren die Trainingsdaten und die Netzparameter wie Lernrate, Epoche und die Aktivierungsfunktion immer fest im Code implementiert. Nachfolgend sollen diese Parameter im neuronalen Netz über eine Benutzeroberfläche einstellbar gemacht werden. Unser Beispiel soll eine Mehrklassen-Klassifikation mit drei verschiedenen Objektklassen abbilden. Bei der hier gezeigten rein programmtechnischen Implementierung wird jeder Anwendungsfall in einem eigenen Visual-Studio-Projekt umgesetzt und das Erstellen der Trainings- und Testdaten erfolgt in Handarbeit.

■ 4.1 Trainings- und Testphase

Das neuronale Netz lernt beim überwachten Lernen (Supervised Learning) im Verlauf der Trainingsphase mithilfe des vorgegebenen Lernmaterials. Infolgedessen werden in der Regel die Gewichte zwischen den einzelnen Neuronen modifiziert. Das verwendete Lernverfahren gibt dabei die Art und Weise an, wie das neuronale Netz diese Veränderungen vornimmt. Das heißt zum Beispiel, dass für den Backpropagation-Algorithmus beim Lernen immer der korrekte Output vorgegeben wird und daran die Gewichte optimiert werden.

In der Testphase werden hingegen keine Gewichte verändert. Stattdessen wird auf Grundlage der bereits modifizierten Gewichte aus der Trainingsphase untersucht, ob das neuronale Netz etwas gelernt hat. Dazu füttert man die Neuronen im Input Layer mit Daten und prüft, welchen Output das neuronale Netz berechnet. Hierbei kann man verschiedene Arten von Daten untersuchen:

- Ausgangsdaten: Durch eine erneute Präsentation der zu lernenden Ausgangsdaten kann geprüft werden, ob das neuronale Netz die Trainingsdaten erfasst hat.
- Neue Daten: Durch die Verwendung neuer Daten kann man feststellen, ob das Netz über die zu lernenden Werte hinaus in der Lage ist, Aufgaben zu lösen. Somit kann überprüft werden, ob eine Generalisierung auf Grund der neuen Daten im neuronalen Netz möglich ist.

4.1.1 Generalisierung

Um in der Testphase auf neue Datensätze zurückgreifen zu können, bedient man sich in der Praxis eines einfachen Tricks. Man nutzt hierfür die Hold-Out Validation (Bild 4.1), bei der es sich um die einfachste Art der Kreuzvalidierung handelt. Bei der Kreuzvalidierung teilen Sie die Daten nach dem Zufallsprinzip in zwei Teile auf. Das neuronale Netz wird dann auf den Trainingsdatensatz trainiert und über den Testdatensatz ausgewertet.

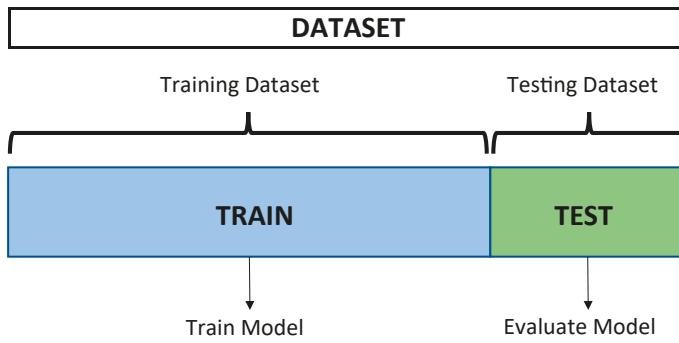


Bild 4.1
Bei der Hold-Out Validation dient der Gesamtdatensatz für Training und Test.

Somit dienen 20 Prozent des Gesamtdatensatzes für den Test, wie Bild 4.1 zeigt. Mit diesen Testdaten kann man dann Rückschlüsse auf unbekannte Daten ziehen. Das bedeutet, wenn beide Ergebniswerte aus dem Trainings- und Testdatensatz im Idealfall gleich sind, arbeitet das Modell des neuronalen Netzes mit unbekannten Daten genauso wie mit bekannten Daten. Zu den typischen Verhältnissen, die zur Aufteilung des Datensatzes verwendet werden, gehören 60:40, 80:20 usw. Der Trainingsdatensatz sollte aber immer größer als der Testdatensatz sein. Die Hold-Out-Validation-Methode findet immer dann Verwendung, wenn in dem Modell des neuronalen Netzes keine Hyperparameter bestimmt werden müssen. Liegen Hyperparameter vor, so greift man auf die Methode der k-fachen Kreuzvalidierung zurück. Dabei wird der Datensatz zufällig in k-Gruppen aufgeteilt. Eine der Gruppen wird als Testdatensatz und die übrigen als Trainingsdatensatz verwendet. Das Modell im neuronalen Netz wird dann auf den Trainingsdatensatz trainiert und auf den Testdatensatz bewertet. Dann wird der Vorgang wiederholt, bis jede einzelne Gruppe als Testdatensatz verwendet wurde.

4.1.2 Dimensionsreduzierung

Benötigt man für sein neuronales Netz eine weniger präzise Generalisierung für unbekannte Daten, so kann man in der Trainings- und Testphase auch eine Beschränkung der Parameteranzahl vornehmen. Es kann auch von Vorteil sein, die Daten auf wenige Dimensionen zu reduzieren und diese für den Anwender für die Überprüfung besser sichtbar zu machen. Des Weiteren führt eine Verringerung der Dimensionen und der Parameter zu einem schnelleren Training.

Bei einer Überprüfung des Gesamtdatenbestandes kann es auch sein, dass man feststellt, dass bestimmte Werte nicht zwingend erforderlich sind. Wenn sich zum Beispiel viele Datensätze um die Nullachse bewegen, kann man eventuell auf diese ohne großen Verlust für das Endergebnis verzichten.

■ 4.2 Batch-, inkrementelles und Mini-Batch-Training

Für das Training eines neuronalen Netzes gibt es drei Hauptansätze, die zumeist als Batch-Training bzw. Offline Learning, inkrementelles Training bzw. Online Learning und Mini-Batch-Training bezeichnet werden.

4.2.1 Batch-Training

Beim Batch-Training handelt es sich um eine Stapelverarbeitung. Daher wird diese Trainingsprozedur auch als Offline Learning bezeichnet. Beim Batch-Verfahren werden die Parameter bzw. das Modell erst angepasst, nachdem der gesamte Stapel an Datensätzen (Batch) das Training durchlaufen hat. In Pseudocode ausgedrückt funktioniert das Batch-Training folgendermaßen:

```
Schleife bis zum Abschluss
    Für jeden Trainingsdatensatz
        Fehler berechnen und Gesamtfehler akkumulieren
    Ende Für
    Gesamtfehler verwenden, um Gewichte und Bias-Werte zu aktualisieren
Ende Schleife
```

Beim Batch-Training wird das Gradientenabstiegsverfahren in der Stapelverarbeitung genutzt. Dabei wird der Gradient für die gesamte Trainingsdatenmenge berechnet. Durch diese Vorgehensweise erlaubt das Batch-Training eine sehr schnelle Ausführung, da die Fehlermetrik direkt aus dem gesamten Trainingssatz bestimmt wird.

4.2.2 Inkrementelles Training

Beim inkrementellen Training wird nicht über einen Stapel trainiert, sondern jeder einzelne Datensatz wird beim Training einzeln zur Laufzeit hinzugefügt. Diese Art von Training wird auch häufig als Online Learning bezeichnet. In Pseudocode ausgedrückt funktioniert das inkrementelle Training wie folgt:

```
Schleife bis zum Abschluss
    Für jeden Trainingsdatensatz
        Berechnungsfehler für den aktuellen Datensatz
    Datensatzfehler verwenden, um Gewichte und Bias-Werte zu aktualisieren
Ende Schleife
```

Das heißt, beim inkrementellen Training werden die Gewichte und Bias-Werte für jeden Trainingsdatensatz aktualisiert. Beim Backpropagation-Algorithmus nutzt man in den meisten Fällen das inkrementelle Training. Auch das nachfolgende Beispiel, Mehrklassenklassifikation

für Predictive Maintenance, nutzt den Backpropagation-Algorithmus mit einem Momentum-Faktor und dem inkrementellen Vorgehen.

Der Trainingsprozess läuft somit immer iterativ ab und bestimmt den Fehlerwert für jeden einzelnen Datensatz. Das neuronale Netz lernt bei diesem Verfahren quasi in Echtzeit, da es sofort mit neuen Daten versehen werden kann, wenn diese zur Verfügung stehen. Es kann somit auch auf Änderungen reagieren und Trainingsdaten verwerfen, da sie bereits in die Lernphase eingeflossen sind.

4.2.3 Mini-Batch-Training

Das Mini-Batch-Training stellt einen dritten Ansatz für das Training von neuronalen Netzen dar. Mini-Batch ist eine Kombination aus Batch- und inkrementellem Training. Mini-Batch stellt den Mittelweg dar und ist in der Lage, Trainingsdatensätze in kleine Blöcke, zum Beispiel immer 50 Datensätze in einem Stapel (Batch), zu sammeln und zu verarbeiten. Der Pseudocode sieht folgendermaßen aus:

```

Schleife bis zum Abschluss
  Schleifen n-mal
    Fehlerwert für den aktuellen Datensatz berechnen
    Fehler akkumulieren
  Ende Schleife
  Aktuelle akkumulierte Fehler verwenden, um Gewichte und Bias-Werte zu
  aktualisieren
  Mini-akkumulierte Fehler auf 0 zurücksetzen
Ende Schleife

```

Bei diesem Verfahren bestimmt somit die Stapelgröße die Anzahl der Trainingsdatensätze, bevor eine Gewichts- und Bias-Wert-Aktualisierung durchgeführt wird. Beim Mini-Batch handelt es sich um eine Zusammenführung aller drei Ansätze.

■ 4.3 Lernprozess beim Backpropagation-Algorithmus

Auch beim Einsatz des Backpropagation-Algorithmus in einem neuronalen Netz müssen die Trainings- und die Testphase abgearbeitet werden. Somit wird für den Backpropagation-Algorithmus in der Trainingsphase eine Trainingsmenge bestehend aus Eingabe- und Ausgabepaaren präsentiert. Das Netz lernt dann in der Trainingsmenge vorhandene Sachverhalte, indem der Anwender ihm mitteilt, welche Ausgabe erwartet wird, wenn eine bestimmte Eingabe vorliegt. Die Testphase dient der Überprüfung der Generalisierungsfähigkeit des gerade trainierten neuronalen Netzes. Der Lernprozess spielt sich auch beim Backpropagation-Algorithmus in der wichtigen und gleichzeitig aufwendigen Testphase ab.

Im nachfolgenden Beispiel werden Sie mithilfe des Backpropagation-Algorithmus mit Momentum-Faktor eine vorausschauende Wartung für ein Wälzlager erstellen, um den optimalen Zustand, einen belasteten Zustand oder einen überlasteten Zustand zu kategorisieren. Des Weiteren soll das neuronale Netz über Parameter in der Benutzeroberfläche konfigurierbar sein.

Wie bereits angesprochen, wird Predictive Maintenance in vielen Bereichen eingesetzt. Eine besonders wichtige Rolle spielt Predictive Maintenance bei der Fahrzeugwartung, hier besonders bei Schienen- und Maschinenfahrzeugen im Bergbau, sowie bei Maschinen für schwere Lasten in der Produktion, wie Kräne und Portale. Die umfangreichen Datenerhebungen vieler verschiedener Sensoren in Motor, Antriebswellen, Getriebe aber auch im Fahrwerk können dazu beitragen, dass teure Reparaturen oder Ausfälle verhindert werden. Dementsprechend können über Predictive Maintenance rechtzeitig vorbeugende Maßnahmen eingeleitet werden.

Wie so etwas funktioniert, lässt sich an einem Wälzlager gut demonstrieren. Führungsgrößen sind hier Axialschub, Drehzahl, Temperatur und Schwingungen. Diese werden kontinuierlich erfasst und so ausgewertet, dass Unregelmäßigkeiten erfasst und ihre Auswirkung auf die Lagerlebensdauer errechnet werden. Infolgedessen lassen sich auch sogenannte Heißläufer per Temperaturüberwachung oder zu starke Schwingungen über Druckpulssensoren ermitteln. Bild 4.2 zeigt einmal die fiktiven erfassten Daten eines Wälzlagers mit entsprechenden Führungsgrößen.

Wälzlager.xlsx - Excel

Optimaler Zustand

	Beschreibung	Marke	Axialschub	Drehzahl	Temperatur	Schwingungen
1	Beschreibung					
2	Amboss Wälzlager	Amboss Werkzeuge	50	10	24	1,5
3	Amboss Wälzlager	Amboss Werkzeuge	50	10	24	1,5
4	Amboss Wälzlager	Amboss Werkzeuge	50	10	30	1,2
5	Amboss Wälzlager	Amboss Werkzeuge	80	10	32	1,4
6	Amboss Wälzlager	Amboss Werkzeuge	80	10	38	1,2
7	Amboss Wälzlager	Amboss Werkzeuge	80	10	24	1,4
8	Amboss Wälzlager	Amboss Werkzeuge	50	10	38	1,2
9	Amboss Wälzlager	Amboss Werkzeuge	50	10	24	1
10	Amboss Wälzlager	Amboss Werkzeuge	80	10	38	1,9

Belastung Zustand

	Beschreibung	Marke	Axialschub	Drehzahl	Temperatur	Schwingungen
13	Beschreibung					
14	Amboss Wälzlager	Amboss Werkzeuge	120	25,4	32	2
15	Amboss Wälzlager	Amboss Werkzeuge	134	20	40	2,2
16	Amboss Wälzlager	Amboss Werkzeuge	150	20	40	2,5
17	Amboss Wälzlager	Amboss Werkzeuge	156	12,7	32	2,7
18	Amboss Wälzlager	Amboss Werkzeuge	160	20	40	2,8
19	Amboss Wälzlager	Amboss Werkzeuge	160	20	60	2
20	Amboss Wälzlager	Amboss Werkzeuge	165	20	60	2
21	Amboss Wälzlager	Amboss Werkzeuge	165	40	60	2,2
22	Amboss Wälzlager	Amboss Werkzeuge	175	42	40	2,8

Überlastung Zustand

	Beschreibung	Marke	Axialschub	Drehzahl	Temperatur	Schwingungen
26	Amboss Wälzlager	Amboss Werkzeuge	250	30	80	3,2
27	Amboss Wälzlager	Amboss Werkzeuge	300	30	80	3,2
28	Amboss Wälzlager	Amboss Werkzeuge	315	30	80	3,4
29	Amboss Wälzlager	Amboss Werkzeuge	350	30	80	3,9
30	Amboss Wälzlager	Amboss Werkzeuge	400	30	80	3,7
31	Amboss Wälzlager	Amboss Werkzeuge	500	30	80	3
32	Amboss Wälzlager	Amboss Werkzeuge	600	30	80	3,5
33	Amboss Wälzlager	Amboss Werkzeuge	700	30	80	3,5
34	Amboss Wälzlager	Amboss Werkzeuge	720	30	80	3,5

Bild 4.2 Führungsgrößen bei einem beispielhaften Amboss-Wälzlager

Dieser Service zum Überwachen von Wälzlagern hat sich in der Praxis schon vielfach bewährt. Zum Beispiel bietet die Firma Schaeffler einen solchen Service für Wälzlager in Windkraftanlagen und für Schienenfahrzeuge an. Die Daten werden in einer eigenen Cloud ausgewertet und können per Internet weltweit abgerufen werden.

4.3.1 Problemstellung

Im nachfolgenden Beispiel geht es darum, den Zustand des Wälzlagers anhand seiner Merkmale Axialschub, Drehzahl, Temperatur und Schwingungen in unterschiedliche Gruppen einzuteilen. Für das Beispiel nehmen wir an, dass der gesuchte Zustandtyp in den Daten erst einmal unbekannt ist. Da das Ziel für das neuronale Netz darin besteht, den Zustand des Wälzlagers anhand der vier numerischen Attribute vorherzusagen, müssen diese Daten in drei mögliche Zustände eingeteilt werden. Es gibt demnach den optimalen Zustand, den Belastungszustand und den Überlastungszustand des Wälzlagers.

4.3.2 Vorbereiten der Daten

Der für das Predictive-Maintenance-Beispiel verwendete Datensatz besteht aus insgesamt 90 Sätzen mit jeweils 30 Einträgen für jede der drei Zustandsarten für das Wälz Lager. Da die Werte für Axialschub, Drehzahl und Temperatur vom Wertebereich gegenüber den Schwingungen stark abweichen, werden die Angaben der Führungsgrößen für das neuronale Netz zur Berechnung normalisiert, indem Sie eine Dezimalskalierung durchführen und die Werte durch 10 teilen.



Die Beispieldaten für das neuronale Netz finden Sie in der *WaelzLager.csv*-Datei unter GitHub:

<https://github.com/DanielBasler/NeuralNetwork>

Da auch die Zustandsarten nur numerisch erfasst werden können, verwenden Sie für die drei möglichen Y-Werte auch entsprechende Zahlen. In den Datensätzen wurde der Zustand „Optimal“ als (0,0,1), „Starke Belastung“ als (0,1,0) und „Überlastung“ als (1,0,0) codiert. Bild 4.3 zeigt einen Ausschnitt der Daten aus der entsprechenden WälzLager-Datei.

```

F:\Waelzlager.csv - Notepad++
Datei Bearbeiten Suchen Ansicht Kodierung Spracher
Waelzlager.csv
1 5;1;2,4;1,5;0;0;1
2 5;1;2,4;1,5;0;0;1
3 5;1;3;1,2;0;0;1
4 8;1;3,2;1,4;0;0;1
5 8;1;3,8;1,2;0;0;1
6 8;1;2,4;1,4;0;0;1
7 5;1;3,8;1,2;0;0;1
8 5;1;2,4;1;0;0;1
9 8;1;3,8;1,9;0;0;1
10 4,5;1;2,2;1,5;0;0;1
11 7,5;1;2,4;1,2;0;0;1
12 6;1;2,4;1,4;0;0;1
13 6,5;1;3;1,4;0;0;1
14 8;1;3,2;1,4;0;0;1
15 8,2;1;3,2;1,3;0;0;1
16 6,6;1;3,2;1,6;0;0;1
17 7;1;3,8;1,8;0;0;1
18 5,5;1;4;1,2;0;0;1
19 5,6;1;3,8;1,5;0;0;1
20 7;1;3,2;1,5;0;0;1
21 5;1;3,2;1,5;0;0;1
22 5;1;3,2;1,6;0;0;1
23 7;1;3,8;1,6;0;0;1

```

Bild 4.3

Auszug aus der CSV-Datei
für das neuronale Netz

4.3.3 Das neuronale Netz programmieren

Jetzt können Sie damit beginnen, das neuronale Netz für die vorausschauende Wartung für das Wälzlager mit Trainings- und Testphase Schritt für Schritt zu programmieren, und zwar auf Basis des Wissens, das Sie sich über neuronale Netze in Kapitel 3 erarbeitet haben.

Nachdem die Aufgabenstellung festgelegt und die Trainingsdaten angepasst sind, wählt man den passenden Machine-Learning-Algorithmus aus. Da es sich bei der Aufgabe um überwachtes Lernen für eine Mehrklassen-Klassifizierung handelt, bietet sich ein neuronales Netz mit einer Feedforward-Architektur an. Die Datenbasis (Beispieldaten) lässt sich sehr gut inkrementell abarbeiten und somit ist die Aufgabe für den Backpropagation-Algorithmus prädestiniert.

Backpropagation mit Momentum

Der im Beispiel eingesetzte Lernalgorithmus verwendet zur Gewichtswert-Anpassung eine konstante Lernrate und einen Momentum-Term (Momentum-Faktor). Um die beste Netzkonfiguration für die Aufgabenstellung zu ermitteln, soll es möglich sein, die Neuronen-Anzahl im Hidden-Layer, die Lernrate und das Momentum zu variieren.

Hierzu soll eine Benutzeroberfläche diese Parameterwerte aufnehmen, sodass unterschiedliche Einstellmöglichkeiten gegeben sind. Des Weiteren soll es machbar sein, die Anzahl der Trainingsdatensätze sowie den Initialwert für die Gewichte und die Anzahl der Iterationen (Epoche) in Prozent vorzugeben. Zur Berechnung des Netzfehlers wird der mittlere quadra-

tische Fehler (MSE; siehe Kasten) verwendet. Dieser Wert ist auch als Parameter einstellbar. Die Auswahl der Aktivierungsfunktion im Hidden Layer zwischen HyperTan und Sigmoid soll möglich sein.



Mittlerer quadratischer Fehler:

Der mittlere quadratische Fehler (engl. Mean Squared Error, daher mit „MSE“ abgekürzt) ist ein Begriff der mathematischen Statistik. Mit dem MSE kann die Abweichung eines Punktschätzers von dem zu schätzenden Wert berechnet werden. Somit ist MSE eine Risikofunktion, die dem erwarteten Wert des quadratischen Fehlverlustes entspricht [9].

So ist es möglich, über die Kombination aller Parameter das beste Netzergebnis zu ermitteln. Bei einer Verwendung von zum Beispiel 80 % der Trainingsdaten werden die 90 Datensätze aus der CSV-Datei nach dem Zufallsprinzip in eine 72-Punkte-Untermenge (entspricht 80 %) aufgeteilt, die für das Training verwendet werden sollen, und eine 18-Punkte Untermenge (20 %), die für die Testphase verwendet werden.

Über die Datensätze der Testphase wird dann die Wahrscheinlichkeit einer korrekten Klassifikation der Daten vorgenommen, die das Netz noch nicht kennt. Beachten Sie bei der Implementierung, dass in dem Beispiel kein Fehler-Handling in Bezug auf fehlende oder fehlerhafte Parametereingaben gemacht wird.

4.3.4 Benutzeroberfläche

Das zu implementierende neuronale Netz mit dem Backpropagation-mit-Momentum-Algorithmus besteht neben *MainWindow* aus den vier weiteren Klassen *MakeMatrikForNeuralNetwork*, *NeuralMath*, *NeuralNetwork* und *PrepareData*. Dieser Abschnitt skizziert, wie die WPF-Benutzeroberfläche aussehen soll. Somit bildet auch hier wieder die *MainWindow.xaml*-Datei das GUI (*Graphical User Interface*) und die *MainWindow.xaml.cs* die Code-Behind-Datei für das Programm, um den Algorithmus auszuführen.

Als Vorlage wird die WPF-App (.NET Framework) verwendet und als Projektname *Multiclasses-Demo* vergeben. Bild 4.4 zeigt die WPF-Oberfläche des neuronalen Netzes nach dem Start mit den Steuerelementen für Aufnahme und Einstellung der Parameterwerte. Der Aufbau kann über Listing 4.1 direkt im XAML-Editor der Klasse *MainWindow* vorgenommen werden. Sie können die Steuerelemente aber auch anders anordnen.

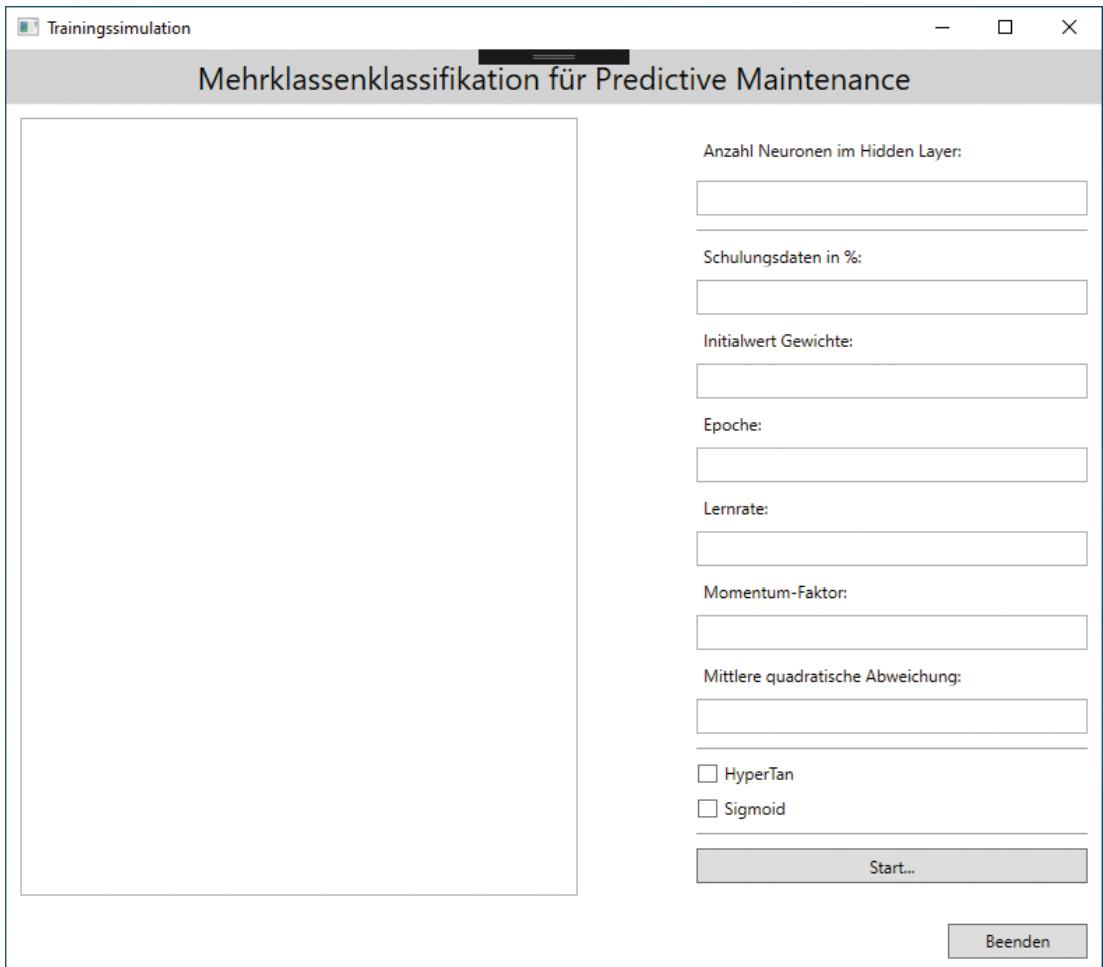


Bild 4.4 Die WPF-Oberfläche des neuronalen Netzes

Listing 4.1 XAML-Code für die WPF-Oberfläche

```
<Window x:Class="MulticlassesDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="Trainingssimulation" Height="700" Width="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
```

```

</Grid.RowDefinitions>
<StackPanel Background="LightGray">
    <Label FontSize="22"
        HorizontalAlignment="Center">Mehrklassenklassifikation für Predictive Maintenance</Label>
</StackPanel>
<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="71*"/>
        <ColumnDefinition Width="11*"/>
        <ColumnDefinition Width="300"/>
    </Grid.ColumnDefinitions>
    <ListBox Name="ListProgramValues" Margin="10" ></ListBox>
    <StackPanel Grid.Column="2" Margin="10">
        <Label Content="Anzahl Neuronen im Hidden Layer:" Height="25" Margin="0 10 0 0" />
        <TextBox Name="teNeuronOfHiddenLayer" Height="25" Margin="0 10 0 0" FontSize="16"/>
        <Separator Margin="0 10 0 0"/>
        <Label Content="Schulungsdaten in %:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teTrainingCourseData" Height="25" Margin="0 5 0 0" />
        <Label Content="Initialwert Gewichte:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teWeights" Height="25" Margin="0 5 0 0" />
        <Label Content="Epoche:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teEpoch" Height="25" Margin="0 5 0 0" />
        <Label Content="Lernrate:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teLearnRate" Height="25" Margin="0 5 0 0" />
        <Label Content="Momentum-Faktor:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teMomentumFactor" Height="25" Margin="0 5 0 0" />
        <Label Content="Mittlere quadratische Abweichung:" Height="25" Margin="0 5 0 0" />
        <TextBox Name="teMeanSquaredError" Height="25" Margin="0 5 0 0" />
        <Separator Margin="0 10 0 0"/>
        <CheckBox Name="cbHyperTan" Margin="0 10 0 0" Content="HyperTan" Checked="CbHyperTan_Checked"/>
        <CheckBox Name="cbSigmoid" Margin="0 10 0 0" Content="Sigmoid" Checked="CbSigmoid_Checked"/>
        <Separator Margin="0 10 0 0"/>
        <Button Name="bStart" Height="25" Margin="0 10 0 0" Click="BStart_Click">Start...</Button>
    </StackPanel>
</Grid>
<StackPanel Grid.Row="3" Orientation="Horizontal" HorizontalAlignment="Right">
    <Button Name="bCancel" Height="25" Width="100" Margin="0 10 10 10" Click="BCancel_Click" >Beenden</Button>
</StackPanel>
</Grid>
</Window>

```

Somit ist die Benutzeroberfläche für die Einstellung der Parameter schon erstellt und Sie können mit der Implementierung der Funktionalitäten für das neuronale Netz beginnen.

4.3.4.1 Code-Behind der MainWindow-Klasse

Im Code-Behind der *MainWindow*-Klasse wird die Programmlogik zum Aufruf für das neuronale Netz implementiert. Da hier gleich zu Beginn auch die Instanzen der weiteren Klassen gebildet werden, sollten Sie als Erstes Ihr Projekt um die benötigten Klassen erweitern. Legen Sie dafür in der Visual Studio Solution die Klassen *MakeMatrixForNeuralNetwork*, *NeuralMath*, *NeuralNetwork* und *PrepareData* an. Der Solution Explorer sollte jetzt so aussehen wie in Bild 4.5.

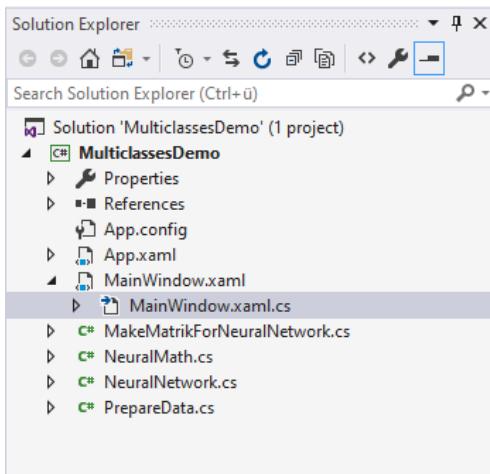


Bild 4.5

Ansicht der Projektstruktur im Solution Explorer

Als Nächstes fügen Sie der *WindowMain*-Klasse die benötigten Felder hinzu. Dafür deklarieren Sie einfach die Variablen aus Listing 4.2.

Listing 4.2 Felder der Klasse WindowMain

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Windows;

namespace MulticlassesDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        NeuralNetwork neuralNetwork = null;
        NeuralMath neuralMath = new NeuralMath();
        PrepareData prepareData = new PrepareData();
        int nNeuronInputLayer = 4;
        int nNeuronOutputLayer = 3;
        int activationFunction = 1;
        int neuronOfHiddenLayer = 0;
```

```

int trainingCourseData = 0;
int initWeights = 0;
int epoch = 0;
double learnRate = 0;
double momentumFactor = 0;
double meanSquaredError = 0;
double[][] allDataItems = new double[89][];
double[][] trainingDataItems = null;
double[][] testingDataItems = null;
List<string> OutputText = new List<string>();

public MainWindow()
{
    InitializeComponent();
}

```

Des Weiteren enthält die Klasse auch die Methoden für die Auswahl der Checkboxen für die Aktivierungsfunktion und das Click-Event zum Beenden des Programms.

Listing 4.3 Event-Handling

```

private void CbHyperTan_Checked(object sender, RoutedEventArgs e)
{
    if (cbHyperTan.IsChecked == true)
    {
        cbSigmoid.IsChecked = false;
        activationFunction = 1;
    }
}

private void CbSigmoid_Checked(object sender, RoutedEventArgs e)
{
    if (cbSigmoid.IsChecked == true)
    {
        cbHyperTan.IsChecked = false;
        activationFunction = 2;
    }
}

private void BCancel_Click(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}

```

Der Event-Handler *BStart_Click* enthält erst einmal die Zuweisungen der Steuerelemente zu den dazugehörigen Variablen mit entsprechender Typumwandlung.

Listing 4.4 Der Event-Handler BStart_Click

```

private void BStart_Click(object sender, RoutedEventArgs e)
{
    OutputText.Add("Start...");
    //No error handling of missing entries
    neuron0fHiddenLayer = Convert.ToInt16(teNeuron0fHiddenLayer.Text);
    OutputText.Add("Input-Neuronen = 4");
    OutputText.Add("Hidden-Neuronen = " + teNeuron0fHiddenLayer.Text);
    OutputText.Add("Output-Neuronen = 3");
    trainingCourseData = Convert.ToInt16(teTrainingCourseData.Text);
    OutputText.Add("Traningsdatenanteil = " + teTrainingCourseData.Text
        + " %");
    initWeights = Convert.ToInt16(teWeights.Text);
    OutputText.Add("Initialisierungswert der Gewichte = " + teWeights.Text);
    epoch = Convert.ToInt16(teEpoch.Text);
    OutputText.Add("Epoche = " + teEpoch.Text);
    learnRate = Convert.ToDouble(teLearnRate.Text);
    OutputText.Add("Lernrate = " + teLearnRate.Text);
    meanSquaredError = Convert.ToDouble(teMeanSquaredError.Text);
    OutputText.Add("Mittlere quadratische Abweichung = "
        + teMeanSquaredError.Text);
    momentumFactor = Convert.ToDouble(teMomentumFactor.Text);
    OutputText.Add("Momentum-Faktor = " + teMomentumFactor.Text);

    if(activationFunction == 1)
    {
        OutputText.Add("Aktivierungsfunktion = HyperTan");
    }
    else
    {
        OutputText.Add("Aktivierungsfunktion = Sigmoid");
    }

    OutputText.Add("");
    OutputText.Add("Lese CSV Daten...");
    ListProgramValues.ItemsSource = OutputText;

    ReadingDataset();
}

```

Die Methode *ReadingDataset* wird verwendet, um die Datensätze aus der CSV-Datei einzulesen. Die Methode liest über *File.ReadAllLines* die Trainingsdaten ein und weist sie dem *double*-Array *allDataItems* zu.

Listing 4.5 Die Methode ReadingDataset

```

private void ReadingDataset()
{
    try
    {
        if (File.Exists(@"c://temp//waelzlager.csv"))
        {

```

```

        int index = 0;
        foreach (var line in File.ReadAllLines(@"c://temp//waelzlager.csv"))
        {
            try
            {
                var splittedLine = line.Split(';');
                allDataItems[index] = new double[]
                {
                    Convert.ToDouble(splittedLine[0]),
                    Convert.ToDouble(splittedLine[1]),
                    Convert.ToDouble(splittedLine[2]),
                    Convert.ToDouble(splittedLine[3]),
                    Convert.ToDouble(splittedLine[4]),
                    Convert.ToDouble(splittedLine[5]),
                    Convert.ToDouble(splittedLine[6])
                };
                index++;
            }
            catch (Exception) { } // ToDo
        }
    }
    catch (Exception e)
    {
        //ToDo
    }
}

```

4.3.4.2 Nutzen der Hold-Out Validation

Über die Methode *ReadingDataset* ist jetzt das benötigte Dataset an Trainingsdaten für das neuronale Netz verfügbar. Jetzt implementieren Sie die Hold-Out-Validation-Funktion, um die verfügbaren Daten in Trainings- und Testdaten aufzuteilen. Hierfür nutzen Sie die Klasse *Preparedata* (Vorbereiten von Daten).

Listing 4.6 Die Klasse PrepareData

```

using System;

namespace MulticlassesDemo
{
    public class PrepareData
    {
        double[][] trainingDataItems = null;
        double[][] testingDataItems = null;

        public void PreparingTraningData(double[][] allDataItems,
                                         int seed, int traingCourseData)
        {
            decimal courseData = Convert.ToDecimal(traingCourseData) / 100;
            Random rnd = new Random(seed);
        }
    }
}

```

```
int totalRows = allDataItems.Length;
int numberCols = allDataItems[0].Length;

int trainingRows = (int)(totalRows * courseData);
int testingRows = totalRows - trainingRows;

trainingDataItems = new double[trainingRows][];
testingDataItems = new double[testingRows][];
double[][] copy = new double[allDataItems.Length][];

for (int i = 0; i < copy.Length; ++i)
{
    copy[i] = allDataItems[i];

}

for (int i = 0; i < copy.Length; ++i)
{
    int r = rnd.Next(i, copy.Length);
    double[] tmp = copy[r];
    copy[r] = copy[i];
    copy[i] = tmp;
}

for (int i = 0; i < trainingRows; ++i)
{
    trainingDataItems[i] = new double[numberCols];

    for (int j = 0; j < numberCols; ++j)
    {
        trainingDataItems[i][j] = copy[i][j];
    }
}

for (int i = 0; i < testingRows; ++i)
{
    testingDataItems[i] = new double[numberCols];

    for (int j = 0; j < numberCols; ++j)
    {
        testingDataItems[i][j] = copy[i + trainingRows][j];
    }
}

public double[][] GetItemsOfTrainingData()
{
    return trainingDataItems;
}

public double[][] GetItemsOfTestingData()
{
    return testingDataItems;
}
}
```

Die Methode *PreparingTrainingData* in der Klasse *Preparedata* funktioniert wie folgt. Als Erstes enthält sie eine Matrix des Datasets, also alle Daten der CSV-Datei, und legt hierfür eine Matrix von Trainingsdaten und Testdaten an. Über den *seed*-Parameter wird der Startwert für das Random-Objekt für das zufällige Zusammenstellen der Daten übergeben.

Der Parameter *trainingCourseData* in der Methode *PreparingTrainingData* legt die Anzahl der Trainingssätze fest. Dann werden die Anzahl der Zeilen für die Trainingsmatrix und für die Testmatrix berechnet. Anschließend wird eine Referenzkopie von *allDataItems* erstellt und die Reihenfolge der Datensätze durcheinandergewürfelt. Ist die Referenzkopie erstellt, werden die Datensätze der Ausgangskopie in der veränderten Reihenfolge in das Array für die Trainingsdaten kopiert. Dasselbe passiert auch mit den Testdaten.

Bei der verwendeten Methode *PreparingTrainingData* handelt es sich um den Fisher-Yates-Shuffle-Algorithmus. Dieser Algorithmus ermöglicht das einfache Randomisieren von Arrays durch die Veränderung der Elementreihenfolge. Das heißt, das *double*-Array wird durch die *for*-Schleife neu angeordnet, sodass es sich nicht mehr in seiner ursprünglichen Reihenfolge befindet. Über die Methoden *GetItemsOfTrainingData* und *GetItemsOfTestingData* kann man die benötigten Arrays abrufen. Für das Anlegen der Trainings- und Testdaten zur Laufzeit erweitern Sie den *BStart_Click*-Event-Handler in der *MainWindow*-Klasse.

Listing 4.7 Aufruf der Methoden in der PrepareData-Klasse

```
prepareData.PreparingTraningData(allDataItems, initWeights, trainingCourseData);
trainingDataItems = prepareData.GetItemsOfTrainingData();
testingDataItems = prepareData.GetItemsOfTestingData();
```

Es werden auch entsprechend die Arrays *trainingDataItems* und *testingDataItems* mit den Matrixinformationen gefüllt. Bevor Sie mit der Implementierung der Klasse *NeuralNetwork* beginnen, erstellen Sie vorher noch die Klasse *MakeMatrixForNeuralNetwork*.

Listing 4.8 Klasse zum Erzeugen einer Matrix

```
namespace MulticlassesDemo
{
    public static class MakeMatrikForNeuralNetwork
    {
        public static double[][] CreateMatrix(int rows, int cols)
        {
            double[][] result = new double[rows][];
            for (int r = 0; r < result.Length; ++r)
            {
                result[r] = new double[cols];
            }
            return result;
        }
    }
}
```

Die Methode *CreateMatrix* gibt eine Matrix (Zeile, Spalte) zurück, die für entsprechende Rechenoperationen im neuronalen Netz benötigt wird.

4.3.5 Programmablauf

Ehe Sie jetzt die Klasse *NeuralNetwork* in Angriff nehmen, skizzieren wir noch einmal kurz, wie das neuronale Netz funktioniert und die Trainingsphase durchlaufen soll.

Im ersten Schritt wird die Initialisierung des Netzes mit Zufallszahlen durchgeführt. Durch diesen Vorgang stellen Sie sicher, dass alle Gewichte und Bias-Werte nun einen Wert zwischen null und eins tragen.

Im zweiten Schritt wird der Trainingsdatensatz x ins Netz eingespeist. Als dritter Schritt folgt dann der Feedforward-Pass. Aus den Eingabewerten des Datensatzes x können jetzt mithilfe der Gewichte die kumulierten Werte durch das Bias und die Aktivierungsfunktion die Ausgabewerte jedes Neurons Layer für Layer bis zum Output berechnet werden.

Im vierten Schritt wird dann die Berechnung der gewünschten Kostenfunktion vorgenommen. Da Gewichte und Bias im ersten Schritt als Zufallszahlen initialisiert worden sind, ist dieser Fehlerwert im ersten Durchlauf sehr hoch.

Im fünften Schritt wird der Backward-Pass durchlaufen. Der errechnete Fehlerwert aus Schritt vier wird wieder anteilig auf die Gewichte und die Bias-Werte im neuronalen Netz verteilt. Der Prozess läuft vom Output und dann angefangen bei der letzten Schicht im Hidden Layer über alle Schichten bis zum Input.

Schritt sechs aktualisiert die Netzparameter unter Berücksichtigung der Lernrate und dem Momentum-Faktor und Schritt sieben startet einen neuen Durchlauf mit einem neuen Datensatz aus den Trainingsbeispielen und aktualisierten Gewichten und Bias-Werten. Diese sieben Schritte bilden somit das Grundgerüst für die Trainingsimplementierung in der Klasse *NeuralNetwork*.

4.3.6 Das neuronale Netz implementieren

Beginnen Sie also mit der Implementierung der Klasse *NeuralNetwork*. Als Erstes werden die benötigten Felder für die Klasse definiert.

Listing 4.9 Felder der Klasse NeuralNetwork

```
using System;

namespace MulticlassesDemo
{
    public class NeuralNetwork
    {
        NeuralMath neuralMath = new NeuralMath();
        private static Random rnd;
```

```

    public static int activationFunction;

    public static int inputNodes;
    public static int hiddenNodes;
    public static int outputNodes;

    public static double[] inputs;
    public static double[][] inputToHiddenWeights;

    public static double[] hiddenBiases;
    public static double[] hiddenOutputs;
    public static double[][] hiddenToOutputWeights;

    public static double[] outputBiases;
    public static double[] outputs;
    public static double[] outputGradients;

    public static double[] hiddenGradients;
    public static double[][] inputHiddenPrevWeightsDelta;
    public static double[] hiddenPrevBiasesDelta;
    public static double[][] hiddenOutputPrevWeightsDelta;
    public static double[] outputPrevBiasesDelta;

}

}

```

Im Konstruktor der Klasse wird festgelegt, wie viele Knoten im Input Layer, im Hidden Layer und im Output Layer vorhanden sind. Die Angaben definieren die Gestalt und Größe des neuronalen Netzes. Die Anzahl der Neuronen im Hidden Layer lassen sich über den Parameter in der Benutzeroberfläche festlegen. Auch die Art der Aktivierungsfunktion - *HyperTan* oder *Sigmoid* - wird entsprechend festgelegt. Die Ermittlung der Anzahl der Neuronen, die im Hidden Layer verwendet werden, ist eine knifflige Angelegenheit im neuronalen Netz. Die Auswahl eines passenden Wertes für die Anzahl der Neuronen ist meist eine Frage von Versuch und Irrtum. Daher ist es auf jeden Fall sinnvoll, diesen Wert als Parameter zu verwenden, um ihn bei jeder Trainingssimulation entsprechend anpassen zu können.

Listing 4.10 Der Konstruktor der Klasse NeuralNetwork

```

public NeuralNetwork(int nInput, int nHidden, int nOutput, int act)
{
    rnd = new Random(0);
    activationFunction = act;

    inputNodes = nInput;
    hiddenNodes = nHidden;
    outputNodes = nOutput;

    inputs = new double[inputNodes];
    inputToHiddenWeights = MakeMatrikForNeuralNetwork.
        CreateMatrix(inputNodes, hiddenNodes);
}

```

```

hiddenBiases = new double[hiddenNodes];
hiddenOutputs = new double[hiddenNodes];
hiddenToOutputWeights = MakeMatrikForNeuralNetwork.
    CreateMatrix(hiddenNodes, outputNodes);

outputBiases = new double[outputNodes];
outputs = new double[outputNodes];

InitializeWeights();

hiddenGradients = new double[hiddenNodes];
outputGradients = new double[outputNodes];
inputHiddenPrevWeightsDelta = MakeMatrikForNeuralNetwork.
    CreateMatrix(inputNodes, hiddenNodes);

hiddenPrevBiasesDelta = new double[nHidden];
hiddenOutputPrevWeightsDelta = MakeMatrikForNeuralNetwork.
    CreateMatrix(hiddenNodes, outputNodes);

outputPrevBiasesDelta = new double[outputNodes];
}

```

Die aufzurufende Methode *InitializeWeights* sorgt für die erste Initialisierung der Gewichte und Bias-Werte im neuronalen Netz.

Listing 4.11 Die Methoden InitializeWeights und SetWeights

```

private void InitializeWeights()
{
    int numberWeights = (inputNodes * hiddenNodes)
        + (hiddenNodes * outputNodes)
        + hiddenNodes + outputNodes;

    double[] initialWeights = new double[numberWeights];
    double lo = -0.01;
    double hi = 0.01;

    for (int i = 0; i < initialWeights.Length; ++i)
    {
        initialWeights[i] = (hi - lo) * rnd.NextDouble() + lo;
    }

    SetWeights(initialWeights);
}

private void SetWeights(double[] initialWeights)
{
    int numberWeights = (inputNodes * hiddenNodes)
        + (hiddenNodes * outputNodes)
        + hiddenNodes + outputNodes;
}

```

```

        if (initialWeights.Length != numberWeights)
            throw new Exception("Fehlerhafte Array-Länge!");

        int z = 0;

        for (int i = 0; i < inputNodes; ++i)
        {
            for (int j = 0; j < hiddenNodes; ++j)
            {
                inputToHiddenWeights[i][j] = initialWeights[z++];
            }
        }

        for (int i = 0; i < hiddenNodes; ++i)
        {
            hiddenBiases[i] = initialWeights[z++];
        }

        for (int i = 0; i < hiddenNodes; ++i)
        {
            for (int j = 0; j < outputNodes; ++j)
            {
                hiddenToOutputWeights[i][j] = initialWeights[z++];
            }
        }

        for (int i = 0; i < outputNodes; ++i)
        {
            outputBiases[i] = initialWeights[z++];
        }
    }
}

```

Somit ist die Instanziierung des neuronalen Netzes abgeschlossen. Da jetzt in der Klasse *NeuralNetwork* alle benötigten Felder definiert sind, können Sie nun die Klasse *NeuralMath* implementieren, um alle Berechnungen und Aktivierungsfunktionen in eine eigene Klasse auszulagern. Die Klasse *NeuralMath* beinhaltet alle benötigten Methoden für die Berechnung im neuronalen Netz.

Listing 4.12 Die Klasse NeuralMath

```

using System;

namespace MulticlassesDemo
{
    public class NeuralMath
    {
        public double MeanSquaredError(double[][] trainingDataItems, int inputNodes,
                                       int hiddenNodes, int outputNodes)
        {
            double sumSquaredError = 0.0;
            double[] xValues = new double[inputNodes];
            double[] tValues = new double[outputNodes];
        }
    }
}

```

```
for (int i = 0; i < trainingDataItems.Length; ++i)
{
    Array.Copy(trainingDataItems[i], xValues, inputNodes);
    Array.Copy(trainingDataItems[i], inputNodes, tValues, 0, outputNodes);
    double[] yValues = CalculateExpenses(xValues);

    for (int j = 0; j < outputNodes; ++j)
    {
        double err = tValues[j] - yValues[j];
        sumSquaredError += err * err;
    }
}

return sumSquaredError / trainingDataItems.Length;
}

public double[] CalculateExpenses(double[] xValues)
{
    if (xValues.Length != NeuralNetwork.inputNodes)
    {
        throw new Exception("xValues Array haben eine fehlerhafte Länge.");
    }

    double[] hSums = new double[NeuralNetwork.hiddenNodes];
    double[] oSums = new double[NeuralNetwork.outputNodes];

    for (int i = 0; i < xValues.Length; ++i)
    {
        NeuralNetwork.inputs[i] = xValues[i];
    }

    for (int j = 0; j < NeuralNetwork.hiddenNodes; ++j)
    {
        for (int i = 0; i < NeuralNetwork.inputNodes; ++i)
        {
            hSums[j] += NeuralNetwork.inputs[i]
                * NeuralNetwork.inputToHiddenWeights[i][j];
        }
    }

    for (int i = 0; i < NeuralNetwork.hiddenNodes; ++i)
    {
        hSums[i] += NeuralNetwork.hiddenBiases[i];
    }

    for (int i = 0; i < NeuralNetwork.hiddenNodes; ++i)
    {
        if(NeuralNetwork.activationFunction == 1)
        {
            NeuralNetwork.hiddenOutputs[i] = HyperTan(hSums[i]);
        }
        else
        {

```

```

        NeuralNetwork.hiddenOutputs[i] = sigmoid(hSums[i]);
    }
}

for (int j = 0; j < NeuralNetwork.outputNodes; ++j)
{
    for (int i = 0; i < NeuralNetwork.hiddenNodes; ++i)
    {
        oSums[j] += NeuralNetwork.hiddenOutputs[i]
                    * NeuralNetwork.hiddenToOutputWeights[i][j];
    }
}

for (int i = 0; i < NeuralNetwork.outputNodes; ++i)
{
    oSums[i] += NeuralNetwork.outputBiases[i];
}

double[] softOut = Softmax(oSums);
Array.Copy(softOut, NeuralNetwork.outputs, softOut.Length);
double[] retResult = new double[NeuralNetwork.outputNodes];
Array.Copy(NeuralNetwork.outputs, retResult, retResult.Length);

return retResult;
}

private double[] Softmax(double[] oSums)
{
    double max = oSums[0];

    for (int i = 0; i < oSums.Length; ++i)
    {
        if (oSums[i] > max) max = oSums[i];
    }

    double scale = 0.0;

    for (int i = 0; i < oSums.Length; ++i)
    {
        scale += Math.Exp(oSums[i] - max);
    }

    double[] result = new double[oSums.Length];

    for (int i = 0; i < oSums.Length; ++i)
    {
        result[i] = Math.Exp(oSums[i] - max) / scale;
    }

    return result;
}

private double HyperTan(double activation)
{
    if (activation < -20.0)

```

```
{  
    return -1.0;  
}  
else if (activation > 20.0)  
{  
    return 1.0;  
}  
else return  
    Math.Tanh(activation);  
}  
  
private static double sigmoid(double activation)  
{  
    return 1 / (1 + Math.Exp(-activation));  
}  
  
public double Forecast(double[][] testingDataItems,  
                      int inputNode, int outputNode)  
{  
    int correct = 0;  
    int incorrect = 0;  
  
    double[] xValues = new double[inputNode];  
    double[] tValues = new double[outputNode];  
    double[] yValues;  
  
    for (int i = 0; i < testingDataItems.Length; ++i)  
    {  
        Array.Copy(testingDataItems[i], xValues, inputNode);  
        Array.Copy(testingDataItems[i], inputNode, tValues, 0, outputNode);  
  
        yValues = CalculateExpenses(xValues);  
        int maxIndex = MaxIndex(yValues);  
  
        if (tValues[maxIndex] == 1.0)  
        {  
            ++correct;  
        }  
        else  
        {  
            ++incorrect;  
        }  
    }  
  
    return (correct * 1.0) / (correct + incorrect);  
}  
  
public int MaxIndex(double[] yValues)  
{  
    int bigIndex = 0;  
    double biggestVal = yValues[0];  
  
    for (int i = 0; i < yValues.Length; ++i)  
    {  
        if (yValues[i] > biggestVal)
```

```
        {
            biggestVal = yValues[i];
            bigIndex = i;
        }
    }

    return bigIndex;
}
}
```

Nach der Implementierung sind die mathematisch benötigten Methoden vorhanden, um mit der Umsetzung der Trainingsmethode für das Netz anfangen zu können.

Die Methode Training

Als Nächstes wird die Methode *Training* mit Leben gefüllt. Dieser Code ist etwas komplexer und umfasst auch noch eine weitere Methode *UpdateWeights* für das Update der Gewichte und Bias-Werte sowie einen weiteren Fisher-Yates-Shuffle-Algorithmus mit dem Methodennamen *RandomPlay*, der die Trainingsdaten in zufälliger Reihenfolge aufbaut.

Listing 4.13 Implementierung der Methode Training

```
public void Training(double[][][] trainingDataItems, int maxEpochs, double learnRate,
double momentumFactor, double mSquaredError)
{
    int epoch = 0;

    double[] xValues = new double[inputNodes];
    double[] tValues = new double[outputNodes];

    int[] sequence = new int[trainingDataItems.Length];

    for (int i = 0; i < sequence.Length; ++i)
    {
        sequence[i] = i;
    }

    while (epoch < maxEpochs)
    {
        double mse = neuralMath.MeanSquaredError(trainingDataItems,
                                                   inputNodes, hiddenNodes, outputNodes);
        if (mse < mSquaredError) break;

        RandomPlay(sequence);

        for (int i = 0; i < trainingDataItems.Length; ++i)
        {
            int idx = sequence[i];
            Array.Copy(trainingDataItems[idx], xValues, inputNodes);
            Array.Copy(trainingDataItems[idx], inputNodes, tValues, 0,
                      outputNodes);
        }
    }
}
```

```

        neuralMath.CalculateExpenses(xValues);
        UpdateWeights(tValues, learnRate, momentumFactor);
    }

    ++epoch;
}
}

private void UpdateWeights(double[] tValues, double learnRate, double momentum)
{
    if (tValues.Length != outputNodes)
    {
        throw new Exception("Zielwerte haben nicht die gleiche Länge wie die Output-Neuronen in der Methode UpdateWeights");
    }

    for (int i = 0; i < outputNodes; ++i)
    {
        double derivative = (1 - outputs[i]) * outputs[i];
        outputGradients[i] = derivative * (tValues[i] - outputs[i]);
    }

    for (int i = 0; i < hiddenNodes; ++i)
    {
        double derivative = (1 - hiddenOutputs[i]) * (1 + hiddenOutputs[i]);
        double sum = 0.0;

        for (int j = 0; j < outputNodes; ++j)
        {
            double x = outputGradients[j] * hiddenToOutputWeights[i][j];
            sum += x;
        }

        hiddenGradients[i] = derivative * sum;
    }

    for (int i = 0; i < inputNodes; ++i)
    {
        for (int j = 0; j < hiddenNodes; ++j)
        {
            double delta = learnRate * hiddenGradients[j] * inputs[i];
            inputToHiddenWeights[i][j] += delta;

            inputToHiddenWeights[i][j] += momentum
            * inputHiddenPrevWeightsDelta[i][j];

            inputHiddenPrevWeightsDelta[i][j] = delta;
        }
    }

    for (int i = 0; i < hiddenNodes; ++i)
    {
        double delta = learnRate * hiddenGradients[i];
    }
}
}

```

```

        hiddenBiases[i] += delta;
        hiddenBiases[i] += momentum * hiddenPrevBiasesDelta[i];

        hiddenPrevBiasesDelta[i] = delta;
    }

    for (int i = 0; i < hiddenNodes; ++i)
    {
        for (int j = 0; j < outputNodes; ++j)
        {
            double delta = learnRate * outputGradients[j] * hiddenOutputs[i];

            hiddenToOutputWeights[i][j] += delta;

            hiddenToOutputWeights[i][j] += momentum
                * hiddenOutputPrevWeightsDelta[i][j];

            hiddenOutputPrevWeightsDelta[i][j] = delta;
        }
    }

    for (int i = 0; i < outputNodes; ++i)
    {
        double delta = learnRate * outputGradients[i] * 1.0;

        outputBiases[i] += delta;

        outputBiases[i] += momentum * outputPrevBiasesDelta[i];

        outputPrevBiasesDelta[i] = delta;
    }
}

private void RandomPlay(int[] sequence)
{
    for (int i = 0; i < sequence.Length; ++i)
    {
        int r = rnd.Next(i, sequence.Length);
        int tmp = sequence[r];
        sequence[r] = sequence[i];
        sequence[i] = tmp;
    }
}
}

```

Die Methode trainiert das neuronale Netz durch ihren Aufbau als Mehrklassen-Klassifikator unter Verwendung der Lernrate und dem Momentum-Faktor für die dynamische Anpassung bei den Gewichten und Bias-Werten.

Über den Parameter *maxEpochs* wird die Anzahl der Iterationen für den Trainingsprozess übergeben. Die Werte *learnRate* und *momentumFactor* steuern die Geschwindigkeit, mit der die Rückwärtsausbreitung, also der Backward-Pass, zu einem endgültigen Satz von Gewichts- und Bias-Werten konvergiert. Fällt in der Trainingsphase der Fehlerschwellenwert für die Variable *mSquaredError*, die als Parameter übergeben wird, unter den mittleren quadratischen

Fehler (MSE), so wird das Training beendet. Der mittlere quadratische Fehler (MSE) wird über die Methode *MeanSquaredError* aus der Klasse *NeuralMath* berechnet.

Die *while*-Schleife in der Methode hat zwei Ausgangsbedingungen. Die erste Bedingung wird erfüllt, wenn die Variable *epoch* größer als die vorgegebene Variable *maxEpochs* ist und die zweite Ausgangsbedingung prüft, ob die Variable *mse*, der mittlere quadratische Fehler, größer ist als der übergebene Wert der Variablen *mSquaredError*. Innerhalb der *while*-Schleife befindet sich eine *for*-Schleife, die über jedes Element in den Trainingsdaten iteriert und durch den Shuffle-Algorithmus in der Methode *RandomPlay* jedes Trainingselement in eine zufällige Reihenfolge bringt. In der Methode *neuralMath.CalculateExpenses* wird dann die benötigte Kostenfunktion für den Fehlerwert berechnet und die ausgewählte Aktivierungsfunktion für den Hidden Layer verwendet. In den Neuronen im Output Layer wird die Softmax-Aktivierungsfunktion verwendet. Nach dem Ausführen der Kostenfunktion werden die Gewichte und Bias-Werte über die Methode *UpdateWeights* angepasst.

In der Methode *UpdateWeights* werden als Erstes die Output-Gradienten berechnet, in der zweiten *for*-Schleife die Gradienten im Hidden Layer und in der dritten *for*-Schleife werden dann die Gewichte in Verbindung mit der Lernrate und dem Momentum-Faktor neu berechnet und in einem Array für die Weiterverarbeitung gespeichert.

Die beiden folgenden *for*-Schleifen errechnen die neuen Bias-Werte im Hidden Layer und nachfolgend dann auch die Gewichte. Zum Schluss erfolgt die Berechnung für den Bias-Wert im Output Layer. Somit ist jetzt die Implementierung der Trainingsmethode für das neuronale Netz abgeschlossen. Die Initialisierung und der Trainingsaufruf erfolgen in der *MainWindow*-Klasse im *BStart_Click*-Event-Handler, siehe Listing 4.14.

Listing 4.14 Initialisierung und Trainingsaufruf

```
neuralNetwork = new NeuralNetwork(nNeuronInputLayer, neuronOfHiddenLayer,
nNeuronOutputLayer, activationFunction);
neuralNetwork.Training(trainingDataItems, epoch, learnRate, momentumFactor,
meanSquaredError);
OutputText.Add("");
OutputText.Add("Training durchgeführt.");
ListProgramValues.Items.Refresh();
```

Die Trainingsdaten sind in dem Beispielprojekt im Array *trainingDataItems* als Trainingsmatrix gespeichert. Diese Matrix kann jetzt für die Auswertung des neuronalen Netzes herangezogen werden.

4.3.7 Auswertung ermitteln

Das Hauptziel der Beispielimplementierung ist es zu untersuchen, ob sich mit dem neuronalen Netz ein sehr hoher Prozentsatz für die Vorhersage ermitteln lässt, ob ein verarbeiteter Datensatz einen „optimalen“, „belasteten“ oder „überlasteten“ Zustand für das Wälzlager beschreibt.

Das heißt, nachdem der Klassifikator des neuronalen Netzes trainiert wurde und so einen Satz von optimalen Gewichts- und Bias-Werten hervorgebracht hat, wird im nächsten Schritt bestimmt, wie exakt das neuronale Netz mit den Testdaten funktioniert.

Die Auswertung hierfür erfolgt über die Methode *Forecast* in der Klasse *NeuralMath*, die Sie schon implementiert haben. Hierbei erfolgt die Berechnung im *BStart_Click*-Event-Handler einmal für die Trainingsdaten und einmal für die Testdaten.

Listing 4.15 Aufruf der Methode Forecast

```
double trainingForecast = neuralMath.Forecast(trainingDataItems,
                                              nNeuronInputLayer, nNeuronOutputLayer);
OutputText.Add("Genauigkeit der Trainingsdaten = "
              + Convert.ToString(trainingForecast));

OutputText.Add("");
double testingForecast = neuralMath.Forecast(testingDataItems,
                                              nNeuronInputLayer, nNeuronOutputLayer);

OutputText.Add("Genauigkeit der Testdaten = " + Convert.ToString(testingForecast));

OutputText.Add("");
OutputText.Add("Berechnung beendet... ");
ListProgramValues.Items.Refresh();
```

Die Methode ordnet die übergebenen Datensätze wieder in einer Referenzkopie an und errechnet dann über die Kostenfunktion das Neuron im Output, das am stärksten auf den Eingabewert reagiert. Das heißt, die Neuronen im Output Layer konkurrieren miteinander. Das Neuron, das die größte Übereinstimmung seines Gewichtswertes mit dem Eingabemuster feststellt, gewinnt und ist somit 1, alle anderen Neuronen verlieren und sind 0. Diese Technik bezeichnet man auch als Ermittlung des Siegerneurons bzw. als Winner-Takes-All. Um die Genauigkeit in der Methode zu verbessern, wird die Methode *MaxIndex* verwendet, um die Position des größten berechneten Y-Wertes zu finden.

■ 4.4 Simulationsergebnis

Sind alle Klassen für das Beispiel implementiert, können Sie das neuronale Netz starten und mit der Auswertung des Simulationsergebnisses beginnen.

Wie Sie bei der Anwendung des neuronalen Netzes zur Klassifizierung des Zustandes ersehen können, erreichte es, unter Verwendung von

- 5.000 durchgeführten Berechnungen,
 - 9 Neuronen im Hidden-Layer,
 - einer Lernrate von 0,05 und einem Momentum-Faktor von 0,01,
 - einem mittleren quadratischen Fehlerwert von 0,04
 - und der Verwendung der HyperTan-Aktivierungsfunktion in der Zwischenschicht
- eine Genauigkeit bei den Trainingsdaten von 0,98... und bei den Testdaten von 0,944. Bild 4.6 zeigt das errechnete Ergebnis mit den entsprechenden Netzparametern.

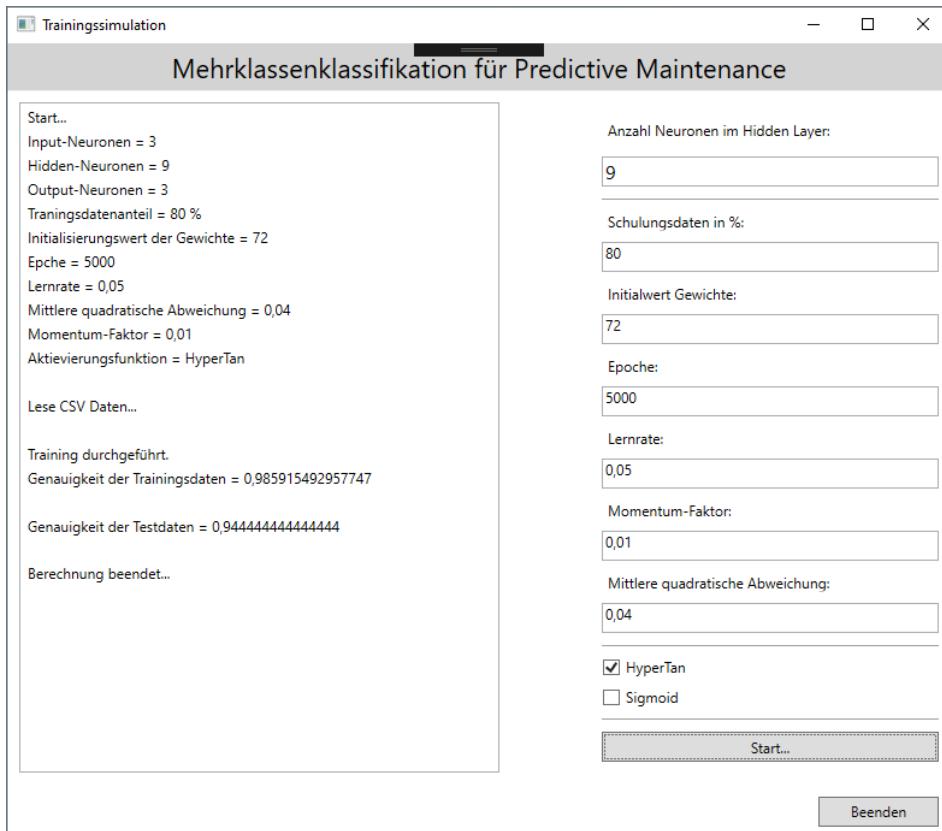


Bild 4.6 Das Simulationsergebnis

Die Trefferquote liegt bei 5.000 Epochen mit 0,9859 bei fast 99 %. Das ist ein wirklich sehr gutes Ergebnis für dieses kleine neuronale Netz mit Backpropagation-Algorithmus unter Zuhilfenahme des Momentum-Faktors.

Dieses Ergebnis kann man aber nur erzielen, wenn man sich beim Testen des Netzes mit den Netzparametern beschäftigt und mit jedem neuen Testdurchlauf eine Optimierung der Parameter durchführt, um ein besseres Ergebnis zu erzielen.



Hyperparameter

Die über die Benutzeroberfläche erstellten Parameter für das neuronale Netz definieren die Hyperparameter, weil sie die Eigenschaften des Modells bestimmen (siehe Abschnitt 2.3, „Die Schritte in einem Machine-Learning-Projekt“).

■ 4.5 Parameteranpassungen

Die Erfolgsquote von fast 98 % bei den vorliegenden Datensätzen und den aufgezeigten Parametern kann sich in der Praxis schon sehr gut sehen lassen. Allerdings muss man sich, um ein solches Ergebnis zu erzielen, an die Netzparameter herantasten. Sie beginnen in Ihrem Projekt ja ohne eine Vorgabe der Parameterwerte. Das heißt, Sie müssen mit den Parameterwerten experimentieren, um ein entsprechend gutes Ergebnis zu erzielen. So ist zum Beispiel das Optimieren der Lernrate sehr wichtig. Das heißt, als erste Verbesserung könnte man versuchen, die Lernrate anzupassen. So könnte man mit einer Lernrate von 0,02 beginnen und diese dann in einzelnen Schritten weiter anpassen, ohne die anderen Werte zu verändern. So liegt die Erfolgsquote bei einer Lernrate von 0,02 bei 97 % (schon nicht schlecht), aber bei einer Lernrate von 0,2 liegt sie plötzlich nur noch bei 0,3521, also bei 35 %, und ist somit viel schlechter als bei einer sehr niedrigen Lernrate. Offenbar führt eine zu große Lernrate zu einem Überschwingen (engl. Overshooting) während des Gradientenabstiegs.

Setzt man zudem noch den Momentum-Faktor mit ein, so steuern die Werte Lernrate und Momentum die Geschwindigkeit, mit der der Backward-Pass im Backpropagation-Algorithmus arbeitet. Wenn Sie mit beiden Werten experimentieren, werden Sie sehr schnell feststellen, dass der Backward-Pass äußerst empfindlich auf Veränderungen dieser beiden Werte reagiert.

Um die Erfolgsquote eines neuronalen Netzes zu verbessern, können Sie auch das Training mit den Trainingsdaten mehrfach wiederholen. Spielen Sie hierfür mit dem Epoche-Wert in der Benutzeroberfläche. Bei einer minimalen Trainingssitzung von nur 10 Durchläufen sackt die Erfolgsquote stark ab. Je mehr Möglichkeiten jedoch der Gradientenabstieg bekommt, seinen kleinsten Abstieg zu erreichen, umso besser wird das Ergebnis im neuronalen Netz.

Sie können ein neuronales Netz auch optimieren, indem Sie versuchen, die Form des Netzes zu ändern, indem Sie z. B. die Anzahl der Neuronen im Hidden Layer verändern. Welche Auswirkungen hat diese Änderung auf das neuronale Netz?

Die Anzahl der versteckten Schichten ist in neuronalen Netzen nicht begrenzt, denn dort findet das Lernen statt. In der Praxis ist es häufig der Fall, dass bei einer Erhöhung der Anzahl der versteckten Neuronen die Ergebnisse besser werden. Allerdings ist in unserem Beispiel aufgrund der kleinen Menge von Trainingsdaten die Auswirkung auf das Ergebnis nicht sehr groß. Bei einer hohen Anzahl von Neuronen dauert es aber erheblich länger, das Netz zu trainieren.

Es gibt es eine Vielzahl von Möglichkeiten, ein neuronales Netz zu optimieren. Sie müssen sich in einzelnen Schritten an ein Optimum herantasten, das für Sie vom Entwicklungsaufwand her auch noch akzeptabel ist.

5

Recurrent Neural Networks

Bisher wurden für die Klassifizierung von Objekten nur Feedforward-Netze (FNN) betrachtet, also neuronale Netze, deren Neuronen in klar abgetrennte Schichten eingeteilt sind. Sie verfügen somit über die bekannten Input Layer, Output Layer und beliebig viele verborgene Hidden Layer. Sie haben gelernt, dass in einem FNN nur vorwärts gerichtete Verbindungen erlaubt sind, also nur Verbindungen zu Neuronen der jeweils nächsten Schicht, und dass alle Eingänge und Ausgänge unabhängig voneinander sind.

Feedforward-Netze bieten durch ihren Aufbau keine generalisierende bzw. keine universelle Netzarchitektur, mit der sich vielfältige Aufgaben lösen lassen. Hier verhält es sich genauso wie bei den Algorithmen für Machine Learning und Deep Learning. Auch beim Einsatz von neuronalen Netzen wird für die Verarbeitung und für Problemlösungen eine Vielzahl unterschiedlicher Netzarchitekturen verwendet, um den jeweiligen Anforderungen gerecht zu werden.

Für die Verarbeitung von sequenziellen Daten, hierzu zählen Texte, Musikstücke, aber auch Börsenkurse und Wetterdaten, ist ein Feedforward-Netz in der Regel nicht geeignet, da es jeden Berechnungsschritt unabhängig vom vorherigen durchführt. Dadurch ist es bei einem FNN nicht möglich, Abhängigkeiten zwischen zeitlich versetzten Eingaben in Sequenzen zu erkennen.

Das Besondere an Sequenzen in Form von Texten oder Zeitreihen, wie sie bei Wetterdaten vorkommen, ist, dass hintereinander folgende Datenpunkte nicht unabhängig voneinander sind. Wörter erhalten erst im Satzzusammenhang mit anderen Wörtern ihren Sinn. Auch zeitliche Verläufe besitzen häufig gewisse Regelmäßigkeiten. Zum Lernen einer solchen Abfolge muss das System Informationen aus einer Menge von Ausgangsdaten in zeitlicher Reihenfolge analysieren. Für diese Art von Aufgaben wurden Recurrent Neural Networks (rekurrente neuronale Netze), kurz RNNs, entwickelt. Diese Netze können ihre Ausgaben bei jedem Abschnitt wieder als Eingabe nutzen.

■ 5.1 Sequenzen und Rückkopplung

Im Gegensatz zu einem Feedforward-Netz können RNNs von der vorwärts gerichteten Verarbeitungsrichtung abweichen. Dadurch sind sie in der Lage, sich mittels Rückkopplungen selbst zu beeinflussen. Bild 5.1 zeigt den Unterschied im Informationsfluss zwischen Feedforward und Recurrent Neural Networks.

Wie Sie in Bild 5.1 sehen, verarbeiten RNNs die Eingabe nicht nur von Schicht zu Schicht, sondern leiten die Gewichte und Bias-Werte (Verzerrungen) auch von höhere in niedrigere Schichten oder auch zu sich selbst zurück. Dies wird auch als zyklische Verbindung bezeichnet und erlaubt eine Rückkopplung im neuronalen Netz.

Ein gutes Beispiel dafür ist die Spracherkennung. Gesprochene Sätze bestehen aus einer Abfolge von Lauten, die sich häufig erst im Kontext sinnvoll interpretieren lassen. Zu dem Zeitpunkt, an dem in einem Feedforward-Netz das Wort „Rekurrent“ den Buchstaben „k“ erreicht, hat es „R“ und „e“ schon wieder vergessen, somit ist es nicht mehr möglich eine Aussage zu treffen, welcher Buchstabe im nächsten Schritt folgt.

Das RNN behebt dieses Problem. Das RNN bildet sozusagen ein neuronales Netz mit Schleifen (Bild 5.1), in dem die Information bestehen bleibt. Somit können Datenwerte der Gegenwart und aus der Vergangenheit betrachtet werden. Dies ist wichtig, weil die Reihenfolge der Daten wichtige Informationen darüber enthält, wie sich die Daten im zeitlichen Verlauf entwickelt haben. So erhält man zum Beispiel bei einer Zeitreihenanalyse von Wetterdaten für den Zeitraum von 2010 bis 2019 den Datenverlauf von Merkmalen wie Luftdruck, Temperatur und Windgeschwindigkeit in entsprechender zeitlicher Reihenfolge für die Analyse. Die Schleifen im RNN kann man sich wie eine Art Kurzzeitgedächtnis vorstellen, welches das Netz dazu befähigt, Abfolgen von Eingaben zu verarbeiten.

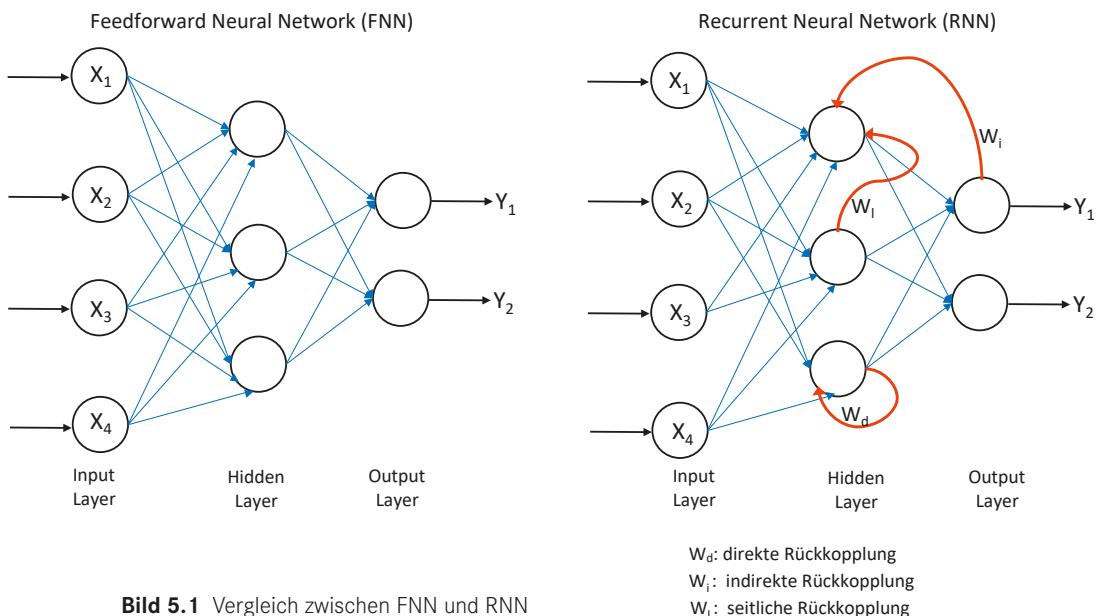


Bild 5.1 Vergleich zwischen FNN und RNN

Durch die Fähigkeit, sequenzielle Daten zu verarbeiten, werden Recurrent Neural Networks nicht nur bei Apple Siri und Google Voice Search verwendet, sondern auch in vielen anderen Bereichen wie der Analyse von Zeitreihen, Sprache, Text, Finanzdaten, Audio, Video und Wetterdaten. Ein häufiger Anwendungsfall für sequenzielle Daten ist die Zeitreihenanalyse, bei der eine Reihe von Datenpunkten ausgewertet wird, die in zeitlicher Reihenfolge aufgelistet sind.

Bei FNNs ist die Ein- und Ausgabe immer ein einziger fester Vektor. RNNs hingegen sind in der Lage Sequenzen von Vektoren ein- und auszugeben.

Das bedeutet, dass Sie zum Beispiel Wetterdaten eingeben können und das rekurrente neuronale Netz versucht eine Schätzung für die nächste Woche oder den nächsten Monat zu ermitteln. Das RNN erkennt Muster in der Eingabesequenz und erfährt über die Zeitreihenfolge, wann diese wahrscheinlich wieder auftreten werden. Natürlich können Wetterdaten stark schwanken, wenn etwas Ungewöhnliches passiert. RNNs sind nicht in der Lage, etwas aus Ereignissen zu lernen, die ganz selten oder noch nie zuvor passiert sind, oder was nicht in einer Art von Intervallen auftritt. Das Ziel eines RNN ist es, mögliche Abhängigkeiten in sequenziellen Daten zu erkennen. Das RNN versucht also Korrelationen zwischen verschiedenen Punkten innerhalb einer Sequenz zu finden.

Des Weiteren unterscheidet man bei RNNs zwei Arten von Abhängigkeiten. Kurzfristige Abhängigkeiten beschreiben eine Bedingtheit in der jüngeren Vergangenheit. Langfristige Abhängigkeiten hingegen sind Korrelationen zwischen Zeitpunkten, die weit voneinander entfernt sind. Leider gibt es bei der Bestimmung der Abhängigkeiten keine klare Definition von kurz- und langfristig, sodass Sie nicht mit Sicherheit bestimmen können, wann kurzfristig endet und wann langfristig beginnt. Das Auffinden der Abhängigkeit ermöglicht es jedoch, Muster in den sequenziellen Daten zu erkennen und diese vom RNN erkannten Informationen zur Vorhersage eines Trends zu verwenden.

Rückkopplung

In einer Sequenz wird ein bestimmter Punkt als Zeitschritt bezeichnet. Die Gesamtzahl der Punkte ergibt dann die Sequenzlänge. Für jeden Zeitschritt in der Sequenz gibt es einen Merkmalsvektor, der aus den Werten besteht, die Sie verfolgen möchten. Daher ist eine Sequenz mindestens zweidimensional in der Form [Zeitschritt, Merkmalswert]. Sie können aber auch mehrere verschiedene Merkmale nutzen.

Wie weiter oben schon erwähnt, lassen sich in einem Recurrent Neural Network zyklische Verbindungen zwischen den verschiedenen Schichten herstellen. Hierdurch lassen sich RNNs in verschiedene Arten von Rückkopplungen unterteilen. Bild 5.2 zeigt die Darstellung der drei wichtigsten Rückkopplungen. Man unterscheidet grundsätzlich:

- **Direkte Rückkopplung (direct feedback):** Der Neuronen-Ausgang wird als weiterer Eingang genutzt. Es entsteht eine Verbindung von einem Neuron zu sich selbst.
- **Indirekte Rückkopplung (indirect feedback):** Sie verbindet den Ausgang mit einem Neuron der vorhergehenden Schicht.
- **Seitliche Rückkopplung (lateral feedback):** Der Ausgang des Neurons wird mit einem Neuron derselben Schicht verbunden.
- **Vollständige Verbindung:** Jeder Neuronen-Ausgang hat eine Verbindung zu jedem anderen existierenden Neuron.

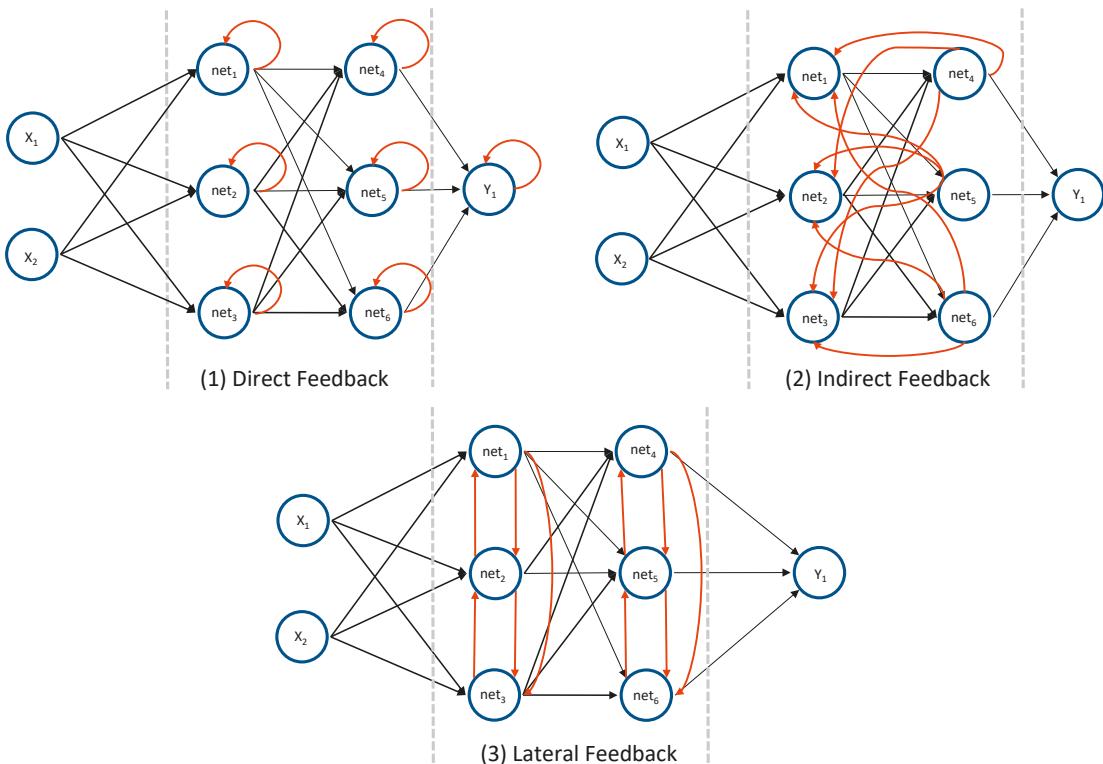


Bild 5.2 Die Rückkopplungsarten im Recurrent Neural Network (RNN)

■ 5.2 Architektur eines RNN

Die Struktur eines Recurrent Neural Network wird als Zelle bezeichnet. Die Minimalversion einer RNN-Zelle funktioniert fast genauso wie ein gewöhnliches neuronales Netz. Die Zelle besteht lediglich aus einer Ebene mit anschließender Aktivierungsfunktion. Bild 5.3 zeigt die vereinfachte Darstellung einer RNN-Zelle.

Die RNN-Zelle ist in der Lage, die Ausgabe des Netzes zurück zur Eingabe zu leiten. Das RNN verwendet somit den Ausgang im Moment t , um den Ausgang des Netzes im Moment $t+1$ zu berechnen.

Bei einer RNN-Zelle wird unter einer Eingabesequenz eine Folge $X = (X_1, \dots, X_n)$ und unter einer Ausgabesequenz eine Folge $Y = (Y_1, \dots, Y_n)$ verstanden. Der Index $t \in \{1 \dots n\}$ gibt den Zeitpunkt innerhalb der Sequenz an.

Somit lässt sich der Zeitschritt einfach als mathematische Formel darstellen:

$$h_t = f(h_{t-1}, X_t)$$

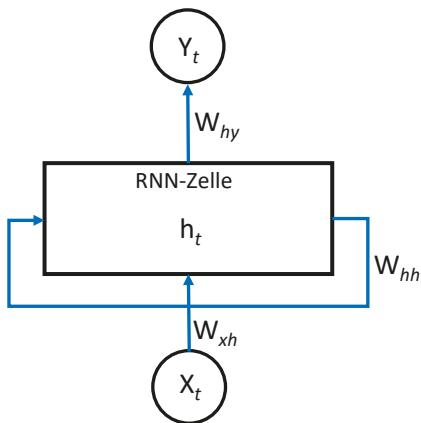


Bild 5.3
Die vereinfachte RNN-Zelle

Die Funktion f ist normalerweise eine nichtlineare Aktivierungsfunktion wie tanh oder ReLU. Der Zustand des Netzes im aktuellen Zeitschritt wird zum Eingangswert für den nächsten Zeitschritt. Die folgende Formel gibt die Aktivierung einer RNN-Zelle zum Zeitpunkt t an:

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} X_t)$$

Der Aktivierungsfunktion tanh wird die Eingabe X_t und die vergangene Aktivierung h_{t-1} übergeben. W_{hh} sind die Gewichte der rekurrenten Neuronen und W_{xh} die Gewichte der Eingangsneuronen.

Bild 5.4 zeigt ein entfaltetes RNN. Bei einem entfalteten oder auch abgerollten RNN wird das Netz für die gesamte Sequenz ausgeschrieben. Wenn die Sequenz zum Beispiel ein Satz mit drei Wörtern ist, wird das Netz zu einem dreischichtigen neuronalen Netz entfaltet werden, eine Schicht für jedes Wort. Hierbei sollten Sie Folgendes beachten:

- Den nicht sichtbaren Bereich in der RNN-Zelle können Sie sich als internen Speicher vorstellen. Hier werden Informationen darüber erfasst, was in allen vorherigen Zeitschritten geschehen ist. Die Schrittausgabe wird ausschließlich auf der Grundlage des zeitlichen Speichers berechnet.
- Im Gegensatz zum Feedforward-Netz, das auf jeder Schicht unterschiedliche Parameter verwendet, teilt ein RNN die gleichen Parameter über alle Schritte hinweg. Das stellt sicher, dass bei jedem Schritt die gleiche Aufgabe ausgeführt wird, nur mit unterschiedlichen Eingaben.

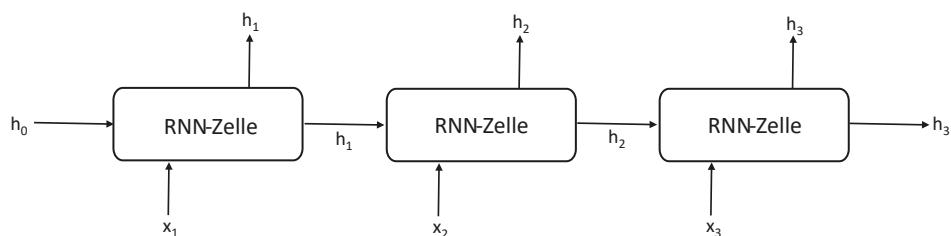


Bild 5.4 Entfaltetes Recurrent Neural Network

Die Mathematik hinter einem einfachen RNN ist, wie beschrieben, nicht ungeheuer komplex, sondern lässt sich sehr einfach in C#-Code für eine Klasse als RNN-Zelle übertragen. Sie müssen hierfür nur eine Methode in der Klasse implementieren, die einen Zeitschritt zur Verfügung stellt, mit dem Sie die Zeit simulieren können. Sie würden weiterhin noch eine Eingabe X benötigen, welche die Eingabe zu diesem Zeitschritt darstellt. Die implementierte Methode würde dann eine Ausgabe Y zurückgeben. Innerhalb der Klasse würden Sie die Informationen über frühere Eingaben und Netzzustände behalten. Eine allgemeingültige C#-Implementierung könnte daher wie folgt aussehen:

Listing 5.1 Implementierung einer einfachen RNN-Zelle

```
public class RNN
{
    private Matrix<double> state;
    private Matrix<double> inputWeights;
    private Matrix<double> recurrentWeights;
    private Matrix<double> outputWeights;

    public RNN(Matrix<double> initialInputWeights,
               Matrix<double> initialRecurrentWeights,
               Matrix<double> initialOutputWeights)
    {
        inputWeights = initialInputWeights;
        recurrentWeights = initialRecurrentWeights;
        outputWeights = initialOutputWeights;
    }

    public Matrix<double> TimeStep(Matrix<double> input)
    {
        state = Matrix<double>.Tanh(state.Multiply(recurrentWeights)
            + input.Multiply(inputWeights));
        return state.Multiply(outputWeights);
    }
}
```

In der Beispielimplementierung wird als Aktivierungsfunktion tanh benutzt, innerhalb des Methodenaufrufs wird der aktuelle Zustand mit den wiederkehrenden Gewichten multipliziert und die Summe mit der Eingabe für die entsprechenden Eingabegewichte gebildet. Der Ausgang wird dann durch die Multiplikation des aktuellen Zustands mit den Ausgangsgewichten berechnet.

■ 5.3 Backpropagation Through Time

Bei einem aufgefalteten RNN kann wie auch bei einem Feedforward-Netz den Backpropagation-Algorithmus zur Minimierung der Fehlerfunktion anwenden. Da in diesem Fall noch die Zeitdimension hinzukommt, verwenden wir den erweiterten Backpropagation-Through-Time-Algorithmus, kurz BPTT.

Das bedeutet, Sie müssen bei einem RNN in der Zeit zurückgehen und alle Gewichte der vorherigen Zeitschritte anpassen. Bild 5.5 zeigt die schematische Darstellung des BPTT-Algorithmus.

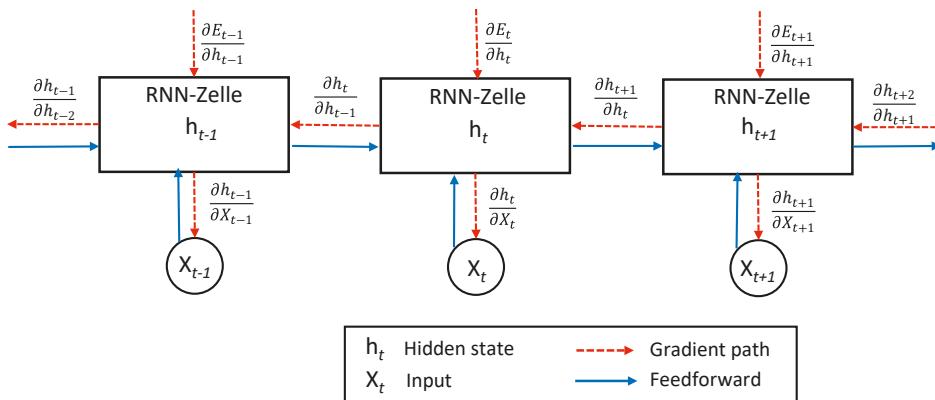


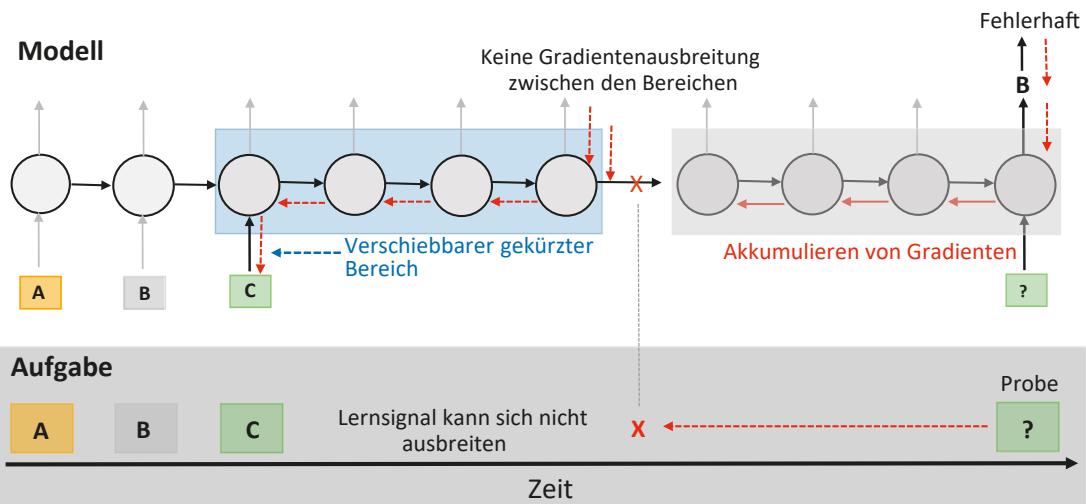
Bild 5.5 Backpropagation Through Time

Zunächst wird für jede Ausgabe des Netzes der Fehler über die bekannte Fehlerfunktion (Cost Function) berechnet. Nach der Vorwärtspropagation werden die Gewichte durch Gradientenabstieg auf der Fehlerfunktion berechnet und dann die Anpassung der Gewichtsmatrix W_{xh} , W_{hh} und W_{hy} ermittelt und mit dem Backpropagation-Algorithmus optimiert.

Somit ähnelt der BPTT schon stark dem Backpropagation-Algorithmus in einem Standard-Feedforward-Netz.

Der einzige Unterschied besteht darin, dass Sie die Gradienten des Fehlers für alle Zeitschritte zusammenfassen. Dies kann so einfach gemacht werden, da die Parameter über alle Schichten im RNN hinweg gleich sind. Somit wird in der Regel die gesamte Datenfolge als ein Trainingsbeispiel betrachtet. Dies ermöglicht, den Fehler in jedem Zeitschritt und den globalen Fehler als Summe aller Fehler zu berechnen.

BPTT kann dann zu einem Problem werden, wenn die Sequenz sehr lang wird und Sie nach jeder Vorhersage den ganzen Weg zurückpropagieren müssen. Truncated Backpropagation Through Time, kurz TBPTT, löst dieses Problem durch die Aufspaltung der Sequenz. Bild 5.6 zeigt die schematische Darstellung der Aufspaltung. Jedes Mal, wenn die Backpropagation angewendet wird, müssen Sie nur um die Länge der abgeschnittenen Teilsequenz zurückgehen, in der Sie sich grade befinden.

Modell**Bild 5.6** Truncated Backpropagation Through Time

Die Schwierigkeit hier ist, dass das Netz jetzt nur noch Abhängigkeiten innerhalb dieser Teilsequenzen lernen kann. Daher ergeben sich aus der Architektur eines RNNs einige grundsätzliche Probleme. Außerdem ist die praktisch zu nutzende Sequenz sehr klein, da der Einfluss einer bestimmten Eingabe auf die Ausgabe mit der Sequenzlänge entweder laufend abnimmt oder exponentiell zunimmt.

Daher tritt bei RNNs sehr schnell das Problem der verschwindenden Gradienten auf, wenn Sie eine Sequenz zurückpropagieren. Je weiter Sie in der Sequenz zurückgehen, desto weniger Bedeutung haben die gelernten Werte für die aktuellen Vorhersagen. Somit ist es nicht möglich, dass ein RNN langfristige Abhängigkeiten lernt. In der Forschung spricht man auch von einer gravierenden Vanishing- bzw. Exploding-Gradient-Problematik.

**Vanishing Gradients**

Von Vanishing Gradients (verschwindenden Gradienten) spricht man, wenn die Werte eines Gradienten zu klein sind und das neuronale Netz nicht mehr lernt oder ein extrem langes Laufzeitverhalten zustande kommt. Vanishing Gradients treten immer dann auf, wenn viele im BPTT-Algorithmus berechnete Ableitungen kleiner als 1 sind. Da der Gradient mittels der Kettenregel aus dem Produkt der Ableitung berechnet wird, nimmt dieser im Verlauf immer kleinere Werte an.

Bei der Kettenregel handelt es sich um eine Ableitungsregel, die immer dann anzuwenden ist, wenn zwei Funktionen miteinander verkettet (ineinander verschachtelt) sind.



Exploding Gradients

Exploding Gradients (explodierende Gradienten) treten immer dann auf, wenn der Algorithmus den Gewichten eine zu hohe Bedeutung zuweist. Infolgedessen wächst der Gradient sehr schnell und es werden die Gewichte innerhalb einzelner Schritte stark verändert. Somit ist auch hier kein zielführendes Training mehr möglich. Das Problem kann allerdings gelöst werden, indem Sie den Truncated-Backpropagation-Through-Time-Algorithmus (TBPTT-Algorithmus) anwenden.

Das Gradienten-Problem der Vanishing Gradients wird durch eine spezielle Art von rekurrenten neuronalen Netzen gelöst. Long Short-Term Memory Networks sind RNNs, die durch ihre Architektur in der Lage sind, Langzeitabhängigkeiten zu erfassen.

■ 5.4 Long Short-Term Memory Networks

Long Short-Term Memory Networks , kurz LSTMs, stellen eine Erweiterung für RNNs dar, die in der Lage ist, durch einen speziellen Speicher auch langfristige Abhängigkeiten zu erlernen. Sie wurden 1997 von Hochreiter & Schmidhuber eingeführt und im Laufe der letzten Jahre von vielen Forschern und Spezialisten verbessert. Sie sind besonders durch ihren Einsatz im Machine Learning und Deep Learning bekannt geworden. Sie haben in den letzten Jahren große Fortschritte bei Vorhersagesystemen gemacht, die mit Sequenzdaten arbeiten, wie Spracherkennung, Videoverarbeitung und Texterzeugung.

Das Problem bei RNNs ist, wie oben erwähnt, dass beim Backpropagation-Through-Time-Algorithmus der Einfluss, den ein früherer Wert auf eine spätere Berechnung haben kann, stark eingeschränkt ist. Es gibt keine Möglichkeit, die Information unmittelbar weiterzugeben.

LSTMs sind explizit darauf ausgerichtet, dieses Problem zu vermeiden. Somit stellt das Speichern von Informationen über lange Zeiträume das Standardverhalten von LSTMs dar. Das heißt, dass zeitliche Informationen in einem LSTM gezielt vergessen oder gemerkt werden können, sodass nicht alle Informationen im Zeitverlauf als gleich wichtig verarbeitet werden, sondern Informationen zeitabhängig gezielt gewichtet werden können.

Das Ziel der LSTM-Architektur ist die Erzeugung eines konstanten Fehlerrückflusses innerhalb der Backpropagation und somit die Vermeidung von Vanishing Gradients. Erreicht wird dies durch die Verwendung eines internen Zustands, der unverändert an den nächsten Zeitschritt weitergegeben und nur durch sogenannte Gates verändert wird.

Alle rekurrenten neuronalen Netze besitzen eine Architektur in Form einer Kette von sich wiederholenden Zellen eines neuronalen Netzes. Unter den Standard-RNNs, welche Sie in Abschnitt 5.2 kennengelernt haben, hat diese sich wiederholende Zelle eine sehr einfache Struktur, siehe Bild 5.7. Hier ist zum Beispiel nur eine einzelne hyperbolische Tangens-Schicht (\tanh) in der Zelle aufgeführt.

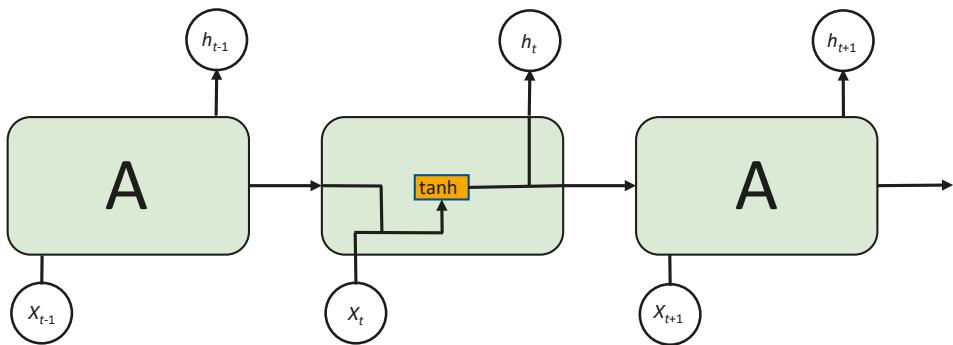


Bild 5.7 Die Zelle im ausgerollten Standard-RNN

LSTMs besitzen auch diese kettenartige Struktur, aber die einzelne LSTM-Zelle besitzt statt einer einzelnen neuronalen Schicht insgesamt vier Schichten, die auf besondere Art und Weise interagieren. Bild 5.8 zeigt die schematische Darstellung einer LSTM-Zelle.

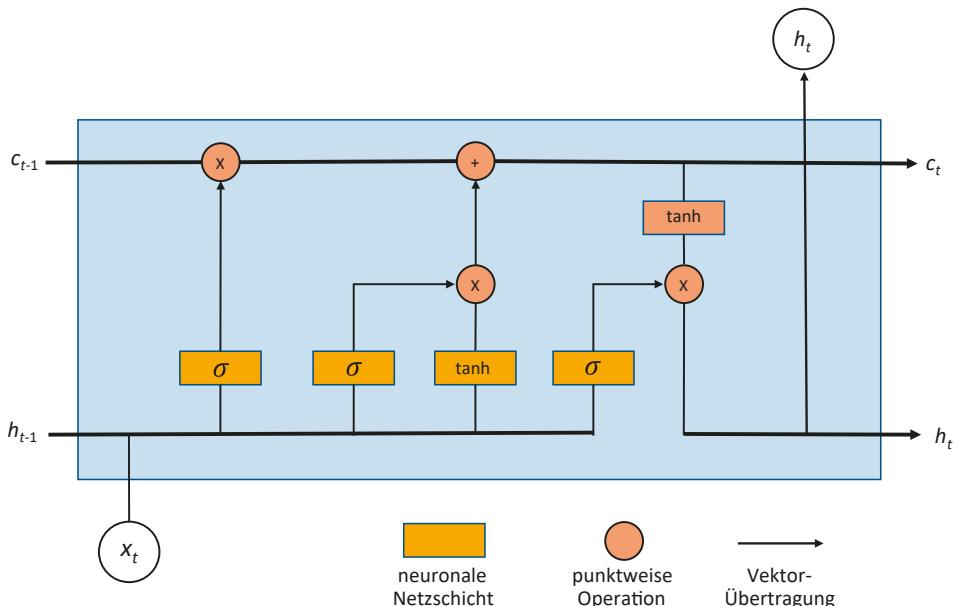


Bild 5.8 Die vollständige LSTM-Zelle

In Bild 5.8 trägt jede Linie einen ganzen Vektor, vom Eingang bis zum Ausgang eines Knotens. Die Kreise stellen die punktweisen bzw. elementweisen Operationen wie zum Beispiel die Vektoraddition oder Multiplikation dar. Die gelben Rechtecke repräsentieren die neuronale Netzschicht mit den entsprechenden Aktivierungsfunktionen Sigmoid oder hyperbolischer Tangens (tanh).

5.4.1 Funktionsweise von LSTMs

Mithilfe der Zelle ist es möglich ein rekurrentes neuronales LSTM-Netz zu konstruieren. Der entsprechende Zellzustand wird durch die horizontale Linie dargestellt, die durch den oberen Rand der Darstellung in Bild 5.9 verläuft.

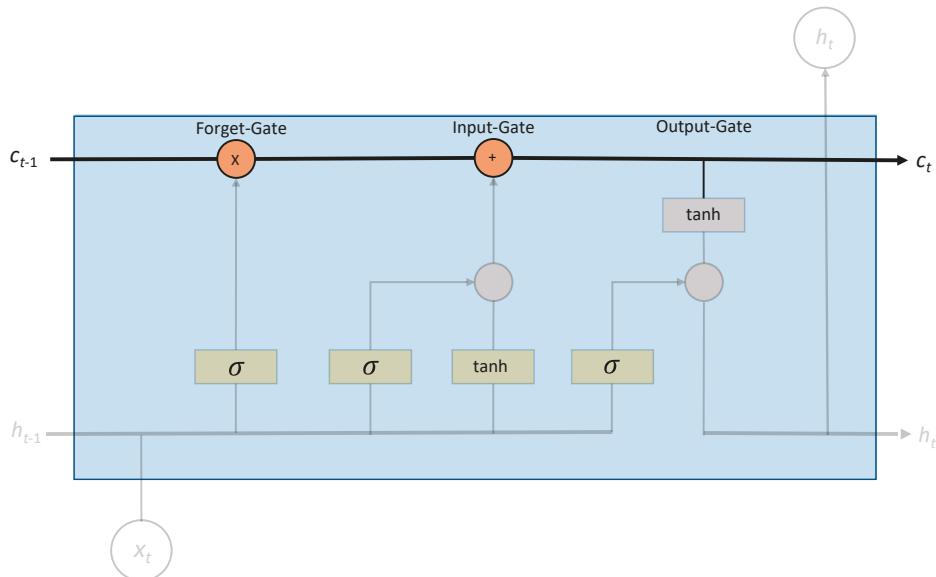


Bild 5.9 Zellzustand der LSTM-Zelle

Der Zellzustand verläuft hier geradlinig entlang der gesamten Zelle, mit nur einigen geringfügigen linearen Wechselwirkungen. In den meisten Fällen ist der Informationsfluss in diesem Status unverändert. Die LSTM-Architektur ist in der Lage, dem Zellzustand Informationen zu entnehmen oder hinzuzufügen. Dies wird sehr sorgfältig durch Strukturen reguliert, die als Gates bezeichnet werden.

Durch diese Gates ist es möglich, Informationen optional durchzulassen. Gates bestehen aus einer Schicht eines sigmoiden neuronalen Netzes und einer punktweisen Multiplikation-Operation. Die Sigmoid-Aktivierungsfunktion gibt Werte zwischen 0 und 1 aus und beschreibt damit, wie viel von jeder Komponente durchgelassen wird (Bild 5.10).

Eine Standard-Long-Short-Term-Memory-Zelle besitzt drei Gates, um den Zellzustand zu schützen bzw. zu speichern und zu kontrollieren.

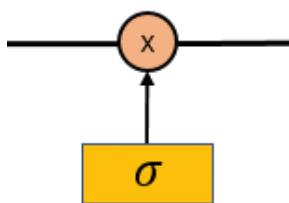


Bild 5.10
Sigmoid-Schicht in der LSTM-Zelle

5.4.1.1 Forget-Gate

Der erste Schritt in der LSTM-Zelle besteht darin, zu entscheiden, welche Information aus dem Zellzustand entfernt werden soll. Diese Entscheidung wird von einer Sigmoid-Schicht getroffen, die allgemein als *forget-gate layer* bezeichnet wird. Sie betrachtet h_{t-1} und X_t (Bild 5.11) und gibt für jeden Wert im Zellenstatus C_{t-1} eine Zahl zwischen 0 und 1 aus. Eine 1 steht für „Wert vollständig beibehalten“, während eine 0 für „vollständig vergessen“ bzw. „nicht durchlassen“ steht.

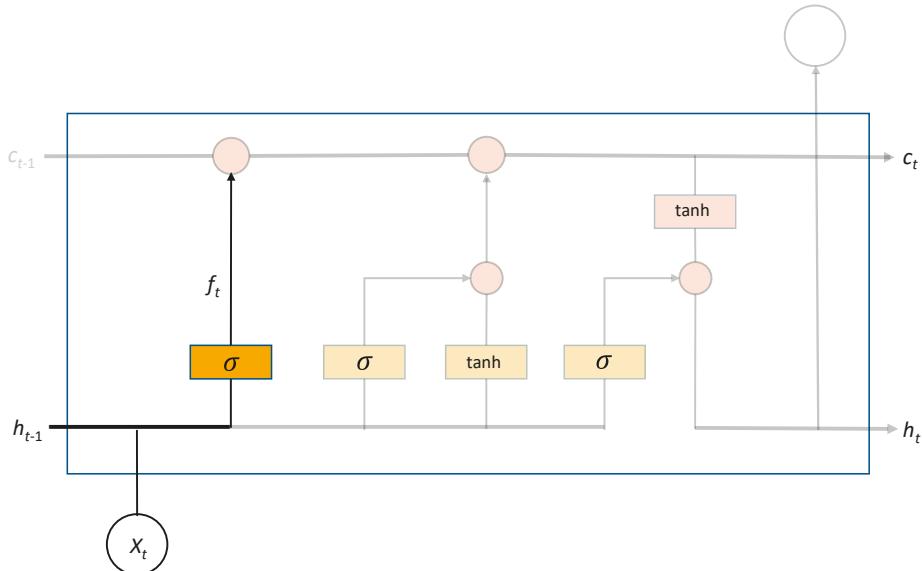


Bild 5.11 Forget-Gate in der LSTM-Zelle

Die Berechnung der Funktion im Forget-Gate wird wie folgt implementiert:

$$f_t = \sigma(W_f \cdot [h_{t-1}, X_t] + b_f)$$

Wobei σ für die Sigmoid-Funktion, X_t für die Eingaben, W für die Gewichte und b_f für den Bias-Wert, also für die Verzerrungen, steht.

5.4.1.2 Input-Gate

Der nächste Schritt in der LSTM-Zelle besteht darin, zu entscheiden, welche neuen Informationen Sie im Zustand der Zelle speichern möchten. Dieser Schritt wird in zwei Teilen durchgeführt. Zunächst entscheidet auch hier eine Sigmoid-Schicht (Bild 5.12), die als *Input-Gate Layer* bezeichnet wird, welche Werte aktualisiert werden. Danach erzeugt die Schicht mit der hyperbolischen Tangens-Aktivierungsfunktion (\tanh) einen Vektor mit neuen Werten C^-_t .

Die Berechnung der Funktionen ergibt sich wie folgt:

$$i_t = \sigma(W_i \cdot [h_{t-1}, X_t] + b_i)$$

$$C^-_t = \tanh(W_c \cdot [h_{t-1}, X_t] + b_c)$$

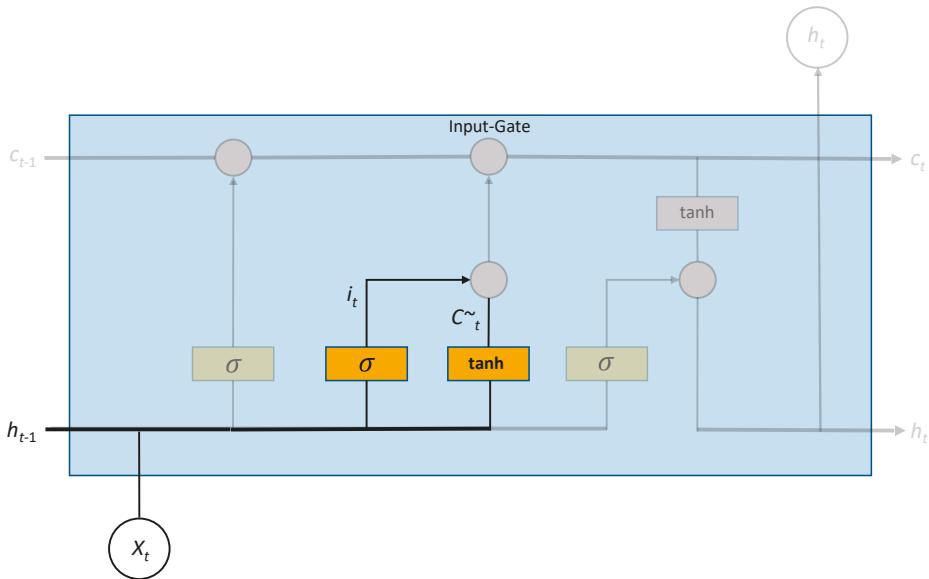


Bild 5.12 Input-Gate in der LSTM-Zelle

Das heißt, eine hyperbolische Tangens-Schicht bestimmt aus X_t und h_{ct-1} die hinzuzufügende Information C^{\sim}_t , die mit der gleichen Kombination aus Sigmoid-Schicht (i_t) und punktweiser Multiplikation gefiltert wird. Die so gefilterte Eingabe wird mithilfe einer punktweisen Addition dem internen Zellzustand hinzugefügt (Bild 5.13).

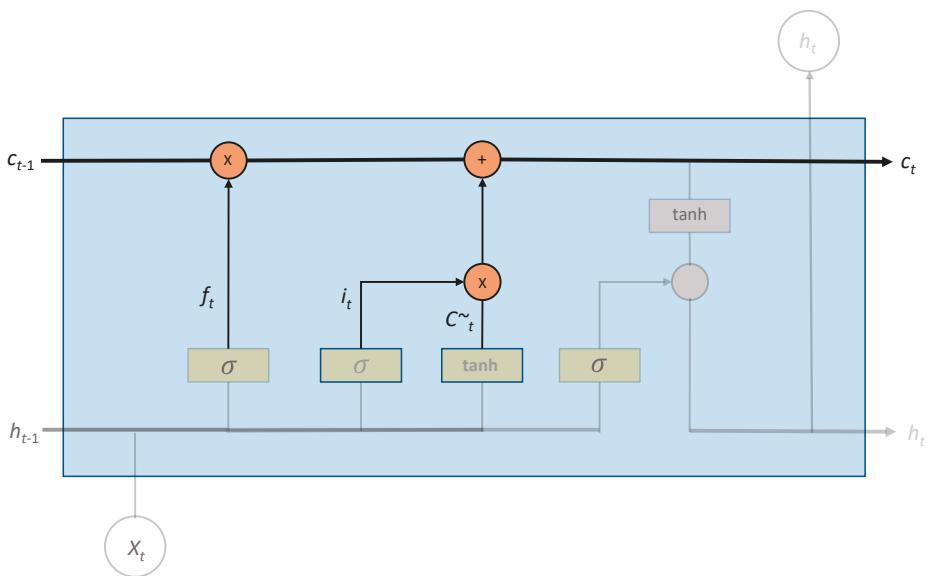


Bild 5.13 Berechnung des internen Zellzustands

Die Berechnung hierbei erfolgt über:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot C^-_t$$

Somit können Sie jetzt über das Output-Gate entscheiden, was ausgegeben werden kann.

5.4.1.3 Output-Gate

Auch das Output-Gate in Bild 5.14 folgt dem gleichen Prinzip. Auch hier wird der Zellzustand c_t durch eine hyperbolische Tangens-Schicht (tanh) in die Ausgabe transformiert, die wieder durch die Sigmoid-Schicht (O_t) und punktweiser Multiplikation gefiltert wird.

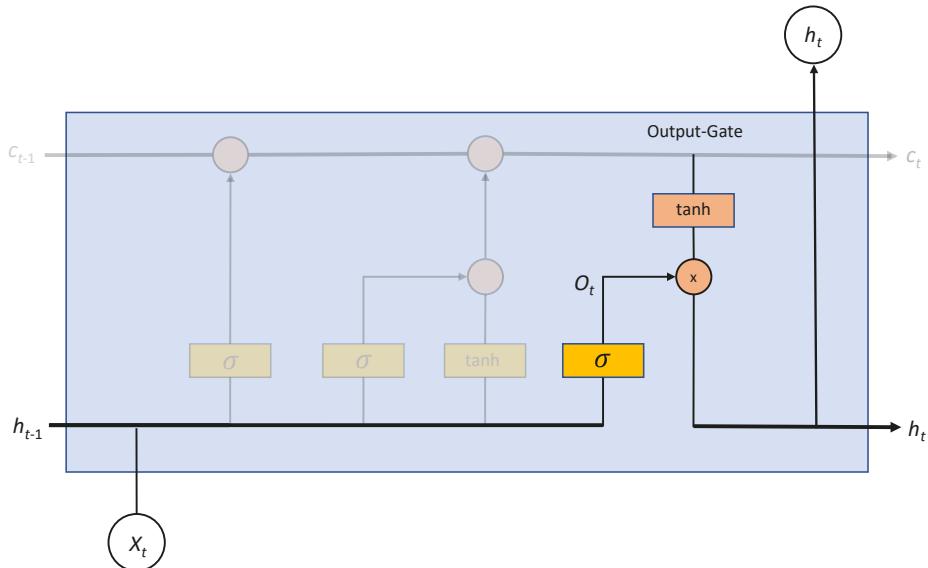


Bild 5.14 Output-Gate in der LSTM-Zelle

Die Berechnung wird wie folgt vorgenommen:

$$O_t = \sigma(W_O \cdot [h_{t-1}, X_t] + b_O)$$

$$h_t = O_t \cdot \tanh(C_t)$$

Die so gefilterte Ausgabe der aktuellen LSTM-Zelle wird sowohl in die darunterliegende Schicht, als auch an die in zeitlicher Dimension nächste LSTM-Zelle weitergegeben.

5.4.1.4 Zusammenfassung

Die Idee der LSTM-Architektur ist, dass der Zustandsvektor in zwei Hälften gespalten wird, und zwar wie folgt:

- In den Langzeitspeicher, der Informationen unverändert über beliebige Schichten weiterreichen kann, wobei die Weitergabe durch die Gates kontrolliert und optimiert wird.
- In den Arbeitsspeicher, der in jedem Rechenschritt komplett berechnet wird, wie in einem Recurrent Neural Network.

Mit den Gates in der LSTM-Architektur können somit folgende Eigenschaften erzielt werden:

- Das Forget-Gate ermöglicht es, nicht mehr benötigte Informationen zu löschen und andererseits wichtige Informationen unverändert bestehen zu lassen.
- Das Input-Gate erlaubt es, neue Informationen in das Netz einzuspeichern.
- Das Output-Gate ermittelt die Aktivierung der LSTM-Zelle unter der Berücksichtigung des Zellzustandes.

Somit ist das Gate in der LSTM-Architektur eine sigmoide Einheit, die wie der Eingangsknoten vom aktuellen Datenpunkt X_t sowie von der verborgenen Schicht (Hidden Layer) im vorherigen Zeitschritt aktiviert wird. Ist der Wert des Gates 0, so wird der Fluss des Knotens abgeschnitten, ist der Wert 1, wird dieser durchgelassen bzw. weitergereicht.

5.4.2 Gradient Clipping

Das Problem der Vanishing Gradients lässt sich wie oben gezeigt mit der LSTM-Architektur lösen. Es bleibt also noch die Frage, wie man mit Exploding Gradients umgeht. Ein dafür verwendeter Lösungsansatz ist das Gradient Clipping (Mikolov, 2012).

Beim Gradient Clipping werden die Gradienten-Werte (elementweise) auf einen bestimmten Minimal- oder Maximalwert gezwungen, wenn der Gradient einen erwarteten Wertebereich überschreitet. Das heißt, wenn der Gradientenabstieg im Algorithmus vorschlägt, einen sehr großen Schritt zu machen, greift das Gradienten-Clipping-Verfahren ein, um die Schrittgröße so sehr zu verringern, dass es weniger wahrscheinlich ist, dass sie außerhalb des Bereichs liegt, in dem der Gradient die Richtung des annähernd steilsten Abstiegs anzeigt. Dieses Vorgehen bewirkt eine Stabilisierung des Trainings, da große Schwankungen nach oben ausgeglichen werden.

5.4.3 Varianten

Da LSTM-Zellen sehr komplex ausfallen, gibt es in der Praxis zudem Dutzende Implementierungsvarianten, die sich aber glücklicherweise nur geringfügig voneinander unterscheiden. Eine sehr beliebte LSTM-Variante ist die von Gers und Schmidhuber [10] aus dem Jahr 2000. Hier wurde die LSTM-Architektur um sogenannte Peephole-LSTMs erweitert. Hierbei handelt es sich um Beobachtungsverbindungen, die es erlauben, den Zellzustand in den Gates-Schichten zu betrachten.

Eine weitere Variante ist die Verwendung von gekoppelten Forget-Gates und Input-Gates. Anstatt getrennt zu entscheiden, was vergessen und was mit neuen Informationen versehen werden soll, wird diese Entscheidung gemeinsam getroffen. In diesem Fall wird nur vergessen, wenn Sie einen neuen Input hinzufügen. Somit werden nur neue Werte den Zustand der Zelle verändern, wenn gleichzeitig ältere Informationen vergessen werden.

Man trifft in der Praxis auch sehr häufig auf die von Cho 2014 [11] eingeführte Gate Recurrent Unit, kurz GRU. Auch hier besteht die interne Struktur der Speicherzelle aus Gates. Im Gegensatz zu den LSTMs werden hier die Forget-Gates und Input Gates zu einem einzigen Update Gate kombiniert. Somit werden auch der Zellzustand und der verborgene

Zustand zusammengeführt. Das hieraus resultierende Modell ist etwas weniger komplex als ein Standard-LSTM-Modell und erfreut sich daher zunehmender Beliebtheit.

5.4.4 LSTM-Implementierung

Durch die große Verfügbarkeit von Machine Learning Frameworks, die LSTM-Netze unterstützen und entsprechend viele Funktionen zur Optimierung, Visualisierung und zum Batch Processing anbieten, werden Sie wahrscheinlich nie in die Verlegenheit kommen, ein LSTM-Netz von Grund auf in C# zu implementieren. Das Verständnis der Funktionsweise hilft aber bei der Nutzung einer LSTM-Architektur mit einer entsprechenden Code-Bibliothek wie Microsoft ML.NET oder Google TensorFlow.

Es ist natürlich auch möglich, ein LSTM vollständig in C# zu erstellen. Listing 5.2 zeigt einmal beispielhaft, wie eine LSTM-Zelle (LSTM-Layer) aufgebaut sein könnte.

Listing 5.2 LSTM-Layer für ein neuronales Netz (Beispiel)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LstmDemo
{
    public class LstmLayer : INetworkLayer
    {
        int inputDimension;
        int outputDimension;

        readonly Matrix wix;
        readonly Matrix wih;
        readonly Matrix inputBias;
        readonly Matrix wfx;
        readonly Matrix wfh;
        readonly Matrix forgetBias;
        readonly Matrix wox;
        readonly Matrix woh;
        readonly Matrix outputBias;
        readonly Matrix wcx;
        readonly Matrix wch;
        readonly Matrix cellWriteBias;

        Matrix _hiddenContext;
        Matrix _cellContext;

        readonly IForNoLinearity _inputGateActivation = new SigmoidUnit();
        readonly IForNoLinearity _forgetGateActivation = new SigmoidUnit();
        readonly IForNoLinearity _outputGateActivation = new SigmoidUnit();
        readonly IForNoLinearity _cellInputActivation = new TanhUnit();
        readonly IForNoLinearity _cellOutputActivation = new TanhUnit();
```

```

public LstmLayer(int inputDimension, int outputDimension, double
initParamsStdDev, Random rng)
{
    this.inputDimension = inputDimension;
    this.outputDimension = outputDimension;
    wix = Matrix.Random(outputDimension, inputDimension,
                        initParamsStdDev, rng);
    wih = Matrix.Random(outputDimension, outputDimension,
                        initParamsStdDev, rng);
    inputBias = new Matrix(outputDimension);
    wfx = Matrix.Random(outputDimension, inputDimension,
                        initParamsStdDev, rng);
    wfh = Matrix.Random(outputDimension, outputDimension,
                        initParamsStdDev, rng);
    forgetBias = Matrix.Ones(outputDimension, 1);
    wox = Matrix.Random(outputDimension, inputDimension,
                        initParamsStdDev, rng);
    woh = Matrix.Random(outputDimension, outputDimension,
                        initParamsStdDev, rng);
    outputBias = new Matrix(outputDimension);
    wcx = Matrix.Random(outputDimension, inputDimension,
                        initParamsStdDev, rng);
    wch = Matrix.Random(outputDimension, outputDimension,
                        initParamsStdDev, rng);
    cellWriteBias = new Matrix(outputDimension);
}

public Matrix Activate(Matrix input, Graph g)
{
    //input gate
    Matrix sum0 = g.Mul(wix, input);
    Matrix sum1 = g.Mul(wih, _hiddenContext);
    Matrix inputGate = g.Nonlin(_inputGateActivation,
                               g.Add(g.Add(sum0, sum1), inputBias));

    //forget gate
    Matrix sum2 = g.Mul(wfx, input);
    Matrix sum3 = g.Mul(wfh, _hiddenContext);
    Matrix forgetGate = g.Nonlin(_forgetGateActivation,
                                 g.Add(g.Add(sum2, sum3), forgetBias));

    //output gate
    Matrix sum4 = g.Mul(wox, input);
    Matrix sum5 = g.Mul(woh, _hiddenContext);
    Matrix outputGate = g.Nonlin(_outputGateActivation,
                                 g.Add(g.Add(sum4, sum5), outputBias));

    //write operation on cells
    Matrix sum6 = g.Mul(wcx, input);
    Matrix sum7 = g.Mul(wch, _hiddenContext);
    Matrix cellInput = g.Nonlin(_cellInputActivation,
                               g.Add(g.Add(sum6, sum7), cellWriteBias));
}

```

```

//compute new cell activation
Matrix retainCell = g.Elmul(forgetGate, _cellContext);
Matrix writeCell = g.Elmul(inputGate, cellInput);
Matrix cellAct = g.Add(retainCell, writeCell);

//compute hidden state as gated, saturated cell activations
Matrix output = g.Elmul(outputGate,
                        g.Nonlin(_cellOutputActivation, cellAct));

//rollover activations for next iteration
_hiddenContext = output;
_cellContext = cellAct;

return output;
}

public void ResetState()
{
    _hiddenContext = new Matrix(outputDimension);
    _cellContext = new Matrix(outputDimension);
}

public List<Matrix> GetParameters()
{
    List<Matrix> result = new List<Matrix>();
    result.Add(wix);
    result.Add(wih);
    result.Add(inputBias);
    result.Add(wfx);
    result.Add(wfh);
    result.Add(forgetBias);
    result.Add(wox);
    result.Add(woh);
    result.Add(outputBias);
    result.Add(wcx);
    result.Add(wch);
    result.Add(cellWriteBias);
    return result;
}
}
}

```

In dieser Klasse wird die LSTM-Zelle über die Methode *LSTMLayer* erstellt und die Aktivierung der Forget, Input und Output-Gates vorgenommen. Den vollständigen Code einer LSTM-Architektur für die Lösung des XOR-Problems finden Sie unter GitHub. Dort können Sie die einzelnen Schritte der Implementierung durcharbeiten.



Die Beispielimplementierung *LSTMDemo* finden Sie als Visual-Studio-Projekt auf GitHub:

<https://github.com/DanielBasler/LSTMDemo>

6

Convolutional Neural Networks

Das Convolutional Neural Network (neuronales Faltungsnetzwerk), kurz CNN oder auch ConvNet genannt, ist ein häufig eingesetzter Algorithmus für Deep Learning, wenn es um Bild- beziehungsweise um Objekterkennung geht. CNNs sind besonders nützlich, um Muster in Bildern zu finden und so Objekte, Gesichter und Szenen zu erkennen. Die Stärke eines CNN liegt darin, dass es mithilfe von Filtertechniken wichtige Merkmale wie Formen, Farben und Strukturen und deren Relationen zueinander erkennen kann.

Das heißt, CNNs sind prädestiniert für Anwendungen wie selbstfahrende Fahrzeuge, Objekterkennung und Anwendungen zur Gesichtserkennung. Daher werden CNNs heute sehr gerne im Bereich der Computer Vision eingesetzt.

Computer Vision ist ein Teilgebiet der künstlichen Intelligenz, das sich damit beschäftigt, Informationen aus visuellen Daten zu extrahieren. Zu diesen Daten zählen unter anderem Fotos, Scans, Videosequenzen, aber auch mehrdimensionale Daten wie zum Beispiel aus medizinischen Scannern. Man versucht, mit Convolutional Neural Networks bestimmte Objekte in Bildern zu identifizieren, zu lokalisieren und zu klassifizieren.

Die Verwendung von CNNs in diesem Bereich liegt in der Fähigkeit des Systems, sich die optimale Auswahl und Konfiguration von Hunderten bis Tausenden von Filtern (siehe Kasten) allein auf Basis der Trainingsdaten, in diesem Fall Bilder, selbst beizubringen. Daher lernt ein CNN, Bilder nicht nur als große Ansammlung von Farbwerten auf PixelEbene zu betrachten, sondern abstrahiert Muster und Objekte. Das Ergebnis wird dann im CNN letztlich durch die Auswertung der erkannten Strukturen abgeleitet.



Filter

Im Kontext von CNNs ist ein Filter ein Satz von zu erlernenden Gewichten, die mithilfe des Backpropagation-Algorithmus gelernt werden. Ein Filter wird durch einen Vektor von Gewichten dargestellt, mit dem das Netz die Eingabe faltet. Der Filter, ähnlich wie ein Filter in der Signalverarbeitung, liefert ein Maß dafür, wie sehr ein Eingangssignal einem Merkmal ähnelt. Das Merkmal, das mithilfe des Filters identifiziert werden kann, wird nicht manuell konstruiert, sondern durch den Lernalgorithmus aus den Daten abgeleitet.

Das heißt, im CNN extrahieren die einzelnen Schichten des Netzes spezielle, zu lernende Merkmale von Bildern, die dann bei der Erkennung des gewünschten Objekts helfen. Daher ist bei CNNs eine manuelle Merkmalsextraktion nicht nötig, da die Merkmale unmittelbar vom neuronalen Netz gelernt werden. Hierdurch können diese Netze sehr schnell mit neuen

Aufgaben neu trainiert werden und so auf bereits vorhandene Machine-Learning-Modelle aufbauen. Die einfachste Form eines Deep Learning Workflow ist also die Übergabe von Bildern an das CNN, welches die Merkmale automatisch lernt und dann Objekte klassifizieren kann. Bedenken Sie aber, dass es in der Bildanalytik doch sehr komplex zugehen kann, vor allem im anspruchsvollen Bereich der Objektklassifizierung. Sie benötigen einen Algorithmus, der robust genug ist, alle Merkmale eines Objekts wiederzuerkennen, und zwar unabhängig vom Bildhintergrund.

Ein solches robustes Convolutional Neural Network kann über Dutzende oder Hunderte von Schichten (Layer) verfügen, die jeweils lernen, unterschiedliche Merkmale eines Bildes zu erkennen. Hierfür werden Filter auf jedes Trainingsbild in verschiedenen Auflösungen angewendet, und die Ausgabe jedes „gefalteten“ Bildes wird als Eingabe für die nächste Ebene verwendet. Die Filter in einem CNN können mit einfachen Funktionen für Merkmale wie Helligkeit und Kanten starten und in der Folge in ihrer Komplexität zur Bestimmung von Merkmalen zunehmen, um ein Objekt eindeutig wiederzuerkennen.

Bei CNNs geht man in den meisten Fällen davon aus, dass die Eingaben, die an das neuronale Netz erfolgen, Bilder sind. Daher kann man im Vorfeld schon bestimmte Eigenschaften in die Architektur von CNNs aufnehmen, die das Verarbeiten von Bilddaten vereinfachen. Die Ausnahme stellen CNNs für die Text- und Audio-Analyse dar (siehe Abschnitt 10.5.3 „Texterkennung mit CNN“).

■ 6.1 Aufbau eines CNN

In den vorherigen Kapiteln haben Sie Techniken kennengelernt, wie man über Hyperparameter, Regularisierung und Optimierung neuronale Netze erstellen und verbessern kann. Im einfachsten Fall eines Feedforward Neural Network erhalten neuronale Netze einen Input in Form eines Vektors und transformieren ihn durch eine Reihe von Hidden Layern. Jeder Hidden Layer, also die verborgene Schicht, besteht aus einer Reihe von Neuronen, wobei jedes Neuron vollständig mit allen Neuronen in der vorherigen Schicht verbunden ist. Hierbei fällt auf, dass sich ein solches einfaches Feedforward-Netz nicht wirklich dazu eignet, ein Vollbild zu skalieren.

Das Hauptproblem beim Arbeiten mit Fotos oder Bildern ist, dass die Eingabedaten in das Netz sehr groß werden können. Angenommen, ein Bild hat die Größe $68 * 68 * 3$ (68 breit, 68 hoch, 3 Farbkanäle), so hätte ein einzelnes vollständig verbundenes neuronales Netz in seinem ersten Hidden Layer $68 * 68 * 3 = 13.872$ Gewichte. Okay, diese Menge könnte gerade noch zu bewältigen sein, aber daran ist schon zu erkennen, dass eine vollständig verbundene Struktur in einem einfachen neuronalen Netz nicht in der Lage ist, ein großes Bild zu skalieren. Ein Bild mit nur einer Größe von $650 * 450 * 3$ würde zu einer Anzahl von 877.500 Gewichten führen.

Wenn Sie nun eine so große Eingabe an ein neuronales Netz übergeben, schwilkt die Anzahl der Parameter auf eine riesige Zahl an, natürlich auch in Abhängigkeit von den verborgenen Neuronen im Hidden Layer. Dies führt nicht nur zu mehr Rechen- und Speicherbedarf,

sondern auch zu einem instabilen und schwerfälligen neuronalen Netz. Des Weiteren würde die große Anzahl von Parametern sehr schnell zu einer Überanpassung (Overfitting) führen und das Netz wäre nicht mehr lernfähig.

Bei Convolutional Neural Networks macht man sich die Tatsache zunutze, dass die vorgenommenen Eingaben aus Bildern bestehen und man für diesen Fall schon im Vorfeld die richtige Architektur dafür anwenden kann. Bild 6.1 vergleicht den Aufbau eines Feedforward-Netzes mit dem Aufbau eines CNNs.

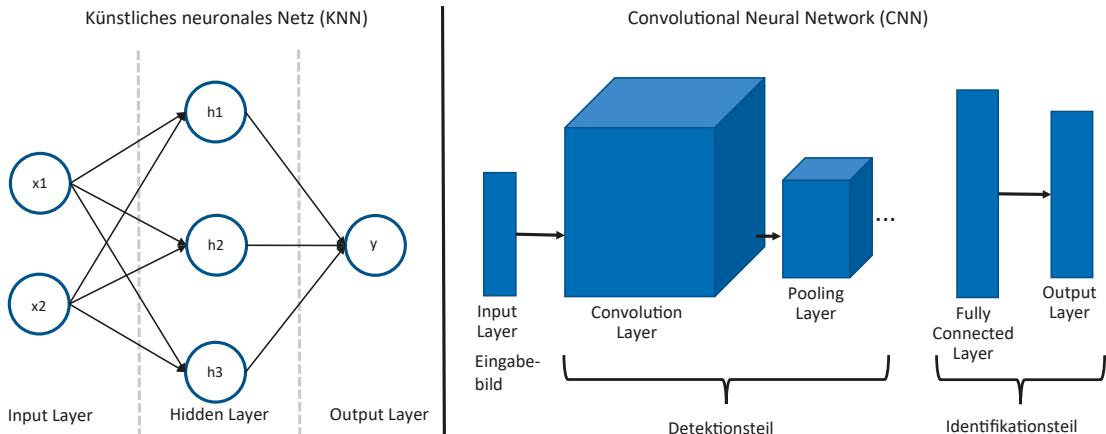


Bild 6.1 Künstliches neuronales Netz (KNN) vs. Convolutional Neural Network (CNN)

Wie Sie sehen, sind die Schichten eines CNN im Gegensatz zu einem KNN in drei Dimensionen (Block-Darstellung) angeordnet: Breite, Höhe, Tiefe.

Das heißt, ein CNN ordnet seine Neuronen in drei Dimensionen (Breite, Höhe, Tiefe) an, daraufhin wandelt es das 3D-Eingangsvolumen in ein 3D-Ausgangsvolumen von Neuronen-Aktivierungen um. Der Detektionsteil besteht dabei aus einer Folge von Convolutional Layern (Faltungsschicht) und Pooling Layern. Diese Schichten extrahieren die komplexen Eigenenschaften eines Bildes. Der Identifikationsteil wiederum besteht aus einem herkömmlichen Feedforward-Netz und nimmt die entsprechende Klassifizierung vor.



2D/3D-Volumen

Da ein Convolutional Neural Network in den meisten Fällen für Bilder verwendet wird, spricht man auch von einem Eingangs- und Ausgangsvolumen, da ein Bild immer aus Breite und Höhe (2D-Volumen) sowie aus einer Anzahl von Farbkanälen (3D-Volumen) besteht. Hierbei handelt es sich um entsprechende 2D- oder 3D-Matrizen, die für die Berechnung im neuronalen Netz benötigt werden.

Oder auch mit anderen Worten, die erste Schicht kann die Kanten im Bild erkennen. Tiefere Schichten sind dann in der Lage, weitere Objekttypen im Bild zu erkennen und weitere Schichten können dann das komplett Objekt, zum Beispiel das Gesicht einer Person, erkennen. Wie das im Einzelnen funktioniert, erläutere ich in Abschnitt 6.2.1 an einem Beispiel für die Kantenerkennung.

■ 6.2 Detektionsteil

Wie in Bild 6.1 gezeigt, besteht der Detektionsteil aus dem Convolutional Layer mit der entsprechenden Aktivierungsfunktion und dem Pooling Layer, der auch als Subsampling Layer bezeichnet wird. Dieser Komplex kann mehrfach hintereinandergeschaltet werden, um die entsprechenden Strukturen im Bild zu erkennen.

6.2.1 Kantenerkennung

Wie oben schon erwähnt, besteht ein einfaches CNN auch aus einer Abfolge von Schichten, und jede Schicht wandelt durch eine differenzierbare Funktion ein 3D/2D-Volumen von Aktivierungen in ein 3D/2D-Ausgangsvolumen um. In diesem Abschnitt geht es am Beispiel von vertikalen und horizontalen Kanten darum, zu zeigen, wie Sie bestimmte Strukturen im Bild erkennen können.

Wir haben ein Graustufenbild (Bild 6.2) mit den Abmessungen $6 * 6 * 1$, da nur ein Farbkanal benutzt wird.

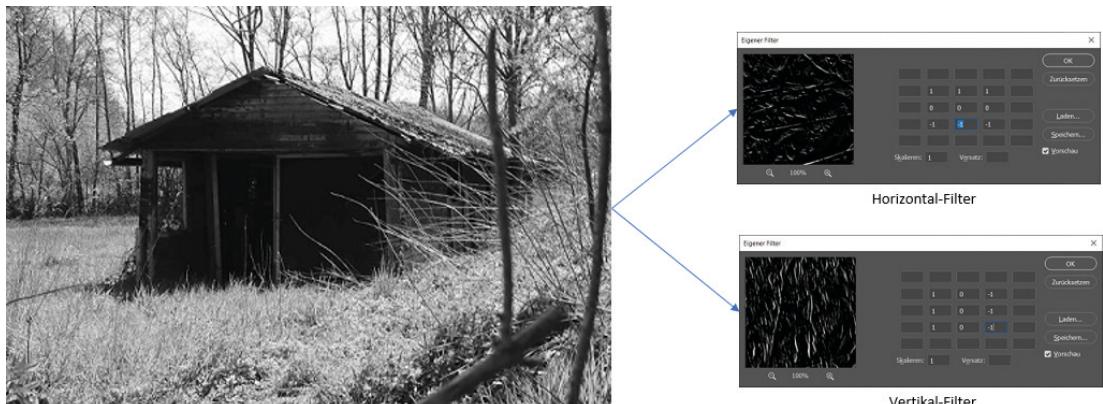


Bild 6.2 Graustufenbild mit Filter

Um jetzt die Kanten im CNN erkennen zu können, wird das vorliegende Bild als Eingangs-matrix benutzt. Dafür transformieren Sie die Pixelwerte des Bildes in eine entsprechende $6 * 6$ -Matrix (Bild 6.3). Diese Matrix stellt den Eingangsvektor für den ersten Convolutional Layer im CNN dar. Für die Transformation der Pixelwerte verwenden Sie einfach eine *Copy-Pixel*-Methode für Bitmaps und konvertieren so das Bild in eine Matrix.

Der erste Convolutional Layer im neuronalen Netz besteht aus einer großen Menge von Neuronen, wobei jedes auf einen kleinen Ausschnitt des Bildes reagiert. Der Ausschnitt eines Bildes im Convolutional Layer ist typischerweise eine $3 * 3$ - oder $5 * 5$ -Matrix, die über das ganze Bild läuft. Jedes Neuron einer Ebene reagiert aber nur auf ein bestimmtes Muster, das durch den $3 * 3$ - bzw. $5 * 5$ -Filter vorgegeben ist. Daher falten Sie die $6 * 6$ -Matrix des Eingangsvektors mit einem $3 * 3$ -Filter (Bild 6.4).

Pixelwerte						Matrixwerte					
3	0	1	2	7	4	3	0	1	2	7	4
1	5	8	9	3	1	1	5	8	9	3	1
2	7	2	5	4	3	2	7	2	5	4	3
0	3	5	8	7	1	0	3	5	8	7	1
4	4	3	9	4	6	4	4	3	9	4	6
2	4	5	4	3	9	2	4	5	4	3	9

Bild 6.3 Vom Pixelwert zur Matrix

$$\text{Matrixwerte } 6 \times 6 \text{ Image} \quad * \quad \text{Filter } 3 \times 3$$

$$\left[\begin{array}{cccccc} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 4 & 3 \\ 0 & 3 & 5 & 8 & 7 & 1 \\ 4 & 4 & 3 & 9 & 4 & 6 \\ 2 & 4 & 5 & 4 & 3 & 9 \end{array} \right] \quad *$$

Bild 6.4 6×6 -Matrix mit 3×3 -Filter

Die Verwendung eines Filters in Form einer 3×3 - oder 5×5 -Matrix bezeichnet man in der Bildverarbeitung als diskrete Faltung. Daher auch der Name Convolutional Neural Network (neuronales Faltungsnetzwerk), da das Eingangsbild gefiltert wird, um entsprechende Merkmale zu extrahieren.

Nach der Berechnung mit dem Filter erhalten Sie ein 4×4 -Bild. Das erste Element der 4×4 -Matrix wird wie in Bild 6.5 dargestellt errechnet.

3 ¹	0 ⁰	1 ¹
1 ¹	5 ⁰	8 ¹
2 ¹	7 ⁰	2 ¹

Bild 6.5Das Ergebnis der 4×4 -Matrix

Die Berechnung wird wie folgt vorgenommen. Sie nehmen die erste 3×3 -Matrix aus dem 6×6 -Bild und multiplizieren Sie mit dem Filter. Nun ist das erste Element der 4×4 -Ausgabe die Summe des elementweisen Produkts dieser Werte:

$$3 \cdot 1 + 0 \cdot 1 + 1 \cdot -1 + 1 \cdot 1 + 5 \cdot 0 + 8 \cdot -1 + 2 \cdot 1 + 7 \cdot 0 + 2 \cdot -1$$

Um jetzt das zweite Element der 4×4 -Ausgabe zu berechnen, verschieben Sie den Filter einen Schritt nach rechts und erhalten so wiederum die Summe des elementweisen Produkts. Diesen Vorgang wiederholen Sie für das gesamte Bild und erhalten dann die gewünschte Gesamtausgabe für die 4×4 -Matrix. Bild 6.6 zeigt die komplett berechnete 4×4 -Matrix.

0 ¹	1 ⁰	2 ¹
5 ¹	8 ⁰	9 ¹
7 ¹	2 ⁰	5 ¹

Bild 6.6

Schritt zwei der Berechnung

Das heißt, wenn Sie einen $6 * 6$ -Eingangsvektor betrachten und diesen mit einem $3 * 3$ -Filter falten, erhalten Sie eine $4 * 4$ -Matrix als Output. Da höhere Pixelwerte im Bild den helleren Teil des Bildes repräsentieren, können Sie auf diese Weise eine vertikale Kante im Bild erkennen.

Der im Beispiel festgelegte Filtertyp hilft bei der Erkennung der vertikalen Kanten im Bild. Sie können aber auch eine Vielzahl von Filtern verwenden, wobei jeder Filter lernt, bei der Eingabe nach etwas Anderem zu suchen. Wenn zum Beispiel der erste Convolutional Layer das Roh-Bild als Input annimmt, dann können verschiedene Neuronen von unterschiedlich orientierten Kanten oder Farbklecksen aktiviert werden.

Bild 6.7 zeigt die benutzten Filter für vertikale- und horizontale Kanten. Sie können aber auch die aus der Bildverarbeitung bekannten Sobel- und Scharr-Operatoren [12] als Filter einsetzen. Mit diesen Filtern wird aus dem Originalbild ein Gradienten-Bild erzeugt. Bei der Verwendung des Operators werden die hohen Frequenzen im Bild mit Grauwerten dargestellt.

$\text{Vertikal} \quad \left\{ \begin{array}{ccc} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{array} \right\}$	$\text{Horizontal} \quad \left\{ \begin{array}{cccc} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{array} \right\}$
$\text{Sobel} \quad \left\{ \begin{array}{ccc} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{array} \right\}$	$\text{Scharr} \quad \left\{ \begin{array}{ccc} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{array} \right\}$

Bild 6.7
Einsatz von Filtern

Das heißt, der Eingangsvektor, der als Hyperparameter definiert ist, wird von einer festgelegten Anzahl von Filtern analysiert, die dann mit einer konstanten Schrittweite über die Matrix des Imports scannen. Dabei wandern die Filter von links nach rechts über den Eingangsvektor und springen nach jedem Durchlauf in die nächsttiefe Zeile. Mit dem so genannten Pooling (siehe Abschnitt 6.2.2) wird dabei festgelegt, wie sich der Filter verhalten soll, wenn er an den Rand der Matrix stößt.

6.2.2 Pooling

Bei der gemachten Berechnung können Sie sehen, dass die Filterung einer Eingabe von $6 * 6$ -Dimensionen mit einem $3 * 3$ -Filter einen Output von $4 * 4$ ergibt. Diese Annahme können Sie jetzt verallgemeinern und bestimmen, dass, wenn die Eingabe $n * n$ und die Filtergröße $f * f$ ist, die Output-Größe wie folgt sein muss:

$$\text{Output: } (n - f + 1) * (n - f + 1)$$

Dies führt allerdings zu zwei Nachteilen:

- Jedes Mal, wenn Sie eine weitere Filteroption anwenden, schrumpft die Größe des Bildes.
- Die am Rand des Bildes vorhandenen Pixel werden im Vergleich zu den zentralen Pixeln nur wenige Male während der Filterung verwendet. Daher kann es am Rand zu Informationsverlust kommen.

Um diese Probleme zu umgehen, werden in der Praxis sehr häufig an den Bildrändern Pixel hinzugefügt. So wird aus einer $6 * 6$ -Matrix für die Eingabe eine $8 * 8$ -Matrix erzeugt. Wendet man darauf jetzt die Filterung mit einer $3 * 3$ -Matrix an, so ergibt sich eine $6 * 6$ -Matrix, die die ursprüngliche Form des Bildes darstellt.

Dieser Vorgang nennt sich Pooling (p) und wird dann wie folgt in der Ausgabe berechnet:

$$\text{Output: } (n + 2p - f + 1) * (n - 2p - f + 1)$$

Somit gibt es zwei gängige Berechnungen für das Ausgabevolumen, einmal ohne Pooling, was auch jederzeit gültig ist, oder mit Pooling für ein genaueres Ergebnis. Sie können das Pooling auch so angeben, dass die Ausgabegröße mit der Eingabegröße übereinstimmt:

$$\text{Somit folgt: } n + 2p - f + 1 = n, \text{ also } p = (f - 1) / 2$$

Der Filter hat für jeden Punkt in seinem Sichtfenster ($3 * 3$ bzw. $5 * 5$) ein festes Gewicht, und er errechnet aus den Pixelwerten im aktuellen Sichtfenster und diesen Gewichten eine Ausgabematrix. Die Größe dieser Ausgabematrix ist somit abhängig von der Größe des Filters, vom Pooling und vor allem von der Schrittweite.

6.2.3 Schrittweite

Neben der entsprechenden Anzahl von Filtern im CNN, müssen Sie auch noch die Schrittweite (Stride = Schritt), mit der Sie den Filter verschieben möchten, als Hyperparameter angeben. Wenn die Schrittweite 1 ist, dann verschieben Sie den Filter jeweils um ein Pixel. Wenn die Schrittweite 2 beträgt, dann springen die Filter beim Verschieben jeweils um 2 Pixel. Eine Schrittweite von über 3 kommt in der Praxis fast nicht vor. Durch den Einsatz einer großen Schrittweite werden räumlich gesehen kleinere Ausgabevolumen erzeugt. Der Output wird mit der Schrittweite (s) wie folgt berechnet:

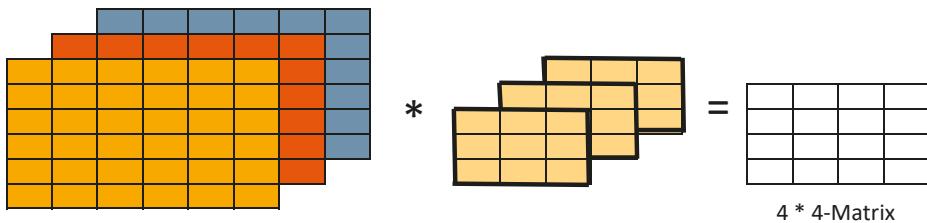
$$\text{Output: } \left(\frac{n + 2p - f}{s} + 1 \right) * \left(\frac{n + 2p - f}{s} + 1 \right)$$

6.2.4 2D- und 3D-Volumen

Bisher haben Sie nur ein 2D-Bild für den Eingangsvektor betrachtet. Nachfolgend spielen wir das Ganze mit einem 3D-Eingangsbild in der Form $6 * 6 * 3$ durch. Hier verwenden Sie dann anstatt eines $3 * 3$ -Filters einen 3D-Filter in der Form $3 * 3 * 3$. Daraus ergibt sich:

- Eingabe: $6 * 6 * 3$
- Filter: $3 * 3 * 3$

Die Abmessungen stellen die Breite, Höhe und die Farbkanäle in der Eingabe und im Filter dar. Sie sollten bei der Arbeit mit einem CNN immer darauf achten, dass die Anzahl der Farbkanäle im Eingang und im Filter gleich ist. Auch dieses Verfahren führt bei 3D-Volumen zu einem Output von $4 * 4$. Bild 6.8 zeigt die visuelle Darstellung.

**Bild 6.8** 3D-Eingabe und 3D-Filter

Da der Eingang drei Farbkanäle besitzt, sollte also auch der Filter unbedingt drei Farbkanäle haben. Nach der Faltung (Filterung) ist die Ausgangsform eine $4 * 4$ -Matrix. Das erste Element der Ausgabe ist somit die Summe des elementweisen Produkts der ersten 27 Werte aus der Eingabe, bestehend aus 9 Werten aus jedem Farbkanal und den 27 Werten aus dem Filter. Diesen Schritt wiederholen Sie für das gesamte Bild.

Wenn Sie für das Beispiel einen weiteren Filter heranziehen, ändert sich auch die Ausgabedimension. Statt eines $4 * 4$ -Outputs hätten Sie einen $4 * 4 * 2$ -Output, beim Einsatz von drei Filtern einen Output von $4 * 4 * 3$ und immer so weiter. Auch die Berechnung der Dimension lässt sich in einer Formel verallgemeinern. Hierbei stellt nc die Anzahl der Farbkanäle im Eingang und im Filter dar, während nc' die Anzahl der Filter angibt:

- Eingabe: $n * n * nc$
- Filter: $f * f * nc$
- Pooling: p
- Schrittweite: s
- Ausgabe: $\left(\frac{n+2p-f}{s}+1\right) * \left(\frac{n+2p-f}{s}+1\right) * nc'$

6.2.5 Aktivierungsfunktion

Auch bei einem Convolutional Neural Network benötigt jedes Neuron eine Aktivierungsfunktion. Das heißt, sobald Sie einen Output nach der Filterung des gesamten Bildes erhalten, fügen Sie diesem Output einen Bias-Wert für die Verzerrung hinzu und wenden dann darauf, wie schon bekannt, eine Aktivierungsfunktion an.

In der Praxis hat sich für die Aktivierung im Convolutional Layer die Rectified Linear Unit, kurz ReLU, als am besten geeignet gezeigt. Bestimmt man dann den Formelbuchstaben c für das Eingangssignal, so kann man die Aktivierungsfunktion sehr einfach bestimmen:

$$\text{ReLU: } f_{\text{akt}}(c) = f_{\text{ReLU}}(c) = \max(0, c) * c$$

Auch bei einem CNN dient die Aktivierung in der ersten Schicht als Input für die zweite Schicht und so weiter. Somit wird ersichtlich, dass die Anzahl der Parameter bei einem CNN unabhängig von der Größe eines Bildes ist. Die Parameteranzahl hängt hier also eindeutig von der Filtergröße ab. Das heißt, bei der Verwendung von 10 Filtern in der Form $3 * 3 * 3$ hat jeder Filter schon einmal 27 Parameter. Hinzu kommt jetzt noch der hinzugezogene

Bias-Wert, so ergibt sich eine Gesamtparameteranzahl von 28 pro Filter. Da Sie 10 Filter verwenden, beträgt die Anzahl der Gesamtparameter für die erste Schicht $28 * 10 = 280$. Demnach ist es also egal, wie groß das Eingangsbild ist, die Parameter hängen nur von der Filtergröße und der Filteranzahl ab.

Infolgedessen können Sie jetzt über die gezeigten Notationen Filtergröße, Pooling, Schrittweite und Anzahl der Filter ein einfaches CNN aufbauen.

6.2.6 Ein sehr einfaches CNN

Bisher haben Sie nur einen Convolutional Layer betrachtet. Je tiefer die Netze sind, umso mehr Convolutional Layer, Pooling Layer bzw. Subsampling-Kombinationen reihen sich aneinander. Die TU Eindhoven [13] (Bild 6.9) zeigt ein sehr komplexes Convolutional Neural Network, in dem Stufe für Stufe immer komplexere Merkmale gelernt werden. Das Durchlaufen der Schritte ermöglicht dann, zahllose Basisstrukturen zu extrahieren. Am Ende kombinieren dann voll verknüpfte Neuronen-Layer die Merkmale zu einer Klassifizierung. Das CNN wird bei der TU für die Klassifikation von Zuständen des Tokamak-Plasmaeinschlusses [14] verwendet.

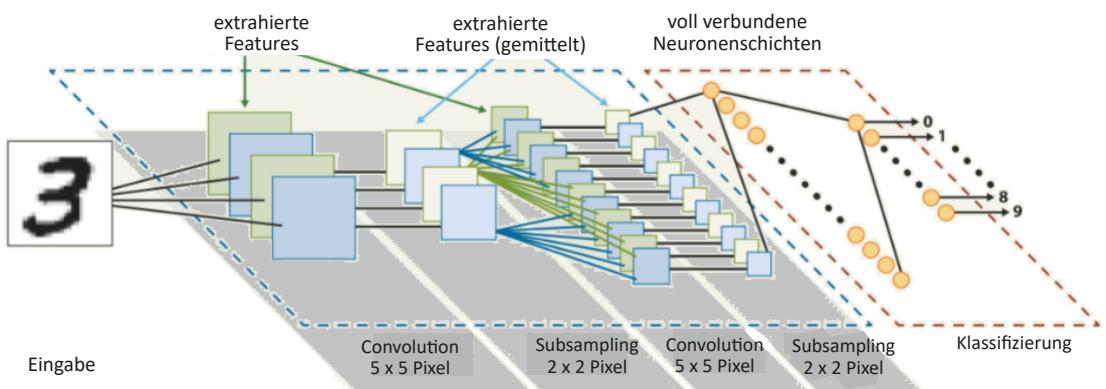
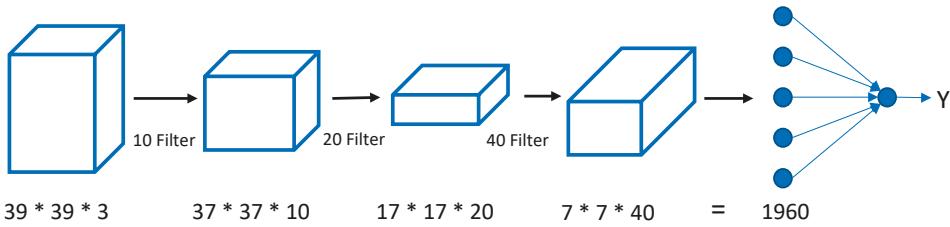


Bild 6.9 Ein komplexes CNN (Quelle: TU Eindhoven)

Wie sieht der Vorgang jetzt aber bei einem einfachen Eingabebild der Größe $39 * 39 * 3$ und 10 Filtern in der Form $3 * 3 * 3$ sowie einer Schrittweite von 1 aus. Es wird kein Pooling verwendet, das heißt, Sie füllen die Ränder nicht mit Null-Pixeln auf.

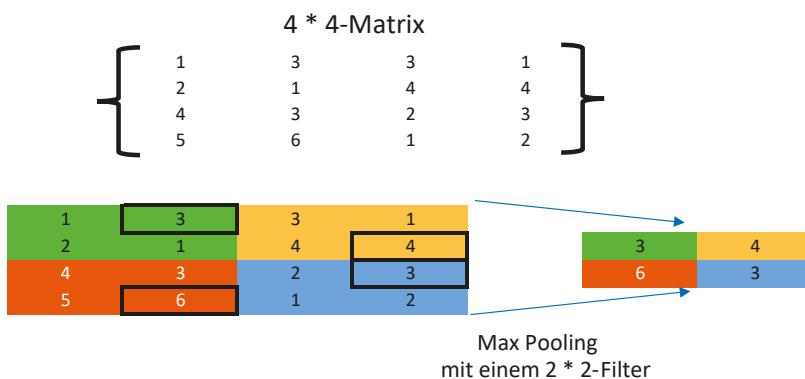
Sie erhalten eine erste Ausgabe von $37 * 37 * 10$. Jetzt filtern Sie diesen Output ein weiteres Mal und erhalten dann einen Output von $7 * 7 * 40$. Das Ergebnis des Outputs beträgt somit 1960 Parameter. Diese werden jetzt als Vektor an einen Klassifikator weitergegeben, der auf Grund von Merkmalen eine Klassifizierung vornimmt. Bild 6.10 zeigt die einfache schematische Darstellung des einfachen CNN.

Durch die Verwendung einer Reihe von Hyperparametern wie Anzahl der Filter, Größe der Filter, der zu verwendenden Schrittweite und natürlich dem Pooling lassen sich vielfältige Einstellungen im Vorfeld für das neuronale Netz festlegen. Aber welche Aufgabe hat das Subsampling?

**Bild 6.10** Ein einfaches CNN

6.2.7 Subsampling

Das Subsampling, in einem CNN auch als Pooling Layer bezeichnet, wird verwendet, um die Größe der Eingabe zu reduzieren und damit die Berechnung zu beschleunigen. Das heißt, Ziel des Subsampling ist es, die Dimensionalität zu reduzieren, indem die davorliegende Schicht gefiltert und deren Ausbreitung in x- und y-Richtung reduziert wird. Betrachten Sie hierfür die $4 * 4$ -Matrix aus Bild 6.11.

**Bild 6.11** $4 * 4$ -Matrix mit Ausführung des Max Pooling beim Subsampling

Für das Subsampling wendet man jetzt auf diese Matrix ein Pooling an – daher auch die Bezeichnung Pooling Layer –, was zu einer Ausgabe als $2 * 2$ -Matrix führt. Das Subsampling passt die Größe räumlich an, indem die MAX-Operation des Pooling verwendet wird. Die gebräuchlichste Form ist ein Subsampling mit Filtern der Größe $2 * 2$, die mit einer Schrittweite von 2 Abtastungen sowohl in der Breite als auch in der Höhe die Eingabe vornimmt.

Diese Form wird als Maximal Pooling bezeichnet. Zusätzlich zum maximalen Pooling können diese Einheiten, oder auch Units genannt, noch weitere Funktionen erfüllen, wie zum Beispiel ein Average Pooling (Durchschnittswert) oder sogar ein L2-Norm-Pooling bei der Verarbeitung von Textdaten. Das Average Pooling wurde in der Anfangszeit der CNNs häufig verwendet, ist aber inzwischen vom Maximal Pooling abgelöst worden. Daher wird jetzt auch im Buch die Bezeichnung Pooling Layer verwendet, da es sich um eine eigene Schicht im CNN handelt.

6.2.8 CNN mit Pooling Layer

Bild 6.12 zeigt ein CNN mit Convolutional Layer und Pooling Layer. Die Eingabe liegt in Form einer $32 * 32 * 3$ -Matrix vor.

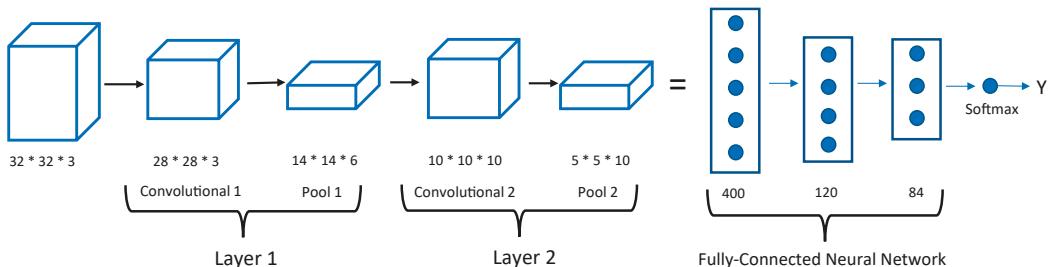


Bild 6.12 CNN mit Pooling Layer

Dieses neuronale Netz besteht aus einer Kombination von Convolutional Layer und Pooling Layer am Anfang und ein paar vollständig verbundenen Neuronen-Lagern am Ende und schließt mit einem Softmax-Klassifikator, um den Input in verschiedene Kategorien einzutragen. Die Hyperparameter für die Filter sind jeweils 5 und 2 und die Schrittweite beträgt 1 im Convolutional Layer und 2 im Pooling Layer.

Durch den Einsatz des Pooling Layers schrumpft die Breite und Höhe der Eingabe mit zunehmender Tiefe des Netzwerks von $32 * 32$ auf $5 * 5$, während die Anzahl der Kanäle von 3 auf 10 ansteigt. Durch die Faltung, also das Filtern im Netz, hat man zwei wesentliche Vorteile. Als Erstes die gemeinsame Nutzung von Parametern und als Zweites eine entsprechende Sparsamkeit bei der Neuronen-Verbindung.

Wenn Sie das oben genannte Beispiel ohne Filterung verwenden und somit eine vollständig verbundene Schicht aufbauen würden, wäre die Anzahl der Parameter $32 * 32 * 3 * 28 * 28 * 6 = 14.450.688$, also etwas mehr als 14 Millionen Parameter. Das würde allerdings beim Training des Netzes zu einer erheblichen Rechenzeit führen und wäre in diesem Fall für eine Bildanalyse nicht optimal.

Betrachten Sie die Anzahl der Parameter aber im Zusammenhang mit der Filterung, so kommen Sie beim Einsatz von 6 Filtern auf $(5 * 5 + 1) * 6 = 156$ Parameter. Durch das Falten reduzieren Sie die Anzahl der Parameter enorm und beschleunigen das Training des Modells erheblich.

Der zweite Vorteil der Faltung ist die Sparsamkeit bei den Neuronen-Verbindungen. Für jede Schicht in einem CNN hängt jeder Output-Wert von einer kleinen Anzahl von Eingängen ab, anstatt alle Eingänge zu berücksichtigen.

■ 6.3 Identifikationsteil

Der Identifikationsteil in einem CNN besteht aus einem vollständig verbundenen (fully-connected) neuronalen Netz und entspricht somit in seiner Funktion einem Feedforward Neural Network (FNN). Wie Sie wissen, besitzen in FNNs die Neuronen in einer vollständig verbundenen Schicht eine vollständige Verbindung zu allen Aktivierungen in der vorherigen Schicht. Die Aktivierung kann daher mit einer Matrix-Multiplikation berechnet werden, gefolgt von dem Bias-Wert für die benötigte Verzerrung.

Der einzige Unterschied zwischen einer FNN- und einer CNN-Schicht besteht darin, dass die Neuronen in der CNN-Schicht nur mit einer lokalen Region im Eingang verbunden sind und dass viele der Neuronen gemeinsame Parameter haben. Die Neuronen in beiden Schichten berechnen sich aber nach wie vor aus dem Punktprodukt, sodass ihre funktionelle Form identisch ist. Daher ist es möglich, zwischen FNN-Schichten und CNN-Schichten zu konvertieren.

Dense und Flattening

Der Klassifizierer in einem CNN wird auch als Dense Layer bezeichnet. Der Dense Layer tastet sich von der letzten Pooling-Schicht aus abwärts. Damit das nachgeschaltete neuronale Netz aber mit den Parametern aus der letzten Pooling-Schicht arbeiten kann, ist eine Verflachung notwendig. Das heißt, der mehrdimensionale Output aus der Pooling-Schicht muss in einen eindimensionalen Vektor überführt werden. Diesen Vorgang nennt man Flattening.

Dropout

Über einen sogenannten Dropout Layer ist es möglich, während der Trainingsphase eine Überanpassung (Overfitting) zu vermeiden. Hierfür werden zufällig einige Einheiten (Units) und ihre Input- und Output-Verbindungen aus dem neuronalen Netz entfernt (drop out). So will man im Netz sicherstellen, dass die Einheiten sich nach Möglichkeit individuell voneinander unterscheiden, um so die Überanpassung zu verhindern.

Auswertung

Die Ausgabe der letzten Schicht im CNN basiert auf der Umsetzung der Mehrklassen-Klassifikation wie Sie sie schon in Abschnitt 3.6.1.2 kennengelernt haben. Das CNN liefert immer Ausgangsvektoren mit genauso vielen Elementen zurück, wie Klassen vorgegeben sind. Auch hier wird für das Lösen des Mehrklassenproblems im Output die logistische Regression per Softmax-Funktion durchgeführt.

■ 6.4 Schlussbemerkung

In diesem Kapitel haben Sie die Grundlagen von Convolutional Neural Networks kennengelernt. Sie haben erfahren, wie ein CNN funktioniert und wie die verschiedenen Bausteine ineinander greifen. Inzwischen gibt es eine Vielzahl von Convolutional Networks, die in den Machine Learning Frameworks der verschiedenen Anbieter zur Verfügung stehen. Abschnitt 10.3, „Convolutional Neural Networks für die Objekterkennung“, zeigt die Implementierung eines CNN in C#.

7

Machine Learning Frameworks

Als Google 2015 das Framework TensorFlow für Machine Learning und Deep Learning veröffentlichte, befanden sich die Systeme für künstliche Intelligenz für die breite Masse von Entwicklern und Forschern noch in der Erprobungsphase bzw. im Experimentierstadium. Dank günstiger Cloud-Umgebungen und leistungsfähiger Hardware ist die Anzahl der Machine und Deep Learning Frameworks in den letzten Jahren rasant gestiegen. Langsam, aber sicher etabliert sich ML in vielen Unternehmen, da heute Workflows und Best Practices für ML zur Verfügung stehen und mithilfe der Frameworks sich viele Business-Aufgaben schneller und effizienter erledigen lassen.

Auch in branchenspezifischen Anwendungen spielen ML-Funktionen zunehmend eine entscheidende Rolle, aber trotz der vielfältigen Anwendungsmöglichkeiten und dem starken Interesse an Machine-Learning-Lösungen mangelt es vielfach noch an der Umsetzung.

Diese Zurückhaltung kann unterschiedlichste Gründe haben, allerdings haben nahezu alle Machine Learning Frameworks eines gemeinsam: Sie nutzen jeweils nur ihr eigenes Datenformat, um entwickelte Modelle zu persistieren (speichern) und bereitzustellen. Das führt bei der Fülle der Angebote an ML-Modellen zu noch mehr Unübersichtlichkeit. Hier schafft der *Open Neural Network Exchange* (ONNX)-Standard eine Möglichkeit, um ein ML-Modell, das zum Beispiel mit TensorFlow entwickelt und trainiert wurde, zu exportieren, um es dann mit Microsofts ML.NET Framework wieder zu laden und auszuführen, um es dann in einer Anwendung bereitzustellen.

Es gibt mittlerweile eine sehr große Auswahl an Machine Learning Frameworks und jedes davon bietet unterschiedliche Vor- und Nachteile. Daher sollte auf jeden Fall die Art des Projekts, in dem man Machine Learning einsetzen möchte, für die Auswahl des Frameworks ausschlaggebend sein.

Berücksichtigen Sie bei Ihrem Projekt auch immer, dass es zum Trainieren der ML-Algorithmen in vielen Anwendungsgebieten großer Datenmengen bedarf und somit für das Training mit dem entsprechenden ML-Modell eine leistungsstarke Hardware zu empfehlen ist. Da zum Optimieren und Austesten der ML-Modelle auch immer wieder Anpassungen an den Hyperparametern durchgeführt werden, macht sich vor allem in der Entwicklungsphase eine schnelle Hardware auf zwei oder mehr Grafikkarten (Graphics Processing Unit – GPU) rasch bezahlt.

Alle großen Cloud-Anbieter stellen inzwischen *Machine Learning as a Service* (MLaaS) zur Verfügung. So ist es für sehr rechenintensive ML-Modelle möglich, diese auf massiv paralleler Hardware in der Cloud auszuführen, um die Rechenzeit zu verringern.

Des Weiteren fällt auf, dass fast jedes Framework und jede API (*Application Programming Interface*) im Bereich von Machine Learning eine Verbindung zur Programmiersprache Python hat.

Zurzeit ist Python immer noch die am meisten verwendete Sprache, wenn es um künstliche Intelligenz geht.

Wenn Sie sich nicht unbedingt mit objektorientierter Programmierung sowie Frontend- und Backend-Entwicklung beschäftigen möchten, sondern nur aus Daten effektiv und einfach neue Erkenntnisse gewinnen möchten, so könnte Python die Programmiersprache Ihrer Wahl im Bereich Data Science und Machine Learning sein.

Somit könnte man auf den ersten Blick annehmen, C# spielt keine allzu große Rolle im Machine-Learning-Umfeld, einmal abgesehen von ML.NET. Da Machine Learning aber immer mehr als Teil bestehender Softwaresysteme eingesetzt werden soll und man somit auch nach Möglichkeit einen reibungslosen Ablauf der Integration in bestehende Produktivsysteme gewährleisten will, stellt C# zusammen mit dem .NET Framework und dem .NET Core Framework, Visual Studio und MS SQL Server ein sehr gutes Ökosystem dar. Daher wird auch C# im Umfeld von ML immer wichtiger und die Sprache ist auf jeden Fall die erste Wahl, wenn es um den Einsatz in produktiven Systemen geht oder schon bestehende Systeme erweitert werden sollen.

■ 7.1 Einbindung von ML-Frameworks in C#

Für die Integration eines entwickelten Machine-Learning-Modells bzw. eines ML-Frameworks in ein C#-Produktivsystem existieren unterschiedliche Möglichkeiten. Die drei gängigsten Methoden sind:

- Das ML-Modell lässt sich für das C#-Projekt kapseln. Dafür sorgen die Zugriffe über eine entsprechende Programmierschnittstelle (API) auf das ML-Modell oder Framework. Die klar definierten Schnittstellen sorgen für wenige Abhängigkeiten und eine saubere Trennung der Komponenten. Zu diesen Schnittstellen zählen aber nicht nur Application Programming Interfaces (API) oder der Zugriff über einen entsprechenden ML-Service bei den Cloud-Anbietern, sondern auch Software Development Kits (SDKs), wie sie zum Beispiel für den Zugriff auf Amazons Web Service verwendet werden.
- Eine weitere Option ist der Export des ML-Modells und der anschließende Import in eine der für C# zur Verfügung stehenden ML-Bibliotheken. Hierfür wird in vielen Fällen das Open Neural Network Exchange (ONNX)-Format benutzt. Der Nachteil hierbei ist: Wird das ML-Modell auf neue Daten trainiert oder müssen Modifikationen an dem Modell vorgenommen werden, so muss ein erneuter Export des Modells vorgenommen werden und es muss wieder neu in das Produktivsystem importiert werden.
- Sie können das ML-Modell aber auch selber in C# vollständig für den Produktionseinsatz neu entwickeln. Hier kommt es vor allem auf die Komplexität des ML-Modells und des Projekts an, ob sich eine Eigenentwicklung rechnet oder der komplette Kosten- und Zeitrahmen gesprengt wird. Vielleicht kann hier auch der Einsatz des ML.NET Frameworks für die schnellere Umsetzung einer lokalen Anwendung hilfreich sein.

Wichtig bei der Verwendung von ML-Modellen und -Frameworks ist es, sie weitgehend nahtlos in ein entsprechendes Projekt zu integrieren. Es wäre in einem Produktivsystem sehr

wünschenswert, wenn sich die Verwendung eines ML-Modells als einfacher Funktionsaufruf gestalten ließe.

Durch den Einsatz von C# ist es problemlos möglich, im ML-Umfeld zu experimentieren, schnell zu iterieren und gleichzeitig robusten und gut wartbaren Code zu schreiben. Mit C# lässt sich ein ML-Workflow vom Entwurf bis zur Produktivumgebung sehr gut umsetzen. Um aber nicht jedes Mal das Rad wieder neu erfinden zu müssen, kann der Einsatz eines Machine Learning Frameworks von großem Vorteil sein.

■ 7.2 TensorFlow

Wenn man sich heute mit Machine Learning beschäftigt, trifft man sehr schnell auf Googles ML-Framework TensorFlow. Das Framework steht als Open-Source-Projekt unter der Apache-2.0-Lizenz zur Verfügung.

TensorFlow ist zurzeit sicher das beliebteste und bekannteste ML Framework. Es ist aktuell in der Version 2.0 freigegeben. Allerdings kann die Nutzung auf Grund der Low-Level-Programmierung für ungeübte Entwickler sehr schnell etwas irreführend und unhandlich sein. Diesen Umstand versuchte man durch die Bereinigung von APIs und der Überarbeitung von diversen Modi in der Version 2 zu lösen, um so die Programmierung zu vereinfachen.

Entwickler können TensorFlow als Framework nutzen, um verschiedene Modelle für das Machine Learning zu entwickeln. Aufgrund dessen ist TensorFlow auch für die sogenannte Datenstromorientierte Programmierung ausgelegt. Das Framework arbeitet mit einem kontinuierlichen Datenstrom und implementiert daraus einen Berechnungsgraphen, um neuronale Netzyzyklen frei darzustellen. Mathematisch und technisch ausgedrückt heißt das, dass TensorFlow komplexe, sich immer wiederholende parallele Berechnungen aus Tensoren (Daten) ausführen kann. Daraus folgt, dass eine Berechnung in TensorFlow immer wie ein Datenfluss-Berechnungsgraph aufgebaut ist. Der Graph besteht hierbei aus Knoten und Kanten. Die Knoten des Graphen stellen Operationen dar, wie etwa Addieren oder Multiplizieren. Somit findet alles bei TensorFlow in einem vorher definierten Berechnungsgraphen statt. In Abschnitt 3.3.4, „Tensor“, haben Sie erfahren, dass es sich bei einem Tensor um einen mehrdimensionalen Vektor handelt. Bild 7.1 zeigt die numerische Berechnung eines gerichteten Datenfluss-Berechnungsgraphen in TensorFlow.

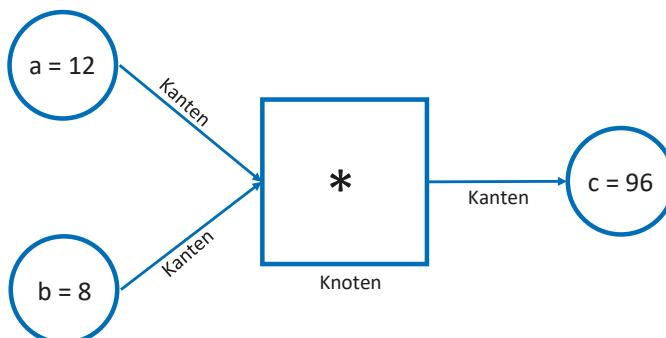


Bild 7.1
Berechnungsgraph
in TensorFlow

Wie Sie in Bild 7.1 sehen, bezeichnet der Berechnungsgraph eine abstrakte Darstellung des zugrunde liegenden mathematischen Problems in Form eines gerichteten Diagramms.

Das Diagramm besteht in seiner Form aus Kanten und Knoten, die miteinander verbunden sind. Die Eingangssignale werden durch Kanten in den Knoten eingespeist, verändert und ausgegeben. Durch die richtige Verbindung der Knoten kann ein Berechnungsgraph erstellt werden, der die notwenigen Daten und mathematischen Operationen zur Erstellung eines neuronalen Netzes beinhaltet.

In dem oben gezeigten Beispiel werden einfach zwei Zahlen miteinander multipliziert. Hierfür werden die Zahlen in den Variablen a und b gespeichert. Die Variablen fließen über ihre Kanten durch den Berechnungsgraphen bis zu dem Knoten, an dem die Multiplikation durchgeführt wird. Das Ergebnis wird dann in der Variable c gespeichert. Die aufgezeigten Variablen werden in TensorFlow einfach als *placeholder*, also Platzhalter, bezeichnet.

Somit werden alle Zahlen im Berechnungsgraphen nach dem gleichen Ablauf verarbeitet. Die interne Struktur von TensorFlow setzt sich wie folgt zusammen:

- Die Tensoren stellen Objekte bzw. Datenstrukturen dar, die Vektoren oder mehrdimensionale Matrizen enthalten.
- Die Knoten des Berechnungsgraphen entsprechen mathematischen Operatoren und sorgen so für die Berechnung.
- Die Knoten des Berechnungsgraphen bilden die Tensoren und geben diese an andere Knoten weiter.

7.2.1 Ablauf in TensorFlow

Die Arbeit mit TensorFlow kann immer in zwei wesentliche Phasen unterschieden werden:

1. **Erstellen eines Berechnungsgraphen:** In dieser Phase wird über die zur Verfügung stehenden Ressourcen (Daten etc.) und die gewünschte Bedingung ein Modell modelliert.
2. **Ausführen des Berechnungsgraphen:** Nachdem das Modell erstellt ist, können Sie dieses aufführen.

Alle Berechnungen im Graphen finden in TensorFlow in einer gekapselten Session statt. Hierdurch ist es auch möglich, in der Session den fertigen Berechnungsgraphen zu laden, oder sogar nach und nach über entsprechende API-Aufrufe einen neuen Berechnungsgraphen zur Laufzeit zu erzeugen. Ist der Berechnungsgraph erzeugt und initialisiert, so interagiert TensorFlow nur noch durch das Aufrufen von Operationen im Berechnungsgraphen.

Die Knoten, die die Operationen repräsentieren, beinhalten die von TensorFlow zur Verfügung gestellten Operatoren wie Addition, Subtraktion, Multiplikation aber auch die für neuronale Netze benötigten mathematischen Hyperbelfunktionen. Eine ausführliche Dokumentation aller Operatoren finden Sie unter [15].

7.2.2 Das TensorBoard

Es ist in TensorFlow mithilfe des sogenannten *TensorBoard* (Bild 7.2) möglich, den Berechnungsgraphen grafisch darzustellen.

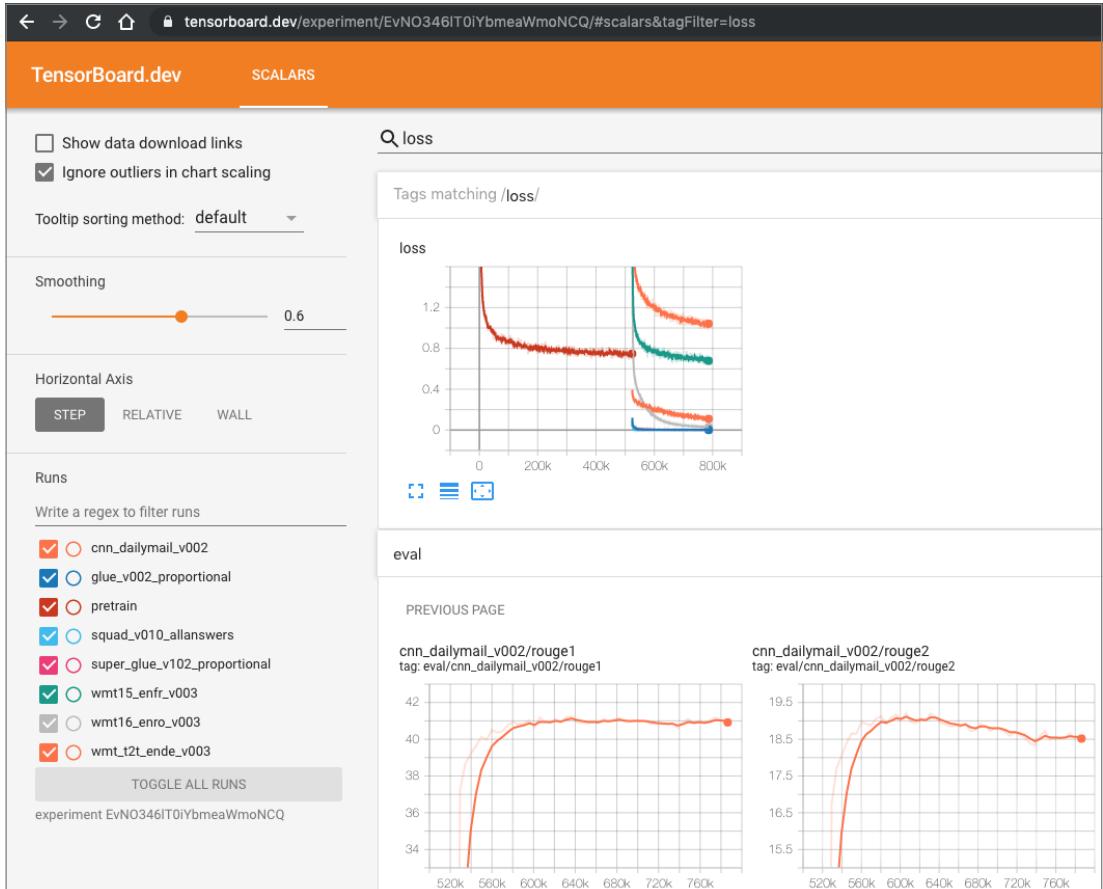


Bild 7.2 Das TensorBoard im Einsatz (Quelle: *TensorFlow.org*)

Somit kann über die grafische Aufbereitung der Status des Trainingsfortschrittes visuell nachverfolgt werden. Das Tensor Board kann neben dem Fortschritt auch Bilder anzeigen, die durch den Berechnungsgraphen fließen. Auch das Debugging im Berechnungsgraphen ist mit dem Tool zur Laufzeit möglich. Des Weiteren stehen noch folgende Visualisierungen zur Verfügung:

- Verfolgen und Visualisieren von Metriken wie Verlust und Genauigkeit
- Visualisieren des Modellgraphen mit allen Operationen und Schichten.
- Anzeigen von Histogrammen von Gewichten, Verzerrungen und anderen Tensoren.
- Anzeigen von Bildern, Text und Audiodaten

7.2.3 Begriffe

In der Praxis treffen Sie bei der Arbeit mit TensorFlow immer wieder auf folgende wichtige Begriffe:

- **Tensor:** Der Tensor stellt die Grundlage für die Berechnung in TensorFlow dar. Alle Daten, die TensorFlow intern nutzt, sind in Tensoren gekapselt.
- **Berechnungsgraph (kurz Graph):** Hierunter verbirgt sich die Definition von TensorFlow-Berechnungsabläufen.
- **Inferenz:** Unter Inferenz versteht man so viel wie ein bestimmtes Verfahren anzuwenden. Dazu zählen die von TensorFlow zur Verfügung gestellten Machine-Learning-Verfahren wie Klassifizieren, Vorhersagen, Übersetzen oder auch die Bildanalyse und vieles mehr.
- **Modell:** Das Modell stellt das Ergebnis des Lernprozesses dar. So zum Beispiel das mit Daten trainierte neuronale Netz.
- **Session:** Die Session bildet den eigentlichen Kontext, in dem TensorFlow ausgeführt wird. Das heißt, damit Sie überhaupt mit TensorFlow arbeiten können, müssen Sie eine Session erzeugen, in die der Berechnungsgraph geladen wird, der dann mit einem Modell initialisiert wird.

TensorFlow ist ein sehr umfangreiches Machine Learning Framework, welches sich für Anwendungsfälle in vielen Bereichen nutzen lässt. So zum Beispiel in der Medizin, im Finanzsektor, aber auch in der Industrie und in der Wirtschaft. Auch Google selbst nutzt TensorFlow für die Bildidentifikation, für Übersetzungen und in Google Maps.

Wie bereits erwähnt, benötigt man für die Nutzung von TensorFlow durch den Funktionsumfang und die vielfältigen Möglichkeiten eine gewisse Einarbeitungszeit und die Lernkurve ist auch entsprechend steil. TensorFlow bildet für Sie als Entwickler sozusagen das Backend für Ihr ML-Modell ab. Unter TensorFlow findet der Lernprozess statt, bei dem das neuronale Netz optimiert wird. Daher ist es für Sie als C#-Entwickler, viel einfacher und effektiver über eine entsprechende Programmierschnittstelle den Zugriff auf TensorFlow zu realisieren. Inzwischen lässt sich TensorFlow auch sehr gut mit Microsoft ML.NET, SciSharp-TensorFlow.NET oder auch Microsoft Azur ML Studio nutzen. Um den Einsatz von SciSharp-TensorFlow.NET geht es in Kapitel 8.

7.2.4 TensorFlow Playground

TensorFlow hat mit dem Playground (Bild 7.3) eine Möglichkeit geschaffen, einfach und schnell ohne Softwareinstallation oder Programmierkenntnisse mit ML-Modellen im Webbrowser zu experimentieren. So können Sie unter [16] erste Experimente mit TensorFlow durchführen.

Über den Playground können Sie Ein- und Ausgabe-Parameter einstellen, eine kleine Netzwerktopologie aufbauen, Aktivierungsfunktionen und Lernrate bestimmen. Durch den TensorFlow Playground und den Experimenten sind Sie in der Lage, sich ein wenig mit der Funktionsweise von TensorFlow auch ohne Programmierung vertraut zu machen.

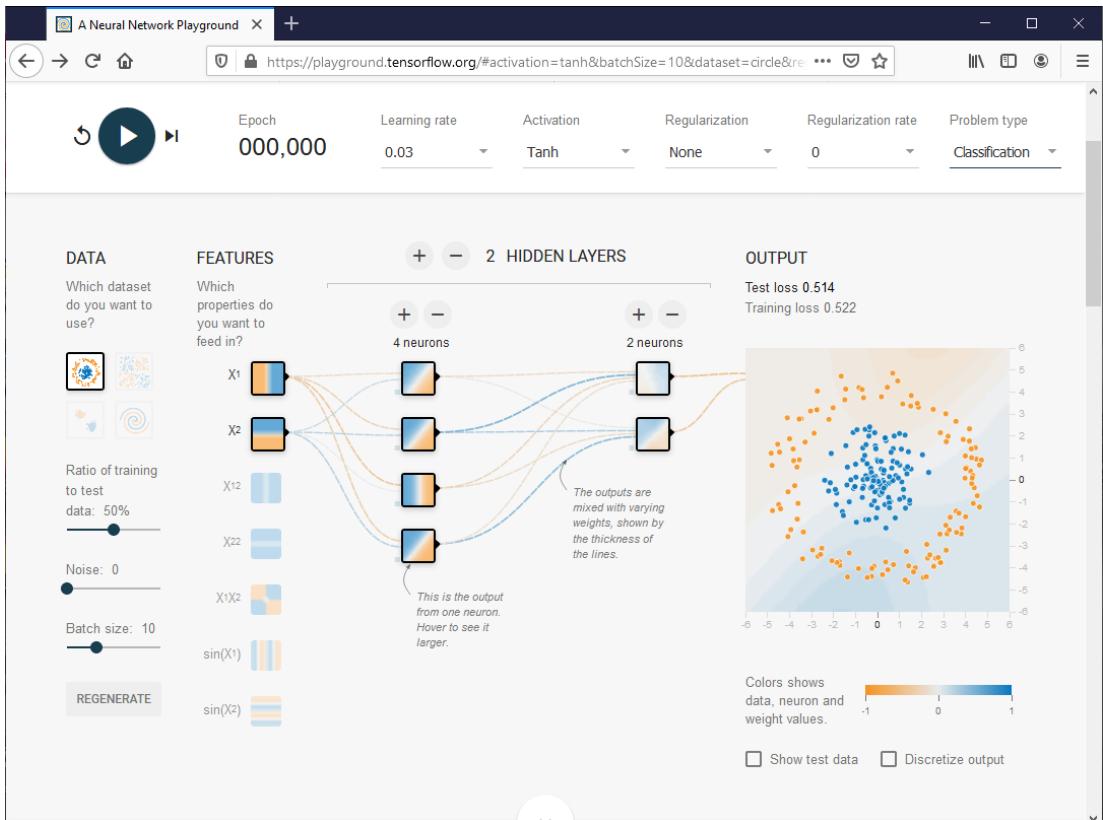


Bild 7.3 TensorFlow Playground im Einsatz

■ 7.3 Keras

Beschäftigt man sich mit TensorFlow, so trifft man automatisch auf Keras. Hierbei handelt es sich um eine Open-Source-Bibliothek, die von Netflix, Uber, Square und vielen anderen verwendet wird. Der große Unterschied zu den meisten ML-Frameworks ist, dass Keras nicht eigenständig eingesetzt wird, sondern als eine Art Interface in TensorFlow Verwendung findet.

Keras wurde zum größten Teil von dem Google-Entwickler Francois Chollet entwickelt und dient der schnellen Implementierung eines neuronalen Netzes. Die Bibliothek steht als High-Level-API bereit und unterstützt sowohl Convolutional Neural Networks (CNN) als auch Recurrent Neural Networks (RNN) sowie eine Kombination aus beiden.

Die zentralen Merkmale von Keras sind Einfachheit, Erweiterbarkeit und Modularität. Während der Entwicklung der jetzt aktuellen Version von Keras, lag der Fokus der Entwickler vor allem auf einer besseren Bedienbarkeit von Keras. Ein neues Design vereinfacht jetzt

die Erstellung und die Verwendung von ML-Modellen in Keras. Das heißt, das Designprinzip der aktuellen Keras-Version unterstützt die Anwender in der Art und Weise, wie sie Modelle erstellen, Schichten definieren oder mehrere Input-Output-Modelle einrichten. Nach der Gestaltung Ihres ML-Modells kompiliert Keras also Ihr ML-Modell mit Verlust- und Optimierungsfunktionen und unterstützt Sie beim Trainingsprozess. So können Sie mit einfachen Mitteln neuronale Netze erstellen und konfigurieren, ohne sich detailliert mit dem zugrunde liegenden Backend beschäftigen zu müssen. Inzwischen ist Keras vollständig in TensorFlow integriert und somit direkt in TensorFlow einsetzbar.

Auch Keras lässt sich auf unterschiedlichen Betriebssystemen und Plattformen nutzen. Als C#-Entwickler finden Sie entsprechende Unterstützung für Keras im Microsoft ML.NET Framework und im SciSharp-Keras.NET-Technologie-Stack.

■ 7.4 Infer.NET

Bei Infer.NET handelt es sich um ein plattformübergreifendes Machine Learning Framework, das als Open-Source-.NET-Bibliothek auf GitHub unter einer MIT-Lizenz freigegeben ist. Die Anfänge von Infer.NET gehen auf das Jahr 2004 zurück und seitdem hat es sich von einem Forschungsprojekt zu einem Machine Learning Framework entwickelt. Es ist inzwischen auch Teil des ML.NET Frameworks und wird in den Microsoft-Produkten Office, Xbox und Azure eingesetzt.

Infer.NET ermöglicht einen modellbasierten Ansatz zum maschinellen Lernen. Dieser Ansatz umfasst eine Vielzahl von Möglichkeiten, die sich in Darstellung, Formulierungsgrad und Wahl der Werkzeuge, zum Beispiel der Beschreibungssprache, unterscheiden. In der Modellierungsphase können die Anwender auch direkt Ihr Domänenwissen in das Modell von Infer.NET integrieren. Aus dem Modell lässt sich dann im Infer.NET Framework ein maschineller Lernalgorithmus erstellen. Sie können neben den domänenspezifischen Problemen auch Standardaufgaben wie Klassifikation, Empfehlungen oder Clustering lösen.

Der modellbasierte Ansatz im Infer.NET Framework basiert auf der Ausführung von Bayesschen Inferenzen (Rückschlüsse bzw. Schlussfolgerungen) in grafische Modelle. Folglich können in Infer.NET verschiedene Bayessche Modelle wie Bayes-Point-Machine-Klassifikatoren [17], TrueSkill-Matchmaking [18], Hidden-Markov-Modelle [19] und Bayessche neuronale Netze, auch als Bayesian Neural Network (BNN) bezeichnet, implementiert werden.



Inferenzen

Die Bayessche Inferenz ist eine Methode der statistischen Inferenzen (Schlussfolgerungen), bei der der Satz von Bayes verwendet wird, siehe Abschnitt 2.4.5, „Bayes-Klassifikation“. Mit anderen Worten stellt die Bayessche Inferenz (Bayes-Statistik) einen Zweig in der Statistik dar, in der mit dem Bayesschen Wahrscheinlichkeitsbegriff und dem Satz von Bayes Fragestellungen in der Stochastik untersucht werden.

Der Vorteil eines neuronalen Netzes auf Basis von Bayes ist, dass der Bayessche Wahrscheinlichkeitsbegriff keine unendlich oft wiederholbaren Zufallsexperimente voraussetzt, sodass Bayessche Methoden auch bei kleinen Datenmengen anwendbar sind. Weitere Informationen finden Sie auch auf der Infer.NET-Webseite unter *Resources and References* [20].

Ein Bayessches Netz ist ein gerichteter azyklischer Graph, das heißt, der gerichtete Graph enthält keinen Zyklus. Hier beschreiben die Knoten Zufallsvariablen und die Kanten bedingte Abhängigkeiten zwischen den Variablen. Das BNN versucht, die gemeinsame Wahrscheinlichkeitsverteilung aller beteiligten Variablen unter Ausnutzung bedingter Unabhängigkeiten möglichst kompakt zu repräsentieren.

Nachdem man mit Infer.NET sein gewünschtes Modell konzipiert hat, wird es mithilfe der Modellierungs-API des Frameworks in ein probabilistisches Programm übersetzt.

7.4.1 Probabilistische Programmierung

Bei der probabilistischen Programmierung handelt es sich um eine relativ neue Richtung im Bereich Data Science und Machine Learning. Mithilfe der probabilistischen Programmierung möchte man komplexe statistische Schlussfolgerungen und Machine Learning einer breiteren Zielgruppe zugänglich machen. Das Projektteam von Infer.NET definiert die probabilistische Programmierung als eine Möglichkeit, statistische Modelle von Prozessen aus der realen Welt zu erstellen [21].

Durch die probabilistische Programmierung soll es für den Entwickler möglich sein, Wahrscheinlichkeitsmodelle leichter im Vorfeld zu modellieren und diese Modelle anschließend über einen Algorithmus zu lösen. Mit der probabilistischen Programmierung können sogenannte unsichere und unvollständige Daten verarbeitet werden. Infer.NET fungiert in diesem Fall als eine sogenannte *Inference Engine*, die in der Lage ist, spezielle Inferenzalgorithmen auf ein probabilistisches Programm anzuwenden. Hierdurch erreicht man eine gewollte und nützliche Trennung zwischen Modellbildung und Inferenz.

Die probabilistische Programmierung ist also darauf ausgelegt, mit entsprechenden Unsicherheiten, also fehlenden Daten, Ausreißern in Datenwerten, fehlenden Informationen für die Entscheidungsfindung usw., umzugehen. Sie erlaubt dadurch eine bessere Interpretation der Ergebnisse einer Prognose.

Die erzeugten Modelle werden zwar nicht genauer, aber man wird in die Lage versetzt, das Ergebnis besser einzuschätzen, ob es große Differenzen zwischen dem tatsächlichen Wert und der Prognose gibt. Die probabilistische Programmierung implementiert hierfür ein Verfahren, das auf Zufallszahlen basiert, die ungewisse Werte darstellen, wodurch auch die Standarddatentypen wie *int*, *bool*, *string* erweitert werden können. Jede Zufallsvariable stellt eine Menge oder einen Bereich (Set) von möglichen Werten und eine dazu passende zugehörige Verteilung dar. Diese Verteilung weist jedem möglichen Wert eine Wahrscheinlichkeit zu. Die vorhandene Verteilung verkörpert das Spektrum der Variablenwerte und ermöglicht statistische Analysen, um das Verhalten der Zufallsvariablen besser zu verstehen.

Durch den modellbasierten Ansatz in Infer.NET ist auch eine Interpretierbarkeit des Algorithmus möglich. Hat man das Modell selbst erstellt, so lässt sich dieses auch Schritt für Schritt bei der Ausführung im Algorithmus nachvollziehen, und man kann genau erklären, wie der Lernalgorithmus sich in einer bestimmten Weise verhält oder bestimmte Vorhersagen trifft.



Zufallsvariablen

In der Stochastik ist eine Zufallsvariable eine Größe, deren Wert vom Zufall abhängig ist [22]. Die Zufallsvariablen X sind dadurch gekennzeichnet, dass sie verschiedene Werte $X_1, X_2, X_3 \dots$ annehmen können, wobei jeder dieser Werte selbst ein zufälliges Ergebnis darstellt und mit einer bestimmten Wahrscheinlichkeit $P_1, P_2, P_3 \dots$ auftritt.

Die Funktion, die jedem Wert von X die Wahrscheinlichkeit für sein Eintreten zuordnet, wird Wahrscheinlichkeitsverteilung genannt. Das heißt, in einem Beispiel kann eine boolesche Zufallsvariable mit einer Wahrscheinlichkeit von 65 Prozent wahr sein und mit 35 Prozent falsch.

Die Wahrscheinlichkeitsverteilung [23] gibt an, wie sich die Wahrscheinlichkeit auf den möglichen Wert einer Zufallsvariablen verteilt.

Einfach ausgedrückt ist ein probabilistisches Programm die formale Darstellung einer Wahrscheinlichkeitsverteilung. Es existieren ja schon für die Normalverteilung oder die Binomialverteilung sehr effiziente Algorithmen. Aber besonders bei Modellen, die komplexe Wechselwirkungen zwischen verschiedenen Zufallsvariablen aus unterschiedlichen Verteilungen als Datenwerte enthalten, lohnt sich ein Blick auf eine mögliche Implementierung mit Infer.NET. Probabilistische Programme steuern autonome Roboter und selbstfahrende Autos, dienen der Vorhersage von Wirtschaftstrends, ermöglichen die Beschreibung von Sicherheitsmechanismen, implementieren zufallsgesteuerte Algorithmen und werden daher auch im Bereich Machine Learning vermehrt eingesetzt. Auch mit TensorFlow ist eine probabilistische Programmierung möglich. Dort heißt das Framework TensorFlow Probability.

7.4.2 Arbeitsweise von Infer.NET

Das Infer.NET Framework bietet Ihnen als Entwickler die Möglichkeit, grafische statistische Modelle als C#-Code übersichtlich darzustellen. Des Weiteren erhält man eine Inferenz-Engine, die komplexe mathematische Ausdrücke wie zum Beispiel Mittelwert und Präzision mit Gauß- und Gamma-Prioritäten berechnet und eine Vielzahl von Algorithmen zur Durchführung von Inferenz (Schlussfolgerungen) zu Verfügung stellt. Infer.NET bietet folgende Vorteile bei der Entwicklung von Modellen:

Umfangreiche Modellierungssprache

Infer.NET bietet eine Unterstützung für einzelne und mehrere unabhängige Variablen. Außerdem können die Modelle aus einem breiten Spektrum von Faktoren konstruiert werden. Dazu zählen arithmetische Operationen, lineare Algebra, Bereichs- und Positivitätsbeschränkungen, Boolesche Operatoren und vieles mehr.

Inferenz-Algorithmen

Das Framework stellt eine Fülle von fertigen Algorithmen für die Ermittlung von Schlussfolgerungen zur Verfügung. Dazu zählen Expectation Propagation [24], Belief Propagation [25], Variational Message Passing [26] und Gibbs-Sampling [27].

Möglichkeit groß angelegter Inferenzen

Infer.NET kompiliert die Modelle direkt in einen Inferenzquellcode, der unabhängig und ohne Overhead ausgeführt werden kann. Hierdurch kann der Code auch direkt in Ihre Anwendung integriert werden.

Erweiterbarkeit durch Plug-in-Architektur

Das Framework verwendet eine offene und anpassungsfähige Plug-in-Architektur. Somit ist es möglich, die breite Palette von Modellen und Inferenzoperationen zu erweitern. Es kommt in der Praxis immer wieder zu Sonderfällen, in denen ein neuer Faktor oder Verteilungstyp bzw. ein neuer Algorithmus benötigt wird. In diesem Fall können Sie mit Ihrem Code unter Zuhilfenahme der Plug-in-Methoden das Infer.NET Framework erweitern.

Das Framework funktioniert, indem es eine zuvor erstellte Modelldefinition in den Quellcode kompiliert, der zur Berechnung einer Reihe von Schlussfolgerungen, den bezeichneten Inferenzabfragen, benötigt wird. Bild 7.4 zeigt den dafür notwendigen Prozess.

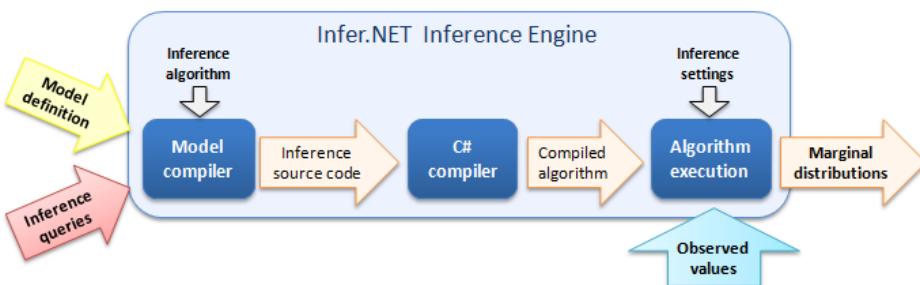


Bild 7.4 Arbeitsweise der Inference Engine (Quelle: Infer.NET, <https://bit.ly/3qtC2y5>)

Hierbei werden die Schritte wie folgt durchlaufen:

- Der Benutzer erstellt unter Verwendung der Modelling-API eine Modelldefinition und spezifiziert eine Reihe von Inferenzabfragen in Bezug auf das Modell.
- Der Benutzer erzeugt mithilfe der Modelldefinition und dem Modell-Compiler den Quellcode des angegebenen Inferenzalgorithms. Der erzeugte Quellcode kann in einer Datei ausgelagert oder bei Bedarf direkt in der Anwendung verwendet werden.
- Unter der Verwendung von Datensätzen und deren beobachteten Werten, wie zum Beispiel Werte aus Datenbankfeldern, wird der Algorithmus in der Inferenz-Engine ausgeführt. Dieser Vorgang kann für verschiedene Einstellungen der beobachteten Werte wiederholt werden, ohne den Algorithmus neu kompilieren zu müssen.

7.4.3 Infer.NET-Architektur

Die Infer.NET-Architektur basiert vollständig auf sogenannten Inferenzkomponenten. Das Framework verfügt schon von Grund auf über eine große Anzahl von eingebauten Komponenten. Bild 7.5 gibt einen Überblick über die Komponenten-Architektur.

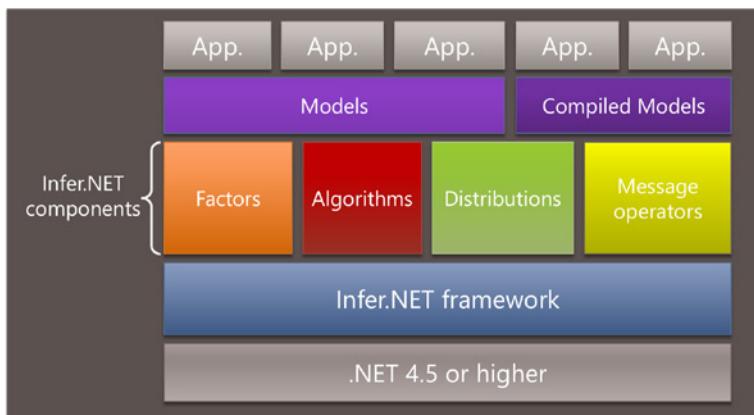


Bild 7.5 Die Infer.NET-Architektur (Quelle: Infer.NET, <https://bit.ly/3q9zghw>)

Der große Vorteil von Infer.NET sind die schon erwähnten Plug-in-Funktionen, die es ermöglichen, dass Sie auch Ihre eigenen Komponenten hinzufügen können. Es stehen vier Arten von Komponenten zur Verfügung:

- **Factors:** Unter Factors (Faktoren und Einschränkungen) findet man die Grundbausteine von Modellen.
- **Algorithmen:** Unter diesen Komponenten finden Sie alle Algorithmen, die zur Durchführung der Inferenz (Schlussfolgerung) verwendet werden können.
- **Distributions:** Die Distributions beschreiben die Verbindungen, die verwendet werden können, um eine Unsicherheit in einer Zufallsvariable auszudrücken.
- **Message operators:** Unter den Message operators versteht man bei Infer.NET die atomaren Operationen eines Algorithmus. In der Regel benötigen Sie für jeden Factor und für jeden Algorithmus einen entsprechenden Message operator.

Durch diese Architektur ist eine strikte Trennung von Modell und Inferenz-Algorithmus möglich. In der Praxis wird ein Infer.NET-Modell in einem Codeblock definiert und in einer separaten C#-Klasse gekapselt. Somit ist es möglich, dieses Modell sehr schnell in einer Anwendung zu übernehmen und auch mit verschiedenen Abfragen zu verwenden. Auch die Pflege, Wartung und Anpassung ist hier genauso wie Sie es von einer C#-Klasse gewohnt sind. Somit ist es für Sie als Entwickler ganz einfach möglich, das Modell anzupassen ohne Einfluss auf die Inferenz-Engine des Frameworks zu nehmen.

7.4.4 Infer.NET Modelling-API

Die API von Infer.NET erlaubt es Ihnen, sehr schnell ein probabilistisches Modell mit C#-Code zu erstellen. Sie können auch ein hochkomplexes Modell aus einzelnen einfachen Modellen zusammensetzen. Es ist also nicht nötig, das ganze Modell auf einmal zu entwerfen. Das Modell und auch die Datensatzgröße lassen sich sukzessive aufbauen, bis ein vollständig implementiertes Modell vorliegt.

Wenn Sie Infer.NET einsetzen, so definieren Sie als Erstes ein probabilistisches Modell für Ihren Lösungsansatz. Über die API legen Sie die benötigten Modellvariablen an und stellen die entsprechenden Beziehungen zueinander her. Das folgende Code-Snippet zeigt die Verwendung in Bezug auf die API zur Definition eines Modells für das Lernen des Mittelwerts und der Präzision eines Gaußschen Modells.

Listing 7.1 Erstellung eines Infer.NET Modells

```
Variable<double> mean = Variable.GaussianFromMeanAndVariance(0, 100);
Variable<double> precision = Variable.GammaFromShapeAndScale(1, 1);
VariableArray<double> data = Variable.Constant(new double[] { 11, 5, 8, 9 });
Range i = data.Range;
data[i] = Variable.GaussianFromMeanAndPrecision(mean, precision).ForEach(i);
```

Das kurze Codebeispiel erzeugt zwei Zufallsvariablen, Mittelwert und Präzision mit Gauß- und Gamma-Prioritäten. Das entsprechende Faktordiagramm des Modells in Bild 7.6 zeigt die Beziehungen zwischen den Variablen.

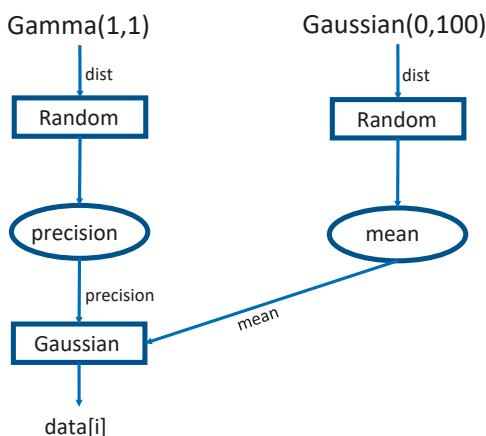


Bild 7.6
Faktordiagramm des Infer.NET-Modells

Des Weiteren wird über `Variable.Constant` ein Array erzeugt, bei dem die Werte in diesem Array in das Modell eingefügt werden und bei der Ausführung des Modells nicht veränderbar sind. Der i -Data-Bereich geht über den Datenindex von 0 bis 3. Die abschließende Linie verbindet alle oben genannten Punkte miteinander, indem sie die Datenpunkte, die von einer Gaußverteilung mit Mittelwert und Präzision gezeichnet werden sollen, einschränkt.

Das hier gezeigte Modell ermöglicht das Lernen von A-posteriori-Verteilungen [28] über den Mittelwert und die Präzision einiger Daten. Das Beispiel verdeutlicht, wie mit wenig Aufwand ein Modell mit dem Infer.NET Framework und C# entwickelt werden kann. Eine ausführliche Dokumentation über den Einsatz des Infer.NET Frameworks und der Modellierungsmöglichkeiten finden Sie im Infer.NET User Guide [29].

7.4.5 Lernen und Trainieren

Infer.NET unterstützt als Trainingsmethode die Bayessche Inferenz und somit den zugrunde liegenden Satz von Bayes. Darüber hinaus kann auch das Online-Verfahren zum Einsatz kommen, indem nicht über einen Batch trainiert wird, sondern inkrementell jeder einzelne Datensatz dem Training hinzugefügt wird (siehe Abschnitt 4.2 Batch, Inkrementell und Mini-Batch). Da die Bindung der Datenquellen direkt im C#-Code erfolgen kann, hat man auch noch eine entsprechend große Auswahl von Datenquellen bei der Datenbindung und kann damit sehr flexible Anwendungsszenarien umsetzen.

Des Weiteren ermöglicht Infer.NET das sogenannte hierarchische Lernen. Hierbei wird ein vortrainiertes Modell speziell für eine jeweilige rekursive Entität trainiert. Auch die Modell Selektion mit Evidence (Beweis) wird unterstützt. Hiermit lässt sich eine sehr gute Balance zwischen Modell-Komplexität, Qualität und Performanz finden [30].

Wie schon beschrieben, beruht bei Infer.NET die Hauptidee auf dem modellbasierten Ansatz, bei dem alle Kriterien über den Problembereich in Form des erstellten Modells fixiert und in einer präzisen mathematischen Form ausgedrückt werden. Diese Kriterien umfassen die Anzahl und die Typen von Variablen sowie die Art und Weise, wie diese Variablen einander beeinflussen, wenn man eine Änderung vornimmt. Somit lassen sich bei Infer.NET das Modell und die entsprechende Lösung des Algorithmus beim Lernvorgang nachvollziehen.

7.4.6 Infer.NET in der Anwendung

Die autonome Navigation stellt heute neben der Interaktionsfähigkeit mit mobilen Robotern immer noch eine der anspruchsvollsten Disziplinen dar. Die Möglichkeiten von Infer.NET und der probabilistischen Programmierung können in der Robotik helfen. In der Praxis spricht man von der probabilistischen Robotik, die sich mit der Wahrnehmung, also wie Roboter sehen und ertasten, und der Steuerung von Robotern beschäftigt. Die zentralen Aufgaben sind in diesem Fall die Entwicklung von Algorithmen zur Sensordatenverarbeitung, Lokalisierung, Pfadplanung, Navigation und Aktionsplanung von mobilen Robotern.

Hierbei ist immer noch das grundlegendste Problem die Lokalisierung. Bei der Auswertung von Sensoren wird zwar von einer guten Kalibrierung ausgegangen, jedoch ist es Teil des probabilistischen Ansatzes, die erfassen Werte immer mit einer konservativen Fehlerverteilung zu verarbeiten. Das heißt, man versucht, Unsicherheiten in der Roboterposition mithilfe von Methoden aus der Wahrscheinlichkeitsrechnung zu modellieren. Probabilistische Filter werden eingesetzt, um die Qualität der vorhandenen Daten für die Lokalisierung zu verbessern.

Die Methoden der probabilistischen Robotik haben auch in der Bildverarbeitung zur Objektverfolgung einen großen Verwendungsbereich. So steht immer noch die Kollisionsvermeidung bei autonomen mobilen Robotern an erster Stelle. Hier ist man auf ein effektives, schnelles, robustes und leicht zu implementierendes Berechnungsframework zur Planung von Roboterbewegungen angewiesen.

Eines der besten Beispiele für kooperierende mobile Roboter wurde von der Firma Kiva Systems (inzwischen von Amazon übernommen) für den Bereich der Lager-Logistik im Versandhandel realisiert. Die autonomen Transportroboter fahren unter die entsprechenden Regale, heben diese an und fahren selbstständig durch die Lagerhallen zum Zielplatz [31]. Das Anwendungsbereich von mobilen Robotern ist inzwischen riesig. Zu den mobilen Robotern zählen autonome Fahrzeuge, Staub- und Mähdrohner, mobile Roboter im Bereich der Logistik, Landwirtschaft oder auch der Krankenpflege.

Im nachfolgenden Beispiel soll kein autonomer mobiler Roboter programmiert werden, das wäre dann für ein Buch doch etwas zu umfangreich, sondern es soll mithilfe einer einfachen Anwendung die Leistung eines Roboters ermittelt werden. In dem Beispiel werden die den Robotern bekannten Kollisionen als Ergebnis herangezogen. Für das Modell wird vorausgesetzt, dass die latente Kollision jedes Roboters normal verteilt ist und dass das Kollisionsverhalten bei einer bestimmten autonomen Fahrt eine abweichende Leistung darstellt. Die Auswertung der Daten legt nahe, dass die Kollisionsvermeidung besser ist als eine Kollision mit einem Objekt.

7.4.7 Das Modell entwerfen

Für das Beispiel fahren jeweils zwei Roboter einen Parcours entlang. Sie benötigen zur Auswertung eine Liste der abgeleiteten Kollisionen jedes Roboters in jeder Fahrrunde sowie die Varianz für die Ungewissheit in Bezug auf die Kollision. Bild 7.7 zeigt die Beispieldaten der einzelnen Runden der Roboter gegeneinander.

Entwurf des Modells zur Analyse der Kollisionserkennung

Fahrstrecke	keine Kollision	Kollision
1	Roboter 0	Roboter 1
2	Roboter 1	Roboter 2
3	Roboter 0	Roboter 2
4	Roboter 3	Roboter 0
5	Roboter 4	Roboter 0
6	Roboter 4	Roboter 1
7	Roboter 2	Roboter 4
8	Roboter 3	Roboter 4
9	Roboter 3	Roboter 1

Bild 7.7 Das Modell für das Beispiel

Jetzt müssen Sie die Beispieldaten aber schon sehr genau betrachten, um den Roboter mit den wenigen Kollisionen herauszufinden, obwohl es sich im Moment nur um neun Datensätze handelt. Mithilfe von Infer.NET und der probabilistischen Programmierung lässt sich die Rangfolge in den Datensätzen schnell analysieren. Im Beispiel beginnt die Zählweise bei Roboter 0, da der Index im Infer.NET Framework nullbasiert ist. Der verwendete Algorithmus wird in erweiterter Form mit sehr viel mehr Parametern häufig genutzt, um Qualifikationsbewertungssysteme in den verschiedensten Bereichen zu implementieren.

7.4.8 Infer.NET anwenden

Das vorgestellte Beispiel kann man natürlich auch ohne Algorithmus lösen. Wird ein probabilistisches Modell aber aufgrund vieler Zufallsvariablen und unterschiedlicher Datentypen komplexer, so benötigen Sie auf jeden Fall ein gutes statistisches Verständnis und sehr gute Kenntnisse der Bayesschen Inferenz und der numerischen Analyse, um die Aufgabe zu lösen. Glücklicherweise kann hier die Inferenz-Engine von Infer.NET effektive Unterstützung bieten. Nachdem Sie das Modell konzipiert haben, wird es mithilfe der API von Infer.NET als probabilistisches Programm ausgedrückt. Hierfür sind mit Infer.NET und C# nur einige wenige Schritte zu erledigen.

- Das probabilistische Modell wird direkt mit C# in einem Programm bzw. in einer Klasse erstellt.
- Einer oder mehreren Zufallsvariablen werden mit der Eigenschaft *ObservedValue* Werte zugewiesen.
- Eine Inferenz-Engine erstellt die benötigte Abfrage. Hier reicht in Infer.NET eine einzige Code-Zeile. Die Inferenz-Engine erledigt die Arbeit im Zusammenspiel mit dem ausgewählten Algorithmus.
- Die zurückgelieferten Werte können weiterverarbeitet bzw. ausgewertet werden.

Um Infer.NET nutzen zu können, müssen Sie als Erstes das *Microsoft.ML.Probabilistic.Compiler*-Paket in ein Visual-Studio-Projekt einbinden. Hierfür können Sie Infer.NET als Open-Source-Lösung komplett von GitHub [32] übernehmen und dann einbinden. Noch einfacher geht es über den NuGet-Paketmanager in Visual Studio.

Erstellen Sie ein Konsolen-Projekt mit C# als Console App (.NET Framework). Das Beispielprojekt erhält den Namen *RoboterRangliste*. Nach dem Erstellen der Solution wählen Sie über *Tools|Nuget Package Manager* den Punkt *Manage NuGet Packages for Solution* aus und installieren das Paket *Microsoft.ML.Probabilistic.Compiler* (Bild 7.8). Bestätigen Sie die Installation und öffnen Sie dann die Klasse *Program.cs* in Visual Studio.

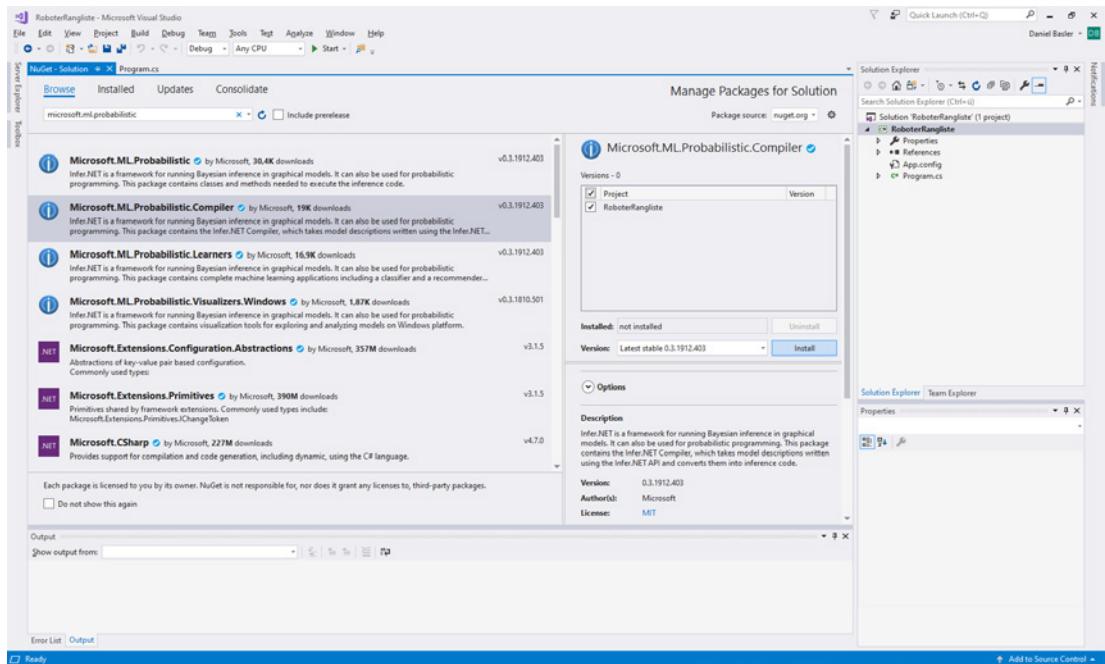


Bild 7.8 Einbinden von Infer.NET in das Projekt

Fügen Sie jetzt der Program-Klasse das Code-Beispiel aus Listing 7.2 hinzu.

Listing 7.2 Der Code für die Rangliste

```
using Microsoft.ML.Probabilistic.Distributions;
using Microsoft.ML.Probabilistic.Models;
using System;
using System.Linq;

namespace RoboterRangliste
{
    class Program
    {
        static void Main(string[] args)
        {
            var collisionAvoidedData = new[] { 0,1,0,3,4,4,2,3,3 };
            var collisionData = new[] { 1,2,2,0,0,1,4,4,1 };

            var drivingRoute = new Range(collisionAvoidedData.Length);
            var robots = new Range(collisionAvoidedData.Concat(collisionData).Max() + 1);
            var roboterSkills = Variable.Array<double>(robots);
            roboterSkills[robots] = Variable.GaussianFromMeanAndVariance(9, 12).
               ForEach(robots);

            var collisionAvoided = Variable.Array<int>(drivingRoute);
            var collision = Variable.Array<int>(drivingRoute);
```

```
using (Variable.ForEach(drivingRoute))
{
    var collisionAvoidedPerformance = Variable.GaussianFromMeanAndVariance
        (roboterSkills[collisionAvoided[drivingRoute]], 1.0);
    var collisionPerformance = Variable.GaussianFromMeanAndVariance
        (roboterSkills[collision[drivingRoute]], 1.0);
    Variable.ConstrainTrue(collisionAvoidedPerformance >
collisionPerformance);
}

collisionAvoided.ObservedValue = collisionAvoidedData;
collision.ObservedValue = collisionData;

var inferenceEngine = new InferenceEngine();
var inferredSkills = inferenceEngine.Infer<Gaussian[]>(roboterSkills);

var orderedRoboterSkills = inferredSkills
    .Select((s, i) => new { Roboter = i, Skill = s }).OrderByDescending
        (rs => rs.Skill.GetMean());

foreach(var roboterSkill in orderedRoboterSkills)
{
    Console.WriteLine($"Roboter {roboterSkill.Roboter} skill:
{roboterSkill.Skill}");
}
Console.ReadLine();
}
}
}
```

Als Erstes binden Sie über die `Using`-Direktive den benötigten Namespace für die probabilistische Programmierung ein. Dann werden über die Variable `collisionAvoidedData` und `collisionData` die Beispieldaten aus dem Modellentwurf übernommen und danach das statistische Modell als probabilistisches Programm definiert.

Bei der probabilistischen Programmierung wird, wie bereits oben erläutert, eine Zufallsvariable verwendet, die in Infer.NET Framework als Schlüsselwort `Variable` definiert ist. Die Variable ist ein Element aus dem Framework, welches durch eine Klasse `Variable<T>` umgesetzt wird. Die Infer.NET-Variablen sind streng typisiert. Die `Variable<T>` kann sowohl mit einem Wert, wie im Beispiel, initialisiert werden und verhält sich dadurch deterministisch, oder auch einfach nur mit einem rein zufälligen Wert belegt werden. Sie können im Programmcode sehen, dass sich eine Infer.NET-Zufallsvariable ganz einfach über den Methodenaufruf `Variable.GaussianFromMeanAndVariance` mit den Bedingungen der Gaußschen Normalverteilung mit Varianz erstellen und nutzen lässt.



Normalverteilung (Gauß-Verteilung)

Die Normalverteilung ist die wichtigste Verteilung in der Statistik und wird verwendet, wenn die tatsächliche Verteilfunktion unbekannt ist [33]. Die Normalverteilung ist somit eine stetige Wahrscheinlichkeitsverteilung und bildet daher eine Funktion, die dabei hilft, die Wahrscheinlichkeit für alle möglichen Werte zu berechnen, die eine zufällige Variable annehmen kann.

Über die Methode `Variable.ForEach(drivingRoute)` wird die Leistung der Roboter bewertet. Dann werden die Daten zur Beobachtung an das Modell geheftet. Dies erfolgt über die Methode `ObservedValue`.

Das Ergebnis des Modells wird dann über die Inferenz (Schlussfolgerung) errechnet. Hierfür nutzt man einfach die Inferenz-Funktion von Infer.NET, die mit `new InferenceEngine()` in C# instanziert wird. Da es bei der Schlussfolgerung noch entsprechende Unsicherheiten gibt, nutzt man die Varianz in der Normalverteilung. Über den Befehl `Console.WriteLine` wird das Ergebnis in der Konsole ausgegeben.

Wird das Programm jetzt das erste Mal kompiliert, so wird durch das Infer.NET Framework ein probabilistisches Modell mit Inferenz-Abfragen und einem automatisch gewählten Inferenz-Algorithmus erzeugt und als neue C#-Klasse generiert. Diese Klasse stellt dann einen Algorithmus zur Verfügung, der speziell für das definierte Modell zusammengestellt und optimiert ist.

Das probabilistische Modell wird durch Infer.NET automatisch in C#-Code erstellt und bei der Ausführung des Programms übersetzt und in eine für dieses Modell spezifische C#-Klasse mit dem Namen `Model_EP.cs` (Bild 7.9) im Ausführungsverzeichnis des Projekts (`.\bin\Debug\GeneratedSource\...`) angelegt, die dann für die Inferenz-Operationen benutzt wird. Startet man das Programm, so wird das Modell zum ersten Mal übersetzt und die Rangliste der Roboter ermittelt. Bild 7.10 zeigt das Ergebnis in der Konsolen-App.

```

RoboterRangliste - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Model EP.cs X NuGet - Solution Programs.cs Model:Model_EP
Solution Explorer
Search Solution Explorer (Ctrl+Q)
Solution RoboterRangliste (1 project)
  > RoboterRangliste
    > References
    > App.config
    > packages.config
      > Programs
        > Program
Properties Team Explorer
Solution Explorer Properties
Output
Show output from Debug
'RoboterRangliste.exe' (CLR v4.0.30319): RoboterRangliste.exe: Loaded 'C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.ServiceModel\Internal\v4.0.0.0__31b38d59d364e355\RoboterRangliste.exe'. (CLR v4.0.30319; RoboterRangliste.exe) Loaded 'C:\Beispiele_Kapitel_7\RoboterRangliste\RoboterRangliste\bin\Debug\System.CodeDom.dll'. Skipped loading symbols.
'RoboterRangliste.exe' (CLR v4.0.30319): RoboterRangliste.exe: Loaded 'pugixml'. Symbols loaded.
'RoboterRangliste.exe' (CLR v4.0.30319): RoboterRangliste.exe: Loaded 'C:\Windows\Microsoft.NET\assembly\GAC_MSIL\mscorlib.resources\v4.0_0.0.0_de_b77a5c561934e00f\mscorlib.resources'. The program '[4448] RoboterRangliste.exe' has exited with code -1073741812 (0xCCCCCCCC).

```

Bild 7.9 Die von Infer.NET erzeugte Model_EP-Klasse

```
C:\Beispiele_Kapitel_7\RoboterRangliste\RoboterRangliste\bin\Debug
Compiling model...done.
Iterating:
.....|.....|.....|.....| 50
Roboter 3 skill: Gaussian(12,24, 4,381)
Roboter 0 skill: Gaussian(8,747, 1,719)
Roboter 4 skill: Gaussian(8,684, 1,612)
Roboter 2 skill: Gaussian(7,718, 1,705)
Roboter 1 skill: Gaussian(7,607, 1,775)
```

Bild 7.10 Die Anwendung wird ausgeführt.

Durch seinen modellbasierten Ansatz ist der Einstieg in das Infer.NET Framework auch für Neulinge im Bereich ML sehr gut zu meistern. Des Weiteren hat das Infer.NET Framework den Vorteil, dass es klein und kompakt ist und man auch auf nicht ganz so leistungsstarker Hardware damit noch gut zurechtkommt. Das Infer.NET Framework erstellt aus dem Modell einen maßgeschneiderten Algorithmus für das maschinelle Lernen, der dann auch lokal verwendet werden kann. Das Einsatzgebiet für das Framework reicht von Modellen in der Statistik, wie der vorgestellte Algorithmus, der sich aus der Normalverteilung ableitet, über Algorithmen für die Spamfilter-Analyse bis hin zur Prüfung von Textinhalten, Empfehlungssystemen und vielem mehr. Das Microsoft Research Cambridge-Team hat ein kostenloses Onlinebuch [34] herausgebracht, das eine gute Einführung in das Thema statistische Modelle bietet.

■ 7.5 ML.NET mit AutoML und ModelBuilder

ML.NET ist ein Open-Source- und plattformübergreifendes Framework (Windows, Linux, MacOS) für maschinelles Lernen speziell für .NET-Entwickler und unterstützt im Gegensatz zu Infer.NET eine Vielzahl von unterschiedlichen ML-Algorithmen, so zum Beispiel Klassifikation und Regression. Es entstand ursprünglich bei Microsoft Research und wurde das erste Mal auf der Build 2018 vorgestellt. Zurzeit steht es in der Version 1.5.1 zur Verfügung.

Mit ML.NET können .NET-Entwickler maschinelle Lernmodelle erstellen und verwenden, um beispielsweise Prognosen, Empfehlungen, Betrugserkennung, Bildklassifizierung und viele weitere Aufgaben umzusetzen. Erklärtes Ziel von ML.NET und Microsoft war es immer, .NET-Entwicklern ein Framework für Machine Learning an die Hand zu geben, welches eine Alternative zu den Python Libraries für Machine Learning darstellt. Aus diesem Grund unterstützt das Framework ein breites Open-Source Ökosystem, indem es zum Beispiel die Integration mit gängigen Deep Learning Frameworks wie TensorFlow und Interoperabilität durch ONNX (Open Neural Network Exchange) mitbringt.

ML.NET besteht aus Kernkomponenten für die Datenrepräsentation, für unterschiedliche ML-Szenarien wie Regression, binäre Klassifikationen und Clustering. Des Weiteren zählen noch die Datentransformation für die *Feature Selection* (Verwendung einer Teilmenge der verfügbaren Features für einen Lernalgorithmus) und die Normalisierung dazu.

Microsoft will mit ML.NET erreichen, dass maschinelles Lernen mehr und mehr in .NET-Anwendungen eingesetzt werden kann, ohne sich im Detail mit der zugrunde liegenden Mathematik und der Implementierung der ML-Algorithmen auskennen zu müssen. ML.NET bietet eine AutoML-Funktion, die Entwicklern helfen soll, den für ihre jeweilige Anwendung passenden Algorithmus sowie Einstellungen, Modelle und Transformation zu finden. Somit stehen Daten und Anwendungsfall hier im Vordergrund und nicht die eigentliche Implementierung des ML-Algorithmus. Als Entwickler können Sie AutoML entweder über eine eigene API ansprechen oder die Tools ML.NET Model Builder bzw. ML.NET CLI (*Command Line Interface*) verwenden. ML.NET ab der Version 1.4 ermöglicht auch die Ausführung in einer .NET Core 3.0-Anwendung. Ab .NET Core 3.0 wird für den praktischen Einsatz die Hardware-Funktion, mit der .NET-Code mathematische Operationen mithilfe prozessorspezifischer Anweisungen beschleunigt, unterstützt. Damit erreicht man unter .NET Core 3.0 eine schnelle Ausführung von komplexen mathematischen Funktionen.

7.5.1 Einbinden von ML.NET

ML.NET kann als Cross-Plattform-Framework unter Windows, Linux und macOS mit .NET Core oder unter Windows auch mit dem .NET Framework ausgeführt werden. Für die Cross-Plattform und viele Hintergrundaufgaben wie Projektanlage und Code-Generierung bringt ML.NET zur Ausführung ein Command Line Interface (CLI) mit. In der Windows-Welt lässt sich ML.NET jedoch auch ganz einfach mit Visual Studio verwenden. Binden Sie daher für die nachfolgenden Beispiele immer in ihr Visual-Studio-Projekt über den NuGet-Paketmanager das *Microsoft.ML*-Paket ein. In manchen Fällen kann es vorkommen, dass noch zusätzliche Pakete, insbesondere, wenn native Komponenten erforderlich sind, installiert werden müssen, da sich Funktionen in unterschiedliche *Microsoft.ML.**-NuGet-Pakete unterteilen.

Die mitgelieferte CLI dient auch als Werkzeug, mit dem sich ML.NET-Modelle mit AutoML erstellen lassen. Des Weiteren gibt es ab der Version 1.0 auch ein grafisches Benutzerinterface. Dieses Model Builder Tool stellt eine grafische Schnittstelle zum Erstellen von ML-Modellen in Verbindung mit AutoML zur Verfügung. Nach der Erstellung des Modells generiert das Tool anschließend Code zum Trainieren und Verwenden der Modelle. Die Model-Builder-Erweiterung für Visual Studio setzt unter der Haube auch das ML.NET Command Line Interface ein.

7.5.2 Was ist AutoML

Automated Machine Learning (*AutoML*) ist ein Prozess, der den Machine Learning Workflow vereinfachen und beschleunigen und dem Anwender ohne spezifische ML-Kenntnisse das Erstellen von Machine-Learning-Systemen vereinfachen soll.

Durch die Verwendung von AutoML wird der Entwickler bei der Erstellung eines automatisierten Modells enorm entlastet. Nicht die Programmierung des ML-Modells steht im Vordergrund, sondern die Problemlösung. Das spart Zeit und Kosten und sorgt vor allem durch die genaue Nachverfolgung der Prozesskette für eine entsprechende hohe Akzeptanz bei den Anwendern in den Fachbereichen. Mit Auto.ML setzt man als Entwickler auf erprobte

Modelle, leichte Installation und Nutzung. Sie können so Ihr ML-Projekt schneller erfolgreich abschließen und eine stabile und fehlerfreie Machine-Learning-Anwendung liefern.

Neben der Integration von AutoML in ML.NET Framework gibt es inzwischen mehrere Tools von verschiedenen Anbietern für das automatisierte Machine Learning. Dazu zählen unter anderem TPOT für scikit-learn, AutoKeras oder auch Google Cloud AutoML. Dieses Buch geht jedoch nur auf AutoML im ML.NET Framework ein.

Ohne AutoML sieht der klassische ML-Prozess in der Regel folgendermaßen aus:

- Datenerhebung
- Datensichtung
- Vorbereitung der Daten
- Feature Engineering
- Auswahl des passenden ML-Algorithmus
- Training des Modells
- Optimierung der Hyperparameter
- Deployment des Modells

Alle Schritte laufen im ML-Workflow immer getrennt voneinander ab. Das Ziel von AutoML ist es, die einzelnen Blöcke automatisch auszuführen. Das heißt, AutoML versucht automatisch, Features zu erkennen und zu selektieren. Die selektierten Features fließen dann in einen Algorithmus. Auch hier versucht AutoML, den passenden ML-Algorithmus zu finden und diesen mit einer Optimierung der Hyperparameter für das Modell automatisch aufzubereiten.

Einige Prozessschritte wie die Datenerhebung, Datensichtung und die Vorbereitung der Daten sind heute noch schwer zu automatisieren. In der Praxis hat man für AutoML folgende typische automatisierbare Prozessschritte ausgearbeitet, die so auch schon Verwendung finden:

- Feature-Engineering
- Auswahl des ML-Algorithmus
- Optimierung der Hyperparameter für das Modell
- Ergebnisanalyse und eventuell Visualisierung
- Deployment des Modells

AutoML bietet ein großes Potenzial für ganz unterschiedliche Anwendungsbereiche, die von der einfachen Klassifizierung über Regression bis hin zur Robotik reichen. Durch den Einsatz von AutoML wird versucht, den Vorgang des Machine Learning soweit zu abstrahieren, dass tiefgreifende Kenntnisse von Machine Learning überhaupt nicht mehr notwendig sein sollen. Auch Microsoft möchte, dass .NET-Entwickler Machine Learning in jede Art von Anwendung, sei es Web, Desktop oder Mobile, integrieren können ohne ML-Vorkenntnisse besitzen zu müssen.

7.5.3 Model Builder

Mit dem ML.NET Model Builder bietet Microsoft für das Framework eine leicht verständliche grafische Erweiterung in Visual Studio zum Erstellen, Trainieren und Bereitstellen benutzerdefinierter maschineller Lernmodelle an. Hierbei sind Vorkenntnisse im Bereich Machine Learning nicht erforderlich und die Einstiegshürde für den Anwender ist dadurch sehr niedrig.

Das Model Builder Tool unterstützt AutoML aus dem ML.NET Framework und somit auch automatisiertes maschinelles Lernen. Es unterstützt verschiedene ML-Algorithmen und Einstellungen, um so ein optimales Modell für die jeweilige Problemstellung erzeugen zu können.

Model Builder erzeugt ein trainiertes Modell sowie den benötigten Code, um das Modell zu laden und für Vorhersagen zu nutzen. Das erzeugte Modell wird als .zip-Datei gespeichert und kann so von Ihrer .NET Anwendung verwendet werden. Der benötigte Code wird als neues Projekt Ihrer Visual Studio Solution hinzugefügt.

Somit steht Ihnen auch der Code zur Verfügung, mit dem Sie Ihr Modell mit einem neuen Datensatz trainieren können, ohne den gesamten Model-Builder-Prozess noch einmal durchlaufen zu müssen. Des Weiteren fügt der Model Builder auch automatisch eine Konsolenanwendung hinzu, die Sie direkt in Visual Studio ausführen können, um Ihr Modell zu evaluieren.

Damit können Sie AutoML in Verbindung mit dem Model Builder aus dem ML.NET Framework sehr gut für folgende Anforderungen verwenden:

- Implementieren von ML-Lösungen ohne umfangreiche Programmierkenntnisse
- Einsparungen bei der Entwicklungszeit und von Ressourcen
- Nutzen von bewährten Data-Science-Methoden
- Bereitstellen von flexiblen Lösungen

Der große Vorteil von ML.NET mit seinen Tools ist, dass es konform mit dem .NET-Standard ist und aufgrund dessen überall dort verwendet werden kann, wo .NET-Code zum Einsatz kommt.

7.5.4 Einbinden in das Projekt

Der Model Builder kann wie viele andere .NET-Bibliotheken einfach in Visual Studio integriert werden. Um das Model Builder Tool in der Entwicklung zu nutzen, sollten Sie Visual Studio ab der Version 2017 benutzen. Für das nachfolgende Beispiel wird Visual Studio 2019 benutzt. Model Builder steht zurzeit noch als Preview-Version zur Verfügung und ist als Workload in Visual Studio eingebunden. Sollten Sie Visual Studio noch nicht mit dem Model Builder erweitert haben, so können Sie diesen auch ganz einfach als Extension über das Visual-Studio-Menü *Extensions | Manage Extensions* installieren (Bild 7.11). Nach der Installation starten Sie Visual Studio einmal neu und Sie können das Model Builder Tool in Ihrem Projekt einsetzen.

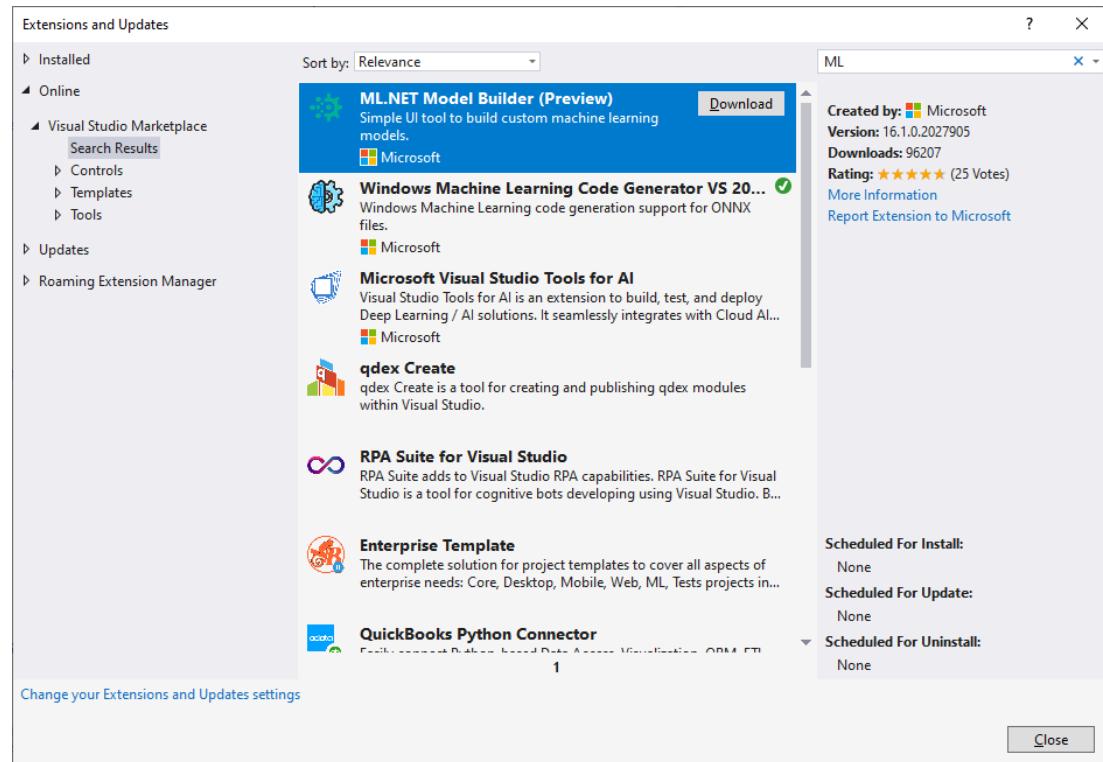


Bild 7.11 Das Model Builder Tool als Extension

7.5.5 Szenario

Nach einer erfolgreichen Installation kann man jetzt den Model Builder direkt in seinem Visual-Studio-Projekt verwenden. Das Model Builder Tool bietet einen einfachen Assistenten, der Sie durch fünf Schritte führt. Hierzu zählen die Auswahl der Aufgabe, das Hinzufügen von Daten, das Ausführen des Machine-Learning-Trainings, die Bewertung des ML-Modells und die Generierung des ML-Modells als C#-Code. Als Erstes wählen Sie für Ihre Problemstellung das Szenario (Aufgabe) aus, das am besten passt. Ein Szenario stellt im Umfeld von ML.NET die Beschreibung für die Art der Vorhersage dar, die Sie mit Ihren Daten treffen möchten. Hierzu zählen:

Textklassifizierung (Text classification)

Diese Klassifizierung dient der Unterteilung von Daten in Kategorien. Man kann über die Textklassifizierung erzeugte Textdaten in Kategorien klassifizieren, um zum Beispiel vorherzusagen, ob Kommentare positiv oder negativ sind.

Regression (Value prediction)

Sie können die Regression verwenden, um Zahlen vorherzusagen. Somit wird eine Vorhersage eines numerischen Wertes aus Ihren Daten (Regression) bestimmt. Die Regression wird oftmals für die Vorhersage von Nachfrage, Preis und Verkaufszahlen eines Produktes eingesetzt.

Bildklassifizierung (Image classification)

Die Bildklassifizierung im Model Builder kann verwendet werden, um Bilder unterschiedlicher Kategorien zu identifizieren. Beispiel hierfür sind unterschiedliche Arten von Gelände, Tieren aber auch zum Beispiel Fertigungsfehler in der Qualitätssicherung.

Empfehlung (Recommendation)

Mit dem Empfehlungsszenario wird eine Liste vorgeschlagener Elemente für einen bestimmten Benutzer vorhergesagt. Das heißt, Sie können eine Liste mit Vorschlägen für einen bestimmten Benutzer erstellen, um diesem zum Beispiel Kaufartikel, Filme, Bücher oder auch TV-Sendungen und Videos zu empfehlen.

Im Model Builder wird dann einfach der gewünschte Typ des Szenarios ausgewählt. Allerdings unterstützen der Model Builder und AutoML noch nicht alle Einsatzszenarien, die sich programmiertechnisch mit ML.NET lösen lassen. Dazu zählen zum Beispiel die Multiklassenklassifizierung, Clustering oder auch die Anomaly Detection Models, die der Anomalie-Erkennung in Strom- und Kommunikationsnetzen dient. An diesen Szenarien wird zurzeit gearbeitet und es ist schon bald damit zu rechnen, dass sie den Weg in das Model Builder Tool finden.

Der Model Builder kann für alle aufgeführten Szenarien das Machine Learning Model lokal auf Ihrem Computer trainieren. Bedenken Sie also bei der Durchführung des lokalen Trainings, dass Sie innerhalb der Grenzen Ihrer Computerressource arbeiten. Für das Szenario der Bildklassifizierung wird alternativ auch das Training in der Azure Cloud angeboten.

Im zweiten Schritt im Model Builder wird die entsprechende Arbeitsumgebung für den Trainingsschritt angezeigt, d. h. es erfolgt nur eine Anzeige der Ressourcen des lokalen Computers. Sie können somit direkt mit *Punkt 3. Data* fortfahren. Beim dritten Schritt verlangt der Model Builder die Daten, mit denen das neue Machine Learning Model erstellt und trainiert werden soll.

7.5.6 Daten

Das Model Builder Tool unterstützt Datasets im TSV-, CSV-, und TXT-Format sowie den Zugriff auf das SQL-Datenbankformat. Verwenden Sie eine TXT-Datei, so müssen die Spalten durch Komma, Semikolon oder /t getrennt werden. Des Weiteren muss die Datei eine Kopfzeile aufweisen. Im Bereich der Bildklassifizierung besteht das Dataset aus Bildern und unterstützt die Dateitypen .jpg und .png.

Beachten Sie, dass für eine gute Vorhersage des gesuchten Ergebnisses eine gewisse Grundmenge an Daten vorhanden sein muss, um das Modell zu trainieren. Sie sollten daher, auch wenn es sich nur um ein kleines Beispiel handelt, ein Modell nie mit weniger als 100 Datensätzen trainieren.

Dataset und Aufgabe

Da auch in der heutigen Zeit die Komplexität in der Logistik ständig zunimmt und damit auch in bestimmten Bereichen die Bestandsführung immer komplizierter wird, beschäftigt sich das nachfolgende Beispiel für das Dataset mit einer Artikelbestandsvorhersage. Die Artikelbestandsdaten sind wie in Bild 7.12 aufgebaut.

	A	B	C	D	E	F	G
1	Artikel_Nr	Lagerort	Menge_In	Menge_Out	Artikeltyp	Gesamt Artikelbestand	
2	Art001	1	100	10	Eigenfertigung	90	
3	Art001	2	350	35	Zukauf	315	
4	Art001	3	400	40	Eigenfertigung	360	
5	Art001	4	1500	0	Eigenfertigung	1500	
6	Art002	1	300	50	Eigenfertigung	250	
7	Art002	2	350	70	Eigenfertigung	280	
8	Art002	3	400	100	Eigenfertigung	300	
9	Art002	4	900	150	Zukauf	750	
10	Art003	1	45	20	Eigenfertigung	25	
11	Art003	2	120	20	Eigenfertigung	100	
12	Art003	3	35	10	Eigenfertigung	25	
13	Art003	4	80	30	Eigenfertigung	50	
14	Art004	1	77	0	Eigenfertigung	77	
15	Art004	2	80	20	Eigenfertigung	60	
16	Art004	3	120	55	Eigenfertigung	65	
17	Art004	4	200	25	Eigenfertigung	175	
18	Art005	1	250	25	Eigenfertigung	225	
19	Art005	2	300	70	Eigenfertigung	230	
20	Art005	3	700	100	Zukauf	600	

Bild 7.12 Artikelbestandsdaten

Diese Tabelle wird für das Beispiel als CSV-Datei aufbereitet. In der CSV-Datei werden die Bestandsdetails als *Artikel_NR*, *Lagerort*, *Menge_IN*, *Menge_OUT*, *Artikeltyp* und *Gesamt Artikelbestand* angegeben. Die Tabelle enthält ca. 100 Datensätze als Muster mit den erforderlichen Details. Die daraus erzeugte CSV-Datei dient als Muster für das Machine Learning Model.



Die vollständige CSV-Datei mit dem entsprechenden Dataset finden Sie unter GitHub:
<https://github.com/DanielBasler/NeuralNetwork>

Dieser Datenbestand wird verwendet, um das Ergebnis mit dem Model Builder zu trainieren, zu bewerten und vorherzusagen. Beachten Sie, dass Sie die Auswahl des Labels für die Vorhersage und die entsprechenden Features (Merkmale) fixieren müssen.

Das Dataset stellt immer eine Tabelle mit Zeilen (Rows) als Trainingsbeispiele und Spalten (Columns) mit Attributen dar. Jede Zeile enthält somit:

- Label (das Attribut, das Sie vorhersagen möchten)
- Features (Merkmale, die als Eingaben verwendet werden, um das Label vorauszusagen)

Für das Szenario der Artikelbestandsvorhersage können folgende Features verwendet werden:

- Artikel_NR
- Lagerort
- Menge_IN (Gesamtzahl der am Standort eingegangenen Artikel)
- Menge_OUT (Gesamtzahl der vom Standort gelieferten Artikel)
- Artikeltyp (Eigenfertigung bedeutet als lokal hergestellt und Zukauf kennzeichnet einen Zukaufartikel)

Das Label ist der *Gesamt Artikelbestand* und repräsentiert die Gesamtzahl des Artikelbestandes am Standort. Bild 7.13 zeigt den Aufbau von Label und Features im Beispieldataset noch einmal im Detail.

Das Diagramm zeigt ein Beispiel-Datenset in Form einer Tabelle. Die Tabelle hat 6 Spalten: Artikel_Nr, Lagerort, Menge_IN, Menge_AUS, Artikeltyp und Gesamt-Artikelbestand. Es gibt 4 Zeilen mit Daten für den Artikel Art001 an verschiedenen Lagerorten (1 bis 4). Die Spalten sind unter dem Titel "Features" zusammengefasst, die einzelnen Zeilen unter "Rows". Der letzte Spalte "Gesamt-Artikelbestand" ist als "Label" gekennzeichnet. Unter der Tabelle steht die Beschriftung "Columns".

Artikel_Nr	Lagerort	Menge_IN	Menge_AUS	Artikeltyp	Gesamt-Artikelbestand
Art001	1	100	10	Eigenfertigung	90
Art001	2	350	35	Zukauf	315
Art001	3	400	40	Eigenfertigung	360
Art001	4	1500	0	Eigenfertigung	1500

Bild 7.13 Aufbau des Beispieldatasets

Das Machine-Learning-Modell wird später mit diesen Daten trainiert werden, um einen entsprechenden Artikelbestand für einen Artikel vorauszusagen. Nach der Auswahl der Trainingsdaten erfolgt im Model Builder schon der nächste Schritt für das Trainieren des Modells.

7.5.7 Training und Auswertung

Das Training im Model Builder ist ein vollständig automatisierter Prozess, bei dem das eingebundene AutoML die verschiedensten ML-Algorithmen durchläuft, um den für die Problemstellung am besten passenden auszuwählen, um anschließend das Modell zu trainieren. Nach dem Training kann Ihr Modell dann Vorhersagen auf Basis ihm bisher unbekannter Eingabedaten treffen.

Da der Model Builder auf AutoML aufbaut, ist auch während des Trainings keine Eingabe oder Anpassung von außen notwendig. Allerdings ist es bei AutoML so, dass die Möglichkeit, eine gute und präzise Vorhersage durchzuführen, umso besser ist, je mehr Daten für das Training zur Verfügung stehen. Die Trainingsdauer beim Einsatz des Model Builder ist abhängig von folgenden Faktoren:

- Anzahl der Features (Merkmale/Spaltenanzahl), die als Eingabe für das Modell verwendet werden
- Art des Spaltentyps
- Die Aufgabenstellung, also das zu lösende Machine-Learning-Problem
- Die Rechenleistung des verwendeten Computersystems

Die nachfolgende Auswertung (*Evaluate*) ist der Prozessschritt, bei dem ermittelt bzw. gemessen wird, wie gut das Modell ist. Der Model Builder verwendet das trainierte Modell, um Vorhersagen mit neuen Testdaten zu treffen und anschließend zu messen, wie gut die Vorhersage ist. Je mehr unterschiedliche Daten man für das Trainieren des Modells verwendet, desto mehr statistische Varianz erzeugt man und umso besser wird die Vorhersage bei neuen, noch unbekannten Daten.

Das Model Builder Tool unterteilt automatisch die Trainingsdaten in einen Trainingssatz und einen Testsatz (siehe Abschnitt 4.1, „Trainings- und Testphase“). Die Trainingsdaten werden auch hier nach dem 80/20-Prozent-Muster aufgeteilt. Die Evaluierung erfolgt dann mit den 20 % Testdaten auf das Modell.

7.5.8 Der Code

Nach der Evaluierungsphase werden das finale Modell und der dazugehörige Code in Form von zwei neuen Projekten generiert und der Solution in Visual Studio hinzugefügt.

Das heißt, das finale Modell wird als ZIP-Datei gespeichert, des Weiteren wird ein neues Projekt mit dem entsprechenden Code zum Laden und Verwenden Ihres Modells der Projektmappe hinzugefügt. Darüber hinaus generiert der Model Builder eine Beispiel-Konsolen-App, die Sie ausführen können, um Ihr Modell in Aktion zu sehen. Außerdem gibt der Model Builder auch den Code aus, der das Modell generiert hat, sodass Sie die einzelnen Schritte zur Erzeugung des Modells komplett nachvollziehen können. Somit kann dann auch der Code verwendet werden, um Ihr Modell mit neuen Daten zu trainieren.

7.5.9 Automatisiert modellieren

Nachdem die Problemstellung, also die Aufgabe für das Machine Learning Model, bekannt und die Beispieldaten vorbereitet bzw. heruntergeladen sind, soll jetzt auf dieser Dataset-Basis ein Klassifizierungs-Modell mit dem Model Builder für die Vorhersage des Artikelbestandes erstellt werden.

Beginnen Sie einfach mit einer leeren .NET-Core Console App mit dem Projektnamen DemoApp in Visual Studio 2019. Um jetzt mit AutoML und dem Model Builder durchzustarten, klicken Sie nach der automatischen Erstellung des Console-App-Projekts mit der rechten Maustaste auf den Projektnamen im Solution Explorer und wählen im Kontextmenü *Add | Machine Learning* aus (Bild 7.14). Die Auswahl öffnet den ML.NET Model Builder, wie in Bild 7.15 zu sehen ist.

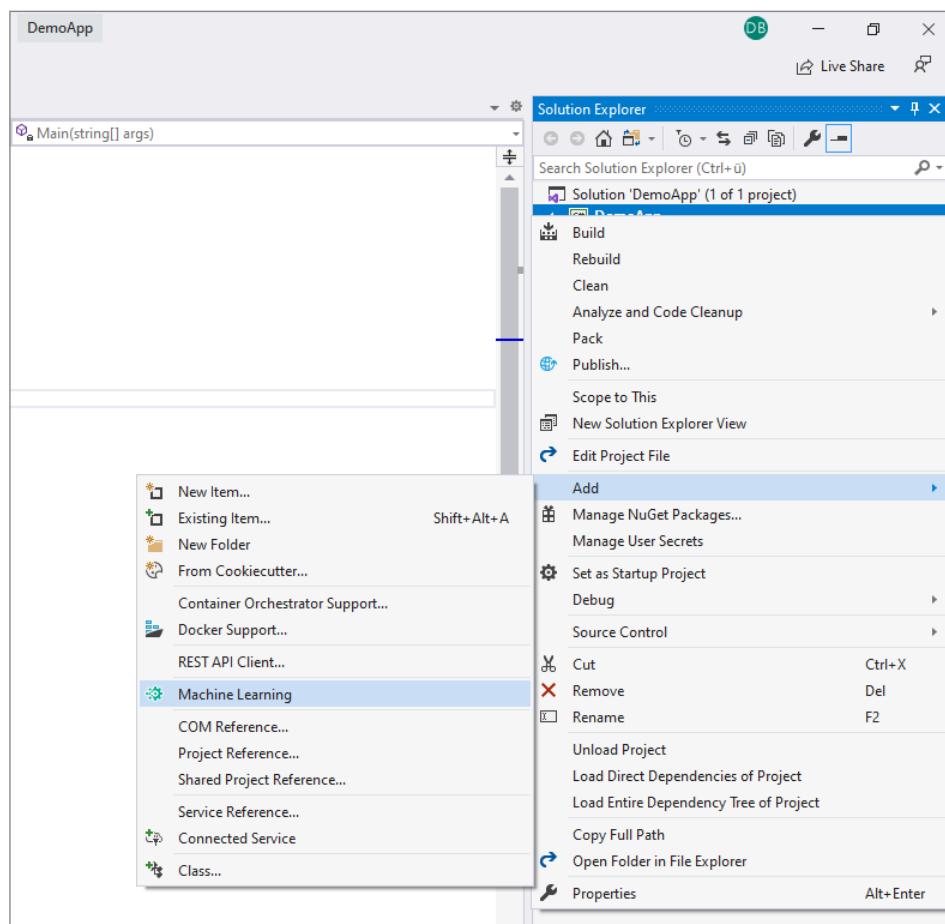


Bild 7.14 Model Builder zum Projekt hinzufügen

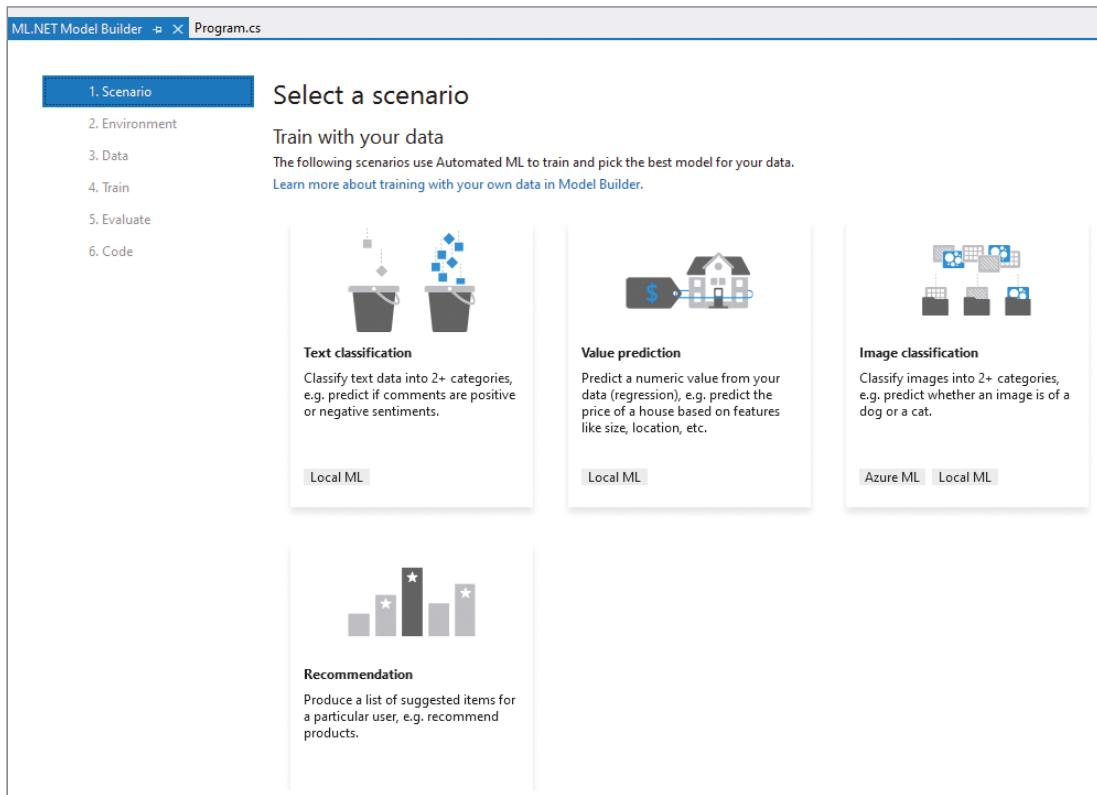


Bild 7.15 Model Builder in Visual Studio 2019

Im Model Builder sehen Sie auf der linken Seite das Menü für die einzelnen schon beschriebenen Schritte *Scenario* (Szenario), *Environment* (Umgebung), *Data* (Daten), *Train* (Training), *Evaluate* (Bewertung) und *Code*.

Aus den verfügbaren Szenarien wählen Sie das ML.NET Model Text classification (beinhaltet auch eine Mehrklassen-Klassifikation) für das Entwicklungsbeispiel. Klicken Sie auf die Kachel *Text classification*, um die Lagermenge für den verfügbaren Artikelbestand vorherzusagen.

Nachdem Sie das entsprechende Szenario ausgewählt haben, können Sie über den Button *Data* die benötigten Daten Ihrem ML-Modell hinzufügen. Belassen Sie die Einstellung auf *File* für das Laden der Daten und wählen Sie über *Select a file* die CSV-Datei der Artikeldaten aus. Legen Sie dann als Erstes die Spalte für die gewünschte Vorhersage, also das für das Modell benötigte Label, fest.

Um den Gesamt-Artikelbestand vorherzusagen, wählen Sie über *Select column* die Spalte *Gesamt_Artikelbestand* aus. Des Weiteren müssen Sie die *Input Spalten* mit den *Features* (Merkmale) auswählen, um das Ergebnis ermitteln zu können. Im Beispiel lassen Sie die vorgeschlagene Einstellung aller fünf Spalten bestehen. Bild 7.16 zeigt die gemachten Einstellungen für das Model-Builder-Dialogfenster *Add data*.

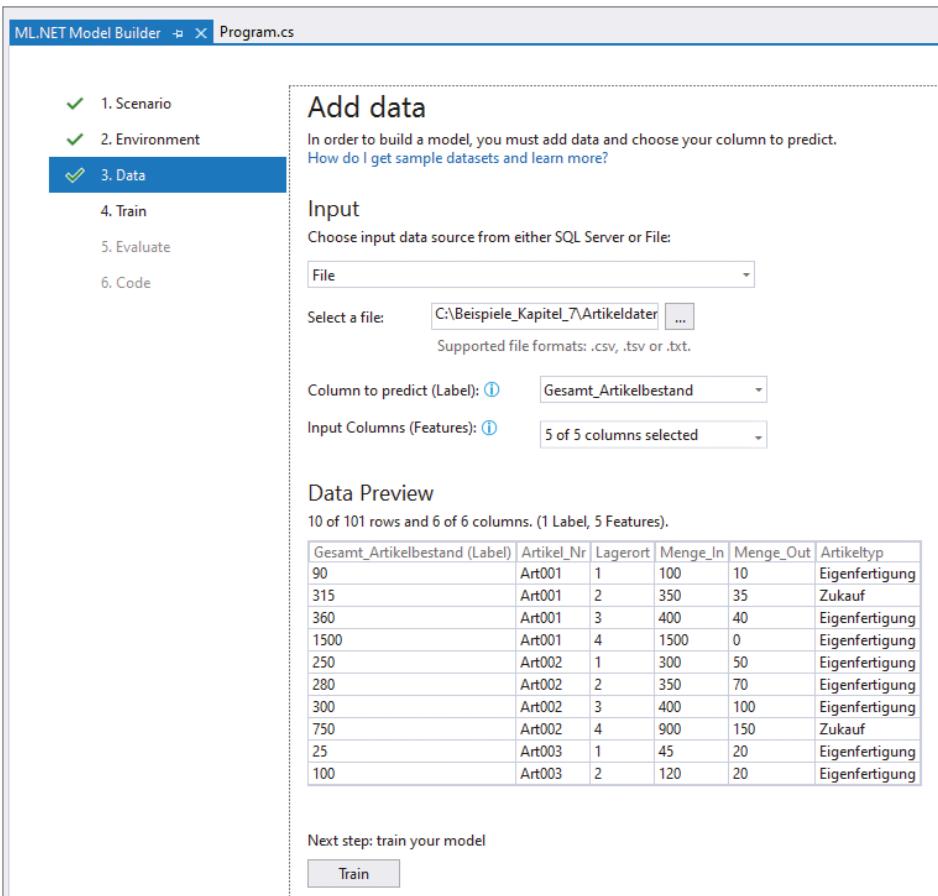


Bild 7.16 Einstellung der gewünschten Vorhersage

Klicken Sie nun auf den Button *Train*, um das Modell mit den zur Verfügung gestellten Daten zu trainieren. Auf dem Train-Dialogbildschirm des Model Builder sehen Sie noch einmal das ausgewählte Szenario, in unserem Beispiel die Klassifizierung, und die Zeit zum Trainieren des Modells. Vorgeschlagen werden hier 10 Sekunden, allerdings ist der Datenbestand nicht groß, dafür aber sehr ähnlich, sodass Sie dem Model Builder ein wenig mehr Zeit zum Trainieren geben sollten. Daher wird für das Beispiel die Zeit auf 100 Sekunden heraufgesetzt, um ein besseres Ergebnis beim ML-Modell zu erzielen. Klicken Sie dann auf die Schaltfläche *Start training* und warten Sie, bis der Model Builder das Training abgeschlossen hat.

Wie schon erwähnt, versucht der Model Builder in der Trainingsphase, unter Verwendung von AutoML für die Algorithmen-Selektion, verschiedene ML-Algorithmen zu durchlaufen und den für das gestellte Problem am besten passenden auszuwählen, um das Modell zu trainieren. Der Model Builder dokumentiert die ML-Algorithmen-Suche im Output-Window von Visual Studio. Nach der Trainingszeit von 100 Sekunden identifiziert der Model Builder den ML-Algorithmus *LightGbmMulti* (Bild 7.17).

The screenshot shows the ML.NET Model Builder interface. In the top navigation bar, 'ML.NET Model Builder' is selected, and the file 'Program.cs' is open. On the left, a sidebar lists steps: 1. Scenario, 2. Environment, 3. Data, 4. Train (selected), 5. Evaluate, and 6. Code.

Train

Specify a time to train for evaluating various models.
How long should I train for?

Training setup summary ▾

Time to train (seconds):

Training results

Best accuracy:	100%
Best model:	LightGbmMulti
Training time:	94,96 seconds
Models explored (total):	1

Next step: evaluate your model

Output

Show output from: Machine Learning

Summary

```
ML Task: multiclass-classification
Dataset: C:\Beispiele_Kapitel_7\Artikeldaten.csv
Label : Gesamt_Artikelbestand
Total experiment time : 94,9585332 Secs
Total number of models explored: 5
```

Top 5 models explored

Trainer	MicroAccuracy	MacroAccuracy	Duration	#Iteration
LightGbmMulti	1,0000	1,0000	11,4	1
FastTreeOva	0,8750	0,8571	32,6	2
SdcaMaximumEntropyMulti	0,6667	0,8000	29,7	3
AveragedPerceptronOva	0,5000	0,5000	11,4	4
SymbolicSgdLogisticRegressionOva	0,0000	0,0000	9,9	5

Code Generated

Bild 7.17 Das AutoML-Ergebnis im Model Builder

Die *MicroAccuracy*- und *MacroAccuracy*-Werte aus Bild 7.17 bieten Ihnen zwei verschiedene Metriken für die Genauigkeit der Modellvorhersage. *MicroAccuracy* ist die normale Genauigkeit, die sich aus der Anzahl der richtigen Vorhersage für die Testdaten, dividiert durch die Gesamtzahl der Elemente ergibt. *MacroAccuracy* ist die durchschnittliche Genauigkeit für die vorherzusagenden Klassen. Die Wertangabe *MacroAccuracy* ist ein nützlicher Hinweis, wenn ein Datensatz zu einer Klasse stark verzerrt ist.

Über den Button *Evaluate* zeigt der Model Builder den besten ermittelten ML-Algorithmus für die Aufgabenstellung im durchgeführten Prozess. Über den Bereich *Try your model* können Sie noch einmal gezielt das trainierte Modell auswerten.

Der *LightGbm*-Algorithmus [35] ist ein sogenanntes Gradientenverstärkungs-Framework, das einen baumbasierten Lernalgorithmus verwendet (siehe Abschnitt 2.6.3, „Entscheidungsbäume“). LightGbm ist ein relativ neuer Algorithmus, der Bäume vertikal wachsen lässt, während andere Algorithmen Bäume ebenenweise wachsen lassen. Des Weiteren entsteht der Baum beim LightGbm jeweils blattweise. Der Algorithmus wählt das Blatt mit dem maximalen Delta-Verlust zum Wachsen aus. Das heißt, die blattweise Strategie teilt das Blatt auf, das den Verlust am stärksten reduziert. Aber Vorsicht, das blattweise Training ist zwar flexibler, aber sehr viel anfälliger für eine Überanpassung, vor allem bei einer kleinen Datenmenge. Das ist zwar für das Beispiel nicht relevant, aber in der Praxis sollten Sie den LightGbm-Algorithmus nur für Daten mit mehr als 10.000 Zeilen verwenden.

Über den Button *Code* wird jetzt das finale Modell und der dazugehörige Code in Form von zwei neuen Projekten generiert und der Solution *DemoApp* hinzugefügt. Die Erzeugung der Übernahme starten Sie, indem Sie auf den Button *Add Projects* klicken. Bild 7.18 zeigt, dass sowohl das Modell *DemoAppML.Model* als auch das Konsolenprojekt *DemoAppML.ConsoleApp* zu dem *DemoApp*-Projekt hinzugefügt wurden.

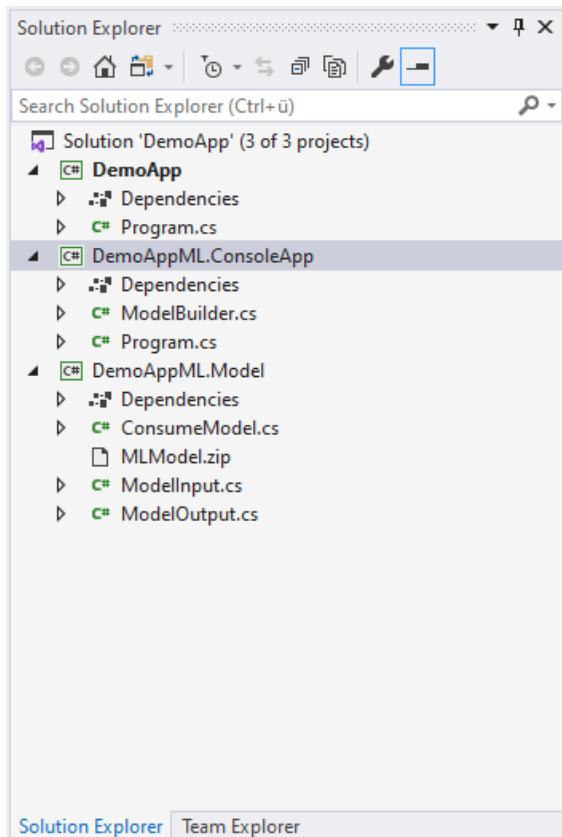


Bild 7.18 Erweiterung des Projekts durch den Model Builder

Der automatisch generierte Code aus Listing 7.3 ist leicht zu interpretieren. Der Beispielcode trifft mit dem ersten Datenelement in der Testdataset-Datei eine Vorhersage. Sie können die Vorlage bearbeiten, um eine neue, bisher noch nicht gemachte Vorhersage aufzurufen. Der bearbeitete Vorhersagecode lädt das trainierte Modell zuerst in den Arbeitsspeicher und erstellt mithilfe des Modells ein *PredictionEngine*-Objekt.

Listing 7.3 Der automatisch erzeuge Code

```
namespace DemoAppML.ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create single instance of sample data from first line of dataset
            // for model input
            ModelInput sampleData = new ModelInput()
            {
                Artikel_Nr = @"Art001",
                Lagerort = 1F,
                Menge_In = 100F,
                Menge_Out = 10F,
                Artikeltyp = @"Eigenfertigung",
            };

            // Make a single prediction on the sample data and print results
            var predictionResult = ConsumeModel.Predict(sampleData);

            Console.WriteLine("Using model to make single prediction --
                Comparing actual Gesamt_Artikelbestand with predicted
                Gesamt_Artikelbestand from sample data...\n\n");
            Console.WriteLine($"Artikel_Nr: {sampleData.Artikel_Nr}");
            Console.WriteLine($"Lagerort: {sampleData.Lagerort}");
            Console.WriteLine($"Menge_In: {sampleData.Menge_In}");
            Console.WriteLine($"Menge_Out: {sampleData.Menge_Out}");
            Console.WriteLine($"Artikeltyp: {sampleData.Artikeltyp}");
            Console.WriteLine($"\\n\\nPredicted Gesamt_Artikelbestand value
                {predictionResult.Prediction} \\nPredicted
                Gesamt_Artikelbestand scores:
                [{String.Join(", ", predictionResult.Score)}]\\n\\n");
            Console.WriteLine("===== End of process,
                hit any key to finish =====");
            Console.ReadKey();
        }
    }
}
```

Am Beispiel der *DemoAppML.ConsoleApp* sehen Sie, wie einfach es ist, den generierten Code mit dem ML-Modell, das der Model Builder erstellt hat, zu verwenden. Sie können diese Klasse in Ihrem eigenen Code verwenden, um das Modell für Ihre Vorhersagen in einer eigenen App zu nutzen.

Experimentieren Sie ausgiebig mit dem Model Builder Tool und AutoML, erweitern und modifizieren Sie die Testdaten, lassen Sie sich mittels neuer Daten und unter Verwendung

von Regressionsalgorithmen numerische Werte vorhersagen oder verwenden Sie Clustering, um Gruppierungen innerhalb von Daten zu finden und vorherzusagen.



UCI Machine Learning Repository

Ohne entsprechend aufbereitete Daten bzw. ohne große Datenmengen ist es immer schwierig, eine datengetriebene Anwendung bzw. ein entsprechendes ML-Modell zu entwickeln und zu testen. Inzwischen bieten viele Universitäten entsprechende Testdatasets an. Auch die Webseite *UC Irvine Machine Learning Repository* stellt für die ML Community eine Vielzahl von Datensätzen aus den verschiedensten Bereichen zur Verfügung. Sie finden dort entsprechende Datasets und Anregungen [36].

7.5.10 Die Kommandozeile (CLI)

Für Leute, die nicht Visual Studio verwenden oder auch nicht unter Windows arbeiten, bietet ML.NET als plattformübergreifendes Tool das Command Line Interface (CLI) als Kommandozeile an. Mit diesem können Sie AutoML zur einfachen Erstellung von ML-Modellen ganz bequem in der Kommandozeile verwenden.

Das CLI ist ein .NET Core Global Tool, das auf .NET Core 2.2 SDK aufsetzt und sich via PowerShell im Mac-Terminal oder in der Linux Bash ausführen lässt. Die Installation erfolgt über den Befehl `dotnet tool install -g mlnet`. Wie auch der Model Builder generiert die ML.NET CLI C#-Beispielcode zum Ausführen des erstellten Modells sowie den C#-Code, der zum Erstellen und Trainieren des Modells verwendet wurde, um den von AutoML gewählten Algorithmus und die Einstellungen einzusehen zu können.

Um mit der ML.NET CLI ein Modell zu generieren, müssen Sie einfach den Befehl `mlnet auto-train` aufrufen und Ihren Datensatz, die ML-Aufgabe und die Zeit zum Trainieren als Parameter angeben. Die CLI erzeugt dann automatisch die gleichen zusammenfassenden Informationen und Klassen wie der Model Builder unter Visual Studio. Mit dem Aufruf `mlnet auto-train -help` lassen sich alle Optionen für die Operation `auto-train` anzeigen.

7.5.11 Die Zukunft von AutoML

AutoML hat in den letzten Jahren erstaunliche Fortschritte erzielt und bietet Anwendern bzw. Entwicklern eine echte Unterstützung in Bezug auf die Verwendung von ML-Algorithmen und Lösungsverfahren an. Wie Sie im oben gezeigten Beispiel gesehen haben, kennt das ML.NET Framework etliche Algorithmen für die Klassifizierungen mit mehreren möglichen Werten.

Der Vorteil bei der Verwendung von AutoML besteht in erster Linie darin, dass nicht nur der Vorlagencode generiert wird, um ein trainiertes Modell zu laden, sondern auch der zugrunde liegende Code, der zum Erstellen, Trainieren und Speichern des Vorhersagemodells verwendet wurde. So ist eine Umsetzung in einer eigenen lokalen Anwendung sehr schnell möglich.

Was AutoML und ML.NET im Zusammenspiel noch nicht beherrschen, ist die Vorhersage auf Grundlage eines neuronalen Netzes. Da neuronale Netze wesentlich komplexer als traditio-

nelle Machine-Learning-Algorithmen sind, wird die Umsetzung von neuronalen Netzen von AutoML noch nicht unterstützt. Es wird aber geforscht, entwickelt und geprüft, um ML.NET und AutoML mit Funktionen für die Entwicklung von neuronalen Vorhersagemodellen zu erweitern. Ein Ansatz ist der Neural Architecture Search (NAS)-Algorithmus.

Die Suche nach neuronaler Architektur

Wie Sie inzwischen erfahren haben, ist die Entwicklung von Modellen für neuronale Netze oft ein erhebliches Architektur-Engineering. Möchte man bei Machine Learning bzw. Deep Learning die bestmögliche Leistung erreichen, ist es heute in der Regel immer noch am besten, ein eigenes ML-Modell zu entwerfen, da dieses ML-Modell dann zu 100 % auf sein zu lösendes Problem spezifiziert werden kann.

Dabei handelt es sich aber um eine entsprechend große Herausforderung. Vielfach sind unbestimmte Faktoren vorhanden, die die Entwicklung schwierig und zeitaufwendig machen, und man verbringt viel Zeit mit dem Trial-and-Error-Ansatz.

Hier kommt dann Neural Architecture Search (NAS) ins Spiel. NAS ist ein Algorithmus, der nach der besten neuronalen Netzwerkarchitektur sucht. Bei NAS beginnen Sie mit der Definition von einer Reihe von Bausteinen, die möglicherweise in Ihrem Netzwerk verwendet werden sollen. Im NAS-Algorithmus tastet dann ein Controller im Recurrent Neural Network (RNN) diese definierten Bausteine ab und setzt sie zu einer Art End-to-End-Architektur zusammen.

Diese neue Netzwerkarchitektur wird dann auf Konvergenz trainiert, um eine gewisse Genauigkeit bei einem definierten Validierungssatz zu erreichen, sodass man sich erhofft, dass der Controller im Laufe der Zeit eine immer besser werdende Architektur erzeugen kann. Man kann also damit rechnen, dass sich im Laufe der nächsten Zeit noch viele weitere Möglichkeiten bei AutoML ergeben werden.

■ 7.6 Benutzerdefiniertes ML.NET

Ohne den Einsatz von Model Builder und AutoML lassen sich mit ML.NET auch benutzerdefinierte Machine-Learning-Modelle erstellen. Der benutzerdefinierte Ansatz stellt – im Gegensatz zu AutoML mit seinen vorgefertigten Szenarien – eine rein programmiertechnische Umsetzung der benötigten ML-Lösung dar und er ist sehr flexibel. Da das Framework außerdem Bibliotheken und NuGet-Pakete zur Verwendung in .NET-Anwendungen umfasst, können Sie ML.NET überall ausführen.

ML.NET möchte es Entwicklern ermöglichen, Ihre vorhandenen .NET-Fähigkeiten zu nutzen, um Machine Learning in entsprechende .NET-Anwendungen zu integrieren. So können Sie mit C# als Programmiersprache Ihrer Wahl eigene leistungsfähige ML-Modelle entwickeln, ohne auf Python oder R umsteigen zu müssen. Mit ML.NET können Sie ML-Modelle lokal oder in einer beliebigen Cloud, wie zum Beispiel Microsoft Azure, erstellen und konsumieren. Es sind auch problemlos Offline-Lösungen möglich, um ML.NET auch in Ihren Desktop-Anwendungen mit UWP, WPF und Windows Forms zu verwenden.

7.6.1 ML.NET-Komponenten

ML.NET bietet neben dem Model Builder und AutoML auch eine .NET API an, die Sie für einen benutzerdefinierten Machine Learning Workflow nutzen können. Die Hauptaktionen sind hier die Erstellung des ML-Modells, um dieses dann für die Erstellung von Vorhersagen in .NET-Anwendungen für den Anwender bereitzustellen. ML.NET stellt für die Entwicklung von Machine-Learning-Anwendungen mehrere Hauptkomponenten zur Verfügung (Bild 7.19).

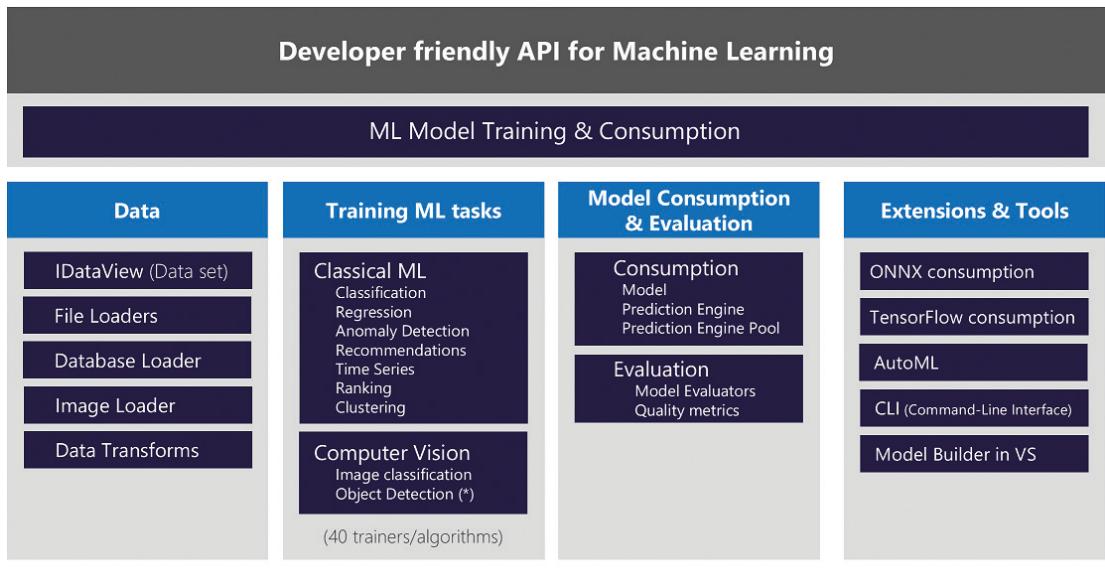


Bild 7.19 Die Hauptkomponenten von ML.NET (Quelle: Microsoft)

Für eine benutzerdefinierte ML.NET-Entwicklung benutzt man in den meisten Fällen folgende Komponenten für den Data-Bereich:

- *IDataView*
- *DataLoaders*
- *DataTransforms*

In ML.NET werden Daten als *IDataView*-Objekt dargestellt. Das *IDataView*-Objekt kann unter ML.NET sehr flexibel und effizient zur Beschreibung und zum Laden von Tabellendaten genutzt werden. Das Objekt dient somit dem Laden von Datensätzen zu weiterer Verarbeitung.

Das *IDataView*-Objekt ist für die Handhabung hochdimensionaler Daten und großer Datensätze konzipiert und beinhaltet die Daten, die während der Datentransformation des Modelltrainings verwendet werden. Für diesen Zweck ist auch ein Streamen der Datenquelle möglich, ohne alle Daten in den Speicher laden zu müssen. Durch diese Möglichkeit ist auch die Verarbeitung großer Datens Mengen gegeben. Das *IDataView*-Objekt kann Zahlen, Texte, Booleans, Vektoren und vieles mehr enthalten.

Sie können Ihre Datensätze aus praktisch jeder Datenquelle in ein *IDataView*-Objekt laden. Diese Aufgabe übernimmt der *DataLoader*, den Sie für alle typischen Quellen wie Text-, Binär- und Bilddateien in ML.NET verwenden können. Über den speziellen *DatabaseLoader* können Sie die Daten direkt aus einer von *System.Data* unterstützten relationalen Datenbank wie SQL Server oder Oracle laden, um die Daten zu trainieren.

Nicht immer lassen sich alle Informationen aus den Daten komfortabel bearbeiten. So kann es immer wieder vorkommen, dass Sie für die Auswertung bzw. Nutzung des Modells eine gezielte Konvertierung zwischen den Datentypen vornehmen müssen. In den meisten Fällen müssen alle Daten in Zahlen oder numerische Vektoren umgewandelt werden. Hierbei kann dann *DataTransforms* hilfreich sein.

ML.NET bietet für die Datentransformation eine Vielzahl von unterstützenden Methoden wie zum Beispiel Text-Featurizer oder auch Hot-Encoder an, um die Daten in ein akzeptables Format für den benutzten ML-Algorithmus zu konvertieren.

Model Training

Für die Trainingsaufgaben bzw. dem Trainieren des Modells werden hauptsächlich die zwei folgenden Komponenten verwendet:

- *Classical ML tasks*
- *Computer Vision*

Unter *Classical ML tasks* versteht man bei ML.NET die klassischen ML-Aufgaben wie zum Beispiel Klassifikation, Regression, Zeitreihen und vieles mehr. ML.NET bietet mehr als 40 Algorithmen für die unterschiedlichsten Bereiche an. Diese Algorithmen sind im Namespace *Microsoft.ML.Trainers* implementiert und auf eine bestimmte Aufgabe ausgerichtet, sodass Sie den spezifischen Algorithmus auswählen und feinabstimmen können, damit er eine höhere Genauigkeit erreicht und Ihre Machine-Learning-Aufgabe besser lösen kann.

Seit Version 1.4 bietet ML.NET unter dem Begriff Computer Vision auch Bildklassifizierung und Bilderkennung für Ihre eigenen benutzerdefinierten Bilder an. Hierbei kommt teilweise noch TensorFlow für das Training zum Einsatz. Microsoft arbeitet aber daran, eine komplett eigenständige Unterstützung für die Objekterkennung zu implementieren.

Model Consumption und Evaluation

Mit der Model Consumption bietet ML.NET, nachdem Sie Ihr erstelltes ML-Modell trainiert haben, eine oder mehrere Möglichkeiten zur Erstellung der Vorhersage an. Je nach Vorhersage und Datenmenge verwendet ML.NET hier unterschiedliche PredictionEngines, also Vorhersagemaschinen. Es wird hierbei von ML.NET unterschieden, ob es sich um Einzelvorhersagen oder um die Unterstützung von Vorhersagen in skalierbaren Multithread-Anwendungen handelt, in der dann ein PredictionEnginePool für den Einsatz herangezogen wird.

Unter Model Evaluation versteht man die Überprüfung des geschulten Modells. Bevor Sie ein ML-Modell produktiv einsetzen, sollten Sie auf jeden Fall sicherstellen, dass die gewünschte Vorhersage auch eine entsprechende Qualität erreicht. ML.NET stellt mehrere Bewertungsmöglichkeiten für jede Machine-Learning-Aufgabe zur Verfügung, damit Sie die Genauigkeit Ihres Modells sowie viele weitere typische Metriken für maschinelles Lernen in Abhängigkeit von der angestrebten ML-Aufgabe herausfinden können.

Tools and Extensions

Die Tools von ML.NET haben Sie ja schon kennengelernt. Hierzu gehören der Model Builder für Visual Studio und die plattformübergreifende Kommandozeile (CLI). Diese Tools verwenden intern die AutoML-API von ML.NET, um das beste ML-Modell für Ihre Daten und der gewünschten Aufgabenstellung zu finden.

Bei den Extensions (Erweiterungen) handelt es sich um die Integration des ONNX-Modells. ONNX (Open Neural Network Exchange) ist ein standardisiertes und interoperables ML-Modellformat. Sie können in ML.NET jedes vortrainierte ONNX-Modell ausführen und bewerten.

Des Weiteren bietet ML.NET noch die Integration von TensorFlow-Modellen an. Mithilfe dieser API können Sie auch jedes vortrainierte TensorFlow-Modell ausführen und bewerten. TensorFlow wird in diesem Fall als Backend für die ML.NET API benutzt.

7.6.2 Benutzerdefinierter Workflow

Die zur Verfügung gestellten Komponenten lassen sich in vielfältiger Form in der Entwicklung von benutzerdefinierten ML-Modellen nutzen. Das Programmieren eines ML-Modells folgt immer den gleichen Regeln. Es werden Daten gesammelt und aufbereitet, um sie dann mithilfe von ML-Algorithmen zu trainieren. Dabei leitet der ML-Algorithmus aus den Daten ein ML-Modell ab, das dann in der Lage ist, bestimmte Muster in den Daten zu erkennen. Der Workflow für die Modellentwicklung mit ML.NET und der iterative Prozess dahinter ist in jedem ML-Projekt einheitlich. Bild 7.20 zeigt den Arbeitsablauf für die Erzeugung eines ML-Modells mit ML.NET.

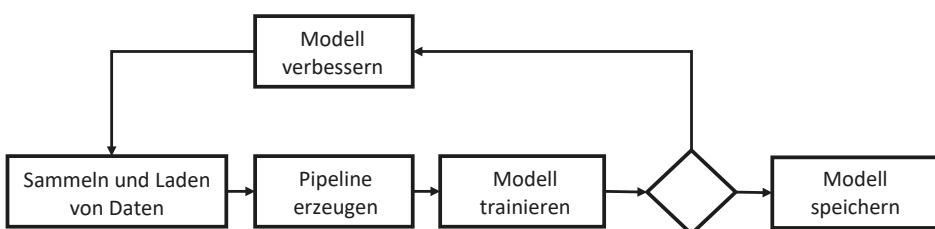


Bild 7.20 Arbeitsablauf für das Erzeugen eines ML-Modells

Aus entwicklungsseitiger Sicht lassen sich die Schritte in Bild 7.20 wie folgt beschreiben. Das Sammeln und Laden von Daten bedeutet, dass die Trainingsdaten erstellt sind und in ein *IDataView*-Objekt von ML.NET überführt werden.

Dann wird eine Pipeline erzeugt, die die Vorgänge zum Extrahieren von Features und das Anwenden eines ML-Algorithmus beinhaltet. Die Pipeline stellt einen Datenstrom zwischen zwei Prozessen durch einen Puffer nach dem First-In-First-Out (FIFO)-Prinzip dar.

Das Trainieren des ML-Modells erfolgt über den Aufruf der *Fit*-Methode aus dem ML.NET-Framework auf der Pipeline. Hiernach wird das ML-Modell bewertet und eventuell noch einmal durch Anpassungen verbessert. Dieser Vorgang wird iterativ durchgeführt, bis das ML-Modell die entsprechende Qualität erreicht hat. Ist dieser Prozessschritt vollzogen, kann das ML-Modell im Binärformat zur Verwendung in einer Anwendung gespeichert werden.

7.6.3 Erstellen einer benutzerdefinierten Anwendung

Wie beschrieben, besitzt ML.NET alle notwendigen Methoden und grundlegende Funktionen für das Erstellen von ML-Modellen mit C#. Das Hinzufügen von ML.NET zu Ihrem C#-Projekt ist ganz einfach. Das Einzige, was Sie tun müssen, ist, das Microsoft.ML-Paket zu installieren.

Öffnen Sie Visual Studio und erstellen Sie eine neue Console-App mit dem Namen *ArtikeldatenApp*. Fügen Sie Ihrer Projekt-Solution die Datei *Artikeldaten.csv* hinzu. Stellen Sie hierbei sicher, dass die Property (Eigenschaft) *Copy to Output Directory*, wie in Bild 7.21 dargestellt, auf *Copy Always*, also immer kopieren, gesetzt ist.

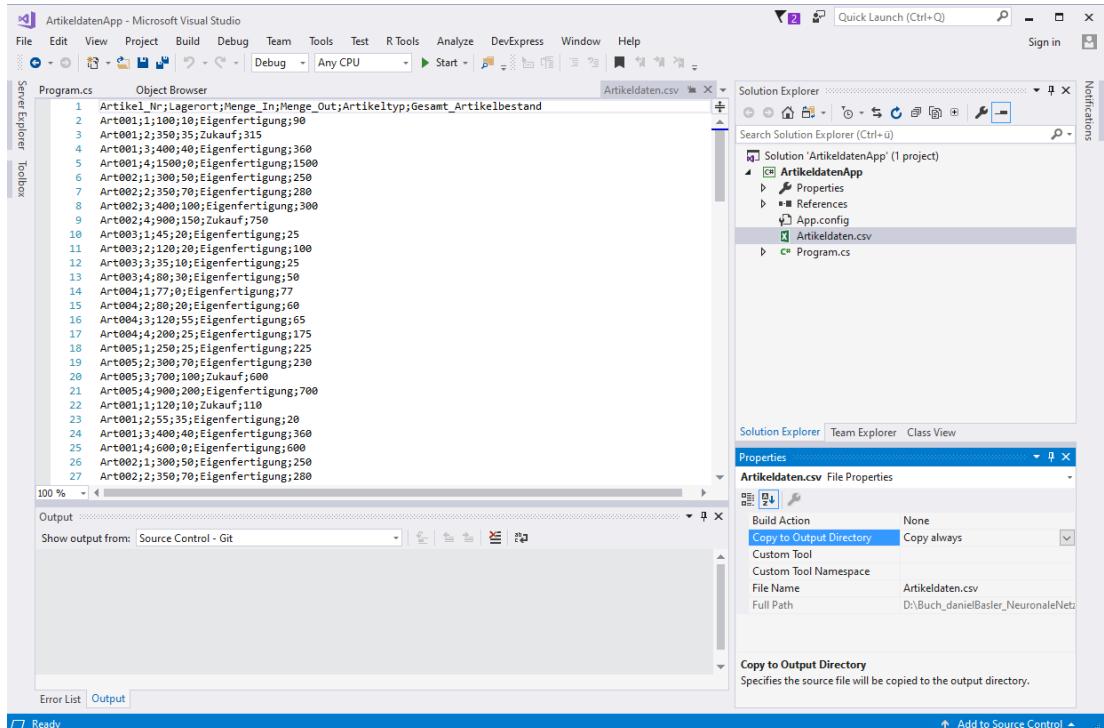


Bild 7.21 CSV-Datei in Visual Studio einbinden

Fügen Sie im nächsten Schritt das NuGet-Package *Microsoft.ML* hinzu. Ist die Installation des Pakets in Ihrem Projekt durchgeführt, so kann man unter *References* einen Blick auf die DLL *Microsoft.ML.StandardTrainers* werfen. Sie sehen dort die Vielzahl der implementierten ML-Algorithmen und die damit verbundenen Einsatzszenarien (Bild 7.22).

An der Vielzahl der ML-Algorithmen können Sie aber auch erkennen, dass es nicht immer leicht ist, die passenden ML-Algorithmen für Ihre Aufgabe bzw. Ihre Problemstellung zu finden. Aus diesem Grund kommen Sie bei der Entwicklung nicht an Trial-and-Error vorbei. Eine komplette Auflistung und Dokumentation aller Algorithmen in ML.NET finden Sie unter [37].

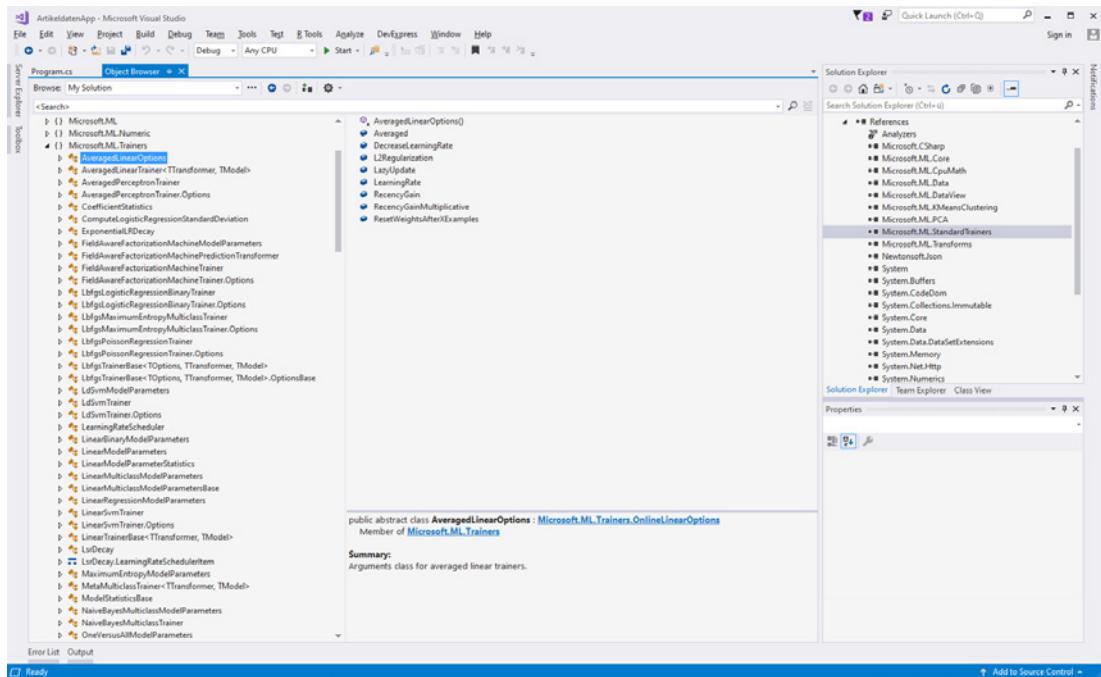


Bild 7.22 Die vielfältigen ML-Algorithmen in ML.NET

Nachdem ML.NET der Solution hinzugefügt worden ist, können Sie die Datei *Program.cs* um folgenden Namespace erweitern.

```
using Microsoft.ML;
using Microsoft.ML.Data;
```

Ist der Namespace ergänzt, so können Sie eine neue ML.NET-Umgebung durch die Initialisierung von *MLContext* erstellen. Das heißt, eine ML.NET Anwendung beginnt immer mit einem *MLContext*-Objekt. Dieses Singletonobjekt enthält sogenannte Kataloge. Ein Katalog stellt unter ML.NET eine Factory zum Laden und Speichern von Daten für Transformationen (Datenvorbereitung), Trainer (hierunter befinden sich die verfügbaren ML-Algorithmen) und Modellvorgangskomponenten dar. Des Weiteren verfügt das Katalogobjekt über Methoden, um die verschiedenen benötigten Komponententypen erstellen zu können. Erweitern Sie daher Ihre Datei *Program.cs* mit dem folgenden Code, um ein *MLContext*-Objekt zu initialisieren.

```
MLContext mlContext = new MLContext();
```

Im nächsten Schritt fügen Sie den Code zum Laden Ihrer Artikeldaten aus der CSV-Datei in einer *IDataView* hinzu. Bevor Sie diese aber laden können, müssen Sie eine Klasse für das benötigte Datenschema Ihres Datensatzes für das ML-Modell definieren. Erweitern Sie daher die Datei *Program.cs* um die neue Klasse *ModelInput*.

```

public class ModelInput
{
    [LoadColumn(0)]
    public string ArtikelNr;
    [LoadColumn(1)]
    public float Lagerort;
    [LoadColumn(2)]
    public float MengeIn;
    [LoadColumn(3)]
    public float MengeOut;
    [LoadColumn(4)]
    public string Artikeltyp;
    [LoadColumn(5)]
    public float GesamtArtikelbestand;
}

```

Ist die Klasse *ModelInput* implementiert, so können Sie mithilfe eines TextLoader Daten aus Ihrer Dataset-Datei in ein entsprechendes *IDataView*-Objekt laden, Erweitern Sie daher die *Main*-Methode in der Datei *Program.cs* wie folgt.

```

IDataView trainData = mlContext
    .Data.LoadFromTextFile<ModelInput>(
        "Artikeldaten.csv", separatorChar: ';',
        hasHeader: true);

```

Sie können alternativ, wie schon erläutert, auch einen *DatabaseLoader* verwenden, um direkt auf eine relationale Datenbank zugreifen zu können.

7.6.4 Datentransformation

Im Beispiel ist die Umsetzung relativ einfach gehalten. Da es sich bei Lagerort, *MengeIn*, *MengeOut* um einfache Integer-Werte handelt, die schon das geeignete Format für den ML-Algorithmus besitzen, muss nur die *ArtikelNr* angepasst werden. *ArtikelNr* besitzt einen String-Wert, der aber für einen ML-Algorithmus nicht verwendet werden kann. Sie benötigen hierfür eine *OneHotEncoding*-Datentransformation, um den String-Typ in das benötigte Format für numerische Vektoren zu konvertieren.

Für diesen Transformationsvorgang erstellen Sie eine Pipeline. In jedem *MLContext*-Katalog befindet sich eine Reihe von Erweiterungsmethoden, die in einer Trainingspipeline verwendet werden können. Im Beispiel fügen Sie ein *OneHotEncoding* in der Pipeline hinzu, mit der Spalte *ArtikelNr* als Eingabe und einer neuen Spalte mit dem Namen *ArtikelNrEncoded* als Ausgabe.

Des Weiteren verarbeiten die Algorithmen in ML.NET die Eingabe aus einer einzigen Spalte als Features. Diese Spalte definiert alle Merkmale, die als Eingaben verwendet werden sollen. In der Pipeline für die Datentransformation können Sie somit eine Spalte mit dem Namen *Features* erstellen und diese Spalte mit den gewünschten Merkmalen (Features) in Ihrem Dataset verketten. Die komplette *dataProcessPipeline* für die Datentransformation wird wie folgt in der *Main*-Methode implementiert.

```
var dataProcessPipeline =
    mlContext.Transforms.Categorical
    .OneHotEncoding(outputColumnName:
    "ArtikelNrEncoded", "ArtikelNr")
    .Append(mlContext.Transforms.Concatenate(
    outputColumnName: "Features", "Lagerort",
    "MengeIn", "MengeOut"));
```

7.6.5 ML.NET-Algorithmus

Nachdem Sie die Datentransformation implementiert haben, können Sie im nächsten Schritt einen ML-Algorithmus auswählen. Die Auswahl erfolgt auch hier über das definierte *mlContext*-Objekt. Über den *mlContext.MulticlassClassification.Trainers* stehen Ihnen mehrere verschiedene ML-Algorithmen über IntelliSense zur Verfügung.

Im Beispiel fügen Sie den *LbfgsMaximumEntropy*-Algorithmus hinzu, wobei Sie die Spalte *GesamtArtikelbestand* als Label und die Spalte *Features* als Merkmale angeben. Die Implementierung erfolgt in der *Main*-Methode mit der nachfolgenden Codezeile.

```
var trainer = mlContext
    .MulticlassClassification.Trainers.LbfgsMaximumEntropy(labelColumnName:
    "GesamtArtikelbestand", featureColumnName: "Features");
```



Maximum-Entropie-Modell

Das Maximum-Entropie-Modell in dem verwendeten Beispiel ist eine Verallgemeinerung der linearen logistischen Regression. Der Hauptunterschied zwischen dem Maximum-Entropie-Modell und der linearen logistischen Regression besteht in der Anzahl der Klassen, die im betrachteten Klassifizierungsproblem unterstützt werden. Das heißt, man verwendet das Maximum-Entropie-Modell, wenn man eine lineare logistische Regression mit mehr als zwei Klassen durchführen möchte.

7.6.6 Erstellen und Trainieren eines ML-Modells

Um jetzt das ML-Modell mit ML.NET zu erstellen und zu trainieren, müssen Sie eine weitere Pipeline erstellen, die sowohl die erforderlichen aufbereiteten Datensätze als auch den Trainingsalgorithmus enthält. Implementieren Sie hierfür den nachfolgenden Programmcode in die *Main*-Methode der *Program.cs*-Datei.

```
var trainingPipeline = dataProcessPipeline.Append(trainer);
var model = trainingPipeline.Fit(trainData);
```

Die Datentransformation und der ML-Algorithmus, den Sie zu diesem Zeitpunkt angeben, werden erst dann ausgeführt, wenn Sie die *Fit*-Methode aufrufen. Die *Fit*-Methode führt das Training aus und gibt ein trainiertes Model zurück.

7.6.7 Modellauswertung

Nach der Modell-Trainingsphase können Sie die Auswertung von ML.NET verwenden, um die Leistung Ihres Modells anhand einer Vielzahl von Metriken zu bewerten. Um das Modell im Beispiel zu bewerten, verwenden Sie die Metrik *MacroAccuracy*, die Sie ja schon aus dem Einsatz mit dem Model Builder kennen.

Die Bewertung des trainierten Modells erfolgt über den Testdatensatz, der in ein *IDataView*-Objekt geladen wird und mit der Methode *Transform* die Vorhersage ausführt. Die Funktion *Evaluate* wird verwendet, um die gewünschte Metrik ausführen zu können. Der Programmcode für die Auswertung des Modells hat folgenden Aufbau.

```
IDataView testData = mlContext.Data.  
    LoadFromTextFile<ModelInput>(  
        "Artikeldaten.csv", separatorChar: ';',  
        hasHeader: true);  
  
IDataView predictions = model.Transform(testData);  
var metrics = mlContext.MulticlassClassification.Evaluate  
    (predictions, "GesamtArtikelbestand");  
  
Console.WriteLine($"Model Ergebnis Marco Accuracy vom " + $"{metrics.MacroAccuracy}");  
Console.ReadLine();
```

Die Auswertung des Modells erfolgt dann einfach in der Konsole der Anwendung. Eine besonders wichtige und nützliche Eigenschaft von *IDataView*-Objekten ist, dass sie in ML.NET verzögert ausgewertet werden. Eine mögliche Datensicht wird nur beim Trainieren und Auswerten des Modells geladen und verarbeitet. Sie können über die *Preview*-Methode beim Schreiben und Testen Ihrer Modell-Anwendung in Visual Studio in der Debugging-Phase einen Blick auf das *IDataView*-Objekt werfen, um dieses zu beobachten und den Inhalt zu prüfen.

```
var debug = testData.Preview();
```

Listing 7.4 zeigt noch einmal den gesamten Beispielcode für die Erstellung eines ML-Modells mit ML.NET.

Listing 7.4 Der vollständige Beispielcode

```
using Microsoft.ML;  
using Microsoft.ML.Data;  
using System;  
  
namespace ArtikeldatenApp  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            MLContext mlContext = new MLContext();
```

```
IDataView trainData = mlContext
    .Data.LoadFromTextFile<ModelInput>(
    "Artikeldaten.csv", separatorChar: ';',
    hasHeader: true);

var dataProcessPipeline =
    mlContext.Transforms.Categorical
        .OneHotEncoding(outputColumnName:
        "ArtikelNrEncoded", "ArtikelNr")
        .Append(mlContext.Transforms.Concatenate(
            outputColumnName: "Features", "Lagerort",
            "MengeIn", "MengeOut"));

var trainer = mlContext
    .MulticlassClassification.Trainers.LbfgsMaximumEntropy(labelColumnName:
    "GesamtArtikelbestand", featureColumnName: "Features");

var trainingPipeline = dataProcessPipeline.Append(trainer);
var model = trainingPipeline.Fit(trainData);

IDataView testData = mlContext.Data.
LoadFromTextFile<ModelInput>(
    "Artikeldaten.csv", separatorChar: ';',
    hasHeader: true);

IDataView predictions = model.Transform(testData);
var metrics = mlContext.MulticlassClassification.Evaluate
    (predictions, "GesamtArtikelbestand");

Console.WriteLine($"Model Ergebnis Marco Accuracy vom "
    + $"{(GesamtArtikelbestand) : {metrics.MacroAccuracy}}");
Console.ReadLine();

}

}

public class ModelInput
{
    [LoadColumn(0)]
    public string ArtikelNr;
    [LoadColumn(1)]
    public float Lagerort;
    [LoadColumn(2)]
    public float MengeIn;
    [LoadColumn(3)]
    public float MengeOut;
    [LoadColumn(4)]
    public string Artikeltyp;
    [LoadColumn(5)]
    public float GesamtArtikelbestand;
}
```



Aktuelle ML-Algorithmen

Nicht jeder verfügbare neue oder sehr aktuelle ML-Algorithmus ist automatisch in ML.NET von Anfang an im Framework enthalten. Sie können aber in vielen Fällen den benötigten ML-Algorithmus über ein NuGet-Paket bequem nachinstallieren.

7.6.8 Modellbereitstellung

Obwohl Sie Ihr Modell in derselben Anwendung trainieren und auch konsumieren können, hat es sich in der Praxis als sinnvoll erwiesen, Modelltrainings- und Anwendungscode in zwei getrennte Anwendungen aufzuteilen. Sie können auch hier, genauso wie schon mit dem Model Builder gezeigt, Ihr trainiertes Modell in eine serialisierte ZIP-Datei speichern und zu einer entsprechenden Endanwendung hinzufügen. Das Speichern erfolgt ganz einfach mit:

```
mlContext.Model.Save(model, trainingData.Schema, "model.zip");
```

Danach können Sie das trainierte Modell Ihrer Anwendung hinzufügen. Die Standardimplementierung wurde ja schon beim Model Builder in Abschnitt 7.5.8 vorgestellt.

Die Codestruktur für die Entwicklung eines ML-Modells in ML.NET sieht fast immer gleich aus. Der iterative Prozess ist wie folgt aufgebaut:

- Sammeln und Laden von Trainingsdaten in ein *IDataView*-Objekt
- Anlegen einer Pipeline zum Erstellen von Features
- Anwenden eines ML-Algorithmus
- Trainieren des Modells durch Aufrufen der *Fit*-Methode auf der Pipeline
- Auswerten des Modells und Iterieren zur Verbesserung
- Speichern des Modells im Binärformat
- Rückladen des Modells in ein *ITransform*-Objekt
- Treffen von Vorhersagen durch Aufrufen von *CreatePredictionEngine.Predict*

So einfach kann Machine Learning mit Visual Studio, ML.NET und C# sein.

7.6.9 TensorFlow, ONNX und ML.NET

Wie Sie an den beiden Beispielen gesehen haben, können Sie mit ML.NET problemlos ein erstelltes ML-Modell exportieren und in einer anderen .NET Anwendung verwenden. Da TensorFlow heute der eigentliche De-facto-Standard für die Erstellung von Machine-Learning-Modellen ist, unterstützt ML.NET die Verwendung von TensorFlow-Modellen für Bildklassifikationsszenarien und Texteingaben (Natural Language Processing).

ML.NET bietet hierfür das NuGet-Paket *Microsoft.ML.TensorFlow* an und nutzt auch hier das Pipeline-Prinzip zum Verarbeiten eines vorab trainierten TensorFlow-Modells und der ML.NET-Bildklassifizierungs-API. Ein entsprechend ausführliches Beispiel und den entsprechenden Code für C# finden Sie direkt unter Microsoft ML.NET [38].

ONNX (Open Neural Network Exchange)

ONNX ist ein offenes Format, das zur Darstellung von Modellen für das Machine Learning entwickelt wurde. ONNX definiert einen gemeinsamen Satz von Operatoren und ein gemeinsames Dateiformat, damit ML-Modelle mit einer Vielzahl von Frameworks und Compilern verwendet werden können.

Das heißt, durch ONNX ist es möglich, ein ML-Modell, das zum Beispiel mit scikit-learn, MxNet oder PyTorch entwickelt und trainiert wurde, zu exportieren und mit ML.NET wieder zu laden, um es in einer .NET-Anwendung bereitzustellen.

In ML.NET findet man ONNX im NuGet-Paket *Microsoft.ML.Onnx.Transformer*. Ein Beispiel steht unter [39] zur Verfügung. Die detaillierte Spezifikation mit allem, was dazugehört, kann im GitHub-Repository von ONNX nachgelesen werden [40].

8

SciSharp Stack

Inzwischen gibt es, neben den schon vorgestellten, auch im .NET-Umfeld weitere ML-Frameworks die sich für den alltäglichen Gebrauch als sehr nützlich erwiesen haben. Besonders interessant ist das Open Source Framework TensorFlow.NET.

Hierbei handelt es sich um ein .NET-basiertes Open-Source-Ökosystem, das es sich zum Ziel gesetzt hat, alle wichtigen ML-Frameworks, die von der Programmiersprache Python unterstützt werden, auch mit .NET und C# zu unterstützen. Das Besondere an SciSharp ist, dass es Portierungen und Bindungen zu den ML-Frameworks TensorFlow, Keras, PyTorch und NumPy in .NET Core zur Verfügung stellt und die APIs der portierten Bibliotheken den Originalen so ähnlich sind, dass Sie alle vorhandenen Ressourcen, Dokumentationen und Community-Lösungen direkt in C# umsetzen können.

So ist es zum Beispiel in TensorFlow möglich, ein ML-Modell zu erstellen, dann zu trainieren und in C# bereitzustellen, und nicht nur wie unter ML.NET ein fertiges TensorFlow-Modell zu importieren und TensorFlow als Backup zu nutzen.

The screenshot shows two side-by-side code editors. The left editor is for 'TensorFlow Python' and the right is for 'TensorFlow.NET'. Both are displaying the same linear regression code, illustrating the similarity between the two frameworks.

```

TensorFlow Python code (Left):


```

Training Data
train_X = numpy.asarray([3.3,4,4.5,5.5,6.71,6.93,4.168,9.779,6.182,7.59,2.167,
7.042,10.791,5.313,7.997,5.654,9.27,3.1])
train_Y = numpy.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.596,2.53,1.221,
2.827,3.465,1.65,2.984,2.42,2.94,1.3])

n_samples = train_X.shape[0]

tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")
```



# Set model weights  
# We can set a fixed init value in order to debug  
var rnd1 = rng.randn<float>();  
var rnd2 = rng.randn<float>();  
W = tf.Variable(-0.06, name="weight");  
b = tf.Variable(-0.73, name="bias")



# Construct a linear model  
pred = tf.add(tf.multiply(X, W), b)



# Mean squared error  
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)



# Gradient descent  
# Note, minimize() knows to modify W and b because Variable objects are trainable=T  
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)



# Initialize the variables (i.e. assign their default value)  
init = tf.global_variables_initializer()



# Start training  
with tf.Session() as sess:  
    # Run the initializer  
    sess.run(init)



# Fit all training data


```

```

TensorFlow.NET code (Right):


```

// Training Data
var train_X = np.array(3.3f, 4.4f, 5.5f, 6.71f, 6.93f, 4.168f, 9.779f, 6.182f,
7.042f, 10.791f, 5.313f, 7.997f, 5.654f, 9.27f, 3.1f);
var train_Y = np.array(1.7f, 2.76f, 2.09f, 3.19f, 1.694f, 1.573f, 3.366f, 2.596f, 2.53f, 1.221f,
2.827f, 3.465f, 1.65f, 2.984f, 2.42f, 2.94f, 1.3f);

var n_samples = train_X.shape[0];

// tf Graph Input
var X = tf.placeholder(tf.float32);
var Y = tf.placeholder(tf.float32);
```



// Set model weights  
// We can set a fixed init value in order to debug  
// var rnd1 = rng.randn<float>();  
// var rnd2 = rng.randn<float>();  
var W = tf.Variable(-0.06f, name: "weight");
var b = tf.Variable(-0.73f, name: "bias");



// Construct a linear model  
var pred = tf.add(tf.multiply(X, W), b);



// Mean squared error  
var cost = tf.reduce_sum(tf.pow(pred - Y, 2.0f)) / (2.0f * n_samples);



// Gradient descent  
// Note, minimize() knows to modify W and b because Variable objects are trainable=T  
var optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost);



// Initialize the variables (i.e. assign their default value)
var init = tf.global_variables_initializer();



// Start training
withSession((tf.Session(), sess =>
{
    // Run the initializer
    sess.run(init);

    // Fit all training data
    for (int epoch = 0; epoch < training_epochs; epoch++)


```

Bild 8.1 Vergleich von TensorFlow mit Python und TensorFlow.NET (Quelle: SciSharp Stack)

Somit bietet TensorFlow.NET aus dem SciSharp Stack [41] ein .NET Standard-Binding, das es ermöglicht, die komplette TensorFlow-API in C# zu implementieren und ein ML-Modell vollständig im .NET-Standard-Framework zu entwickeln. Da es in SciSharp gelungen ist, die APIs so ähnlich wie möglich zu gestalten, können Sie jeden vorhandenen TensorFlow-Code in C# verwenden, ohne sich mit Python beschäftigen zu müssen.

Bild 8.1 zeigt, wie einfach sich ein Python-Skript aus TensorFlow mit TensorFlow.NET in ein C#-Programm übersetzen lässt.

■ 8.1 TensorFlow.NET

Eine der besonders großen Stärken von TensorFlow ist die Modularität, die es erlaubt, ein Machine-Learning-Projekt auf viele unterschiedliche Arten zu erstellen und an individuelle Bedürfnisse anzupassen. Daher wäre es für uns Entwickler toll, wenn diese Vorteile von TensorFlow auch direkt mit C# genutzt werden könnten.

Wie kann man also ohne großen Overhead und Umweg mit C# in TensorFlow einsteigen? Mit TensorFlow.NET bekommt man schnell ein Beispiel mit Visual Studio und C# zum Laufen. Die einfachste Art ist die Implementierung eines klassischen „Hallo TensorFlow!“-Programms, um die Installation und Funktionsweise von TensorFlow.NET kennenzulernen.

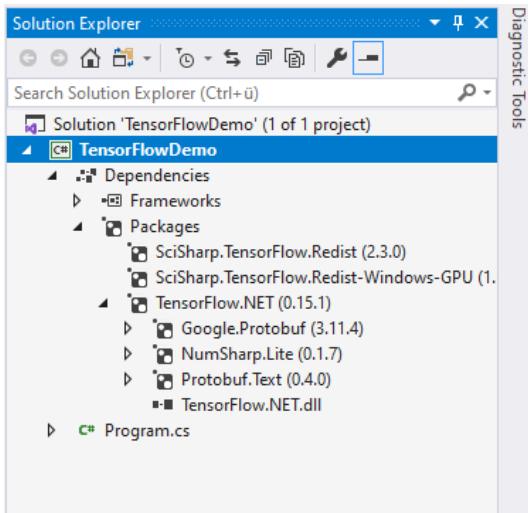
8.1.1 TensorFlow.NET-SDK installieren

Die Installation von TensorFlow.NET setzt auf dem .NET Standard 2.0 auf, sodass Ihr ML-Projekt auf .NET Framework oder auf .NET Core 2.2 basieren kann. Alle nachfolgenden Beispiele verwenden .NET Core 2.2 und Visual Studio Community Edition 2019. Alternativ können Sie auch Visual Studio 2017 mit .NET Core 2.2 verwenden. Die Installation erfolgt über den *NuGet Package Manager* im jeweiligen ML-Projekt in Visual Studio. Erstellen Sie ein neues *Console App (.NET Core)*-Projekt mit dem Namen *TensorFlowDemo* über die entsprechende Vorlage in Visual Studio 2019.

Ist die Console App über die Vorlage in Visual Studio automatisch erstellt worden, wechseln Sie über *Tools | NuGet Package Manager* in die *Package Manager Console*. Führen Sie in der PowerShell die Installation für TensorFlow.NET mit den folgenden drei Paketen durch.

```
PM> Install-Package TensorFlow.NET
PM> Install-Package SciSharp.TensorFlow.Redist
PM> Install-Package SciSharp.TensorFlow.Redist-Windows-GPU
```

Das Paket TensorFlow.NET enthält das benötigte C# Binding. Hinter *SciSharp.TensorFlow.Redist* verbirgt sich das TensorFlow Binary für die CPU-Version und *SciSharp.TensorFlow.Redist-Windows-GPU* erlaubt es, Befehle mit Grafikkartenunterstützung auszuführen, was das Programm erheblich beschleunigen kann. Ist die Installation von TensorFlow.NET erfolgreich abgeschlossen, so finden Sie die entsprechenden Einträge in den Abhängigkeiten (*Dependencies*) in Ihrem Visual-Studio-Projekt (Bild 8.2).

**Bild 8.2**

Hinzufügen von TensorFlow.NET in das ML-Projekt

Um TensorFlow.NET in Aktion zu erleben, reicht das einfache Programm aus Listing 8.1.

Listing 8.1 Das Hallo-TensorFlow-Programm

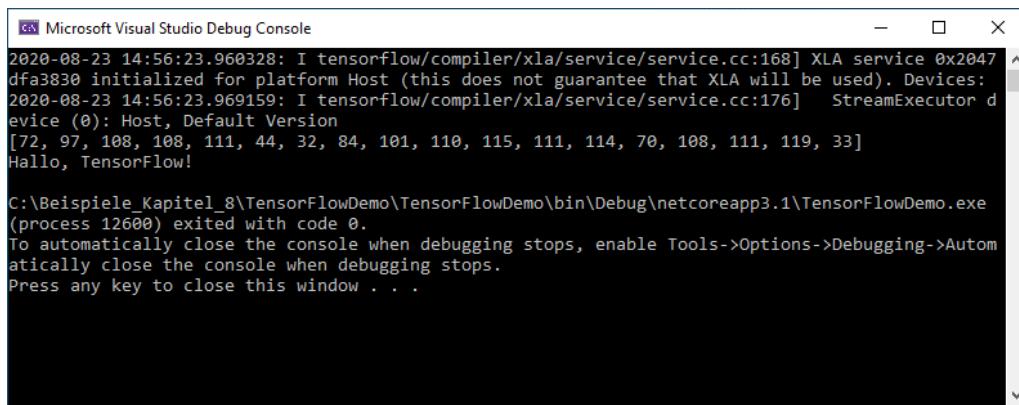
```
using static Tensorflow.Binding;
using System.Text;

namespace TensorFlowDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            ASCIIEncoding ascii = new ASCIIEncoding();
            var hallo = tf.constant("Hallo, TensorFlow!");

            using (var sess = tf.Session())
            {
                var res = sess.run(hallo);
                print(sess.run(hallo));

                string decoded = ascii.GetString(res.ToArray());
                print(decoded);
            }
        }
    }
}
```

Die Konsole (Bild 8.3) gibt „Hallo, TensorFlow!“ als ASCII-Werte und als dekodierten String aus. In allen Codebeispielen von TensorFlow.NET wird TensorFlow über die Abkürzung `tf` importiert. Die Abkürzung hat sich in der TensorFlow Community etabliert und ist so von SciSharp Stack übernommen worden.



The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is as follows:

```
2020-08-23 14:56:23.960328: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x2047
dfa3830 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2020-08-23 14:56:23.969159: I tensorflow/compiler/xla/service/service.cc:176] StreamExecutor d
evice (0): Host, Default Version
[72, 97, 108, 108, 111, 44, 32, 84, 101, 110, 115, 111, 114, 70, 108, 111, 119, 33]
Hallo, TensorFlow!

C:\Beispiele_Kapitel_8\TensorFlowDemo\TensorFlowDemo\bin\Debug\netcoreapp3.1\TensorFlowDemo.exe
(process 12600) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Autom
atically close the console when debugging stops.
Press any key to close this window . . .
```

Bild 8.3 TensorFlow.NET ist erfolgreich installiert.

Wie schon in Abschnitt 7.2.1, „Ablauf in TensorFlow“, beschrieben, benötigt TensorFlow zur Ausführung ein Session-Objekt. Eine Session ist die Ablaufumgebung für die gewünschten Operationen und führt intern einen Tensor aus.

Das heißt, um eine Session in TensorFlow auszuführen, müssen Sie eine oder mehrere Operationen mitgeben. Der Aufruf `sess.run()` führt die übergebene Operation in der Session durch. Die Rückgabewerte erfolgen in der gleichen Reihenfolge wie die übergebenen Operationen.

8.1.2 Tensor

Wie in Abschnitt 7.2, „TensorFlow“, angesprochen, werden die in TensorFlow verwendeten Daten als Tensoren bezeichnet. Bei TensorFlow enthält ein Tensor ein mehrdimensionales Array von Elementen eines einzigen Datentyps, das dem *NDArray* von *NumPy* (siehe Kasten) sehr ähnlich ist.


NDArray

Ein *NDArray* ist in NumPy in der Regel ein mehrdimensionaler Container mit fester Größe. Die Inhalte sind vom selben Typ und derselben Größe. Die Anzahl der Dimensionen und Elemente im Array wird durch seine Form definiert, die ein Tupel von N nicht-negativen ganzen Zahlen ist.

Wenn die Dimension im Array null ist, wird sie als Skalar bezeichnet, ist die Dimension 2, so handelt es sich um eine Matrix und ist die Dimension größer als 2 spricht man von einem Tensor. Es gibt viele Möglichkeiten, ein Tensor-Objekt in TensorFlow.NET zu initialisieren. Das Tensor-Objekt kann von einem Skalar, einem String (Zeichenkette), einer Matrix oder einem Tensor aus initialisiert werden.

Listing 8.2 Tensor

```

using Tensorflow;
using static Tensorflow.Binding;
using NumSharp;

namespace TensorTF
{
    class Program
    {
        static void Main(string[] args)
        {
            Tensor tensor1 = tf.constant(3);
            Tensor tensor2 = tf.constant("Hallo TensorFlow");

            var ndArray = new NDArray(new int[] { 3, 2, 2, 4 });
            Tensor tensor3 = tf.constant(ndArray);

            using (var sess = tf.Session())
            {
                print(sess.run(tensor1));
                print(sess.run(tensor2));
                print(sess.run(tensor3));
            }
        }
    }
}

```

Das Beispiel erstellt einen Tensor mit einem skalaren Wert. Des Weiteren erfolgt eine Initialisierung mit einer Zeichenkette und der dritte Tensor enthält ein *NDArray*, das über die Bibliothek *NumSharp* importiert wird. Die Methode *print* gibt dann die entsprechenden Tensoren auf der Konsole aus.

**NumPy**

Bei NumPy handelt es sich um eine Python-Bibliothek für Machine-Learning-Algorithmen, die mit komplexen multidimensionalen Arrays sowie Matrizen arbeitet und höhere mathematische Funktionen anwenden kann. SciSharp Stack stellt hierfür das Paket *NumSharp.Lite* zur Verfügung.

8.1.3 Platzhalter

Der sogenannte Platzhalter (*tf.placeholder*) von TensorFlow, der als allgemeiner Platzhalter für Daten dient, ist eine einfache Variable, die es ermöglicht, Daten aus externen Quellen in den Berechnungsgraph in TensorFlow zu integrieren. Für *Platzhalter* können ein *Typ (dtype)*, ein *Name (name)* und die *Form (shape)* angegeben werden. Es ist bei der Modellierung eines Berechnungsgraphen aber nicht zwingend notwendig, die Daten am Anfang einzuspeisen. Wie der Name der Variable schon sagt, setzt TensorFlow einfach eine leere Größe ein, die in der zweiten Phase gefüllt werden kann.

Der Einsatz eines leeren Platzhalters ermöglicht es Ihnen in TensorFlow mit variablen Batch-Größen umzugehen. TensorFlow bietet eine einfache Funktion zur Erzeugung eines Platzhalters. In TensorFlow.NET können Sie den gleichen Methodennamen *tf.placeholder* verwenden, um einen Platzhalter zu erstellen. Listing 8.3 zeigt die Implementierung einer Platzhalter-Variablen. Über *print(result)* wird das Ergebnis, also der Wert der Variable, ausgegeben.

Listing 8.3 Placeholder

```
using Tensorflow;
using static Tensorflow.Binding;

namespace Placeholder
{
    class Program
    {
        static void Main(string[] args)
        {
            var placeholder = tf.placeholder(tf.float32);
            var y = placeholder * 10;

            using (var sess = tf.Session())
            {
                var result = sess.run(y, feed_dict: new FeedItem[]
                {
                    new FeedItem(placeholder,2)
                });

                print(result);
            }
        }
    }
}
```

In dem Beispiel kann man erkennen, dass bei einem Platzhalter immer dessen Typ angegeben werden muss. Auf die Form kann man je nach Bedarfsfall bzw. Einsatzzweck verzichten.

8.1.4 Variable

Die Variable wird in TensorFlow verwendet, um variable Parameterwerte im Machine-Learning-Modell darzustellen.

Variablen können durch die Methode *tf.variable* initialisiert werden. Während der Berechnung des Graphen in TensorFlow werden die Variablen durch andere Operationen modifiziert. Die Variable muss hierbei über die Konfiguration zur Laufzeit initialisiert und mit einem Startwert versehen werden. Es kann in vielen Fällen ein fester Wert sein, oder Sie setzen über die Methode *tf.random_uniform* einen zufälligen Wert. Die Methode erstellt über eine zufällige Verteilung eine Matrix der angegebenen Variablengröße. Listing 8.4 zeigt die Verwendung der *tf.variable* in TensorFlow.NET.

Listing 8.4 Variable

```

using Tensorflow;
using static Tensorflow.Binding;

namespace VariableTF
{
    class Program
    {
        static void Main(string[] args)
        {
            var x_fixed = tf.Variable(5, name: "fixed");
            var random = tf.Variable(tf.random_uniform(1, 0, 10,
                TF_DataType.TF_FLOAT, null, "random"));
            using (var sess = tf.Session())
            {
                sess.run(tf.global_variables_initializer());
                print(sess.run(x_fixed));
                print(sess.run(random));
            }
        }
    }
}

```

Der Beispielcode erstellt zwei variable Operationen. Als Erstes muss bei TensorFlow in eine laufende Session die benötigte Variable für den Berechnungsgraphen initialisieren werden. Das geschieht mit der Methode *tf.global_variables_initializer()*. Hierdurch sind die Variablen dem Graphen bekannt und weitere Operationen können sie verwenden. Die Variablen behalten ihre Werte während der ganzen Session. Listing 8.4 ist sehr einfach, zeigt aber gut den internen Prozess, wie TensorFlow mit Variablen umgeht.

8.1.5 Konstante

In TensorFlow ist eine Konstante ein spezieller Tensor, der bei laufendem Berechnungsgraph nicht verändert werden kann. Die Konstante wird mit *tf.constant* implementiert, eine Größe ist nicht zwingend erforderlich, diese ergibt sich wieder aus den Eingabedaten. Die Konstante verfügt über folgende Datenmerkmale:

- **Value:** Skalar-Wert oder Konstanten-Liste.
- **Dtype:** Datentyp.
- **Shape:** Dimension (Abmessung/Größe).
- **Name:** Name der Konstante.

Das nachfolgende Beispiel erzeugt eine Konstante mit einem Skalar-Wert, eine Konstante mit Zeichenkette, eine mit NDArray und jeweils eine Matrix_1 und eine Matrix_2. TensorFlow.NET verwendet hier den gleichen Namen und die gleiche Benennungsmethode wie die Python-Bindung an die TensorFlow API.

Listing 8.5 Konstante

```

using System;
using Tensorflow;
using static Tensorflow.Binding;
using NumSharp;

namespace ConstantTF
{
    class Program
    {
        static void Main(string[] args)
        {
            var constant_1 = tf.constant(3);
            var constant_2 = tf.constant("Hallo TensorFlow");

            var ndArray = np.array(new int[] []
            {
                new int[]{3,6,9}, new int[]{7,8,12}
            });

            var constantArray = tf.constant(ndArray);

            var matrix_1 = tf.constant(new float[,] { { 1, 2 }, { 3, 4 } });
            var matrix_2 = tf.constant(new float[,] { { 5, 6 }, { 7, 8 } });

            var product = tf.matmul(matrix_1, matrix_2);

            using (var sess = tf.Session())
            {
                var result = sess.run(product);
                print(result);
            }
        }
    }
}

```

Der Unterschied zwischen Platzhaltern und Konstanten besteht bei TensorFlow darin, dass Platzhalter Koeffizienten-Werte flexibler angeben können, ohne den Code, der den Berechnungsgraphen aufbaut, zu ändern. Das heißt, auch bei TensorFlow.NET sollten Sie für mathematische Konstanten die Variable *tf.constant* einsetzen.

8.1.6 Berechnungsgraph

Wie schon erläutert, verwendet TensorFlow einen Datenflussgraphen, um Berechnungen in Bezug auf die Abhängigkeiten zwischen einzelnen Operationen darzustellen. Somit definiert der Graph die Berechnung. Wichtig hierbei ist: Der Graph berechnet nichts, er enthält keine Werte, sondern er definiert nur die Operationen, die Sie in Ihrem Code angegeben haben.

Das nachfolgende Beispiel definiert in TensorFlow.NET einen Graphen mit einer Variablen und drei Operationen.

Listing 8.6 Graph

```
with<Graph>(tf.Graph().as_default(), graph =>
{
    var variable = tf.Variable(55, name: "tree");
    tf.global_variables_initializer();
    variable.assign(17);
});
```

Der Ausdruck `var variable` gibt den aktuellen Wert der `tf.variable` zurück. Über `tf.global_variables_initializer` weisen Sie dieser Variablen den Anfangswert 55 zu. Über die Methode `assign` erhält die Variable dann in der Session den neuen Wert 17. TensorFlow.NET simuliert hier die Verwendung des Lebenszyklus des Berechnungsgraphen, der entsorgt wird, wenn die Graph-Instanz nicht mehr benötigt wird.

Auch die Ausführung des Graphen erfolgt dann in einer entsprechenden Session. Die Session wird immer mit `if.Session().run()` gestartet und am Ende mit `if.Session().Close()` geschlossen. Im Beispielcode übernimmt die `Using`-Anweisung den ordnungsgemäßen Einsatz von `IDisposable`-Objekten und stellt das Schließen der TensorFlow.NET Session sicher.

Über die Methode `run` ermöglicht TensorFlow.NET die Ausführung der gewünschten Anweisung. Beachten Sie, dass die Werte Ihrer Variablen nur innerhalb einer Session gültig sind. TensorFlow.NET bietet noch weitere zahlreiche Operationen und Methoden auf Tensoren an, so zum Beispiel auch Matrix-basierte Operationen wie die Multiplikation von zwei Matrizen. Mehr zu den Operationen und Methoden finden Sie in der Dokumentation *The Definitive Guide to TensorFlow.NET* unter [42].

8.1.7 Lineare Regression

An den oben aufgeführten Beispielen können Sie sehen, dass es mit TensorFlow.NET und NumSharp sehr einfach möglich ist, ein Python-Codebeispiel zu nehmen, und es für C# mit geringfügigen Änderungen zum Laufen zu bringen. Somit steht Ihnen als .NET-Entwickler quasi das gesamte TensorFlow-Ökosystem für Machine Learning und Deep Learning offen.

Die lineare Regression ist vielleicht einer der bekanntesten Algorithmen in der Statistik und im Machine Learning, und die Umsetzung in Programmcode ist einfach und übersichtlich. Das nachfolgende Beispiel nutzt TensorFlow.NET, um eine einfache lineare Regression zu erstellen, die eine Regressionslinie an einigen Beispieldaten anpasst.

Das Beispiel zeigt eine lineare Gleichung, die einen bestimmten Satz von Eingabewerten (`dataX`) kombiniert, deren Lösung die vorhergesagte Ausgabe für diesen Satz von Eingabewerten (`dataY`) ist. Als solche sind sowohl die Eingabedaten (`dataX`) als auch die Ausgabedaten numerisch.

Gradientenabstiegsverfahren

Da es in dem Beispiel mehrere Eingabedaten gibt, können Sie den Prozess zur Optimierung der Koeffizienten-Werte verwenden, indem Sie die Fehler des Modells auf Ihren Trainingsdaten iterativ minimieren.

Sie haben diese Operation, Gradient Descent, schon in Kapitel 3 kennengelernt. Hierbei wird die Summe der quadrierten Fehler für jedes Paar von Eingangs- und Ausgangswerten berechnet. Eine Lernrate wird als Skalierungsfaktor verwendet, und die Koeffizienten werden in der Richtung zur Minimierung des Fehlers aktualisiert. Der Vorgang wird so lange wiederholt, bis ein minimaler quadratischer Summenfehler erreicht ist oder keine weitere Verbesserung mehr möglich ist.

8.1.8 Von der Theorie zum Code

Für die Umsetzung der Theorie in Code bietet sich wieder die *.NET Core Console App*-Vorlage von Visual Studio an. Hinzu kommen die benötigten TensorFlow.NET-Pakete und die NumSharp-Bibliothek, die bei der Matrizen- und Vektor-Rechnung hilft.

Erstellen Sie daher ein neues .NET Core-Projekt in Visual Studio 2019 mit dem Namen *LinearRegressionDemo*. Listing 8.7 zeigt den passenden C#-Code, der sich mit den einfachen Werten aus der *np.array*-Methode testen lässt.

Listing 8.7 LinearRegressionDemo

```
using System;
using NumSharp;
using Tensorflow;
using static Tensorflow.Binding;

namespace LinearRegressionDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var dataX = np.array(
                3.3f, 4.4f, 5.5f, 6.93f, 4.168f, 9.779f, 6.182f, 7.59f,
                2.167f, 7.042f, 10.791f, 5.313f, 7.997f, 5.654f, 9.27f, 3.1f);
            var dataY = np.array(
                1.7f, 2.76f, 2.09f, 3.19f, 1.694f, 1.573f, 3.366f, 2.596f, 2.53f,
                1.221f, 2.827f, 3.465f, 1.65f, 2.904f, 2.42f, 2.94f, 1.3f);
            var samples = dataX.shape[0];

            var X = tf.placeholder(tf.float32);
            var Y = tf.placeholder(tf.float32);

            var W = tf.Variable(0.0f, name: "weight");
            var b = tf.Variable(0.0f, name: "bias");

            var model = tf.add(tf.multiply(X, W), b);
            var loss = tf.reduce_sum(tf.pow(model - Y, 2.0f)) / (2.0f * samples);

            var epochs = 1000;
            var learningRate = 0.01f;
            var displayEvery = 50;
```

```
var optimizer = tf.train.GradientDescentOptimizer(learningRate).  
    minimize(loss);  
var init = tf.global_variables_initializer();  
  
using (var sess = tf.Session())  
{  
    sess.run(init);  
  
    Console.WriteLine("Training model...");  
    for (int epoch = 0; epoch < epochs; epoch++)  
    {  
        foreach (var (x, y) in zip<float>(dataX, dataY))  
        {  
            sess.run(optimizer,  
                new FeedItem(X, x),  
                new FeedItem(Y, y));  
        }  
  
        if ((epoch + 1) % displayEvery == 0)  
        {  
            var lossValue = sess.run(loss, new FeedItem(X, dataX),  
                new FeedItem(Y, dataY));  
            Console.WriteLine($" epoch: {epoch + 1}\tMSE =  
                {lossValue}\tW = {sess.run(W)}\t b = {sess.run(b)}");  
        }  
    }  
  
    var trainingLoss = sess.run(loss,  
        new FeedItem(X, dataX),  
        new FeedItem(Y, dataY));  
  
    Console.WriteLine($" Final MSE = {trainingLoss}");  
}  
  
}  
}  
}
```

Der Code wirkt nicht besonders spektakulär und lässt sich dadurch sehr gut nachvollziehen. Als Erstes werden die Trainingsdaten über die *np.array*-Methode eingerichtet. Der Aufruf *dataX.shape[0]* ruft die Länge des Arrays ab, um später die entsprechende Verlustfunktion berechnen zu können.

Dann folgt der Code mit den Platzhaltern, um die Modelleingaben- und -ausgaben einzurichten. Die Variablen *w* und *b* stellen die Gewichte- und Bias-Variablen dar. Diese Variablen werden wie folgt zu einem Modell kombiniert:

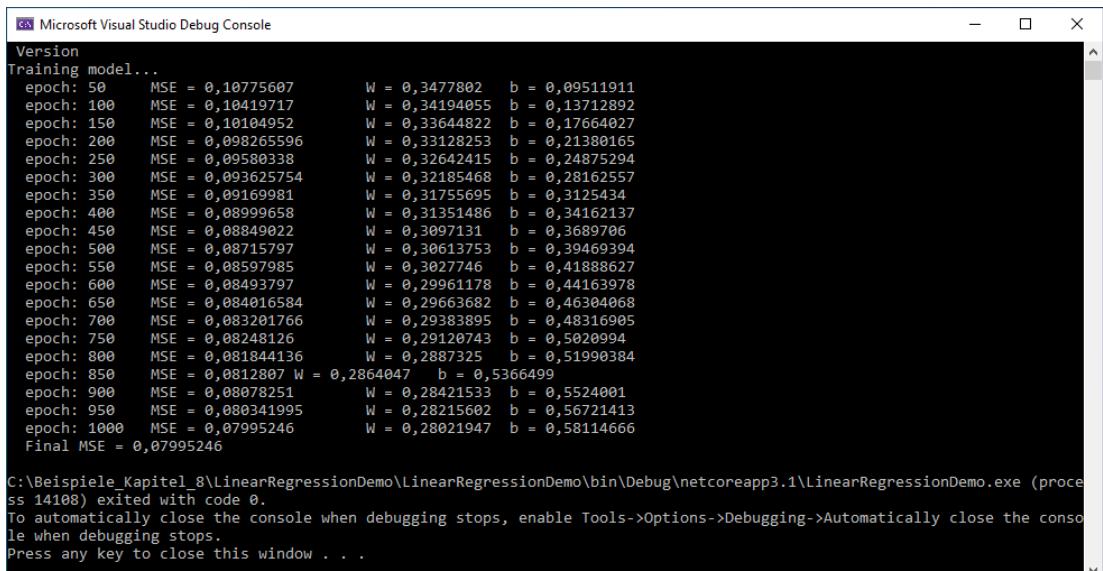
$$Y = wx + b$$

Das erreichen Sie einfach über die TensorFlow-Methoden *.add* und *.multiply*. Sie stellen die einfache Gleichung für die lineare Regression dar. Über die Variable *loss* wird der mittlere quadratische Fehler (Mean Square Error, MSE) errechnet. Die allgemeine Formel dazu lautet:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Es ergibt sich hieraus die Summe des Quadrats der Differenz zwischen den Modellvorhersagen und den tatsächlichen Werten. Des Weiteren werden noch die Epoche und die Lernrate festgelegt.

In der Session wird dann das *Init* ausgeführt, um das Modell über *global_variables_initializer* zu initialisieren. Der Optimierer durchläuft dann 1.000 Epochen, um das Modell zu trainieren, und alle 50 Epochen wird die Verlustfunktion berechnet und der mittlere Trainingsverlust über die Konsole ausgegeben (Bild 8.4).



```
Microsoft Visual Studio Debug Console
Version
Training model...
epoch: 50      MSE = 0,10775607      W = 0,3477802    b = 0,09511911
epoch: 100     MSE = 0,10419717      W = 0,34194055   b = 0,13712892
epoch: 150     MSE = 0,10104952      W = 0,33644822   b = 0,17664027
epoch: 200     MSE = 0,098265596    W = 0,33128253   b = 0,21380165
epoch: 250     MSE = 0,09580338    W = 0,32642415   b = 0,24875294
epoch: 300     MSE = 0,093625754    W = 0,32185468   b = 0,28162557
epoch: 350     MSE = 0,09169981    W = 0,31755695   b = 0,3125434
epoch: 400     MSE = 0,08999658    W = 0,31351486   b = 0,34162137
epoch: 450     MSE = 0,08849022    W = 0,3097131    b = 0,3689706
epoch: 500     MSE = 0,08715797    W = 0,30613753   b = 0,39469394
epoch: 550     MSE = 0,08597985    W = 0,30277446   b = 0,41888627
epoch: 600     MSE = 0,08493797    W = 0,29961178   b = 0,44163978
epoch: 650     MSE = 0,084016584   W = 0,29663682   b = 0,46304068
epoch: 700     MSE = 0,083201766   W = 0,29383895   b = 0,48316905
epoch: 750     MSE = 0,08248126    W = 0,29120743   b = 0,5020994
epoch: 800     MSE = 0,081844136   W = 0,2887325   b = 0,51990384
epoch: 850     MSE = 0,0812807 W = 0,2864047   b = 0,5366499
epoch: 900     MSE = 0,08078251    W = 0,28421533   b = 0,5524001
epoch: 950     MSE = 0,080341995   W = 0,28215602   b = 0,56721413
epoch: 1000    MSE = 0,07995246   W = 0,28021947   b = 0,58114666
Final MSE = 0,07995246

C:\Beispiele_Kapitel_8\LinearRegressionDemo\LinearRegressionDemo\bin\Debug\netcoreapp3.1\LinearRegressionDemo.exe (process 14108) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Bild 8.4 Die Ausgabe der linearen Regression

Fertig ist ein vollständig trainiertes Modell, das Sie jetzt auch mit eigenen neuen Daten für *dataX* und *dataY* testen können. Legen Sie hierfür einfach neue Testdaten als *testDataX* und *testDataY* an und erweitern Sie den Code bzw. ändern Sie ihn entsprechend um.

Wie Sie an diesem Beispiel sehen, ermöglicht es TensorFlow.NET mit wenigen Zeilen Code ein einfaches Modell zu erstellen, zu trainieren und zu evaluieren. Weitere Beispiele für TensorFlow.NET finden Sie unter GitHub [43].

■ 8.2 Keras.NET

Im vorherigen Kapitel wurde Keras als eine High-Level-API für neuronale Netzwerke, die in Python geschrieben wurde und auf TensorFlow ausgeführt wird, vorgestellt.

Das Keras.NET Framework stellt diese High-Level-API für .NET-Entwickler mit entsprechendem Python-Binding zur Verfügung. Somit kann Keras.NET als einheitliche Schnittstelle zur Modelldefinition für TensorFlow genutzt werden.

Auch Keras.NET bietet konsistente und einfache APIs. Es minimiert die Anzahl der Benutzeraktionen, die häufig für die Erstellung von ML-Modellen erforderlich sind, und bietet klare und umsetzbare Rückmeldungen bei Fehlern.

Keras.NET bietet vorgefertigte Funktionen, um zum Beispiel ein Modell zu trainieren, ein Dataset zu laden und mit möglichst wenig Aufwand Schichten und Aktivierungsfunktionen zu definieren. Hierbei spricht man bei Keras.NET von der Modularität. Keras.NET versteht unter einem Modell eine Abfrage oder einen Graphen von eigenständigen, vollständig konfigurierbaren Modulen, die mit möglichst wenig Einschränkungen zusammengesteckt werden können. Zu den eigenständigen Modulen zählen neuronale Schichten, Kostenfunktionen, Optimierer, Initialisierungsschemata, Aktivierungsfunktionen und Regularisierungsschemata. Die Module können frei kombiniert werden und es besteht die Möglichkeit einfach neue nützliche Module zu erstellen.

8.2.1 Keras.NET installieren

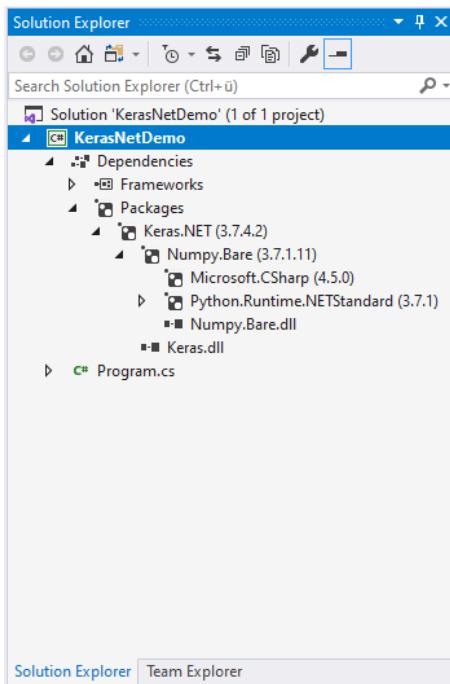
Keras.NET läuft nahtlos auf CPU und GPU und verwendet intern für Berechnungen die Bibliothek NumPy.NET und pythonnet_netstandard. Pythonnet stellt Python für den .NET-Standard als *Python.Runtime.dll* für die jeweilige genutzte Python-Version und das Betriebssystem zur Verfügung.

Bevor Sie Keras.NET in einem Visual-Studio-Projekt verwenden, benötigen Sie ein aktuelles Python-Paket [44] und die Backend-Engine TensorFlow [45]. Keras.NET selbst wird dann einfach über den NuGet Package Manager in Visual Studio über den Befehl

```
PM> Install-Package Keras.NET
```

installiert. Ist die Installation erfolgreich abgeschlossen, verfügt Ihr Projekt jetzt über die neuen Abhängigkeiten von Keras.NET (Bild 8.5).

Nach der Installation muss man aber auf das korrekte Zusammenspiel von Keras.NET und der GPU auf Windows achten. Es kann je nach Grafikkarte notwendig sein, eine Konfiguration durchzuführen, um von der GPU den benötigten dynamischen Speicher zugewiesen zu bekommen.

**Bild 8.5**

Erfolgreiche Installation von Keras.NET

8.2.2 Modelle erstellen

Die Kerndatenstruktur von Keras.NET besteht darin, ein Modell mit Ebenen zu organisieren. Durch diese Ebenen teilt Keras.NET die Implementierung von neuronalen Netzen in mehrere Objekte auf. Dabei werden einzelne Layer (Schichten) als Objektinstanzen zu einem Modell hinzugefügt. Keras.NET gibt bei der Entwicklung zwei Modelle vor:

- Das sequenzielle Modell bildet alle Layer in einer Sequenz ab. Dazu wird das Modell instanziert und die einzelnen Layer werden der Reihe nach dem Modell hinzugefügt. Die Reihenfolge entscheidet, welche Layer ineinander übergehen.
- Das funktionale Modell verknüpft einzelne Layer direkt. Daher muss vor einer Layer-Definition der vorherige Layer bekannt und instanziert sein. Der vorausgegangene Layer wird bei der Instanzierung des aktuellen Layers übergeben. So lassen sich komplexere Abhängigkeiten ausdrücken.

Sind alle Layer dem Modell hinzugefügt, wird es über die Methode *Compile* für den Trainingsprozess vorbereitet und optimiert.

Im folgenden Beispiel wird das XOR-Problem aus Abschnitt 3.4.1, „Multilayer Perceptron“, mit Keras.NET auf einfache Art und Weise gelöst.

Listing 8.8 Lösung des XOR-Problems

```

NDarray x = np.array(new float[,] { { 0, 0 }, { 0, 1 }, { 1, 0 }, { 1, 1 } });
NDarray y = np.array(new float[] { 0, 1, 1, 0 });

var model = new Sequential();
model.Add(new Dense(32, activation: "relu", input_shape: new Shape(2)));
model.Add(new Dense(64, activation: "relu"));
model.Add(new Dense(1, activation: "sigmoid"));

model.Compile(optimizer:"sgd", loss:"binary_crossentropy",
    metrics: new string[] { "accuracy" });
model.Fit(x, y, batch_size: 2, epochs: 1000, verbose: 1);

string json = modelToJson();
File.WriteAllText("model.json", json);
model.SaveWeight("model.h5");

var loaded_model = Sequential.ModelFromJson(File.ReadAllText("model.json"));
loaded_model.LoadWeight("model.h5");

```

Als Erstes werden die Trainingsdaten in einem *NDarray* angelegt. Dann wird das sequenzielle Modell erstellt. Es besteht aus drei *Dense*-Schichten mit jeweils 32 Neuronen, 64 Neuronen und einem Neuron. Eine *Dense*-Schicht stellt eine vollständig verbundene Schicht dar. Dann wird über die Methoden *Compile* und *Fit* das Modell trainiert.

Auch bei Keras.NET können verschiedene Optimizer für das Training des Modells eingesetzt werden. Der Optimizer wird einfach durch seinen Namen, im Beispiel *sgd* für *Stochastic Gradient Descent Optimizer*, festgelegt. Keras.NET bietet weitere Optimizer wie zum Beispiel Adadelta, Adamax oder auch RMSprop an.

Über die Methode *WriteAllText* des File-Objekts wird das Modell gespeichert und mit *model.SaveWeight* werden die Gewichte gesichert. Mit *Sequential.ModelFromJson* und *LoadWeight* kann das Modell wieder geladen und mit den gespeicherten Gewichten ausgeführt werden. Beachten Sie aber, dass Keras.NET sich auf TensorFlow als Backend bezieht. Das heißt, Sie brauchen die Tensoren und den Berechnungsgraphen nicht mehr direkt im Code instanziiieren. Diese Aufgaben erledigt Keras.NET automatisch im Hintergrund. Auch aus diesem Grund eignet sich Keras.NET in Bezug auf ML-Algorithmen und neuronale Netzwerke sehr gut als Tool zur Prototypenentwicklung. Weitere Informationen über die Möglichkeiten von Keras finden Sie direkt auf der Webseite von Keras [46] und für Keras.NET unter dem SciSharp Stack auf GitHub [47].

■ 8.3 NeuralNetwork.NET

Zu den erfreulichen Open Source Tools im Bereich neuronaler Netze gehört auch die NeuralNetwork.NET-Bibliothek. Bei NeuralNetwork.NET handelt es sich um eine .NET-Standard-2.0-Bibliothek, die es C#-Entwicklern erlaubt, sequenzielle Modelle und Berechnungsgraphen mit anpassbaren Schichten (Layern) zu implementieren.

NeuralNetwork.NET bietet einfache APIs für ein schnelles Prototyping, zur Definition von Modellen und zum Trainieren von Modellen. Auch Methoden zum Speichern und Laden eines Modells und seiner Metadaten sind vorhanden.

Für fortgeschrittenere Funktionen kann auch der Grafikprozessor und das *cuDNN*-Toolkit genutzt werden, um die Leistung beim Training oder bei der Verwendung eines neuronalen Netzes erheblich zu steigern.



cuDNN-Toolkit

Das cuDNN- oder auch CUDA-Toolkit (CUDA steht für Compute Unified Device Architecture) bietet eine Entwicklungsumgebung zur Erstellung hochleistungsfähiger GPU-beschleunigter Anwendungen.

Das Toolkit enthält Bibliotheken und Optimierungswerkzeuge zur hardwarenahen Bereitstellung Ihrer Anwendung.

Die NeuralNetwork.NET-Bibliothek stellt auch dem Neuling im Bereich neuronaler Netze einfach zu verwendende Klassen und Methoden zur Erstellung eines neuen neuronalen Netzes zur Verfügung. Aber auch komplexe Netzstrukturen wie zum Beispiel Berechnungsgraphen, die aus TensorFlow bekannt sind, lassen sich abbilden.

Die spezifisch räumliche Struktur eines Berechnungsgraphen erlaubt es, verschiedene Knoten miteinander zu verbinden. So ist es beispielsweise möglich, Daten durch verschiedene parallele Pipelines zu kanalisiieren, die später im Berechnungsgraphen zusammengeführt werden. Eine ausführliche Einführung und passende Beispiele für die NeuralNetwork.NET-Bibliothek finden Sie unter GitHub [48].

Alle bisher vorgestellten Frameworks und Bibliotheken werden stetig weiterentwickelt. Allerdings ist ein direkter Vergleich der Frameworks kaum möglich. Am Ende müssen Sie als Entwickler entscheiden, um welche Art von Projekt es sich handelt, welche Rahmenbedingungen gelten und welches Framework Sie dabei am besten unterstützen kann.

9

Machine Learning as a Service

Die bisher vorgestellten Frameworks konnten immer lokal ausgeführt werden, was je nach Größe des neuronalen Netzes eine entsprechend gute und schnelle Hardware voraussetzt. Eine Alternative, auch in Bezug auf die Rechenleistung, bietet das Machine Learning aus der Cloud.

Machine Learning as a Service, kurz MLaaS, umfasst eine Reihe von Diensten, die fertige Tools zur Entwicklung von wiederverwendbaren ML-Modellen für das maschinelle Lernen bereitstellen. Diese können den jeweiligen Anforderungen entsprechend angepasst werden.

Die größten Cloud-Anbieter, die auch MLaaS zur Verfügung stellen, sind Amazon, Microsoft, Google und IBM (siehe auch Abschnitt 1.3, „KI-Service-Plattformen“). Ein Vorteil der MLaaS-Plattform liegt im Synergieeffekt, da alle Phasen des Machine Learning, einschließlich der Datenspeicherung und -verwaltung, der Modellentwicklung und -bereitstellung, aus einer Hand abgewickelt werden. Somit ist der Projekt- und ML-Prozess entsprechend strukturiert vorgegeben.

Machine Learning als Service kann sowohl für Anfänger als auch für erfahrene ML-Entwickler geeignet sein. Als Neuling profitiert man von der codefreien visuellen Schnittstelle, die meistens im Browser angeboten wird, den vortrainierten Modellen und den vorgefertigten KI-Diensten. Experten können aber auch die codebasierte Umgebung nutzen, um Modelle für ML von Grund auf neu zu entwickeln.

Beachten Sie aber, dass die Eigenschaften und Einsatzmöglichkeiten der jeweiligen Cloud-Plattform unterschiedlich sind. Daher sollten Sie die Lösungen und Dienstleistungen unbedingt vergleichen und prüfen, welche Lösung für Ihr ML-Projekt am besten geeignet ist. Auch die Kostenfrage bzw. Preispolitik der jeweiligen Plattform spielt eine Rolle.

MLaaS wird in der Praxis immer mehr nachgefragt und bereits in verschiedenen Branchen eingesetzt, darunter zum Beispiel im Gesundheitswesen, im Finanzsektor, im Einzelhandel, in der Fertigung, im Transportwesen und in vielen weiteren Bereichen. Auch für die Analyse von Geschäftsprozessen finden die KI-Dienste Anwendung.

Nachfolgend werden Amazon Lex in Verbindung mit dem AWS Toolkit für Visual Studio, die Azure Cognitive Services und das Azure Machine Learning Studio von Microsoft betrachtet.

■ 9.1 Amazon Machine Learning und KI-Services

Amazon Web Services (AWS) bieten eine Vielzahl von vordefinierten KI-Services wie Computer Vision, Sprache, Empfehlungen oder Vorhersagen an. Sie können aber auch den eigenständigen und vollständig verwalteten Dienst Amazon SageMaker verwenden. Dieser bietet jedem Entwickler die Möglichkeit, schnell ML-Modelle für Machine Learning zu erstellen, zu trainieren und bereitzustellen. Des Weiteren bietet Amazon ein Software Development Kit (SDK) für .NET-Entwickler an. Das SDK bringt eine Gruppe von Tools mit, die es Ihnen erlaubt auf die Amazon Services zuzugreifen. Das AWS SDK for .NET unterstützt das .NET Framework 4.5, .NET Standard 2.0, die Portable Class Library und Xamarin. Es unterstützt die meisten Cloud-Produkte von Amazon und wird laufend um weitere Services erweitert. Somit haben Sie einen komfortablen Zugriff auf die Machine-Learning- und KI-Services von Amazon mit Visual Studio und C#.

Zum SDK gesellt sich dann noch das Toolkit for Visual Studio. Hierbei handelt es sich um ein Plug-in für Visual Studio, das Ihnen das Entwickeln, Debuggen und Bereitstellen von .NET-Anwendungen, die Amazon Web Services nutzen wollen, erleichtert. Das Toolkit bietet darüber hinaus noch Visual-Studio-Vorlagen (Templates) für Services wie Lambda und Bereitstellungsassistenten für Webanwendungen und Serverless-Anwendungen.

Das Toolkit ermöglicht es über den AWS Explorer die von der Anwendung verwendeten AWS-Ressourcen anzuzeigen. Wenn Sie AWS-Ressourcen für Ihre Anwendung bereitstellen müssen, können Sie diese manuell mit dem AWS Explorer erstellen oder die im Toolkit enthaltenen *CloudFormation*-Vorlagen verwenden, um Webanwendungsumgebungen bereitzustellen, die auf Amazon EC2 gehostet werden. Das Toolkit for Visual Studio bietet darüber hinaus noch weitere Features für Entwickler an:

- Der AWS Explorer zeigt in einer Baumansicht die AWS-Ressourcen. Sie können sich die Ressourcen im Explorer direkt anzeigen lassen und dort bearbeiten.
- Webanwendungen und Websites können mit einem Klick auf *Publish to AWS Elastic Beanstalk* in der Cloud bereitgestellt werden.
- Durch die Auswahl von *Publish to AWS Lambda* wird eine Serverless-Anwendung in der Cloud bereitgestellt.
- Mit der Amazon-EC2-Instanz-Ansicht können Sie schnell neue Windows-Instanzen und einen Remote Desktop erstellen.
- Sie können die in Ihrem Amazon S3 (Simple Storage Service) gespeicherten Dateien durchsuchen und Dateien hoch- und herunterladen.
- Sie können AWS-IAM-Benutzer- und -Gruppen und entsprechende Zugriffsrichtlinien erstellen.
- Mit dem Toolkit können Sie auch *Amazon DynamoDB*-Tabellen anzeigen, erstellen und löschen.

Somit bietet das AWS Toolkit for Visual Studio nicht nur Hilfe beim Erstellen von .NET-Projekten, sondern erlaubt gleichzeitig die Konfiguration, das Beobachten und Abfragen von Services bei Amazon.



Serverless

Der Begriff bezieht sich auf ein Cloud-Computing-Modell, bei dem der Entwickler keine Server-Infrastruktur einrichten bzw. benutzen, noch eine Skalierung durchführen muss. Diese routinemäßigen Aufgaben abstrahiert der Cloud-Anbieter und Sie können sich als Entwickler voll und ganz auf Ihren Code konzentrieren. Sie sind somit viel schneller produktiv, als wenn Sie traditionelle Server-Anwendungen nutzen.

9.1.1 Amazon Lex

AWS stellt mit dem KI-Service Amazon Lex einen Werkzeugkasten für die Entwicklung von Bots zur Verfügung. Amazon bezeichnet diesen Service als Konversationsschnittstelle für Sprache und Text in verschiedenen Anwendungen. Somit soll es möglich sein, für Ihre Anwendung, egal ob Internetplattform, mobile Anwendung oder IOT-Gerät, einen Chatbot bereitzustellen, der zum Beispiel automatisierten Support liefert und Kundenanfragen beantworten kann. Hierbei unterstützt Amazon Lex sowohl die Entwicklung eines Bots wie auch die Integration von Bots in andere Kommunikationsplattformen wie Facebook Messenger oder über entsprechende REST APIs in eine Vielzahl von anderen Anwendungen.

Amazon bietet mit Lex fortschrittliche Deep-Learning-Funktionen für die automatische Spracherkennung (Automatic Speech Recognition, ASR) zur Umwandlung von Sprache in Text und das natürliche Sprachverständnis (Natural Language Understanding, NLU) zur Erkennung von Textabsichten an. Diese Funktionen sind notwendig, damit ein Chatbot zuverlässig funktioniert.

Die gesprochenen Sätze müssen analysiert und in Text umgewandelt werden, bei geschriebenen Texten dürfen auch Tipp- und Rechtschreibfehler zu keinem Abbruch bei der Erkennung führen. So ist auch die Interpretation der gesprochenen Sätze und Texte sehr wichtig.

Amazon nutzt für die eigene Sprachassistentensoftware Alexa die Deep-Learning-Funktionen von Amazon Lex. Als Entwickler können Sie in Form von Amazon Lex diese Technologie nun auch in Ihren eigenen Anwendungen nutzen.

Die Einsatzmöglichkeiten eines Bots sind vielfältig und reichen von Chat-Antworten bis hin zu Transaktionen wie der Buchung eines Fluges. Einige wichtige Vorteile von Chatbots sind somit:

- **Automatische Antworten:** Diese können so programmiert werden, dass Sie dem Benutzer des Chatbots auf eine Frage automatisch über Sprache oder Text die richtige Antwort geben.
- **Reduzierung von Kosten und Zeit:** In bestimmten Bereichen (Callcenter/Support) können inzwischen Fragen und ihre Antworten vollständig automatisiert werden, sodass Kosten und Zeit eingespart werden.
- **Anfragen und Transaktionen:** Chatbots bieten nicht nur die Möglichkeit, Fragen zu stellen und Antworten zu liefern, sondern können auch entsprechende Transaktionen durchführen.
- **Direkte Interaktion mit APIs:** Die Logik des Bots kann so programmiert werden, dass sie direkt mit einer Backend-API interagiert. Das heißt, Sie können in Ihrem Backend-Programmcode auf eine Aktion im Bot reagieren und einen entsprechenden Vorgang auslösen.

- **Text und Stimme:** Die Interaktion mit dem Chatbot ist nicht nur auf Text beschränkt, sie kann auch so entwickelt werden, dass Sprachanfragen möglich sind. Zu diesem Zweck müssen entsprechende Conversational User Interfaces entworfen werden.



Conversational User Interface (CUI)

Als CUI bezeichnet man eine Konversationsschnittstelle, die es als Benutzeroberfläche für Computer erlaubt, den Eindruck einer Konversation zwischen Anwender und System so zu vermitteln, dass der Anwender den Eindruck hat, er tritt mit dem Computer wie mit einer realen Person in einen Dialog.

Ein entsprechendes Expertenwissen in Bezug auf maschinelles Lernen oder sogar Deep Learning ist nicht Voraussetzung für die Verwendung von Amazon Lex. Sie als Entwickler kümmern sich lediglich um die Definition der Konversation mit dem Endanwender und das Einbinden in Ihre gewünschte Anwendung, während Lex die Funktion für Spracherkennung und natürliches Sprachverstehen übernimmt. Beachten Sie, dass zur Zeit der Erstellung dieses Buches der KI-Dienst Lex noch auf die Region U.S. East beschränkt war.

9.1.2 Die Lex-Chatbot-Struktur

Amazon Lex folgt einer grundlegenden Chatbot-Struktur. Sie müssen zu allererst Intents, Slots, Ihre Dialogformulierung und den gewünschten Dialogablauf erstellen. Der Dialogfluss wird bei Lex als Zustandsmaschine oder Benutzergeschichte bezeichnet. Die benötigten Komponenten für den Chatbot sind in einer Ablaufstruktur angelegt. Das heißt, wenn Sie einen Chatbot entwickeln, werden Sie Schritt für Schritt durch eine vorgegebene Struktur geführt, die Sie bei der Entwicklung des Chatbots unterstützt.

Intents

Bei den *Intents*, den Absichten, handelt es sich um die Hauptkomponenten eines Bots. Intents verwalten die verschiedenen Aufgaben, die mit dem Bot erledigt werden können. Jeder Intent kann separat konfiguriert werden und ist dadurch auch unabhängig von den anderen Intents. Beim Hinzufügen von Intents können diese wie grammatisch gesprochen wie Verben betrachtet werden, die erkennen, welche Absicht der Benutzer verfolgt. Für einen Bankvorgang könnte es unter anderem folgende Intents geben:

- Ich möchte ein Konto eröffnen.
- Ich muss ein Konto schließen.
- Wie ist mein Finanzstatus.
- Ich möchte eine Überweisung durchführen.

Utterances

Unter dem Eintrag *Utterances*, also der eigentlichen Äußerung, versteht man alles, was in den Chatbot eingegeben wird. Die meisten Äußerungen enthalten ein Ziel bzw. eine Absicht. Bei jeder Eingabe versucht der Chatbot die Äußerung einem implementierten Intent zuzuordnen,

um diesen dann auszulösen. Hierfür legen Sie unter *Utterances* entsprechende Textausdrücke fest, die den Intent aktivieren. Intern verwendet Lex Deep-Learning-Mechanismen, um diese Ausdrücke zu erkennen. Daher ist es möglich, dass die gemachten Äußerungen nicht genau dem Text entsprechen müssen, den Sie vom Benutzer des Chatbots erwarten. Ein Intent kann auch mehrere Verben aufnehmen, um auf ähnliche Ausdrücke reagieren zu können.

Bei Lex können Äußerungen auch Slots enthalten. Hierbei handelt es sich um variable Werte, die vom Benutzer bei Fragen an den Chatbot angegeben werden. Im Falle eines Reise-Chatbots wären dann die Angaben Ort, Verkehrsmittel oder Hotel die variablen Werte, die Sie als Slots in den Äußerungen konfigurieren können. Die Prioritäten der Slots lassen sich in Lex festlegen und auch Folgeaufforderungen lassen sich konfigurieren, um sicherzustellen, dass diese auch vom Anwender ausgefüllt werden.

Confirmation prompt (Optional)

Beim Confirmation prompt handelt es sich um eine Bestätigungsabfrage. Diese Frage wird vom Chatbot gestellt, um die Absicht des Benutzers zu bestätigen.

Fulfillment

Bei Fulfillment geht es um die Verarbeitung der Benutzereingabe durch Lex. Hier werden die gesammelten Daten an Ihre AWS-Lambda-Funktion weitergeleitet, in der Sie die Geschäftslogik entwickelt haben, um die Anfrage des Benutzers zu erfüllen.

Response

Nachdem die Anfrage abgearbeitet ist, kann der Benutzer über Response eine Antwort über den Status seiner Anfrage erhalten. Diese Antwort kann in der Lex-Konsole konfiguriert werden, oder Sie können in der Lambda-Funktion eine individuelle Echtzeit-Antwort implementieren.

Session Attributes

Bei den Session Attributes handelt es sich um Attribute, die von der Anwendung, die den Chatbot enthält, gesetzt werden. Hiermit ist es dann möglich, einen entsprechenden programmtechnischen Kontext für den Benutzer herzustellen, das heißt, Ihre Anwendung kann je nach Session-Attribut ein anderes Event auslösen. Die Attribute gelten so lange, bis die Session abläuft oder das Attribut gelöscht wird.

Request Attributes

Über Request Attributes können Attribute für den Chatbot angefordert werden. Im Gegensatz zu den Session Attributes gelten diese Attribute nur für die jeweils aktuelle Anfrage.

Der Aufbau von Struktur und Komponenten in Lex erfolgt in einem sogenannten Skript und wird größtenteils direkt innerhalb von Lex definiert. Sie legen in der Lex-Konsole die umfangreichen Vorgaben für die entsprechenden Absichten, Bestätigungsaufrufe und Folgeaufforderungen schon im Vorfeld an. Für die Verwendung der Sitzungsvariablen, der Dialogverwaltung und der Zustandsaufgaben müssen Sie jedoch eine Verbindung zu einer Lambda-Funktion herstellen.

9.1.3 Entwickeln mit AWS-Lambda-Funktionen

Ein Intent lässt sich mit Amazon Lex auf zwei Arten umsetzen. Entweder durch die Integration in AWS Lambda oder Sie konfigurieren Lex so, dass Intents und Slot-Werte an den Client zurückgesendet werden, um Absichten im Chatbot umzusetzen.

Bei einer Integration in AWS Lambda lassen sich Benutzereingaben mit einer Code-Verknüpfung für die Initialisierung und Validierung implementieren. Amazon spricht diesbezüglich von einem sogenannten *Codehook* bzw. einem Code-Haken. Des Weiteren kann der Codehook zur Einrichtung von Parametern, für Überprüfungen und für Benutzereingaben verwendet werden. Möchten Sie mit einem Back-End über API oder einen Service interagieren, müssen Sie auf jeden Fall eine AWS-Lambda-Funktion entwickeln, die die Eingaben des Bots verarbeitet und die Anfragen an eine API oder einen Service stellt.

Unter AWS Lambda versteht Amazon einen Service, der kleine Code-Segmente in einer serverlosen Infrastruktur einsetzt. Durch den Einsatz der serverlosen Infrastruktur ist es für Sie sehr gut möglich, Ihre eigene Anwendung dann als Microservice aufzubauen. Bei diesem Ansatz implementiert jeder Service (Container) mehrere zusammenhängende und eng verwandte Funktionen. Im Zentrum von Lambda steht die Ausführung von Code-Funktionen. Hierbei nimmt der Service den Sourcecode der Funktion direkt entgegen und führt ihn aus. Die Kommunikation findet mithilfe von Protokollen wie HTTP (REST), nach Möglichkeit auch asynchron statt. Die von AWS Lambda unterstützten Sprachen sind Node.js, Java, Go, C# und Python.

Der Code, der in die Lambda-Funktion einbezogen werden soll, kann aus einer einzelnen Datei oder aus mehreren in einer .ZIP-Datei komprimierten Dateien bestehen. Benötigen Sie bei der Entwicklung weitere Bibliotheken, die auch der Code für die Lambda-Funktion nutzt, so müssen diese unbedingt in der .ZIP-Datei enthalten sein, damit sie für die AWS-Lambda-Funktion verfügbar sind. Der Einstiegspunkt in der AWS-Lambda-Funktion muss immer folgendes Format aufweisen.

```
exports.handler = (event, context, callback) => {
  //ToDo (Hier rufen Sie Ihre API auf, verbinden Sie sich mit externen Diensten, usw.)
  response = ...
  callback(null, response);
};
```

Der Code für die Lambda-Funktion enthält einen Handler, der folgende Parameter unterstützt:

- *Event*
- *Context*
- *Callback*

Das Event-Objekt enthält alle Informationen, die mit dem Ereignis zusammenhängen. Dazu zählen die Session-Attribute, Request-Attribute, die Namen der Intents und die Variablen für die Slots.

Der Parameter *Context* kann entsprechende Informationen über die Lambda-Funktion beinhalten. Bei *Callback* handelt es sich um eine Funktion, die Ihre Antwort erhält. Die Antwort kann eine Nachricht für den Benutzer sein, aber auch weitere Fragen oder Meldungen.

Responses

Über das Code-Segment Response können Sie dem User, der den Bot nutzt, mitteilen, ob es sich um eine erfolgreiche Antwort oder eine fehlgeschlagene Antwort handelt. Alle Antworten werden als JSON-Objekt mit der Struktur aus Listing 9.1 zurückgeliefert.

Listing 9.1 Die Antwort als JSON-Objekt

```
{
  "sessionAttributes": { },
  "dialogAction": {
    "type": "Close",
    "fulfillmentState": "Fulfilled",
    "message": {
      "contentType": "PlainText",
      "content": "Your response message here"
    }
}
```

Je nach Typ des Bots, den Sie entwickeln, können Sie die Elemente im Bereich *Content* aus Listing 9.1 entsprechend anpassen.

sessionAttributes-Objekt

Möchten Sie neue Werte für die *sessionAttributes* setzen, so müssen Sie diese in der Antwort an den User, der den Bot nutzt, zurücksenden. Dies kann ganz einfach über das *sessionAttribut*-Element erfolgen, das einen Satz von Key-Value-Paaren enthält.

dialogAction-Objekt

Das *dialogAction*-Objekt enthält die Einzelheiten zur Antwort. Dazu gehören die Art der Antwort und die nächsten Schritte, die durchgeführt werden sollen. Es sind folgende Attribute möglich:

- *Type*
- *FulfillmentState*
- *Message*

Das Attribut *Type* definiert die Art der Antwort und die vom Bot auszuführenden Aktionen. Mögliche Werte sind hier *Close*, *ConfirmIntent*, *Delegate*, *ElicitIntent* und *ElicitSlot*. Der *FulfillmentState* teilt dem Bot mit, ob die Anforderung erfüllt wurde. Ist dies der Fall, so wird der State auf *Fulfilled* gesetzt, andernfalls auf *Failed*.

Die Message enthält Informationen über die Nachricht, die dem Benutzer angezeigt werden soll. Um dem Benutzer aber eine Antwort geben zu können, müssen folgende Attribute gesetzt werden:

- **contentType:** Nachrichtentyp; gültige Werte sind *PlainText*, *SSML* (in Sprache umgewandelter Text) und *CustomPayload*, welches ein benutzerdefiniertes Format darstellt.
- **Message:** In der Message befindet sich der eigentliche Text, der dem Benutzer angezeigt wird bzw. der in Sprache umgewandelt werden soll.

Weitere ausführliche Informationen zu den Methoden und Funktionen von Amazon Lex finden Sie im Entwicklerhandbuch unter [49]. Als Nächstes erstellen Sie über die Lex-Konsole einen Chatbot und nutzen diesen dann über eine Web-API mit Lambda-Funktion und C#.

■ 9.2 Erstellen eines Lex-Chatbots für .NET

Mit dem hier vorgestellten .NET Core Chatbot erstellen Sie einen Chatbot für eine Wartungsanfrage an eine Maschine. Der Chatbot wird über den Amazon-Lex-Service betrieben. Die eigentliche Anwendung ist eine ASP.NET-Core-MVC-Webanwendung, die das AWS .NET SDK und das AWS Toolkit for Visual Studio verwendet und auf Amazon EC2 bereitgestellt wird. Für das Beispiel werden folgende AWS Services zur Implementierung der Chatbot-Webanwendung verwendet:

- **Amazon Lex:** Für die textbasierte Konversation mit dem Chatbot.
- **Amazon Cognito:** Authentifizierung, Autorisierung und Benutzerverwaltung für Ihre Web-App.
- **Amazon EC2:** Virtuell skalierbare Berechnungsinstanzen in der AWS Cloud, um die Web-App zur Verfügung zu stellen. EC2 steht für Amazon Elastic Compute Cloud.
- **AWS-CodePipeline:** Bereitstellungsdienst für die automatische Erstellung und Bereitstellung von Bots.
- **AWS-CodeBuild:** Ein Build Service zur Kompilierung des Chatbot-Quellcodes und zur Erstellung von Softwarepaketen.

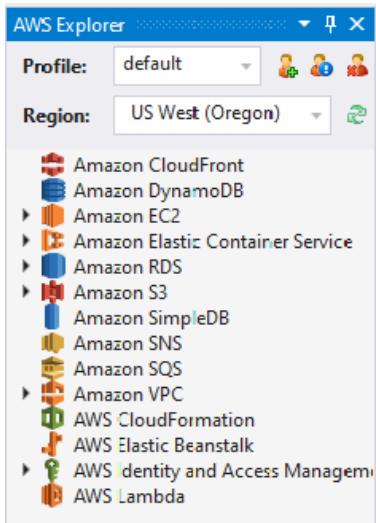
Nachfolgend wird der Chatbot schrittweise mithilfe des Lex Service und der Lex-Konsole im Web erstellt. Alternativ könnte man den Bot auch per Common Line Interface (CLI) über den Lex Service erstellen. Hierzu benötigt man aber einige Erfahrung in der Erstellung eines Bots. Mehr zur Verwendung und Handhabung der CLI entnehmen Sie dem Entwicklerhandbuch [50] von Amazon Lex.

9.2.1 Erste Schritte

Wie schon beschrieben, soll der Chatbot das Erstellen und Anzeigen einer Wartungsanfrage ermöglichen. Im Beispiel wird auf Visual Studio 2019 zurückgegriffen. Zusätzlich benötigen Sie das AWS Tools and SDK for Net [51] und das AWS Toolkit Package [52]. Führen Sie hierzu den Download durch und installieren Sie das SDK und das Toolkit Package. Ist die Installation erfolgreich abgeschlossen, verfügt Visual Studio über neue Projektvorlagen (Templates) für verschiedene Entwicklungsbereiche der Amazon Web Services.

Des Weiteren benötigen Sie ein aktives Amazon-Web-Service-Konto, das Sie unter [53] anlegen können. Beachten Sie hier auf jeden Fall die Preismodelle.

Außerdem müssen Sie, bevor Sie die AWS-Services aus Visual Studio heraus verwenden können, einmalig die Anmeldeinformationen für die Nutzung und Authentifizierung einrichten. Sind alle benötigten Komponenten vorhanden und ist die Anmeldung für AWS in Visual Studio erfolgt, so können Sie im AWS Explorer (Bild 9.1) auf alle Web-Service-Ressourcen von Amazon zugreifen.

**Bild 9.1**

Der AWS Explorer in Visual Studio

9.2.2 Beispiel Chatbot

Der Bot stellt unter Amazon eine Komponente der obersten Ebene dar, die als eine Art Container eingesetzt wird. Der Bot, den Sie im Beispiel entwickeln, wird einen bzw. für Ihre eigene Erweiterung mehrere Intents haben, von denen jeder Intent ein anderes Thema repräsentiert, das der Bot verstehen und demnach eine entsprechende Aktion ausführen kann. Über Slots können Sie dann, je nach Aufgabenstellung, die benötigten variablen Werte definieren, und mit der Lambda-Funktion legen Sie den Ort fest, an dem Sie Ihren Code ausführen möchten, ohne dass Sie einen Server bereitstellen oder verwalten müssen.

Das Anlegen eines Chatbots gestaltet sich in Amazon Lex über das Web Frontend übersichtlich, schnell und komfortabel. Um den Chatbot mit der Konsole zu erstellen, melden Sie sich mit Ihren Benutzerdaten an der AWS-Managementkonsole an. In das Suchfeld *Services finden* tragen Sie den Begriff Amazon Lex ein und wählen diesen dann über den vorgeschlagenen Diensttyp aus.

Nach der Auswahl werden Sie auf die Startseite von Amazon Lex weitergeleitet. Klicken Sie hier auf *Get Started*, um mit der Erstellung des Chatbots zu beginnen. Amazon Lex führt Sie jetzt weiter auf die Dialogseite *Create your bot*. Sie können hier ein vorhandenes Bot-Muster ausprobieren oder über *Custom bot* einen eigenen Bot erstellen.

Da Sie für das Beispiel einen Bot von Grund auf neu erstellen, wählen Sie die Option für den benutzerdefinierten Bot aus (Bild 9.2).

Geben Sie im Feld *Bot name* den Bot-Namen an. Für das Beispiel wurde *DemoServiceBot* verwendet. Wählen Sie für *Output voice* (Sprachausgabe) *None* (keine), da in unserem Beispiel nur ein textbasierter Bot erstellt werden soll. Setzen Sie *Session timeout* auf fünf Minuten. Die IAM-Rolle wurde von AWS automatisch erstellt, sodass für diese Einstellung kein Handlungsbedarf besteht. Die IAM-Rolle ermöglicht es Ihnen, den Zugriff an Benutzer oder Services zu delegieren, die normalerweise keinen Zugriff auf Ihre AWS-Ressourcen haben.

Diese erhalten durch IAM eine temporäre Anmeldeinformation, die es erlaubt, einen AWS-API-Aufruf auszuführen. Für die Einstellung *COPPA* können Sie *No* auswählen.

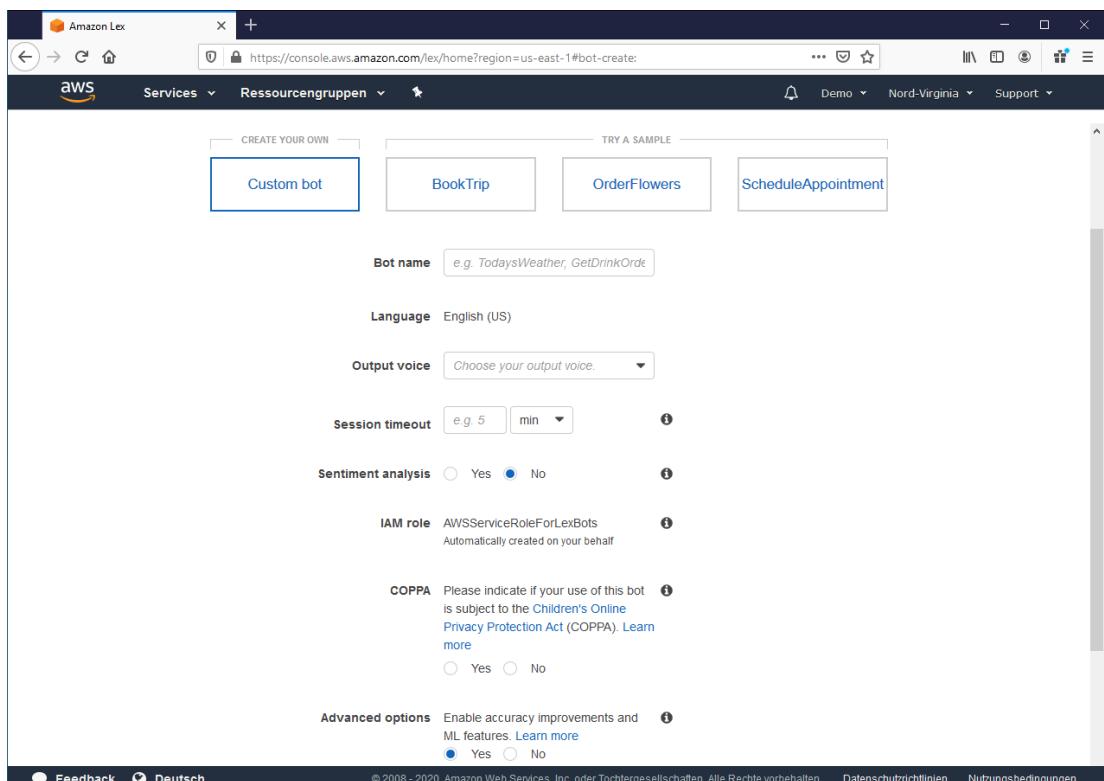


Bild 9.2 Startseite für den benutzerdefinierten Chatbot

➡

COPPA

Der Children's Online Privacy Protection Act (COOPA) steht für das Gesetz zum Schutz der Privatsphäre von Kindern im Internet aus dem Jahr 2000 in den USA [54]. Da die COPPA-Richtlinien in Deutschland nicht gelten, ist in unserem Beispiel ein *No* zulässig.

Die Einstellungen *Advanced options* und *Confidence score threshold* belassen Sie in der Default-Konfiguration. Haben Sie alle benötigten Felder ausgefüllt, können Sie über *Create* den Bot erstellen. Amazon Lex erstellt jetzt den Bot und öffnet die DemoServiceBot-Seite (Bild 9.3).

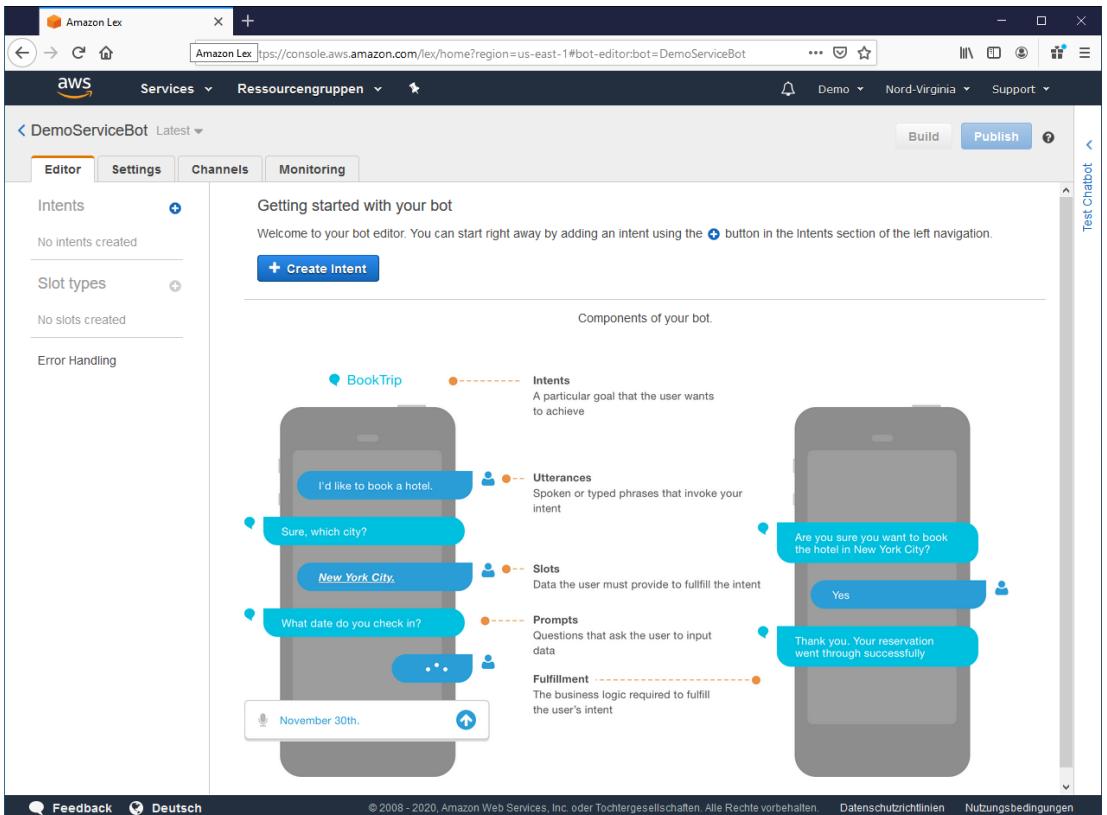


Bild 9.3 Die neue Chatbot-Seite in AWS

9.2.3 Intents

Jetzt können Sie für den Chatbot einen Intent, also die Absicht bzw. Aufgabe, erstellen. Da der Bot nur als Container fungiert, ist ein Intent zwingend notwendig. Wie soll der Bot auch sonst funktionieren und welche Aufgabe soll er lösen? Klicken Sie daher auf die Schaltfläche *Create Intent*, um eine neue Absicht (Aufgabe) zu erstellen.

Im jetzt geöffneten Popup-Fenster (Bild 9.4) werden die Optionen angeboten, einen neuen Intent zu erstellen, Intents zu importieren oder nach vorhandenen Intents zu suchen. Wählen Sie die Option *Create intent*.

Vergeben Sie für den Intent einen eindeutigen Namen. Im Beispiel wird *ServiceHours* gewählt. Der Intent soll den Benutzer über die möglichen Servicezeiten der Herstellerfirma der Maschine informieren. Bestätigen Sie Ihre Eingabe mit *Add*.

Als Nächstes können Sie über *Sample utterances* die Textausdrücke festlegen, die den Intent auslösen sollen. Zum Beispiel können Sie eine Beispieläußerung der Art „Wie sind Ihre Servicezeiten“ oder „Ab wann kann ich jemanden erreichen“ angeben (Bild 9.5).

Das eingebaute Sprachmodell von Lex kann jetzt über Deep-Learning-Mechanismen ähnliche Formulierungen erkennen, sodass man nicht unnötig viele Beispiele angeben muss.

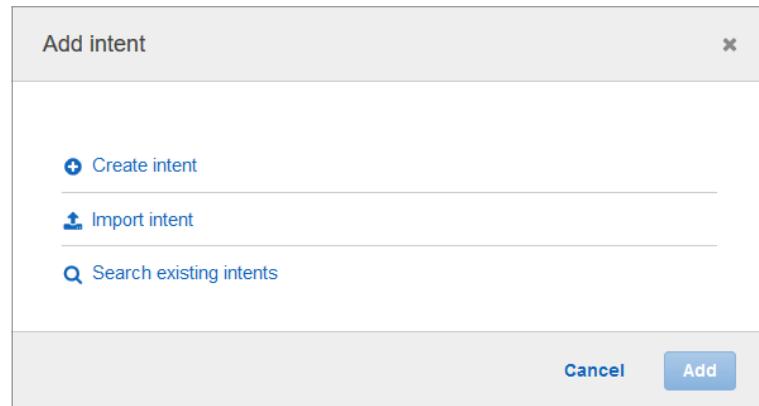


Bild 9.4 Das Popup-Fenster zur Erstellung von Intents

The screenshot shows the AWS Lambda console interface for a bot named "DemoServiceBot". The "Editor" tab is active. On the left, there's a sidebar with "Intents" (selected), "Slot types" (with a plus icon), "No slots created", and "Error Handling". The main area shows an intent named "ServiceHours" with the latest version. Under "ServiceHours", there's a section for "Sample utterances" with three entries: "When can I reach somebody", "What are your service hours", and "What are your service times". Each entry has a delete icon (an X) to its right.

Bild 9.5 Festlegen der Textausdrücke

Confirmation prompt

Etwas weiter unten auf der Dialogseite finden Sie den Eintrag *Confirmation prompt*. Diese Bestätigungsabfrage dient der doppelten Überprüfung mit dem Benutzer. Diese Abfrage sollten Sie für den Dialog mit dem Benutzer aktivieren. Sie können hier eine einfache Frage stellen und wenn der Benutzer mit *Yes* antwortet, können Sie mit der Verarbeitung (Fulfillment) der Benutzereingabe starten.

Wenn der Benutzer mit *No* antwortet, wird die Nachricht, die Sie im Feld *Cancel* festgelegt haben, angezeigt und der weitere Verlauf im Bot für diesen Intent beendet. Im *Confirm*-Feld legen Sie im Beispiel den Text „Möchten Sie die Servicezeiten wissen?“ an und bei *Cancel* tragen Sie einfach „Okay. Danke!“ ein. In Bild 9.6 sind die Meldungen in den Feldern aufgrund der Lex-Sprachversion in Englisch verfasst.

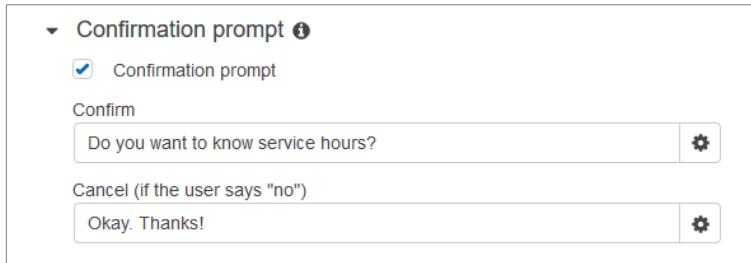


Bild 9.6 Einstellung der Bestätigungsaufrufforderung

Response

Unter *Response* legen Sie die Antworten für den eingerichteten Intent fest. Für unser Beispiel werden einfach die Servicezeiten angegeben. Bild 9.7 zeigt den entsprechenden Eintrag in der Liste.

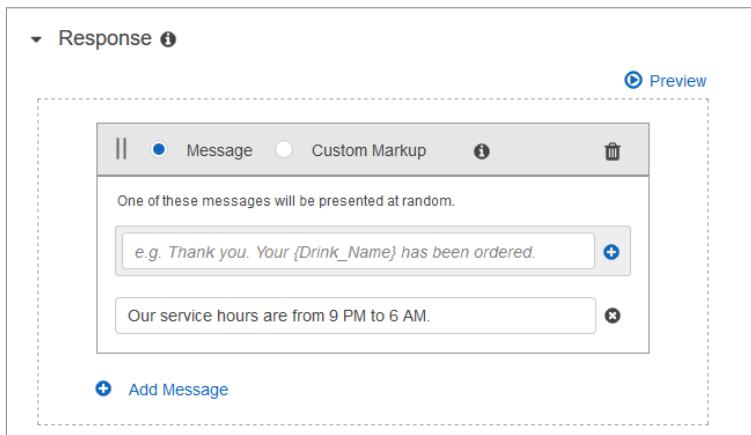


Bild 9.7 Festlegen der Antwort

Somit ist die erste Grundlage für den Chatbot geschaffen. Um jetzt sicherzustellen, dass der Bot funktioniert, sollten Sie den Bot einem Test unterziehen. Speichern Sie hierfür als Erstes Ihre vorgenommenen Einstellungen über den Button *Save Intent* am Ende der Dialogseite ab.

9.2.4 Testen Sie den Bot

Um den Bot in der Lex-Konsole testen zu können, müssen Sie diesen erstellen, indem Sie auf die Schaltfläche *Build* klicken. Sie erhalten eine Hinweismeldung, dass Sie Ihren Bot weiterbearbeiten und nach einem erfolgreichen Build-Vorgang einen Test durchführen können. Bestätigen Sie die Meldung, indem Sie auf *Build* klicken. Nach einem erfolgreichen Build erhalten Sie eine Bestätigungsmeldung (Bild 9.8).

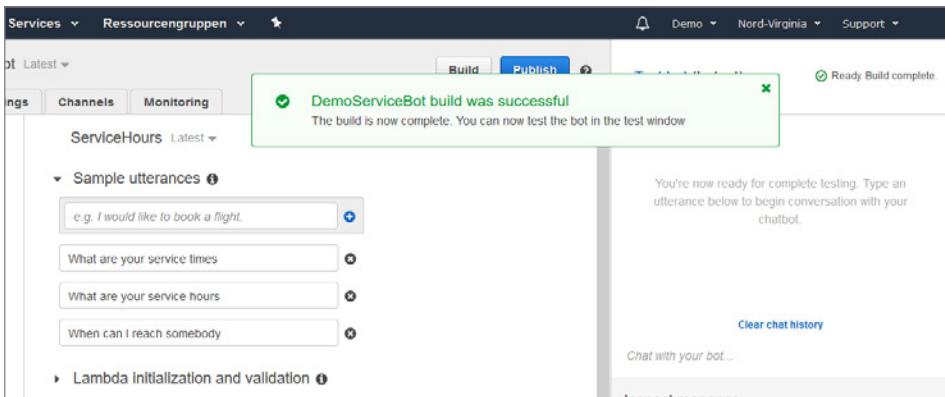


Bild 9.8 Build erfolgreich abgeschlossen

Jetzt können Sie über die rechte Seitenleiste *Test Chatbot* die Funktion überprüfen. Geben Sie im Seitenbereich eine der Textphrasen ein, zum Beispiel „*What are your service time*“, und bestätigen Sie die Eingabe, so wird der Bot entsprechend antworten. Sie können aber von der Phrase abweichen, wie in Bild 9.9 dargestellt. Fragen Sie nach „*What are your service*“, so erhalten Sie aufgrund der Deep-Learning-Mechanismen von Lex ebenfalls die richtige Antwort.

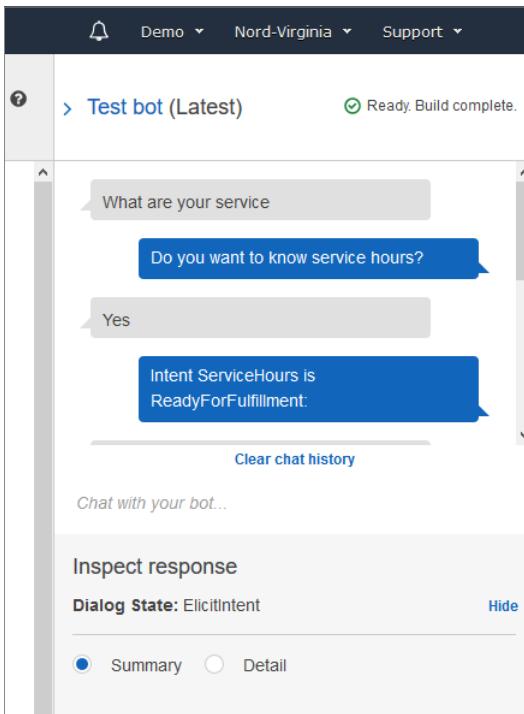


Bild 9.9

Der Chatbot im Test

9.2.5 AWS-Lambda-Funktion

Wie schon angesprochen, stellt AWS Lambda einen serverlosen Datenverarbeitungsservice dar, der Ihren Code beim Eintreten bestimmter Ereignisse ausführt. Dabei führt Lambda Ihren Code automatisch auf hochverfügbaren Infrastrukturen aus und erledigt die gesamte Administration. Der Code, den Sie auf AWS Lambda ausführen, wird bei Amazon als Lambda-Funktion bezeichnet.

In unserem Beispiel konfigurieren Sie die Lambda-Funktion als *Codehook* (Abschnitt 9.1.3), der mit dem Lex Bot verwendet wird, um die Initialisierung und Validierung in der Konfiguration des erstellten Intent durchzuführen. Da Sie im Beispiel den Chatbot als Webanwendung unter .NET hosten wollen, wird die Lambda-Funktion gleich mithilfe des AWS Toolkit for Visual Studio erstellt und hochgeladen.

Um den Chatbot über das Toolkit und Visual Studio nutzen zu können, müssen Sie diesen im Vorfeld veröffentlichen. Ist der Test von Ihrer Seite abgeschlossen, so können Sie den Bot über den Button *Publish* in der Dialogseite veröffentlichen. Vergeben Sie im Popup-Fenster noch einen Alias für den Chatbot. Im Beispiel wird *DemoBot* verwendet. Klicken Sie auf *Publish*, um den Chatbot zu veröffentlichen.

AWS Toolkit for Visual Studio

Sie können eine Lambda-Funktion für Ihren Chatbot auch ganz komfortabel in Visual Studio mit C# erstellen. Der folgende Abschnitt beschreibt dies kurz, für unseren Beispiel-Bot ist dieses Vorgehen aber nicht notwendig.

Starten Sie Visual Studio und wählen Sie in der Vorlagenliste das Projekt *AWS Lambda Project (.NET Core - C#)* aus (Bild 9.10).

Wählen Sie nachfolgend einen Projektnamen und erstellen Sie das Projekt über *Create*. Danach öffnet sich das Dialogfenster *Select Blueprint*, in dem mehrere Lambda-Funktionsvorlagen zur Verfügung gestellt werden. Sie können hier eine Lambda-Funktion von Grund auf neu erstellen oder alternativ auch einen entsprechenden Beispielcode auswählen. Wählen Sie das Muster *Empty Function* und klicken Sie auf *Finish*. Visual Studio erstellt jetzt automatisch ein entsprechendes Projekt für die Lambda-Funktion.

Die Datei *aws-lambda-tools-defaults.json*, die standardmäßig von Amazons Lambda-Tool gelesen wird, legt die Optionen fest. Besonders zu beachten sind die folgenden Felder:

- **Profile:** Der Name eines Profils in Ihrer .NET-Anmeldeinformation.
- **Function-Handler:** Hier wird der *function handler* angegeben.

Die JSON-Datei hat immer genau den Aufbau wie in Listing 9.2.

Create a new project

Recent project templates

-  Console App (.NET Core) C#
-  WPF App (.NET Core) C#
-  WPF App (.NET Framework) C#
-  Blank App (Universal Windows) C#
-  ASP.NET Core Web Application C#

Bild 9.10 Auswahl der Projektvorlage

Listing 9.2 Aufbau der JSON-Datei

```
{
  "Information" : [
    "..."
  ],
  "profile"      : "default",
  "region"       : "us-east-1",
  "configuration": "Release",
  "framework"   : "netcoreapp3.1",
  "function-runtime": "dotnetcore3.1",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "Chatbot::Chatbot.Function::FunctionHandler",
  "function-name": "LambdaFunctionForLex",
  "function-role": "arn:aws:iam::xxxxxxxxx:role/
    lambda_exec_LambdaFunctionForLex",
  "tracing-mode": "PassThrough",
  "environment-variables": ""
}
```

In der Datei *Function.cs* erstellen Sie dann die benötigte Logik für die Implementierung des Lambda-Function-Handlers. Das Muster zeigt nur die automatisch generierte Methode.

Listing 9.3 Aufbau der Klasse Function.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
// into a .NET class.
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.
    DefaultLambdaJsonSerializer))]

namespace Chatbot
{
    public class Function
    {

        /// <summary>
        /// A simple function that takes a string and does a ToUpper
        /// </summary>
        /// <param name="input"></param>
        /// <param name="context"></param>
        /// <returns></returns>
        public string FunctionHandler(string input, ILambdaContext context)
        {
            return input?.ToUpper();
        }
    }
}

```

Ist der Code für die Lambda-Funktion fertiggestellt, können Sie diesen hochladen, indem Sie mit der rechten Maustaste auf den Projekt-Knoten im Solution-Explorer klicken und anschließend *Publish to AWS Lambda* auswählen.

Es öffnet sich das Dialogfenster *Upload Lambda Function*. Wählen Sie hier einen Namen für die Funktion aus und klicken Sie anschließend auf *Next*. Danach können Sie die Konfiguration des Fensters *Advanced Function Details* mit den folgenden Optionen durchführen (Bild 9.11).

Über *Role Name* definieren Sie die IAM-Rolle, die AWS Lambda annimmt, um Ihre Funktion auszuführen. *Memory* legt den Arbeitsspeicher fest, der während der Ausführung für die Funktion verfügbar ist. *Timeout* gibt an, wie lange die Funktion ausgeführt werden darf. *Environment* legt die Key-Value-Paare fest, die Lambda gegebenenfalls in der Ausführungsumgebung benötigt. Klicken Sie dann auf *Upload*, um die Lambda-Funktion bereitzustellen. Bild 9.12 zeigt die zur Verfügung gestellte Funktion in der AWS-Konsole unter dem Servicebereich *AWS Lambda*.

Jetzt können Sie die Lambda-Funktion mit Ihrer implementierten Logik in den Chatbot übernehmen. Dazu müssen Sie nur unter dem Eintrag *Lambda initialization and validation* die erstellte Lambda-Funktion auswählen. Beachten Sie, dass diese Funktion nur die Umsetzung einer Lambda-Funktion mit C# zeigen sollte. Sie können darüber hinaus mit Lambda-Funktionen den Chatbot mit einer benutzerdefinierten Logik versehen oder Ihren Backend-Service erweitern.

The screenshot shows the 'Advanced Function Details' configuration page for an AWS Lambda function. It includes sections for Permissions, Execution, VPC, Debugging and Error Handling, and Environment. The 'Permissions' section allows selecting an IAM role. The 'Execution' section sets memory to 256 MB and timeout to 30 seconds. The 'VPC' section indicates no VPC is selected. The 'Debugging and Error Handling' section shows a DLQ Resource set to '<no dead letter queue>' and an unchecked checkbox for 'Enable active tracing (AWS X-Ray)'. The 'Environment' section shows a KMS Key of '(default) aws/lambda' and an empty environment variable table with an 'Add...' button. At the bottom are 'Close', 'Back', 'Next', and 'Upload' buttons.

Bild 9.11 Erweiterte Details festlegen

The screenshot shows the AWS Lambda 'Funktionen' (Functions) list page. The left sidebar has 'Funktionen' selected under 'AWS Lambda'. The main area displays a table with one function entry:

Funktionsname	Beschreibung	Laufzeit	Code-Größe	Letzte Änderung
LambdaFunctionForLex	.NET Core 3.1 (C#/PowerShell)	20.9 kB	vor 9 Minuten	

Bild 9.12 Die bereitgestellte Lambda-Funktion

9.2.6 Slots

Nach dem Ausflug zu den Lambda-Funktionen können Sie sich wieder auf unseren Chatbot konzentrieren. Über den Bereich *Slots* können Sie flexibel mit variablen Werten arbeiten. Im Abschnitt *Slots* auf der Dialogseite legen Sie den Namen für den Slot, den Typ und die Eingabeaufforderung fest (Bild 9.13).

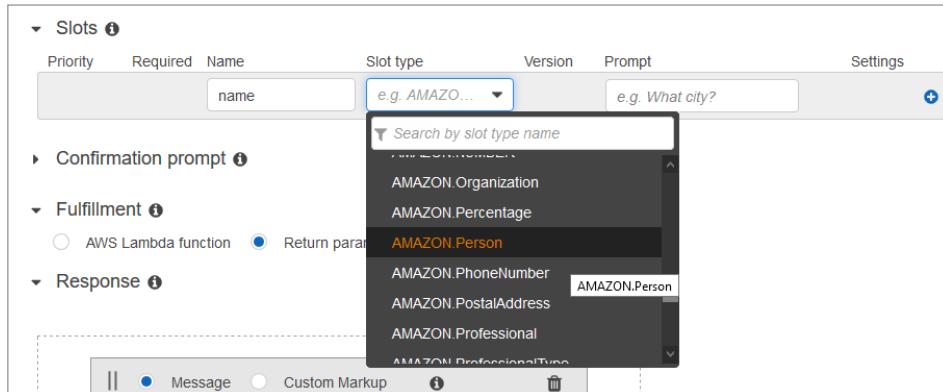


Bild 9.13 Anlegen eines Slots

Amazon bietet im Lex Service eine Vielzahl von unterschiedlichen Arten von Slots an, die Sie verwenden können. Dazu zählen unter anderem die Typen *Person*, *Number*, *EmailAddress*, *Date* und *Time*. So können Sie zum Beispiel den Namen, die Telefonnummer, ein Datum und die Uhrzeit mit Ihrem Chatbot erfragen, um dann den Benutzer zurückrufen zu können. Haben Sie entsprechende Slots angelegt, so können Sie die Einträge speichern, den Bot neu erstellen und Ihre Erweiterung testen.

Sollte die Auswahl der Typkonfiguration nicht ausreichen, so können Sie alternativ über den Eintrag *Slot type* zusätzliche von Ihnen definierte Slots für Ihren Bot erstellen.

9.2.7 Error Handling

Wie Sie bei Ihren Tests mit dem Chatbot sicherlich festgestellt haben, reagiert der Bot nicht besonders hilfreich bei Eingaben, die gar nicht zu den erstellten Textphrasen des Intent passen. Es wird dann immer nur eine voreingestellte Nachricht angezeigt und nach wenigen Aufforderungen teilt der Bot dem Chatbot-Nutzer mit, dass er nicht versteht, was der Nutzer schreibt oder spricht und verabschiedet sich.

Im Bereich *Error Handling* (Bild 9.14) können Sie die Fehlerbehandlung von Lex einstellen bzw. einrichten.

Hier können Sie den Bot jetzt so konfigurieren, dass er in der Lage ist, den Chatbot-Nutzer eine aussagekräftige Information anzuzeigen, wenn der Nutzer etwas eingibt, das der Chatbot nicht versteht. Nach dem Festlegen der benötigten Meldungen erstellen Sie den Bot noch einmal neu und testen Ihre vorgenommenen Einstellungen.

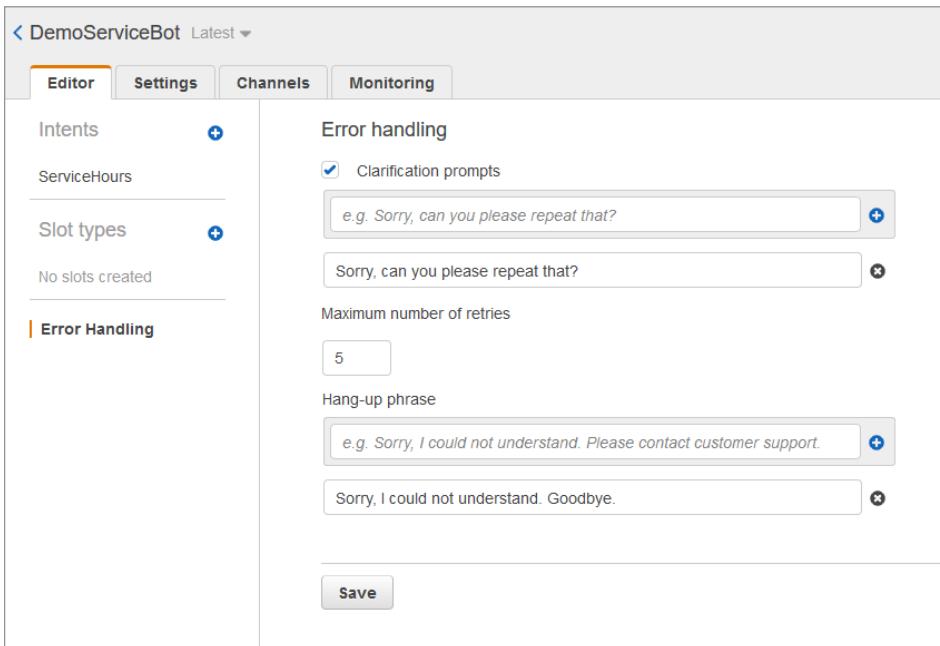


Bild 9.14 Festlegen von Fehlermeldungen

9.2.8 Konfigurieren von Cognito

Somit ist Ihr Basis-Chatbot ohne jeglichen Programmieraufwand erstellt worden. Sie können ihn nach Ihren Wünschen und Anforderungen weiter ausbauen und verbessern.

Nachfolgend soll der Chatbot als Web-App zur Verfügung gestellt werden. Um den Chatbot für die Web-App zu veröffentlichen, benötigen Sie Ihre AWS-Zugangsdaten. Nachdem Sie sich mit Ihren Zugangsdaten angemeldet haben, richten Sie einen AWS Cognito Identity Pool ein. Dabei handelt es sich um einen Speicher für Benutzer-Identitäten, die speziell für Ihr Konto gelten. Um den Identity Pool einzurichten, wählen Sie in der AWS-Konsole den Bereich *Services* aus. Tragen Sie in die Suchleiste „Cognito“ ein und wählen diesen Begriff aus den Suchergebnissen aus. Klicken Sie dann auf den Button *Verwalten von Identitätenpools*.

Vergeben Sie einen eindeutigen Namen, wie zum Beispiel *DemoLexBot*. Markieren Sie das Kästchen *Zugriff für nicht authentifizierte Identitäten aktivieren*. Klicken Sie dann auf die Schaltfläche *Pool erstellen* (Bild 9.15).

Lassen Sie im nächsten Schritt die Standardrollen zu, die AWS automatisch für Sie erstellen möchte, indem Sie einfach auf *Erlauben* klicken.

AWS erstellt jetzt den *Cognito Federated Identity Pool* und zeigt in einem Beispielcode die AWS-Anmeldeinformation und die Pool-ID. Kopieren Sie die Pool-ID und speichern Sie diese dort ab, wo Sie sie leicht abrufen können. Kehren Sie nun zu Ihrem Chatbot zurück und veröffentlichen Sie die neueste Version über *Publish*.

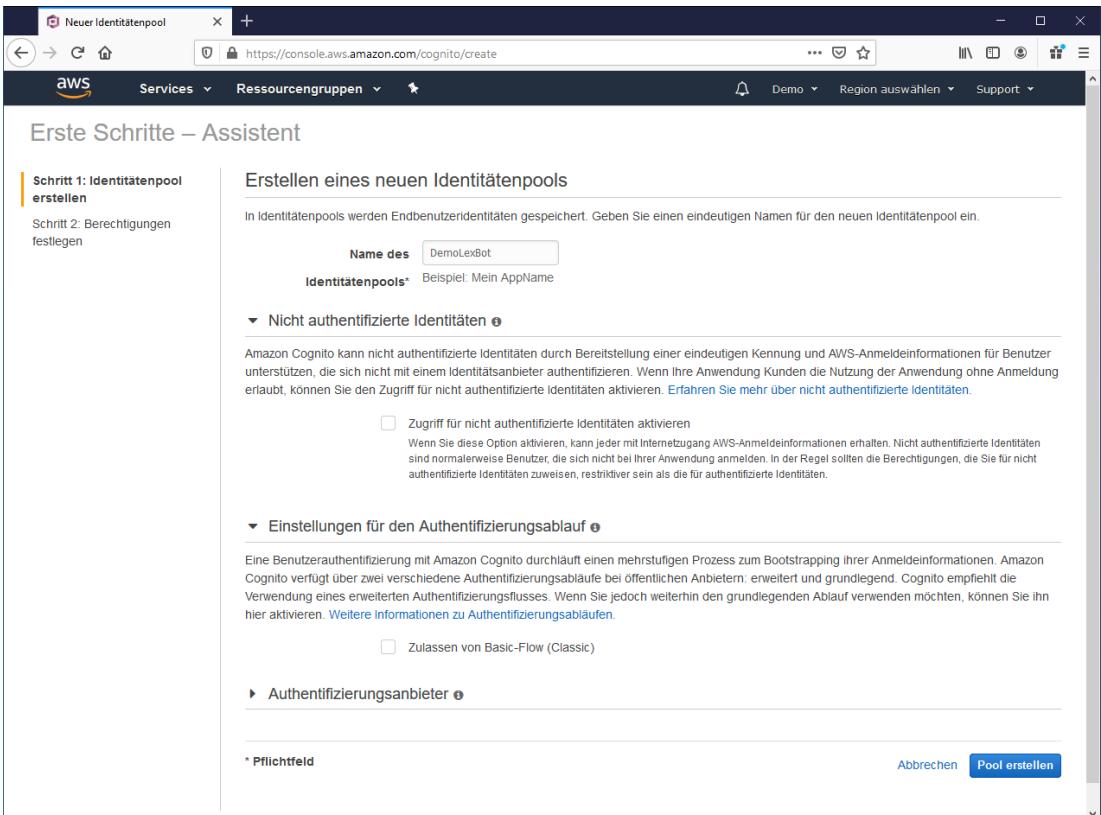


Bild 9.15 Anlegen eines Identitätenpools

9.2.9 Die Web-Applikation

Den erstellten Chatbot in einer ASP.NET Core Web Application zu nutzen, ist jetzt ganz einfach. Amazon bietet hierfür eine fertige Vorlage unter dem Namen *dotnetLexChatBot* an. Der entsprechende Code kann direkt von GitHub unter [55] heruntergeladen werden.

Ist dies geschehen, entpacken Sie das ZIP-Archiv und wechseln in den Unterordner *Code* des Projekts. Rufen Sie hier die Solution *dotnetLexChatBot.sln* mit Visual Studio auf. Nach dem Öffnen in Visual Studio wählen Sie die Datei *appsettings.json* aus.

Listing 9.4 Die JSON-Datei

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
}
```

```

    "AWSConfiguration": {
        "CognitoPoolID": "Enter Your Cognito Identity Pool",
        "LexBotName": "Enter Your Lex Bot Name",
        "LexRole": "",
        "LexBotAlias": "Enter Your Lex Bot Alias",
        "BotRegion": "us-east-1"
    }
}

```

Legen Sie die Angaben für *CognitoPoolID*, *LexBotName* und *LexBotAlias* mit Ihren Daten fest. Das war es schon! Sie können über den Build-Vorgang in Visual Studio das Projekt kompilieren. Achten Sie darauf, dass das Projekt .NET Core 1.1 benötigt. Ist die Kompilierung abgeschlossen, so können Sie die Web-Applikation direkt aus Visual Studio heraus starten und testen. Bild 9.16 zeigt den Startbildschirm der Web-Applikation.

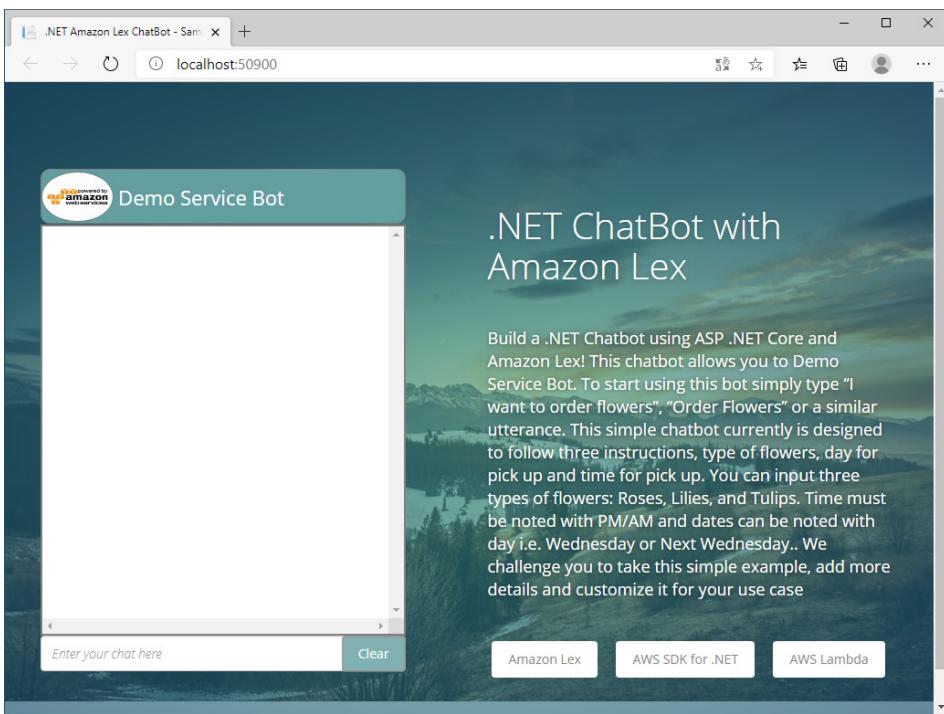


Bild 9.16 Der Bot in einer Web-Applikation

Sie können für die weitere Entwicklung die Amazon-Vorlage nutzen und entsprechend Ihrer eigenen Ideen anpassen.

Dieses Beispiel soll demonstrieren, dass Bots eine faszinierende Spielwiese sind. Der große Vorteil an der Umsetzung mit Amazon Lex ist, dass Sie keine entwicklungstechnischen Grundlagen und Vorkenntnisse über Machine Learning oder Deep Learning und keine komplexe IT-Infrastruktur benötigen, um einen Chatbot zu erstellen und in eigenen Anwendungen zu nutzen. Letztendlich ist es auch egal, für welchen Technologie-Stack Sie sich entscheiden, ob

Amazon, Microsoft oder Google. Die Grundprinzipien für die Entwicklung eines Bots gelten für alle drei Plattformen gleichermaßen.

■ 9.3 Azure Cognitive Services

Auch Microsoft stellt mit den Azure Cognitive Services eine Vielzahl von Cloud-Diensten mit REST APIs und Software Development Kits (SDKs) zur Verfügung, die Ihnen als Entwickler helfen sollen, Funktionen für das maschinelle Lernen in Ihren Anwendung zu implementieren, ohne tiefergehende KI- bzw. Data-Science-Kenntnisse besitzen zu müssen.

Das Ziel von Azure Cognitive Services ist es, Entwicklern zu helfen, Anwendungen zu erstellen, die sehen, hören, sprechen, verstehen und sogar schlussfolgern können. Dazu gehört zum Beispiel das Analysieren von Text auf emotionale Stimmungslagen oder das Analysieren von Bildern, um Objekte oder Gesichter zu erkennen.

Die Dienste werden innerhalb von Azure Cognitive Services in folgende fünf Hauptkategorien aufgeteilt:

- Bildanalyse
- Spracheingabe
- Sprache
- Websuche
- Entscheidungen

Unter Bildanalyse wird das Erkennen, Identifizieren und Untertiteln von Bildern zusammengefasst. Es gibt entsprechende APIs für maschinelles Sehen, Custom Vision und Gesichtserkennung (siehe auch Abschnitt 1.3.3, „Microsoft Cognitive Services“).

Bei der Spracheingabe konvertieren Sie Sprache in Text und Text in natürlich klingende Sprache. Auch das Übersetzen von einer Sprache in eine andere fällt in diesen Bereich. Die zur Verfügung gestellten APIs ermöglichen Ihnen, natürliche Sprache mit vordefinierten Skripts in Ihren eigenen Applikationen zu verarbeiten. Aber auch die Auswertung von Stimmungen anhand der Sprache fällt in diese Rubrik.

In der Kategorie „Websuche“ finden Sie die Bing Search API, die es Ihnen in einer Applikation erlaubt, Webseiten, Bilder, Videos und Nachrichten mit einem einzigen API-Aufruf zu durchsuchen. In der Kategorie „Entscheidungen“ finden Sie ML-Modelle für die Anomalie-Erkennung, Content Moderator und Personalisierung, die das Erstellen von Applikationen ermöglichen, die Empfehlungen geben, um fundierte und effiziente Entscheidungen treffen zu können.

Sie können die Azure Cognitive Services sehr einfach und effektiv kombinieren, um Ihre Anwendungen über verschiedene Services zu verketten. So wird in der Praxis bei Einsatz des Sprach-Service das gesprochene Wort in Text umgesetzt und mithilfe eines Übersetzungsservice in eine andere Sprache übersetzt, um Antworten in dieser Sprache aus einer Search-API zu bekommen. Für diesen Anwendungsfall werden drei Cognitive Services miteinander kombiniert, um eine entsprechende Lösung zu erstellen.

Ein weiteres Beispiel findet sich in der Logistik. So kann man per OCR (Optical Character Recognition) aus einem Bild (Lieferschein, Lagerschein usw.) den gedruckten Text extrahieren und als Text anzeigen bzw. den Text als Sprache ausgeben. Entsprechende Bild- und Sprach-Services für Systeme wie Pick By Voice können für die Intralogistik, Produktion oder Qualitätssicherung eingesetzt werden. Hierbei handelt es sich um eine sogenannte Sprachkommissionierung, die ein belegloses Kommissionierverfahren ermöglicht.

Neben den angebotenen Services ist es aber auch genauso wichtig, einen Cognitive Service auszuprobieren bzw. testen zu können. Die Website der Azure Cognitive Services [56] bietet eine Vielzahl von Beispielen, die auch einen Test mit entsprechenden Demodaten bereitstellen. So können zum Beispiel mit der Computer Vision API eigene Fotos hochgeladen werden und der Cognitive Service ist dann in der Lage, Personen, Tiere, Großstädte und Gegenstände aufgrund der unzähligen Testdaten zu erkennen.

9.3.1 Intelligente kontextbasierte Suchfunktion

Suchfunktionen sind heute in zahlreichen Applikationen oder Lösungen zu finden. Leider erfahren sie oftmals keine entsprechende Akzeptanz bei den Anwendern, da die Qualität der Resultate häufig zu wünschen übrig lässt.

Durch einen Cognitive Service wie die Bing-Websuche lässt sich die Qualität einer Suchabfrage erheblich verbessern. Zu den durchsuchbaren Inhalten gehören verschiedene Arten von News, Entitäten, Bilder und Videos. Durch die Verwendung der Bing Custom Search API (benutzerdefinierte Bing-Suche) können auch spezifische Suchroutinen wie zum Beispiel „Zeige nur News mit dem Inhalt <Deep Learning>“ implementiert werden, die dem Benutzer dann angezeigt werden.

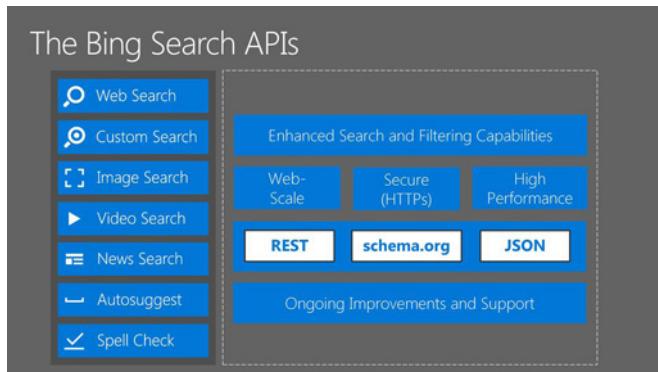
9.3.1.1 Bing-Websuche

Die von Cognitive Services zur Verfügung gestellten Bing Search APIs ermöglichen Ihnen das Erstellen einer Suchmaschine in Ihrer eigenen Applikation, die Webseiten, Bilder, Nachrichten, Orte und vieles mehr findet. Durch das Senden von Suchanforderungen mit den Bing Search APIs können Sie schnell und effektiv relevante Informationen und Inhalte für die Websuche abrufen. Durch die KI-Mechanismen der Cognitive Services, die über die APIs genutzt werden, ist die Qualität der Suchergebnisse, vor allem durch das Herausfiltern der Werbung, sehr gut.

Auch für den Einsatz der meisten Cognitive Services benötigen Sie wie bei den Amazon Services keine speziellen Kenntnisse in Machine oder Deep Learning. Die KI-Mechanismen werden im Hintergrund benutzt, ohne dass Sie als Entwickler damit direkt in Berührung kommen.

Die Cognitive Services Bing Search API sind in inhaltspezifische Bereiche aufgeteilt und geben somit nur bestimmte Inhalte aus dem Internet zurück. Dazu zählen vor allem Bilder, Nachrichten, lokale Unternehmen und Videos. Bild 9.17 zeigt eine Übersicht der einzelnen Bing Search APIs.

Sie können die einzelnen APIs aber auch in einer Applikation kombinieren, um so eine universelle Suchmaschine in Ihrer Anwendung zu implementieren.

**Bild 9.17**

Bing Search APIs
(Quelle: Microsoft)

Web Search

Bei der Bing Web Search API handelt es sich um eine sogenannte Entitäten-Suche. Die API gibt Suchergebnisse mit Entitäten auf Grundlage des gesuchten Begriffs zurück. Dies können Personen, Orte oder Dinge/Objekte sein.

Custom Search

Die benutzerdefinierte Suche (Custom Search) ermöglicht die Erstellung einer Suchfunktion, die ausschließlich die für Sie interessanten Inhalte und Themen umfasst. Hierzu müssen die gewünschten Domänen und Webseiten angegeben werden, die Bing durchsuchen soll. Sie können die Custom-Search-API auch in der Bild- und Videosuche integrieren, um das Suchergebnis anzupassen.

Image Search

Mit der Bildersuche (Image Search) können Sie nach qualitativ hochwertigen statischen und animierten Bildern suchen. Hier können Sie durch den Einsatz von Filtern die Suchvorgänge verfeinern, um Bilder nach Attributen wie Größe, Farbe usw. einzuschränken.

Video Search

Genau wie bei allen anderen Bing Search APIs handelt es sich auch bei der Videosuche (Video Search) um eine inhaltspezifische API. Sie finden über die Suche beliebte Videos, verwandte Inhalte und Miniaturansichten als Vorschau. Die API verfügt des Weiteren über einen Standortfilter und bietet die Möglichkeit, eine sogenannte sichere Suche zu aktivieren, die verhindert, für Kinder und Jugendliche ungeeignete Inhalte anzuzeigen.

News Search

Die Nachrichtensuche (News Search) ermöglicht das Auffinden von Nachrichten. Die API gibt Nachrichtenartikel aus mehreren Quellen oder von bestimmten Domänen zurück. Die API-Version 7 verfügt neuerdings über detaillierte Sortier- und Filteroptionen in Bezug auf Kategorien, Anbieterinformationen und Hinzufügedatum, die das Auffinden bestimmter Informationen vereinfachen.

Autosuggest

Bei der Autosuggest API handelt es sich um eine Vorschlagssuche, die eine Liste vorgeschlagener Abfragen, basierend auf der unvollständigen Zeichenfolge im Suchfeld, in Echtzeit zurückgibt. So ist es für den Anwender noch einfacher, nach dem gewünschten Ergebnis zu suchen. Sie erhöhen mit dieser API auch die Benutzerfreundlichkeit Ihrer Applikation.

Spell Check

Bei der Bing Spell Check API handelt es sich um eine Rechtschreibprüfung, die Grammatik und Rechtschreibung eines Textes überprüft. Hierbei verwendet die Bing API zur Prüfung Machine Learning und die statistische maschinelle Übersetzung, um präzise und kontextbezogene Korrekturen bereitzustellen.

Visual Search

Die Bing Visual Search API ist neu hinzugekommen und erlaubt die Suche mit einem Bild. Hierfür wird das Bild hochgeladen oder über eine URL zur Verfügung gestellt und Sie erhalten als Suchergebnis Webseiten, Weblinks oder ähnliche Bilder, auf denen der Inhalt des hochgeladenen Bildes vorkommt. Die Suche liefert nebenbei auch noch Verkaufspreise, wenn es sich bei dem Bildinhalt um ein identifizierbares Produkt handelt.

Die Bing Search API aus den Azure Cognitive Services können Sie über ein Software Development Kit (SDK) für .NET und C# verwenden. Dieses steht zum Download auf GitHub unter [57] zur Verfügung. Alternativ können Sie auch direkt über das Webportal von Azure eine Abfrage starten, oder Sie benutzen einfach die REST API.

9.3.1.2 Bing Suche über REST API

Am einfachsten lässt sich die Bing Search API zusammen mit Visual Studio verwenden. Als Datenaustausch-Format wird JSON verwendet und die Authentifizierung erfolgt über einen API-Schlüssel.

Sie benötigen zunächst aber ein Microsoft-Azure-Konto. Microsoft stellt verschiedene kostenlose Zugänge zum Testen von Cognitive Services zur Verfügung. Haben Sie das Azure-Konto eingerichtet, können Sie die Cognitive Services über das Azure-Portal konfigurieren (Bild 9.18).

Über den Startpunkt *KI + Machine Learning* können Sie jetzt die benötigte Ressource für die Bing Search API erstellen. Sie erhalten als Ergebnis einen API-Schlüssel, der für die Nutzung der jeweiligen API notwendig ist. Im Fall der Bing-Suche gilt dieser Schlüssel für alle Bing Search APIs, sodass diese in einer Anwendung auch kombiniert werden können.

Sind alle Vorbereitungen abgeschlossen, können Sie ganz einfach eine WPF-Applikation als Suchmaschine erstellen, die über die universelle REST-Schnittstelle (Representational State Transfer) auf die API zugreift.

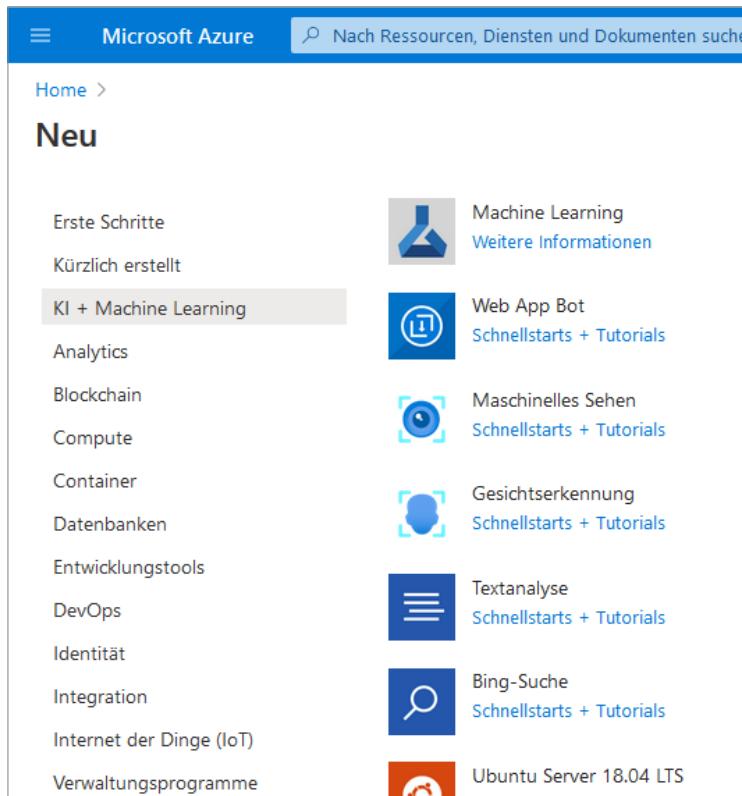


Bild 9.18 Startpunkt für Cognitive Services

9.3.1.3 Die eigene Suchmaschine

Für die Umsetzung der eigenen Suchmaschine erstellen Sie eine WPF-Applikation mit Visual Studio (Bild 9.19). Die WPF-Oberfläche bildet mit der *MainWindow.xaml*-Datei das Programm für die Implementierung des benötigten Codes.

Als Projektname wird *BingWebSearchDemo* verwendet. Erstellen Sie die Oberfläche über den XAML-Editor des *MainWindow*. Nutzen Sie hierfür den Code aus Listing 9.5 für die *Xaml*-Datei. Alternativ können Sie die Oberfläche aber auch ganz nach Ihren eigenen Vorstellungen umsetzen.

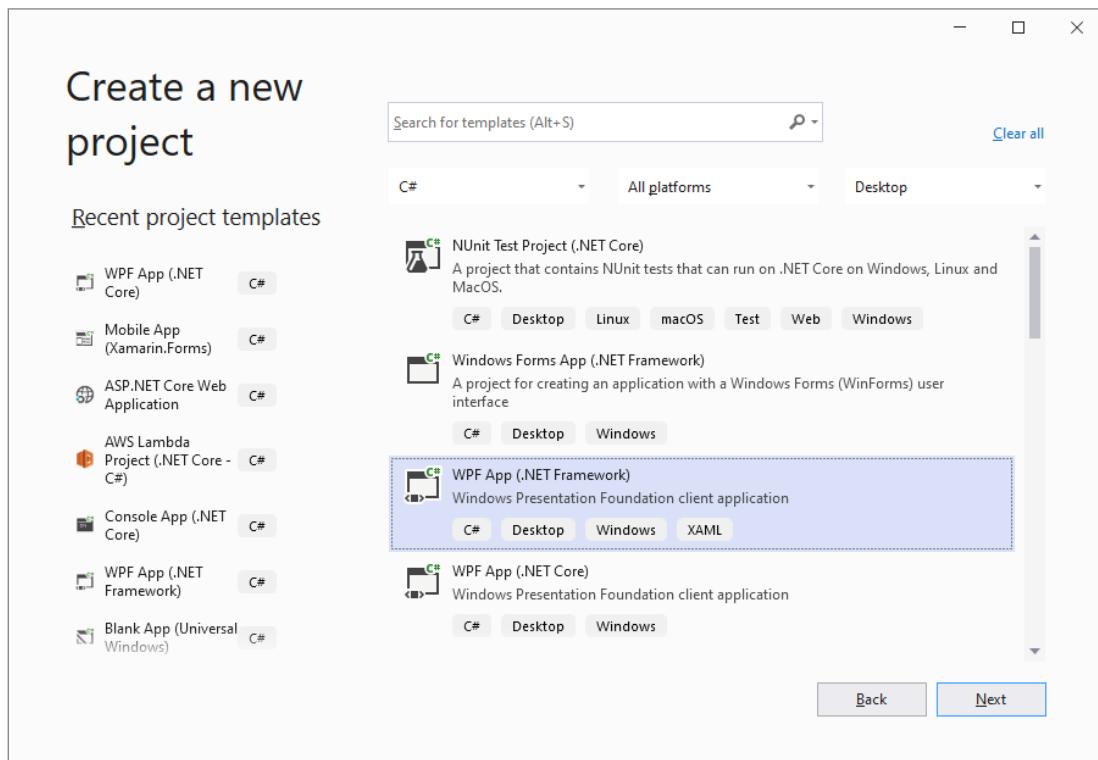


Bild 9.19 Auswahl der passenden Projektvorlage

Listing 9.5 Aufbau der Benutzeroberfläche

```

<Window x:Class="BingWebSearchDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:BingWebSearchDemo"
        mc:Ignorable="d"
        Title="Cognitive Service mit Bing" Height="450" Width="800">
    <Grid>
        <StackPanel>
            <Label Content="Web Suche...." />
            <TextBox Name="tbSearchText" TextChanged="tbSearchText_TextChanged">
                </TextBox>
            <ListView Name="lstView" Margin="0,10">
                <ListView.View>
                    <GridView>
                        <GridView.Columns>
                            <GridViewColumn Header="Beschreibung" Width="350"
                                           DisplayMemberBinding="{Binding Snippet}" />
                            <GridViewColumn Header="Name" Width="300"
                                           DisplayMemberBinding="{Binding Name}" />
                        </GridView.Columns>
                    </GridView>
                </ListView.View>
            </ListView>
        </StackPanel>
    </Grid>

```

```

        </GridView.Columns>
    </GridView>
</ListView.View>
</ListView>
</StackPanel>
</Grid>
</Window>

```

REST APIs, die über HTTP- oder HTTPS-Protokolle verfügbar sind, nutzen JSON oder XML für die Datenformatierung. Da Azure bei der Bing Search API den Wert als JSON-Objekt zurückgibt, müssen Sie noch über den NuGet Manager von Visual Studio dem Projekt das Newtonsoft-JSON-Paket hinzufügen (Bild 9.20).

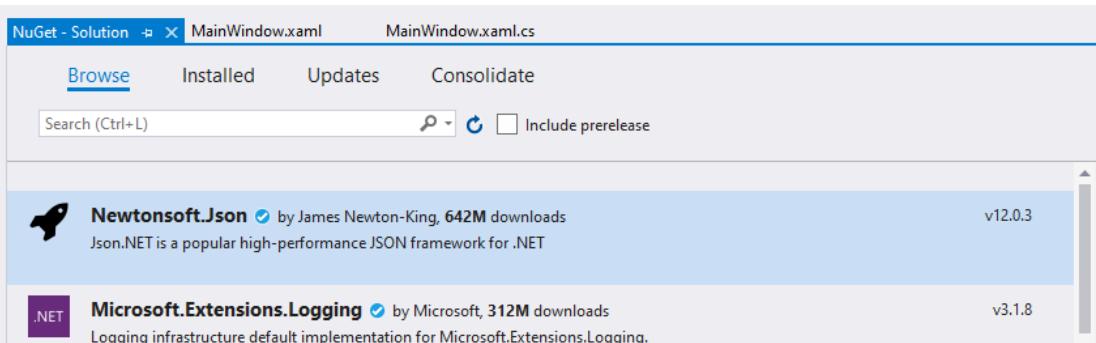


Bild 9.20 Hinzufügen von JSON zum Gesamtprojekt

Die Implementierung des Beispielcodes erfolgt in der Code-Behind-Datei. Da die Bing-Suche im JSON-Objekt eine ID, einen Namen und ein Snippet – in diesem Fall in Form einer Beschreibung – zurückgibt, wird das über eine entsprechende Modellklasse mit dem Namen *WebSearch* gelöst. Listing 9.6 zeigt die komplette Implementierung der Suchmaschine.

Listing 9.6 Der Code für die MainWindow.xaml.cs

```

using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;

namespace BingWebSearchDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window

```

```
{  
    private string WebSearchEndPoint = "https://api.cognitive.microsoft.com/bing/  
v7.0/search?";  
    public MainWindow()  
{  
        InitializeComponent();  
        WebSearchClient = new HttpClient();  
        WebSearchClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key",  
                                         "Ihr API-Schlüssel");  
    }  
  
    public HttpClient WebSearchClient  
{  
    get;  
    set;  
}  
  
    async Task<IEnumerable<WebSearch>> SearchBingWeb(string searchText)  
{  
        List<WebSearch> websearch = new List<WebSearch>();  
  
        string count = "20";  
        string offset = "0";  
        string mkt = "en-us";  
        var result = await RequestAndAutoRetryWhenThrottled(() =>  
            WebSearchClient.GetAsync(string.Format  
                ("{0}q={1}&count={2}&offset={3}&mkt={4}", WebSearchEndPoint,  
                 WebUtility.UrlEncode(searchText), count, offset, mkt)));  
        result.EnsureSuccessStatusCode();  
        var json = await result.Content.ReadAsStringAsync();  
        dynamic data = JObject.Parse(json);  
        for (int i = 0; i < 20; i++)  
        {  
            websearch.Add(new WebSearch  
            {  
                Id = data.webPages.value[i].id,  
                Name = data.webPages.value[i].name,  
                Snippet = data.webPages.value[i].snippet  
            });  
        }  
        return websearch;  
    }  
  
    private async Task<HttpResponseMessage> RequestAndAutoRetryWhenThrottled  
        (Func<Task<HttpResponseMessage>> action)  
{  
        int retriesLeft = 6;  
        int delay = 500;  
  
        HttpResponseMessage response = null;  
  
        while (true)  
        {  
            response = await action();  
        }  
    }  
}
```

```

        if ((int)response.StatusCode == 429 && retriesLeft > 0)
        {
            await Task.Delay(delay);
            retriesLeft--;
            delay *= 2;
            continue;
        }
    else
    {
        break;
    }
}

return response;
}

private async void tbSearchText_TextChanged(object sender,
                                            TextChangedEventArgs e)
{
    try
    {
        if (tbSearchText.Text != null)
            lstView.ItemsSource = await SearchBingWeb(tbSearchText.Text);
    }
    catch (Exception ex)
    {
        //ToDo
    }
}
public class WebSearch
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Snippet { get; set; }

}
}
}

```

Als Erstes wird der Endpunkt über die String-Variablen *WebSearchEndPoint* festgelegt. Endpunkte sind die vollen URIs (Uniform Resource Identifier), aus denen sich die jeweilige REST API zusammensetzt. Die Summe aller Endpunkte ist die API und der einzelne Endpunkt spricht genau eine Ressource an.

Da im Beispiel REST verwendet wird, können Sie einfach eine GET-Anfrage an den angegebenen Endpunkt senden. Diese Funktionalität stellt Ihnen die *HttpClient*-Basisklasse von .NET bereit, die HTTP-Anfragen und -Antworten über die URL (Universal Resource Locator) erhalten bzw. posten kann. Damit Sie die Get-Anfrage für REST verwenden können, müssen Sie im Beispiel Ihren API-Key aus der Bing-Search-API-Ressource eintragen.

Da es bei synchronen Anfragen über HTTP bzw. HTTPS zum Blockieren der Ausführung anderer Skripte kommen kann, wird das Interagieren mit der API-Ressource asynchron umgesetzt. Der *HttpClient* ist auch problemlos in der Lage, mehrere Anfragen gleichzeitig zu verarbeiten.

Der Task *SearchBingWeb* zeigt, wie Sie alle JSON-Daten mithilfe der Bing-Search-API-URL dynamisch parsen können. Über die Variable *count* wird die Anzahl der Suchergebnisse eingeschränkt. Im Beispiel erhalten Sie somit nur eine Ergebnismenge mit 20 Einträgen. Diese werden in der Liste *websearch* zurückgegeben und in der *ListView* in der Benutzeroberfläche angezeigt.

Schon ist die Beispiel-Suchmaschine auf Basis der Bing Search API fertig. Nach dem Kompilieren und dem Eintrag „Deep Learning“ im Textfeld erhalten Sie die entsprechenden Suchergebnisse aus Bild 9.21.

Beschreibung	Name
Deep learning is a class of machine learning algorithms that (pp199–200) uses multiple layers of processing units.	Deep learning - Wikipedia
Deep learning is a type of machine learning that trains a computer to perform human-like tasks.	What is deep learning? SAS
Deep learning is a subset of machine learning, a branch of artificial intelligence that confers human-like visual perception, speech recognition, and decision-making ability.	What Is Deep Learning? PCMag
Deep learning is an AI function that mimics the workings of the human brain in processing unstructured data.	Deep Learning Definition - investopedia.com
Deep learning engineers are highly sought after, and mastering deep learning will give you a competitive edge in the job market.	Deep Learning by deeplearning.ai Coursera
Deep learning has taken the world of technology by storm since the beginning of the decade.	Deep Learning (Adaptive Computation and Machine Learning ...)
Deep learning (deep neural networking): Deep learning is an aspect of artificial intelligence that uses multiple layers of processing units.	What is Deep Learning and How Does it Work?
The demand for Deep Learning skills by employers -- and the job salaries of Deep Learning professionals -- are on the rise.	Deep Learning Professional Certificate edX
The Deep Learning textbook is a resource intended to help students and practitioners learn about deep learning.	Deep Learning
Deep learning super sampling (DLSS) is a technology developed by Nvidia, using deep learning to improve the performance of games.	Deep learning super sampling - Wikipedia
Deep Learning is Large Neural Networks. Andrew Ng from Coursera and Chief Scientist at Google.	What is Deep Learning?
Deep learning engineers are highly sought after, and mastering deep learning will give you a competitive edge in the job market.	Neural Networks and Deep Learning Coursera
"The analogy to deep learning is that the rocket engine is the deep learning models and the fuel is the data."	Deep learning vs machine learning - Zendesk
With just a few lines of MATLAB® code, you can apply deep learning techniques to your own data.	MATLAB for Deep Learning - MATLAB & Simulink
Deep learning defined. Deep learning is a form of machine learning that models patterns of data using multiple processing layers.	What is deep learning? Algorithms that mimic the human brain
Deep learning is a machine learning technique that teaches computers to do what comes naturally.	What Is Deep Learning? How It Works, Techniques ...
Deep Learning is widely used today for Data Science, Data analysis, machine learning, AI, Preventing disease. Building smart cities. Revolutionizing analytics. These are just a few things that Deep Learning can do.	20 Best Books on Deep Learning (2020 Review) - Best Books Hub
Deep Learning, as a branch of Machine Learning, employs algorithms to process data and extract meaningful insights.	Deep Learning & Artificial Intelligence Solutions from NVIDIA
Deep Learning Deep learning is a subset of AI and machine learning that uses multi-layered neural networks.	A Brief History of Deep Learning - DATAVERSITY
Deep Learning Deep learning is a subset of AI and machine learning that uses multi-layered neural networks.	Deep Learning NVIDIA Developer

Bild 9.21 Die fertige Suchmaschine im Einsatz

Auch dieses Beispiel zeigt, wie einfach es sein kann, Cognitive Services in einer Applikation zu verwenden. Dies funktioniert auch problemlos mit den weiteren REST APIs von Azure, so zum Beispiel für die Bereiche Gesichtserkennung oder maschinelles Sehen.

Schauen Sie sich die für Ihr Projekt infrage kommenden kognitiven APIs an und bringen Sie etwas KI in Ihre Business-Anwendung.

■ 9.4 Azure Machine Learning Studio

Das Microsoft Azure Machine Learning Studio, im Folgenden kurz nur noch als Studio bezeichnet, ist ein Webportal für Data Scientists und Entwickler im Bereich Azure Machine Learning. Es basiert auf Microsoft ML Studio aus dem Jahr 2015 und wird stetig weiterentwickelt.

Studio bietet abhängig vom Projekttyp und von der Erfahrung des Benutzers unterschiedliche Arbeitsumgebungen an. Der erfahrene Anwender kann direkt seinen eigenen Python-Code, alternativ auch R-Code, erstellen und diesen über einen verwaltbaren *Jupyter Notebook-Server* ausführen. Einsteiger können einen interaktiven Designer zum Trainieren und Bereitstellen von Machine Learning-Modellen nutzen, ohne Code schreiben zu müssen. Des Weiteren können sie Ihre Modelle, Datasets, die Datenspeicherung und vieles mehr, was Sie für die Entwicklung Ihres Machine-Learning-Projekts benötigen, direkt im Browser verwalten.

Das heißt, Studio unterstützt Drag-and-drop in der Web-Oberfläche, Code SDKs für Python und R, das Erstellen von flexiblen und modularen Pipelines zum Automatisieren von Workflows sowie das automatische Modelltraining und das Optimieren von Hyperparametern sowohl als Code First- wie auch als codefreie Option.

Für die Arbeit mit Studio benötigen Sie ein Azure-Konto. Auch hier reicht zum Experimentieren und Evaluieren ein Gratis-Account aus.

9.4.1 Arbeitsbereich

Um mit Studio zu arbeiten, müssen Sie als Erstes einen Arbeitsbereich erstellen. Hierbei handelt es sich um eine von Microsoft zur Verfügung gestellte Cloud-Ressource zum Experimentieren, Trainieren und Bereitstellen von Machine-Learning-Modellen.

Um einen Arbeitsbereich in Studio zu erstellen, gehen Sie folgendermaßen vor: Melden Sie sich mit Ihrem Konto am Azure-Portal an. Wählen Sie den Dienst *KI + Machine Learning* aus und markieren Sie in der Auswahl *Machine Learning Studio-Arbeitsbereiche*. Klicken Sie hier auf *Erstellen* (Bild 9.22).

Legen Sie auf der nachfolgenden Dialogseite die benötigten Informationen an. Der Arbeitsbereichsname identifiziert Ihren Arbeitsbereich. Tragen Sie hier einen eindeutigen Namen ein. Im Beispiel wird „Demo-ML“ verwendet. Unter *Abonnement* wählen Sie das passende Azure-Abonnement aus.

Unter *Ressourcengruppe* legen Sie eine neue Ressourcengruppe fest, oder Sie wählen eine vorhandene Gruppe aus. Eine Ressourcengruppe enthält verwandte Ressourcen für eine Azure-Lösung. Im Beispiel wird eine neue Gruppe (*Neues Element erstellen*) mit dem Namen *Demo-Aml* erstellt. Als Standort legen Sie *Europa, Westen* fest.

Erstellen Sie für den Arbeitsbereich ein neues Speicherkonto und wählen Sie als Arbeitsbereichstarif *Standard* aus. Hiermit legen Sie bestimmte Features fest, auf die Sie Zugriff haben. Wählen Sie zum Schluss den gewünschten Webdiensttarif aus. Bild 9.23 zeigt die gemachten Einstellungen für den Demo-Arbeitsbereich. Klicken Sie auf den Button *Erstellen*, um den definierten Arbeitsbereich zu erzeugen.

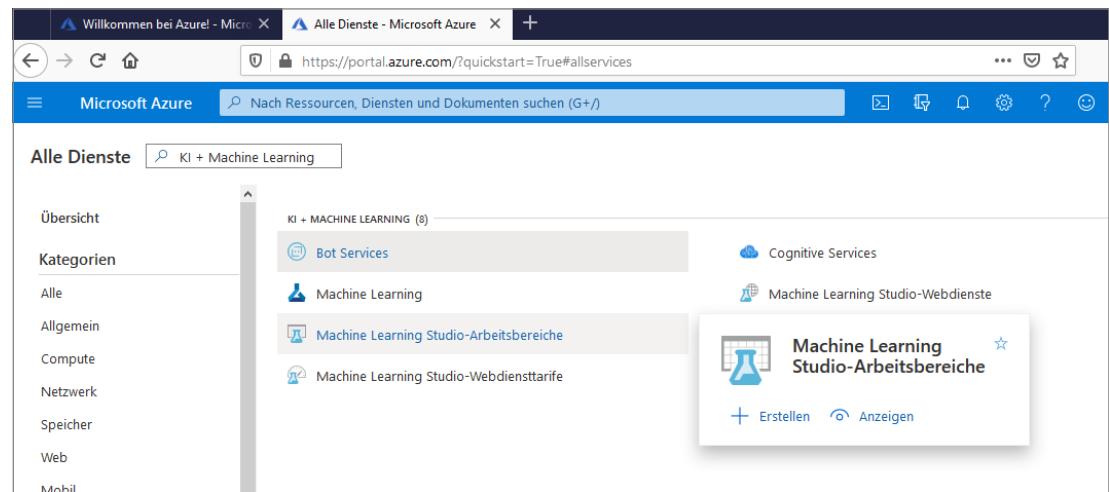


Bild 9.22 Das Erstellen eines Arbeitsbereiches auswählen

Machine Learning Studio-Arbeitsbereich

Arbeitsbereichsname *

 ✓

Abonnement *

 ▾

Ressourcengruppe *

 ▾

Neues Element erstellen

Standort *

 ▾

Speicherkonto *

Neu erstellen

 ✓

Arbeitsbereichstarif

 ▾

Webdiensttarif *

Neu erstellen

 ✓

*Webdiensttarif

DevTest Standard

Erstellen Automatisierungsoptionen

Bild 9.23
Festlegen des Arbeitsbereichs

Sie kehren nun zum Dashboard zurück und erhalten eine Übersicht Ihres Arbeitsbereichs. Hier können Sie dann über den Link *Azure Machine Learning Studio* starten. Als weitere Optionen stehen Ihnen noch der Machine-Learning-Katalog und die Webdienstverwaltung für das Studio zur Verfügung.

Wählen Sie den Link *Machine Learning Studio starten* und klicken Sie bei *Welcome to Azure Machine Learning Studio* auf *SignIn* (Anmelden). Hier können Sie jetzt über die Browser-Oberfläche Ihr Projekt, Experimente und Datenquellen anlegen bzw. verwalten. Über den Menüeintrag *Experiments* und einen Klick auf *+NEW* wird Ihnen eine Reihe von Beispielen vorgeschlagen, die Sie für Ihre Zwecke testen und anpassen können (Bild 9.24).

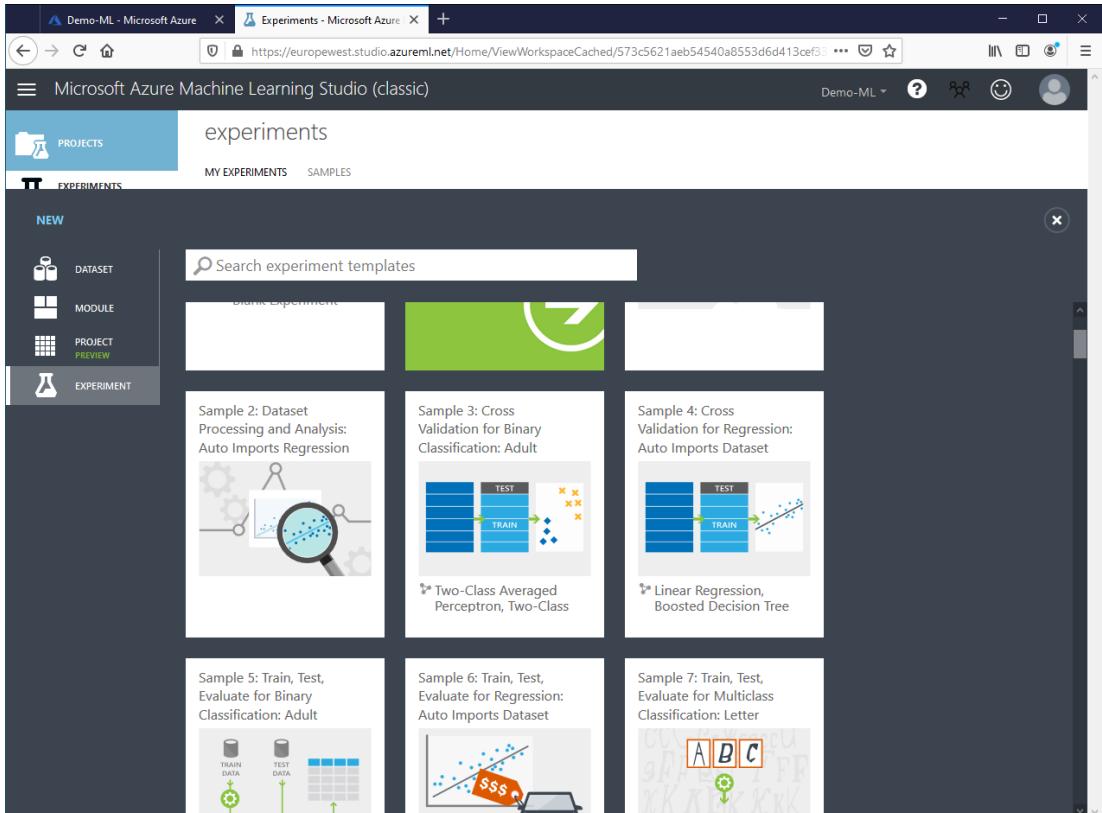


Bild 9.24 Mögliche Beispiele im Azure Machine Learning Studio

Die Eingabedaten und Modelle in den jeweiligen Beispielen kann man in der Browser-Oberfläche durch Drag-and-drop zusammenstellen bzw. bereinigen und anpassen. Wie Sie sehen, stehen Ihnen neben logistischer Regression, Entscheidungsbäumen, Support Vector Machines und neuronalen Netzen eine Vielzahl von Analysemethoden und Algorithmen für die Erstellung von ML-Modellen zur Verfügung. Bild 9.25 zeigt ein Experiment in der Browser-Ansicht.

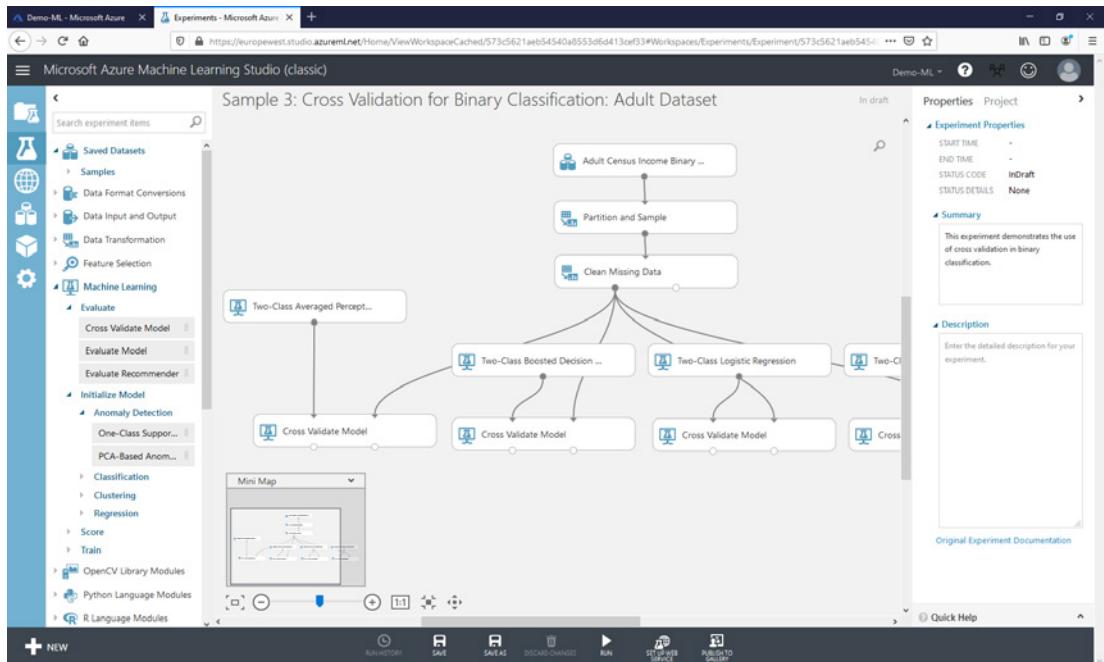


Bild 9.25 Erstellen eines ML-Modells im Azure Machine Learning Studio

Das Studio verfügt über viele Funktionen und Optionen und man benötigt eine gewisse Einarbeitungszeit, um nicht von den Möglichkeiten erschlagen zu werden. Auch sollte man sich bei der Nutzung auf jeden Fall mit maschinellem Lernen auskennen und über einige Programmierkenntnisse in Python oder R verfügen. Die Dokumentation hilft Ihnen über die eine oder andere Hürde hinweg und bietet einen verständlichen Einstieg in das Studio.

Sie haben mit dem Studio vielfältige Möglichkeiten, Modelle zu trainieren, zu evaluieren und bereitzustellen. Sie können das Modell dann über eine entsprechende API nutzen, um es dem Anwender passend zur Verfügung zu stellen. Das Machine Learning Studio stellt somit eine gute Schnittstelle für die Modellierung von ML-Modellen und den Einsatz des fertigen ML-Modells in einer Anwendung dar.

10

Anwendungen entwerfen

Nachdem Sie die Grundlagen neuronaler Netze, von Machine Learning Frameworks und ML-Services kennen gelernt haben, werden in diesem Kapitel einige Beispiele gezeigt, die auf einfache Weise demonstrieren, wie ML-Algorithmen und neuronale Netze eingesetzt werden können. Sie beginnen mit einer einfachen Prognose (Predictive Analytics). Anschließend werden Beispiele für Bildklassifikation und Textanalyse gezeigt. Zum Einsatz kommen sowohl C# ohne weitere ML-Frameworks als auch ML.NET und Microsoft Cognitive Services für die Textanalyse.

Abgesehen von den hier gezeigten Beispielen können Sie mit ML-Algorithmen oder neuronalen Netzen überall dort neue Lösungen entwickeln, wo andere Analysewerkzeuge nicht weiterführen oder nicht zur Verfügung stehen. Bei der Entwicklung neuronaler Netze sollten Sie einige grundlegende Aspekte berücksichtigen, um eine langfristige Sicherheit für das System zu gewährleisten. Dazu zählen sowohl die Systemarchitektur und -konzeption bis hin zur Integrationsfähigkeit in eine bestehende Infrastruktur und die Update-Möglichkeiten auf künftige Technologien. Beachten Sie also auch beim Erstellen von ML-Applikationen unbedingt Ihren Softwareentwicklungsprozess.

■ 10.1 Predictive Analytics

Predictive Analytics beschäftigt sich insbesondere mit dem Erkennen von Mustern und der Vorhersage künftiger Ereignisse. Im Allgemeinen nutzt Predictive Analytics historische Datenquellen, um ein mathematisches Modell zu erstellen, mit dem sich zukünftige Ereignisse vorhersagen lassen. Das erzeugte Modell soll somit Trends oder Muster in den historischen Daten erkennen und diese für die Zukunft vorausberechnen.

Hieraus ergeben sich Anwendungsfälle wie Kurs-, Absatz- und Kostenprognosen, aber auch das sogenannte *Predictive Policing* zur Berechnung der Wahrscheinlichkeit zukünftiger Straftaten sowie die Risikobewertung bei Kreditkarteneinsätzen.

Predictive Analytics wird auch eingesetzt, um Zeit in Prozessen zu sparen oder die Verschwendungen von Ressourcen einzudämmen. Die wichtigsten Bereiche für Predictive Analytics sind derzeit:

- **Markforschung:** Die Identifikation spezifischer Zielgruppen aus Kundendaten und deren Präferenzen für Produkte, Dienstleistungen oder Werbemaßnahmen.

- **Finanzdienstleister und Versicherungen:** Entwicklung von Kreditrisiko- und Versicherungsmodellen.
- **Maschinenbau und Automation:** Prognose von Maschinenausfällen (siehe Abschnitt 3.5, „Predictive Maintenance“).

Predictive Analytics hat in den letzten Jahren viel Aufmerksamkeit erhalten, da durch den Einsatz von Machine-Learning-Algorithmen große Fortschritte erzielt wurden, inzwischen spricht man auch allgemein von Predictive Computing.

10.1.1 Fallbeispiel: Energiebranche

Ein weiterer bekannter Anwendungsbereich für Predictive Analytics ist das intelligente Stromnetz der Zukunft, das in den Medien auch unter dem Begriff *Smart Grid* zu finden ist.

Hierbei möchte man den Stromverbrauch anhand von gespeicherten Verhaltensmustern der Kunden vorhersagen, um die benötigte Einspeisung von Wind- und Wasserkraftenergie exakt zu regulieren. Der Stromverbrauch ist in Deutschland seit Beginn der 1990er-Jahre im Trend immer gestiegen. Daher lohnt sich eine Prognose auf Grundlage historischer Stromverbrauchsdaten.

10.1.2 Zeitreihenanalyse

Zeitreihenanalyse gibt es in zahlreichen Anwendungen. Im Kern geht es darum, in vorgebestimmten Sets von Daten Muster zu finden, die ein tiefergehendes Verständnis von vergangenen Ereignissen oder von vergangenen Verhaltensweisen von Kunden erlaubt. Diese Erkenntnisse werden dann als Grundlage für die Simulation des zukünftigen Verhaltens verwendet. Auch hier gilt, je differenzierter ein Prognosemodell arbeitet, desto genauer kann es Vorhersagen treffen.

Unser Modell zum Stromverbrauch für eine fiktive Wohnstraße soll das Konzept und die Funktionsweise der Zeitreihenanalyse veranschaulichen. Das Beispiel soll anhand der Daten aus den letzten vier Jahren den Stromverbrauch prognostizieren. Die für das Training verwendeten Daten stehen Ihnen wieder unter GitHub zur Verfügung.

Der Trainingsdatensatz ist eine Zeitreihe über den Zeitraum von 2014 bis 2018 mit einem Merkmal (Label) über den Verbrauch mit einer Granularität von einer Stunde. Ein Auszug ist in der Tabelle *Trainingsdaten* (Bild 10.1) aufgelistet.

Bei einer komplexen Zeitreihenanalyse bietet sich ein rekurrentes neuronales Netz zur Vorhersage an. Rekurrente Netze bzw. Long Short-Term Memory (LSTM), siehe Abschnitt 5.4, machen während des Trainings jede Berechnung zum Zeitpunkt t abhängig vom vorherigen Ergebnis ($t - 1$), wodurch sich periodische Merkmale effektiv lernen lassen.

Da in unserem Beispiel im Trainingsdatensatz nur ein bestimmter Zeitstempel und ein Merkmalswert existiert, ist für diesen Fall ein Machine-Learning-Algorithmus effektiver und aufgrund der einfacheren Implementierung besser geeignet.

	A	B
1	Datum	Verbrauch
2	31.12.2014 01:00	11633.0
3	31.12.2014 02:00	11139.0
4	31.12.2014 03:00	10871.0
5	31.12.2014 04:00	10735.0
6	31.12.2014 05:00	10714.0
7	31.12.2014 06:00	10886.0
8	31.12.2014 07:00	11404.0
9	31.12.2014 08:00	12098.0
10	31.12.2014 09:00	12409.0
11	31.12.2014 10:00	12526.0
12	31.12.2014 11:00	12620.0
13	31.12.2014 12:00	12672.0
14	31.12.2014 13:00	12608.0
15	31.12.2014 14:00	12441.0
16	31.12.2014 15:00	12281.0
17	31.12.2014 16:00	12123.0
18	31.12.2014 17:00	12081.0
19	31.12.2014 18:00	12851.0

Bild 10.1

Ausschnitt der Trainingsdaten

Microsoft bietet über das ML.NET Framework den passenden Time-Series-Algorithmus für das Stromverbrauchsbeispiel an. Da Sie nur eine Merkmalsvariable nutzen, können Sie die Vorhersage über einen sogenannten univariaten Zeitreihenanalyse-Algorithmus prognostizieren. Die univariate Analyse (Wahrscheinlichkeitsverteilung) ist ein statistisches Verfahren, bei dem die Merkmalsausprägung einer einzelnen Variable betrachtet wird. In Verbindung mit der Zeitreihenanalyse wird somit nur eine einzelne numerische Beobachtung über einen bestimmten Zeitraum in definierten Intervallen durchgeführt.

Das ML.NET Framework enthält in seinem Algorithmenkatalog für die Prognose die *ForecastBySsa*-Methode, die auf der *Singular Spectrum Analysis*, kurz SSA, basiert. Der SSA-Algorithmus zielt darauf ab, die ursprüngliche Reihe in eine kleinere Anzahl interpretierbarer Komponenten wie Trend, Rauschen oder Saisonalität zu zerlegen. Das heißt, der Algorithmus basiert auf der *Singulärwertzerlegung* einer spezifischen Matrix und zerlegt eine Zeitreihe in ihre Hauptkomponenten. Anschließend werden diese Komponenten rekonstruiert und verwendet, um Werte in der Zukunft vorherzusagen und somit eine sehr genaue Prognose zu erstellen. Dieses Verhalten ermöglicht eine breite Anwendbarkeit des SSA-Algorithmus bei Zeitreihenanalysen.

Zum Auswerten nutzen Sie dann einfach die *TimeSeriesEngine*. Diese erstellt eine Vorhersage-Engine für eine Zeitreihen-Pipeline. Sie aktualisiert den Zustand des Zeitreihenmodells mit Beobachtungen, die in der Vorhersagephase gesetzt wurden, und ermöglicht die Kontrolle des Modells.

10.1.3 Beispielprogramm und Anwendung der Prognose

Für die Umsetzung der Energie-Prognose erstellen Sie einfach eine .NET-Core-Konsolenanwendung mit C#. Starten Sie Visual Studio und wählen Sie über *Create a new project* die Vorlage *Console App (.NET Core)* aus und vergeben Sie den Namen *DemoTimeSeriesForecast*.

Da Sie den ML-Algorithmus aus dem ML.NET Framework nutzen, müssen Sie noch über den NuGet-Manager von Visual Studio dem Projekt die Pakete *Microsoft.ML* und *Microsoft.ML.TimeSeries* hinzufügen.

Die Implementierung des Beispielcodes erfolgt in der *Program.cs*-Datei. Erweitern Sie als Erstes die Datei, um die folgenden *using*-Anweisungen.

```
using System;
using Microsoft.ML.Data;
using Microsoft.ML;
using Microsoft.ML.Transforms.TimeSeries;
```

Als Nächstes erstellen Sie die *ModelInput*-Klasse für die Übernahme der Daten aus der CSV-Datei. Fügen Sie unter der *Program*-Klasse den folgenden Code hinzu.

```
public class ModelInput
{
    [LoadColumn(0)]
    public DateTime Date { get; set; }

    [LoadColumn(1)]
    public float EnergyDemand { get; set; }
}
```

Die *ModelInput*-Klasse enthält die Spalten *Date* für den codierten Zeitstempel der Beobachtung und *EnergyDemand* als Gesamtzahl des Energieverbrauchs in der Stunde. Des Weiteren benötigen Sie noch eine *ModelOutput*-Klasse, welche die vorhergesagten Werte für den Vorphersagezeitraum enthält.

```
public class ModelOutput
{
    public float[] ForecastedEnergy { get; set; }
}
```

In der Main-Methode der Klasse *Program* können Sie die *context*-Variable mit einer neuen Instanz von *MLContext* initialisieren.

```
var context = new MLContext();
```

Die *MLContext*-Klasse ist der Ausgangspunkt für alle ML.NET-Vorgänge. Durch das Initialisieren von *MLContext* wird eine neue ML.NET-Umgebung erstellt, die für mehrere Objekte des Modellerstellungsworflows verwendet werden kann.

Die Trainingsdaten des Typs *ModelInput* werden über die Methode *LoadFromTextFile* der Klasse *ML.Data.TextLoader* geladen.

```
var data = context.Data.LoadFromTextFile<ModelInput>($"C:/temp/Zeitdaten.csv",
    hasHeader: true, separatorChar: ';');
```

Das so erstellte Dataset enthält alle Daten aus der vorgegebenen CSV-Datei.

10.1.4 Definieren der Pipeline

Nach dem Laden der Trainingsdaten definieren Sie im Code die benötigte Pipeline, die *SsaForecastingEstimator* verwendet, um Werte in einem Zeitreihendataset zu prognostizieren.



SsaForecastingEstimator Class

Diese Klasse wird über die Methode *ForecastBySsa* erstellt und implementiert die allgemeine Anomalie-Erkennungstransformation auf der Grundlage der Analyse des Einzelspektrums (SSA).

Es gibt hier nur eine Eingabespalte. Diese muss den Wert *Single* haben, wobei *Single* einen Wert zu einem Zeitstempel in der Zeitreihe angibt. Der Algorithmus erzeugt entweder nur einen Vektor der prognostizierten Werte oder drei Vektoren: einen Vektor der vorhergesagten Werte, einen Vektor der unteren Grenze des Konfidenzintervalls [58] und einen Vektor der oberen Grenze des Konfidenzintervalls.

Die Pipeline wird in der Main-Methode wie folgt implementiert.

```
var pipeline = context.Forecasting.ForecastBySsa(
    nameof(ModelOutput.ForecastedEnergy),
    nameof(ModelInput.EnergyDemand),
    windowSize: 7,
    seriesLength: 30,
    trainSize: 365,
    horizon: 4);
```

Die erstellte Pipeline nimmt für das erste Jahr 365 Datenpunkte (*trainSize*) an und teilt ein Zeitreihendataset in Stichproben von jeweils 30 Tagen (monatlich) auf, wie vom *seriesLength*-Parameter angegeben. Jede dieser Stichproben wird anhand eines 7-tägigen Fensters (*windowSize*) analysiert. Bei der Ermittlung des prognostizierten Wertes für die nächste Zeitspanne werden die Werte der letzten sieben Tage verwendet. Das Modell wird so festgelegt, dass vier Zeitspannen in der Zukunft vorhergesagt werden, wie durch den *horizon*-Parameter definiert.

Das Ergebnis wird aus den Werten der verwendeten Datenbasis gebildet, weshalb die Prognose nicht immer zu 100 % genau sein kann. Als Nächstes verwenden Sie die *Fit*-Methode, um das Modell zu trainieren.

```
var model = pipeline.Fit(data);
```

Um jetzt eine Prognose zu treffen, erstellen Sie über das Modell eine *TimeSeriesPredictionEngine*.

```
var forecastingEngine = model.CreateTimeSeriesEngine<ModelInput, ModelOutput>(context);
```

Nun können Sie über die Methode *Predict* der *PredictionEngine* den Energieverbrauch der nächsten 4 Tage prognostizieren.

```
var forecasts = forecastingEngine.Predict();
```

Die Anzeige der Werte auf der Konsole wird durch das Iterieren in der Vorhersage durchgeführt.

```
Console.WriteLine("Energie-Prognose");
Console.WriteLine("-----");
foreach (var forecast in forecasts.ForecastedEnergy)
{
    Console.WriteLine(forecast);
}
```

Fertig ist das Beispiel für die Prognose des Energieverbrauchs. Sie können dieses Beispiel jetzt noch erweitern und verbessern, so zum Beispiel durch Änderung der Parameter in der Pipeline. Sie können das Modell speichern und es in einer anderen Applikation oder in einer Web-App verwenden. Die Demo-Anwendung zeigt, wie schnell man mit dem ML.NET Framework ein lauffähiges Machine-Learning-Zeitreihenmodell erstellen kann. Listing 10.1 zeigt den kompletten C#-Code für das Erstellen des Modells.

Listing 10.1 DemoTimeSeriesForecast

```
using System;
using Microsoft.ML.Data;
using Microsoft.ML;
using Microsoft.ML.Transforms.TimeSeries;

namespace DemoTimeSeriesForecast
{
    class Program
    {
        static void Main(string[] args)
        {
            var context = new MLContext();

            var data = context.Data.LoadFromTextFile<ModelInput>
                ("C:/temp/Zeitdaten.csv",
                 hasHeader: true, separatorChar: ';');

            var pipeline = context.Forecasting.ForecastBySsa(
                nameof(ModelOutput.ForecastedEnergy),
                nameof(ModelInput.EnergyDemand),
                windowSize: 7,
```

```
seriesLength: 30,  
trainSize: 365,  
horizon: 4);  
  
var model = pipeline.Fit(data);  
  
var forecastingEngine = model.CreateTimeSeriesEngine  
    <ModelInput, ModelOutput>(context);  
  
var forecasts = forecastingEngine.Predict();  
  
Console.WriteLine("Energie-Prognose");  
Console.WriteLine("-----");  
foreach (var forecast in forecasts.ForecastedEnergy)  
{  
    Console.WriteLine(forecast);  
}  
}  
}  
  
public class ModelInput  
{  
    [LoadColumn(0)]  
    public DateTime Date { get; set; }  
  
    [LoadColumn(1)]  
    public float EnergyDemand { get; set; }  
}  
  
public class ModelOutput  
{  
    public float[] ForecastedEnergy { get; set; }  
}  
}
```

■ 10.2 Bildklassifikation

In diesem Abschnitt implementieren Sie ein Beispielprogramm für den Bereich Bildklassifikation. Bei dieser Art von Klassifikation wird ein Bild einer Klasse zugeordnet. Die Klasse bezeichnet typischerweise das einzelne Objekt, das im Bild angezeigt wird.

Die Schwierigkeit der visuellen Mustererkennung zeigt sich in der klassischen Programmierung zum Beispiel schon bei der einfachen Aufgabenstellung, eine handschriftliche Ziffer zu erkennen. Allein schon aufgrund der Ähnlichkeit zwischen den Zahlen 3 und 8, 5 und 6 oder auch 2 und 7 wird man diese Aufgabe mit klassischer Programmierung nicht lösen können.

Dabei handelt es sich beim Erkennen von handgeschriebenen Ziffern um das „Hello World!“-Beispiel für künstliche neuronale Netze. Diese Anwendung hatte ihren Durchbruch schon 1989. Damals gelang die zuverlässige maschinelle Erkennung von Postleitzahlen für die Briefzustellung. Man hatte erkannt, dass es mit Feedforward Neural Network mit Backpropagation möglich ist, eine entsprechende Funktion zu realisieren.

Heute wird das Erkennen von handgeschriebenen Ziffern zum Erlernen von verschiedenen KI-Techniken auch an Berufs- und Hochschulen verwendet. Das nachfolgende Beispiel zeigt, wie Sie mithilfe eines Feedforward Neural Networks und ohne weitere Frameworks ein neuronales Netz zum Erkennen von handgeschriebenen Zahlen implementieren. Sie können das Beispiel auch für weitere Lernzwecke nutzen und es entsprechend erweitern und verbessern. Bild 10.2 zeigt die schematische Lösung der Aufgabe.

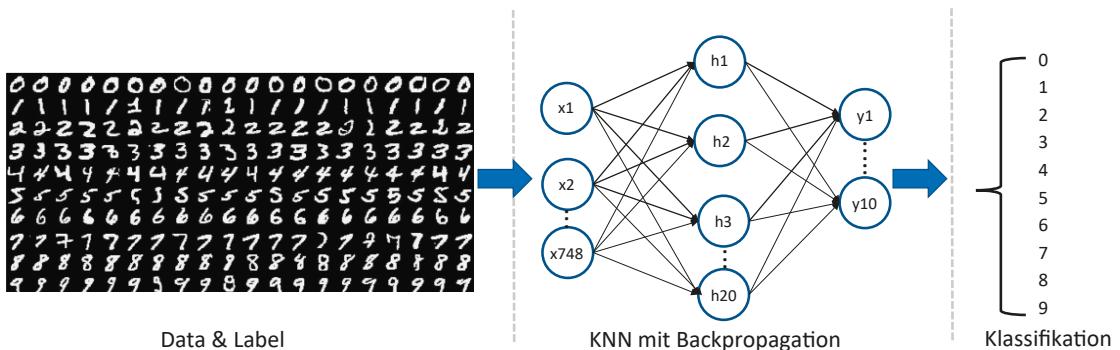


Bild 10.2 Schematische Lösung für die Bildklassifizierung

10.2.1 Benötigte Daten

Für ein aussagekräftiges neuronales Netz benötigen Sie Daten, die idealerweise gelabelt sind. Hierfür nutzen Sie einen vorbereiteten MNIST-Datensatz (*Modified National Institute of Standards and Technology*), der eine Vielzahl von Trainingsdaten als Bilder mit handgeschriebenen Ziffern enthält.

Der originale MNIST-Datensatz verfügt über 60.000 Trainings- und 10.000 Testbeispiele. Bei diesem Datensatz handelt es sich um eine Teilmenge der NIST-Datenbank für handgezeichnete Formen und Zeichen, so beinhaltet die NIST-Datenbank auch handgeschriebene

Formulare und Buchstaben. Der Datensatz wird häufig in entsprechenden Lernprojekten verwendet, auch wenn der MNIST-Datensatz nicht der in Deutschland gängigen Schreibweise für die Ziffern 1 und 7 entspricht. Es existiert für die europäische Schreibweise aber kein alternativer Datensatz. Die ursprünglichen Schwarzweiß-Bilder, in diesem Fall die handgeschriebenen Ziffern aus der NIST-Datenbank, wurden in der Größe so normalisiert, dass sie in einen 20×20 Pixel-Kasten passen und dabei ihr Seitenverhältnis beibehalten. Als Ergebnis werden die resultierenden Bilder durch den Normalisierungsalgorithmus in Graustufenbilder umgerechnet. Die Bilder werden dann in einem 28×28 -Pixel-Bild zentriert, in dem der Massenschwerpunkt der Pixel berechnet und das Bild so verschoben wurde, dass dieser Punkt in der Mitte des 28×28 -Feldes liegt (Bild 10.3).



Bild 10.3 Ausschnitt der handgeschriebenen Ziffern aus dem MNIST-Datensatz

Für unser Beispiel wird der Datensatz auf eine Größe von insgesamt ca. 11.000 Bilder geschrumpft, um das neuronale Netz auch auf kleinen lokalen PC-Systemen ausführen zu können. Der Datensatz ist mehrstufig aufgebaut und besteht aus zehn Klassen, in die Sie Zahlen von 0 bis 9 einordnen können. Er ist ein nützlicher Ausgangspunkt für die Entwicklung und Einübung einer Methodik zur Lösung von Bildklassifikationsaufgaben mithilfe eines Feed-forward-Netzes. Der modifizierte Datensatz für das Beispiel steht unter GitHub zur Verfügung.

10.2.2 Projekt konfigurieren

Für die Umsetzung des neuronalen Netzes nutzen Sie wieder eine einfache WPF-App (.NET Framework) als Rahmen für die Implementierung. Als Projektname wird im Beispiel *DemoZiffernerkennung* verwendet. Erstellen Sie die Oberfläche über den XAML-Editor des *MainWindow*. Nutzen Sie für die XAML-Datei den Code aus Listing 10.2.

Listing 10.2 Aufbau der Benutzeroberfläche

```
<Window x:Class="DemoZiffernerkennung.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
       xmlns:local="clr-namespace:DemoZiffernerkennung"
```

```

mc:Ignorable="d"
Title="KNN mit Backpropagation" Height="450" Width="600"
Loaded="Window_Loaded">

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <StackPanel>
        <Label FontSize="22" HorizontalAlignment="Center">
            Auswertung Ziffernerkennung</Label>
    </StackPanel>
    <Grid Grid.Row="1">
        <GroupBox Header="Hyperparameter und Auswertung" Margin="10" Padding="10">
            <StackPanel>
                <TextBlock>Anzahl Neuronen:</TextBlock>
                <TextBox Name="teNeuron">20</TextBox>
                <TextBlock>Lernrate:</TextBlock>
                <TextBox Name="teLearningRate">0.01</TextBox>
                <TextBlock>Epoch:</TextBlock>
                <TextBox Name="teEpoch" IsReadOnly="True"></TextBox>
                <TextBlock>Genauigkeit:</TextBlock>
                <TextBox Name="teAccuracy" IsReadOnly="True"></TextBox>
            </StackPanel>
        </GroupBox>
    </Grid>

    <StackPanel Grid.Row="2" Orientation="Horizontal" HorizontalAlignment="Right">
        <Button Name="tbTraining" Height="25" Width="100" Margin="10"
Click="tbTraining_Click">Training</Button>
        <Button Name="tbTest" Height="25" Width="100" Margin="10"
Click="tbTest_Click">Auswertung</Button>
        <Button Height="25" Width="100" Margin="10 10 10 10"
Click="Button_Click">Beenden</Button>
    </StackPanel>
</Grid>
</Window>

```

Des Weiteren benötigen Sie für das Feedforward-Netz noch folgende Klassen in Ihrer Solution. Legen Sie diese über *Add | New Item* im Kontextmenü des Solution-Explorers an.

- *ActivationFunction*
- *HiddenLayer*
- *InputLayer*
- *MnistImage*
- *NeuralNetwork*
- *OutputLayer*

Ist die Oberfläche erstellt und sind die neuen Klassen angelegt, sollte Ihr Visual-Studio-Projekt in etwa wie in Bild 10.4 aussehen.

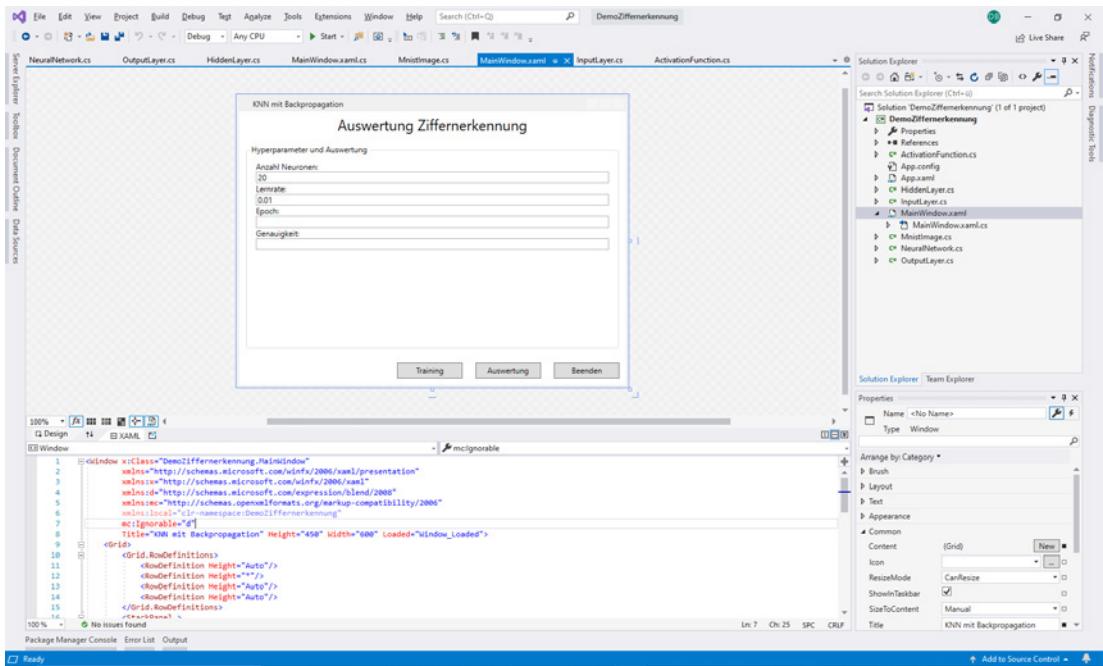


Bild 10.4 Das erstellte Projekt in Visual Studio

10.2.3 Importieren des MNIST-Datensatzes

Der modifizierte MNIST-Datensatz steht in *.jpg-Dateien als Trainings- und Testdatensatz bereit. Sie benötigen eine Programmroutine, die das Einlesen und Extrahieren der Bilder bewerkstelltigt. In der Klasse *MnistImage* werden die Bilder geladen und die Informationen in einer Zeilen- und Spaltenstruktur angeordnet, die es ermöglicht, auf jedes Pixel in jedem Bild der Trainings- und Testdatensätze zuzugreifen.

Jede *.jpg-Datei enthält ca. 960 handgeschriebene Ziffern, die in einem Raster angeordnet sind. Da die einzelnen Ziffern über die konsistente Abmessung von 28×28 Pixel verfügen, ist es möglich, jede Ziffer zu isolieren und die Ziffern-Pixel als eine Liste von Byte-Arrays zu speichern. Listing 10.3 zeigt die vollständige Implementierung der Klasse *MnistImage* für die Aufbereitung der *.jpg-Dateien.

Listing 10.3 Die Klasse *MnistImage*

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;

namespace DemoZiffernerkennung
{

```

```
public class MnistImage
{
    public static int numberWidth = 28;
    public static int nTypes = 10;

    static Random random = new Random(DateTime.Now.Millisecond +
        DateTime.Now.Second);

    public static MnistImage[] trainingImages = new MnistImage[10];
    public static MnistImage[] testImages = new MnistImage[10];

    public Bitmap image = null;
    public byte[] imageBytes;

    List<byte[]> pixelRows = null;

    public static void LoadTrainingImage(int n)
    {
        trainingImages[n].image = new Bitmap(@"c:/Temp/mnist_" + n.ToString()
            + ".jpg");
        trainingImages[n].imageBytes = ByteArrayFromImage(trainingImages[n].image);
    }

    public static void LoadTestImage(int n)
    {
        testImages[n].image = new Bitmap(@"c:/Temp/mnistTest_" + n.ToString()
            + ".jpg");
        testImages[n].imageBytes = ByteArrayFromImage(testImages[n].image);
    }

    public static byte[] ByteArrayFromImage(Bitmap bmp)
    {
        Rectangle rect = new Rectangle(0, 0, bmp.Width, bmp.Height);
        BitmapData data = bmp.LockBits(rect, ImageLockMode.ReadOnly,
            bmp.PixelFormat);
        IntPtr ptr = data.Scan0;

        int numBytes = data.Stride * bmp.Height;
        byte[] image_bytes = new byte[numBytes];

        System.Runtime.InteropServices.Marshal.Copy(ptr, image_bytes, 0, numBytes);

        bmp.UnlockBits(data);

        return image_bytes;
    }

    public static byte[] GetNumberPixels(int column, int row, List<byte[]> pixels)
    {
        byte[] digitPixels = new byte[numberWidth * numberWidth];

        int index = 0;

        for (int y = row * numberWidth; y < (row * numberWidth) + numberWidth; y++)
```

```
        {
            for (int x = column * numberWidth; x < (column * numberWidth)
                + numberWidth; x++)
            {
                digitPixels[index] = pixels[y][x];
                index++;
            }
        }

        return digitPixels;
    }

    public static Bitmap GetImage(int expected, bool testing)
    {
        if (testing)
        {
            return testImages[expected].image;
        }
        else
        {
            return trainingImages[expected].image;
        }
    }

    public static byte[] GetImageNumberPixels(int expected, int column, int row,
        bool testing)
    {
        Bitmap image = GetImage(expected, testing);

        LoadPixelRows(expected, testing);

        return GetNumberPixels(column, row,
            testing ? testImages[expected].pixelRows :
            trainingImages[expected].pixelRows
        );
    }

    public static void LoadPixelRows(int Expected, bool testing)
    {
        if (testing)
        {
            byte[] imageBytes = testImages[Expected].imageBytes;

            Bitmap image = testImages[Expected].image;

            if (testImages[Expected].pixelRows == null)
            {
                testImages[Expected].pixelRows = new List<byte[]>();

                for (int i = 0; i < image.Height; i++)
                {
                    int index = i * image.Width;
                    byte[] rowBytes = new byte[image.Width];
                    for (int w = 0; w < image.Width; w++)

```

```
        {
            rowBytes[w] = imageBytes[index + w];
        }

        testImages[Expected].pixelRows.Add(rowBytes);
    }
}
else
{
    byte[] imageBytes = trainingImages[Expected].imageBytes;

    Bitmap image = trainingImages[Expected].image;

    if (trainingImages[Expected].pixelRows == null)
    {
        trainingImages[Expected].pixelRows = new List<byte[]>();

        for (int i = 0; i < image.Height; i++)
        {
            int index = i * image.Width;
            byte[] rowBytes = new byte[image.Width];
            for (int w = 0; w < image.Width; w++)
            {
                rowBytes[w] = imageBytes[index + w];
            }

            trainingImages[Expected].pixelRows.Add(rowBytes);
        }
    }
}

public static int GetRow(int expected, int index)
{
    Bitmap image = trainingImages[expected].image;

    int nColumns = image.Width / numberWidth;
    int row = index / nColumns;

    return row;
}

public static int GetCol(int expected, int index)
{
    Bitmap image = trainingImages[expected].image;

    int nColumns = image.Width / numberWidth;
    int column = index % nColumns;

    return column;
}

public static byte[] GetImageNumberPixels(int expected, int index,
                                         bool testing)
```

```
{  
    Bitmap image = GetImage(expected, testing);  
  
    int nColumns = image.Width / numberWidth;  
    int nRows = image.Height / numberWidth;  
  
    int row = index / nColumns;  
    int column = index % nColumns;  
  
    if (row < nRows && column < nColumns)  
    {  
        return GetImageNumberPixels(expected, column, row, testing);  
    }  
    else  
    {  
        return null;  
    }  
}  
}  
}
```

10.2.4 Aktivierungsfunktion

Das Beispiel-Feedforward-Netz verwendet die Sigmoid-Aktivierungsfunktion. Die Klasse *ActivationFunction* enthält die Implementierung dieser Funktion zusammen mit ihrer entsprechenden Ableitung, die bei der Backpropagation verwendet wird.

Listing 10.4 Die Klasse ActivationFunction

```
using System;  
  
namespace DemoZiffernerkennung  
{  
    public class ActivationFunction  
    {  
        public static double GetSigmoid(double x)  
        {  
            double sigmoid = 1.0 / (1.0 + Math.Exp(-x));  
            return sigmoid;  
        }  
  
        public static double GetDerivative(double x)  
        {  
            double derivativeOfSigmoidOfX = x * (1.0 - x);  
            return derivativeOfSigmoidOfX;  
        }  
    }  
}
```

10.2.5 Input Layer

Da es sich bei den einzelnen handgeschriebenen Ziffern um 28×28 Pixelfelder mit Graustufen handelt, ist der Eingangsvektor $28 \times 28 = 784$ Felder groß und die Graustufen werden als entsprechende Double-Werte mit dem jeweiligen Schwarzanteil (0.0 = weiß bis 1.0 = schwarz) dargestellt. Bild 10.5 zeigt das vorgenommene Mapping auf den dazugehörigen Eingangsvektor.

```

inputs = new double[newInputs.Length];
if (i < newInputs.Length)
{
    inputs[i] = (double)newInputs[i] / 255.0;
}
for (int i = 0; i < newInputs.Length; i++)
{
    inputs[i] = (double)newInputs[i] / 255.0;
}

```

Index	Value
[0]	0
[1]	0.0392156862745098
[2]	0.0392156862745098
[3]	0
[4]	0.015686274509803921
[5]	0
[6]	0.031372549019607843
[7]	0
[8]	0.015686274509803921
[9]	0.031372549019607843
[10]	0.00392156862745098
[11]	0
[12]	0.043137254901960784
[13]	0.00392156862745098
[14]	0
[15]	0.031372549019607843
[16]	0
[17]	0.015686274509803921
[18]	0.031372549019607843
[19]	0.00392156862745098
[20]	0
[21]	0.043137254901960784
[22]	0.00392156862745098

Bild 10.5

Mapping auf den Eingangsvektor beim Debuggen

Der Eingangsvektor wird somit auf einen 1D-Vektor abgeflacht, der 784 Pixel groß ist und die 0-255-Byte-Information macht aus dem Graustufenbild eine reelle Zahl zwischen 0 und 1. Fügen Sie daher der Klasse *InputLayer* die Codezeilen aus Listing 10.5 hinzzu.

Listing 10.5 Die Klasse InputLayer

```

namespace DemoZiffernerkennung
{
    public class InputLayer
    {
        public double[] inputs;
        public bool[] isDropout;

        public void setInputs(byte[] newInputs)
        {
            inputs = new double[newInputs.Length];

            if (isDropout == null || isDropout.Length != inputs.Length)
            {
                isDropout = new bool[inputs.Length];
            }

            for (int i = 0; i < newInputs.Length; i++)
            {
                inputs[i] = (double)newInputs[i] / 255.0;
            }
        }
    }
}

```

```

        }
    }
}
}
```

10.2.6 Hidden Layer

In der Klasse für die Hidden-Layer-Neuronen existieren die drei Methoden *Activate*, *Backpropagation* und *InitWeights*, die die Gewichte auf Zufallszahlen initialisieren. Die Anzahl der Neuronen wird über die Oberfläche als Parameter festgelegt. Die Voreinstellung beträgt hier 20 Neuronen im Hidden Layer.

Bei der Backpropagation muss der Fehler für jede Ausgabe entsprechend dem Beitrag dieses Neurons zu diesem Fehler berechnet werden. Da sich aber auch die Gewichte der Ausgabeschicht (Output Layer) bei der Backpropagation ändern, müssen die vorherigen Gewichte beim Forward-Pass gespeichert und zur Berechnung des Fehlers verwendet werden (siehe auch Abschnitt 3.7.3, „Backpropagation-Algorithmus“).

Listing 10.6 Die Klasse HiddenLayer

```

namespace DemoZiffernerkennung
{
    public class HiddenLayer
    {
        public int hiddenNeurons = 0;

        public HiddenNeuron[] neurons;

        public NeuralNetwork network;

        public HiddenLayer(int n, NeuralNetwork neuralNetwork)
        {
            network = neuralNetwork;

            hiddenNeurons = n;

            neurons = new HiddenNeuron[hiddenNeurons];

            for (int i = 0; i < neurons.Length; i++)
            {
                neurons[i] = new HiddenNeuron(i, this);
            }
        }

        public void Backpropagation()
        {
            foreach (HiddenNeuron n in neurons)
            {
                n.Backpropagation();
            }
        }
    }
}
```

```
}

public void Activate()
{
    foreach (HiddenNeuron n in neurons)
    {
        n.Activate();
    }
}

public class HiddenNeuron
{
    public bool isDropout = false;
    public int index = 0;
    public double error = 0;

    public double[] weights = new double[MnistImage.numberWidth *
                                              MnistImage.numberWidth];
    public double[] oldWeights = new double[MnistImage.numberWidth *
                                              MnistImage.numberWidth];

    public double sum = 0.0;
    public double sigmoidSum = 0.0;

    public HiddenLayer layer;

    public HiddenNeuron(int i, HiddenLayer h)
    {
        layer = h;
        index = i;
        InitWeights();
    }

    public void InitWeights()
    {
        weights = new double[MnistImage.numberWidth * MnistImage.numberWidth];

        for (int y = 0; y < weights.Length; y++)
        {
            weights[y] = NeuralNetwork.RandomWeight();
        }
    }

    public void Activate()
    {
        if (isDropout) return;
        sum = 0.0;

        for (int y = 0; y < weights.Length; y++)
        {
            if (!NeuralNetwork.inputLayer.isDropout[y])
            {
                sum += NeuralNetwork.inputLayer.inputs[y] * weights[y];
            }
        }
    }
}
```

```

        }

        sigmoidSum = ActivationFunction.GetSigmoid(sum);
    }

    public void Backpropagation()
    {
        if (isDropout) return;

        double sumError = 0.0;

        foreach (OutputNeuron o in NeuralNetwork.outputLayer.neurons)
        {
            sumError += (o.error * o.oldWeights[index]);
        }

        error = ActivationFunction.GetDerivative(sigmoidSum) * sumError;

        for (int w = 0; w < weights.Length; w++)
        {
            weights[w] += (error * NeuralNetwork.inputLayer.inputs[w]) *
                layer.network.learningRate;
        }
    }
}

```

10.2.7 Output Layer

Die Klasse *OutputLayer* verfügt über einen recht einfachen Code. Die Anzahl der Neuronen ist hier auf 10 festgelegt und sollte auch nicht geändert werden, da dies der Anzahl der möglichen Klassifizierungen von 0 bis 9 entspricht.

Auch hier ist es für die Backpropagation notwendig, vor der Anpassung der Gewichte eine Kopie der vorherigen Gewichte aufzubewahren. Dies geschieht über das Array `oldWeights`, damit die Gewichte bei der Fehlerberechnung für die Neuronen im Hidden Layer verwendet werden können. Ebenso werden die Gewichte über die Methode `InitWeights` auf Zufallswerte initialisiert.

Listing 10.7 Die Klasse OutputLayer

```
namespace DemoZiffernerkennung
{
    public class OutputLayer
    {
        public NeuralNetwork network;
        public HiddenLayer hiddenLayer;
        public int outputNeurons = 10;

        public OutputNeuron[] neurons;
```

```
public OutputLayer(HiddenLayer h, NeuralNetwork neuralNetwork)
{
    network = neuralNetwork;
    hiddenLayer = h;
    neurons = new OutputNeuron[outputNeurons];

    for (int i = 0; i < outputNeurons; i++)
    {
        neurons[i] = new OutputNeuron(this);
    }
}

public void Activate()
{
    foreach (OutputNeuron n in neurons)
    {
        n.Activate();
    }
}

public void Backpropagation()
{
    foreach (OutputNeuron n in neurons)
    {
        n.Backpropagation();
    }
}

public class OutputNeuron
{
    public OutputLayer outputLayer;

    public double sum = 0.0;
    public double sigmoidSum = 0.0;
    public double error = 0.0;

    public double[] weights;
    public double[] oldWeights;

    public double expectedValue = 0.0;

    public OutputNeuron(OutputLayer output)
    {
        outputLayer = output;
        weights = new double[outputLayer.hiddenLayer.hiddenNeurons];
        oldWeights = new double[outputLayer.hiddenLayer.hiddenNeurons];
        InitWeights();
    }

    private void InitWeights()
    {
        for (int y = 0; y < weights.Length; y++)
        {
```

```

        weights[y] = NeuralNetwork.RandomWeight();
    }
}

public void Activate()
{
    sum = 0.0;

    for (int y = 0; y < weights.Length; y++)
    {
        if (!outputLayer.hiddenLayer.neurons[y].isDropout)
        {
            sum += outputLayer.hiddenLayer.neurons[y].sigmoidSum * weights[y];
        }
    }

    sigmoidSum = ActivationFunction.GetSigmoid(sum);
}

public void Backpropagation()
{
    CalculateError();

    int i = 0;
    foreach (HiddenNeuron n in outputLayer.hiddenLayer.neurons)
    {
        if (!n.isDropout)
        {
            oldWeights[i] = weights[i];
            weights[i] += (error * n.sigmoidSum) *
                outputLayer.network.learningRate;
        }

        i++;
    }
}

private void CalculateError()
{
    error = ActivationFunction.GetDerivative(sigmoidSum) *
        (expectedValue - sigmoidSum);
}
}
}
}

```

10.2.8 Neural Network

Die Klasse *Neural Network* erstellt das neuronale Netz und implementiert die wichtigsten Methoden zum Ausführen des Trainings. Die Methode *SetExpected* legt den Wert des Ausgangsneurons, das dem gewünschten Ergebnis entspricht, auf 1 fest, und setzt alle anderen Ausgangsneuronen auf 0.

Dieses Vorgehen wird auch als *One Hot Vector* bezeichnet, da der Vektor genauso viele Dimensionen besitzt, wie es Ausgangsneuronen gibt. In ähnlicher Weise arbeitet auch die Methode *Analysis*, indem der Algorithmus das Ausgangsneuron mit dem höchsten Wert findet. Im Idealfall wäre dieser Wert nahe 1 und alle anderen nahe 0, was ja auch dem gewünschten Trainingsziel des neuronalen Netzes entspricht.

Die Methode *TrainTheNeuralNetwork* ruft *SetExpected* auf, um ein Ziel festzulegen und danach die Methoden *Activate* und *Backpropagation*, um ein Ergebnis zu ermitteln und die Gewichte auf der Grundlage des berechneten Fehlers anzupassen.

Listing 10.8 Die Klasse NeuralNetwork

```
using System;
using System.Collections.Generic;

namespace DemoZiffernerkennung
{
    public class NeuralNetwork
    {
        public double learningRate = 0.01;
        public static double learningRateInitial = 0.01;
        public static bool usePositiveWeights = false;
        public int nFirstHiddenLayerNeurons = 30;

        public List<InferenceError> errors = new List<InferenceError>();

        public static bool testing = false;
        public static int expected;

        public static Random rand = new Random(DateTime.Now.Millisecond);

        public static InputLayer inputLayer;
        public static HiddenLayer hiddenLayer;
        public static OutputLayer outputLayer;

        public NeuralNetwork()
        {
        }

        public void InitNeuralNetwork(int hiddenNeurons)
        {
            nFirstHiddenLayerNeurons = hiddenNeurons;
            Create();
        }

        private void Create()
        {
            inputLayer = new InputLayer();
            hiddenLayer = new HiddenLayer(nFirstHiddenLayerNeurons, this);
            outputLayer = new OutputLayer(hiddenLayer, this);
        }

        public void TrainTheNeuralNetwork(int nIterations, int expected)
```

```
{  
    SetExpected(expected);  
  
    for (int n = 0; n < nIterations; n++)  
    {  
        Activate();  
        Backpropagation();  
    }  
}  
  
private void SetExpected(int exp)  
{  
    expected = exp;  
  
    for (int i = 0; i < outputLayer.neurons.Length; i++)  
    {  
        if (i == exp)  
        {  
            outputLayer.neurons[i].expectedValue = 1.0;  
        }  
        else  
        {  
            outputLayer.neurons[i].expectedValue = 0.0;  
        }  
    }  
}  
  
internal static double RandomWeight()  
{  
    double span = 50000;  
    int spanInt = (int)span;  
  
    double magnitude = 10.0;  
  
    if (usePositiveWeights)  
    {  
        return ((double)rand.Next(0, spanInt)) / (span * magnitude);  
    }  
    else  
    {  
        return ((double)(rand.Next(0, spanInt * 2) - spanInt)) /  
               (span * magnitude);  
    }  
}  
  
public void Activate()  
{  
    hiddenLayer.Activate();  
    outputLayer.Activate();  
}  
  
public void Backpropagation()  
{  
    outputLayer.Backpropagation();  
    hiddenLayer.Backpropagation();  
}
```

```
}

internal bool TestInference(int expected, out int estimated,
                           out double confidence)
{
    SetExpected(expected);

    Activate();

    estimated = Analysis(out confidence);

    return (estimated == expected);
}

private int Analysis(out double confidence)
{
    confidence = 0.0;
    double max = 0;

    int result = -1;

    for (int n = 0; n < outputLayer.outputNeurons; n++)
    {
        double s = outputLayer.neurons[n].sigmoidSum;

        if (s > max)
        {
            confidence = s;
            result = n;
            max = s;
        }
    }

    return result;
}

public class InferenceError
{
    public int expected = 0;
    public int n = 0;
    public int estimate = 0;
    public double confidence = 0.0;

    public InferenceError(int e, int neuron, int est, double con)
    {
        expected = e;
        n = neuron;
        estimate = est;
        confidence = con;
    }
}
```

10.2.9 Initialisierung und Auswertung

Über Code Behind in der *MainWindow*-Klasse initialisieren Sie über das *ButtonClick*-Event das Feedforward-Netz und führen über die Methode *TrainAllMnistImages* das Training durch. Die Methode *Evaluation* ermöglicht die Auswertung und gibt nach 10 Epochen die Konfidenz der Ziffern in Prozent aus.

Listing 10.9 Die Klasse MainWindow

```
using System;
using System.Collections.Generic;

namespace DemoZiffernerkennung
{
    public class NeuralNetwork
    {
        public double learningRate = 0.01;
        public static double learningRateInitial = 0.01;
        public static bool usePositiveWeights = false;
        public int nFirstHiddenLayerNeurons = 30;

        public List<InferenceError> errors = new List<InferenceError>();

        public static bool testing = false;
        public static int expected;

        public static Random rand = new Random(DateTime.Now.Millisecond);

        public static InputLayer inputLayer;
        public static HiddenLayer hiddenLayer;
        public static OutputLayer outputLayer;

        public NeuralNetwork()
        {

        }

        public void InitNeuralNetwork(int hiddenNeurons)
        {
            nFirstHiddenLayerNeurons = hiddenNeurons;
            Create();
        }

        private void Create()
        {
            inputLayer = new InputLayer();
            hiddenLayer = new HiddenLayer(nFirstHiddenLayerNeurons, this);
            outputLayer = new OutputLayer(hiddenLayer, this);
        }

        public void TrainTheNeuralNetwork(int nIterations, int expected)
        {
            SetExpected(expected);
        }
    }
}
```

```
for (int n = 0; n < nIterations; n++)
{
    Activate();
    Backpropagation();
}
}

private void SetExpected(int exp)
{
    expected = exp;

    for (int i = 0; i < outputLayer.neurons.Length; i++)
    {
        if (i == exp)
        {
            outputLayer.neurons[i].expectedValue = 1.0;
        }
        else
        {
            outputLayer.neurons[i].expectedValue = 0.0;
        }
    }
}

internal static double RandomWeight()
{
    double span = 50000;
    int spanInt = (int)span;

    double magnitude = 10.0;

    if (usePositiveWeights)
    {
        return ((double)rand.Next(0, spanInt)) / (span * magnitude);
    }
    else
    {
        return ((double)(rand.Next(0, spanInt * 2) - spanInt)) /
               (span * magnitude);
    }
}

public void Activate()
{
    hiddenLayer.Activate();
    outputLayer.Activate();
}

public void Backpropagation()
{
    outputLayer.Backpropagation();
    hiddenLayer.Backpropagation();
}
```

```
internal bool TestInference(int expected, out int estimated,
                           out double confidence)
{
    SetExpected(expected);

    Activate();

    estimated = Analysis(out confidence);

    return (estimated == expected);
}

private int Analysis(out double confidence)
{
    confidence = 0.0;
    double max = 0;

    int result = -1;

    for (int n = 0; n < outputLayer.outputNeurons; n++)
    {
        double s = outputLayer.neurons[n].sigmoidSum;

        if (s > max)
        {
            confidence = s;
            result = n;
            max = s;
        }
    }

    return result;
}

public class InferenceError
{
    public int expected = 0;
    public int n = 0;
    public int estimate = 0;
    public double confidence = 0.0;

    public InferenceError(int e, int neuron, int est, double con)
    {
        expected = e;
        n = neuron;
        estimate = est;
        confidence = con;
    }
}
```

10.2.10 Training und Backpropagation

In dem neuronalen Netz ist bekannt, wie die Arrays für die Ziffern 0 bis 9 aussehen. Wenn am Eingang X eine handgeschriebene Ziffer 3 anlegt, soll am Ausgangsneuron Y auch eine 3 klassifiziert werden. Die auftauchenden Fehler im Netz ergeben sich durch den Vergleich des tatsächlichen Wertes mit dem Sollergebnis Y. Eine entsprechende Abweichung wird auf die bisher rein zufällige Gewichtung verteilt, um den Fehler zu verringern. Hierbei ist die Verteilung des Fehlers, wie aus Abschnitt 3.7.1, „Kostenfunktion“, bekannt, auf die Gewichtung entscheidend. Im ersten Schritt ermittelt man die Korrektur zwischen Ausgabe und Hidden Layer. Das Delta ergibt sich aus dem Fehler und der Steigung an der Stelle des Wertes für die Sigmoid-Funktion. Wenn der Wert entweder groß (nahe 1) oder klein (nahe 0) ist, ist sich das Modell schon sicher und die Fehlerkorrektur ist sehr gering. Dies wird über die abgeleitete Sigmoid-Funktion *GetDerivative* ermittelt. Sollte der Wert um Y nur bei ca. 0,5 liegen, so muss mit einem entsprechend großen Wert korrigiert werden.

Die Differenz der Ausgabe zum gewünschten Wert wird dann in Form einer Fehlerfunktion, hier vom mittleren quadratischen Fehler, rückwärts durch die einzelnen Schichten weitergeleitet (Backpropagation). Somit durchläuft dieses Beispiel vier Schritte, die für eine festgelegte Anzahl von Iterationen (*nIterationLoop* = 50) wiederholt werden:

1. Werte über das Netzwerk weiterleiten
2. Berechnen der Verlustfunktion
3. Übertragen der Werte rückwärts über das Netzwerk (Backpropagation)
4. Aktualisierung der Gewichte

Somit werden bei jedem Trainingsschritt die Gewichte leicht angepasst, um den Verlust für den nächsten Schritt zu verringern. Nach 10 Epochen wird das Training im Beispiel beendet und das Ergebnis des Modells ausgewertet (Bild 10.6).

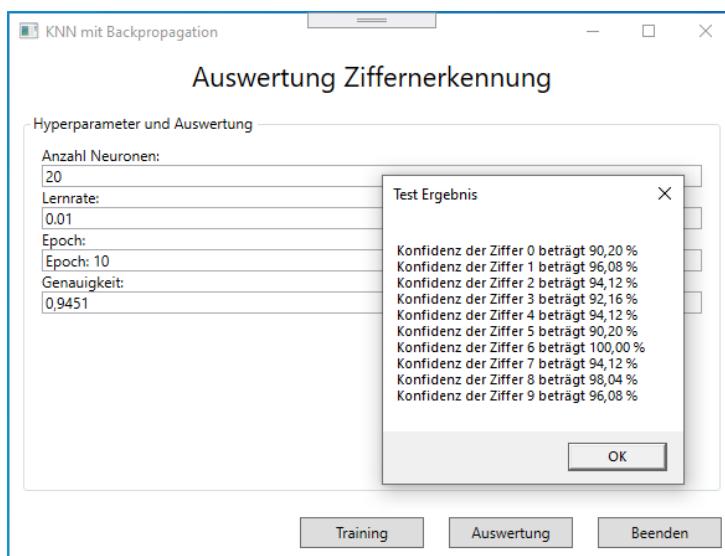


Bild 10.6 Das Ergebnis nach zehn Epochen

10.2.11 Auswertung und Verbesserung

Das Modell durchläuft in der Trainingsphase 10 Epochen mit jeweils 50 Iterationen, das entspricht bei jedem Satz über 5.000 handgeschriebenen Ziffern. Die durchschnittliche Genauigkeit liegt bei etwa 0,9451 und entspricht somit 94 %. Bei der Einzelauswertung kann man auch erkennen (Bild 10.6), dass es Ziffern gibt, die zu 100 % erkannt werden. Das Ergebnis der Prüfung basiert hier auf 50 Iterationen für die Ziffernsätze von 0 bis 9. Wenn Sie den Wert der Variable *nIterationLoop* erhöhen (oder verkleinern) und einen längeren (oder kürzeren) Test durchführen, kann die Genauigkeit nach oben oder nach unten gehen. Auch das Anpassen bzw. Ändern der Epochenzahl beeinflusst die Genauigkeit. Die Lernrate im Beispiel gibt an, wie stark sich die Parameter bei jedem Schritt des Lernprozesses anpassen. Diese Anpassungen stellen die Schlüsselkomponente des Trainings dar. Nach jedem Durchlauf durch das Netz werden die Gewichte leicht abgestimmt, um den Verlust zu reduzieren. Größere Lernraten können schneller konvergieren, können aber auch die optimalen Werte überschreiten, wenn sie aktualisiert werden.

Es gibt also auch hier wieder eine Vielzahl von Möglichkeiten, wie das Ergebnis des Modells verbessert werden kann. Experimentieren Sie, indem Sie die Lernrate verändern oder die Anzahl der Neuronen im Hidden Layer variieren. Sie können alternativ auch eine andere Aktivierungsfunktion wie zum Beispiel ReLU oder Softmax implementieren.

Beachten Sie aber, dass jedes neue Training zu einem anderen Ergebnis führt, da die Gewichte beim Start mit Zufallswerten initialisiert werden. Hier könnte man eine Speichermöglichkeit für das Modell schaffen. Sie sehen, es gibt auch bei diesem kleinen Beispiel noch eine Menge nützlicher Erweiterungsmöglichkeiten.

Nach verschiedenen Trainingsläufen und 20 Epochen erreicht das Modell schließlich eine durchschnittliche Erkennungsrate von ca. 96 %. Es zeigt sich bei den einzelnen Testbildern, dass die Erkennungsrate stark von der Ausrichtung und Darstellung der handgeschriebenen Ziffern abhängig ist (Bild 10.7). Besonders gut erkennt man das bei der Zahl 5, die im Durchschnitt nur auf eine Erkennungsrate von 88 %-92 % kommt.

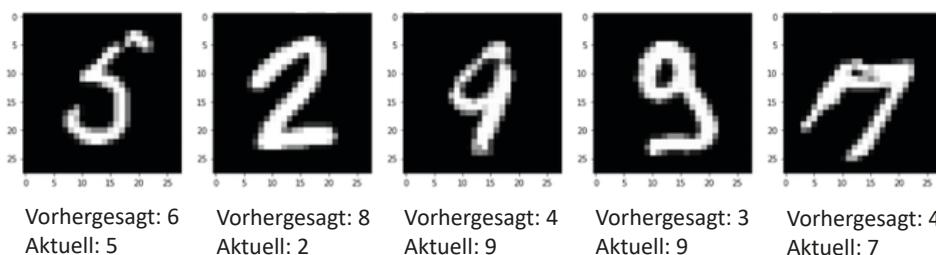


Bild 10.7 Beispiel der fehlerhaften Darstellung der handgeschriebenen Ziffern

Im Beispiel mag eine Erkennungsrate von 96 % ausreichen. In einem Produktionsprozess oder in der Logistik benötigt man aber bei der Objekterkennung eine Quote von über 98 % Genauigkeit, die nur durch die Verwendung eines Convolutional Neural Network (CNN) zu erreichen ist.

■ 10.3 Visuelle Muster erkennen

Im vorherigen Abschnitt haben Sie ein typisches Beispiel eines Feedforward-Netzes für die logistische Regression erstellt, nämlich die Ermittlung, welche Klasse zu einer geschriebenen Ziffer zählt.

In diesem Abschnitt geht es nun um die Bild- und besonders die Objekterkennung in Bildern mit Convolutional Neural Networks (CNNs), welche zur Erkennung von visuellen Mustern mit extremer Variabilität und Robustheit gegenüber Verzerrungen und einfachen geometrischen Transformationen entwickelt wurden. Wie schon in Kapitel 6, „Convolutional Neural Network“, beschrieben, sind CNNs dem KNN Layer (*Fully Connected Neural Network*) vorgeschaltet, um die Bildinformation extrahieren zu können.

Das nachfolgende Beispiel versteht sich als reines Lernprojekt für die Umsetzung der Layer in einem Convolutional Neural Network. Die verschiedenen Frameworks, die CNNs bereits implementieren, isolieren Sie von den einzelnen Details der Schichten und geben Ihnen nur eine abstrakte API an die Hand, um die Komplexität bei der Implementierung zu erleichtern. Daher will dieses Beispiel Sie animieren, auch einmal ein eigenes CNN zu entwickeln – und sei es nur zu Lernzwecken.

Die Umsetzung gelingt allein mit .NET-Bordmitteln, sie zeigt gleichzeitig aber auch, dass die Komplexität des Projekts sehr schnell ansteigen kann. Allerdings können in diesem Beispiel nur der Entwurf und der grundlegende Layer eines CNN vermittelt und nicht alle Besonderheiten oder speziellen Implementierungen berücksichtigt werden.

10.3.1 Aufgabenstellung

Bei diesem Beispiel wird das CNN nur mit C# und Visual Studio erstellt. Sie verwenden unter Visual Studio 2019 ein neues Projekt vom Typ *Console App (.NET Framework)* mit dem Namen *ConvolutionalNeuralNetwork* und einer neuen Klasse mit dem Namen *ConvNet* zur Aufnahme der benötigten Layer (Schichten) und Methoden. Da es sich um ein Lernprojekt handelt, wird das CNN auf das Wesentliche beschränkt und beinhaltet somit nur:

- Convolutional Layer
- Max-Pooling Layer
- Flattern Layer
- Fully Connected Layer
- Rectified Linear Unit (ReLU) als Aktivierungsfunktion
- *Filter*-Methode
- *RandomWeights*-Methode
- *TrainingForConvoltionalNeuralNetwork*-Methode

Im Beispiel wird mit einem Bild, das als Test-JPG zur Verfügung steht, in einer Größe von 32×32 Pixel und mit drei Farbkanälen R, G, B gearbeitet. Beginnen Sie also mit der Implementierung des Convolutional Layers.

10.3.2 Convolutional Layer

Der Convolutional Layer (Faltungsschicht) implementiert den vorgegebenen Filter und integriert diesen in das neuronale Netz. Dabei wird eine Faltungsmatrix (Kernel) über die Pixelwerte gelegt. Die Gewichte der Kernel sind jeweils unterschiedlich dimensioniert. Durch die Verrechnung mit den Eingabewerten können so unterschiedliche Merkmale extrahiert werden.

Listing 10.10 Convolutional Layer

```
public ConvNet()
{
}

public double[,] Convolutional(double[,] input, double[,,] filter)
{
    double[,] output = new double[input.GetLength(0), input.GetLength(1),
                                input.GetLength(2)];

    for (int i = 0; i < filter.GetLength(0); i++)
    {
        for (int j = 0; j < input.GetLength(0); j++)
        {
            for (int k = 0; k < input.GetLength(1); k++)
            {
                for (int l = 0; l < input.GetLength(2); l++)
                {
                    output[j, k, l] = input[j, k, l] * filter[i, j, k, l];
                }
            }
        }
    }

    return output;
}
```

Im Beispiel durchläuft der Code die Forward Propagation und findet entsprechende Ausgabewahrscheinlichkeiten für jede Klasse. Als Filter wird eine $3 \times 3 \times 1$ -Matrix vorgegeben. Der Convolutional Layer stellt die räumliche Beziehung zwischen den Pixeln sicher, indem Bildmerkmale anhand kleiner Quadrate von Eingabedaten gelernt werden. Sie nutzen also den Filter zur Schärfung, zur Kantenerkennung oder zur Unschärfe.

10.3.3 Pooling Layer

Der Pooling Layer dient zur Verallgemeinerung der Daten. Durch den Einsatz von Max-Pooling werden die stärksten Merkmale weitergeleitet.

Listing 10.11 Max-Pooling Layer

```

public double[,] MaxPooling(double[,] input, int filtersize)
{
    double[,] output = null;
    var newHeight = ((input.GetLength(1) - filtersize) / 2) + 1;
    var newWidth = ((input.GetLength(2) - filtersize) / 2) + 1;

    output = new double[input.GetLength(0), newHeight, newWidth];

    for (int j = 0; j <= 2; j++)
    {
        var cuurentY = 0;
        var outY = 0;

        for (int k = 0; k <= 15; k++)
        {
            var cuurentX = 0;
            var outX = 0;

            for (int l = 0; l <= 15; l++)
            {
                double maxValue = MaxValue(input, j, k, l, filtersize);
                output[j, outY, outX] = input[j, k, l] > maxValue ?
                    input[j, k, l] : maxValue;
                cuurentX = cuurentX + 2;
                outX = outX + 1;
            }

            cuurentY = cuurentY + 2;
            outY = outY + 1;
        }
    }

    return output;
}

```

Das Max-Pooling stellt die gebräuchlichste Art des Poolings für CNNs dar. Der Maximalwert wird hier aus einem gegebenen Zahlenfeld entnommen. Zu diesem Zweck teilen Sie die Feature Map in $n \times n$ Felder auf und wählen nur den Maximalwert aus jedem Feld aus. Im Beispiel wird der Maximalwert auf eine Filtergröße von 2×2 angewendet. Die Ermittlung des Maximalwertes wird über die Methode *MaxValue* errechnet.

Listing 10.12 MaxValue

```

private double MaxValue(double[,] input, int j, int k, int l, int filtersize)
{
    double maxValue = 0;

    for (int a = 0; a < k + filtersize; a++)
    {
        for (int b = 0; b < l + filtersize; b++)
        {

```

```

        maxValue = maxValue < input[j, a, b] ? input[j, a, b] : maxValue;
    }

    return maxValue;
}

```

10.3.4 Flatten Layer

In dieser Schicht wird der mehrdimensionale Layer aus der Faltungsmatrix zu einem eindimensionalen Vektor überführt.

Listing 10.13 Flatten Layer

```

public double[] Flatten(double[,] input)
{
    int rgbChannel = input.GetLength(0);
    int rowPixel = input.GetLength(1);
    int columnPixel = input.GetLength(2);
    int length = rgbChannel * rowPixel * columnPixel;
    double[] output = new double[length];

    int count = 0;
    for (int i = 0; i < rgbChannel; i++)
    {
        for (int j = 0; j < rowPixel; j++)
        {
            for (int k = 0; k < columnPixel; k++)
            {
                output[count] = input[i, j, k];
                count = count + 1;
            }
        }
    }

    return output;
}

```

Da es auch bei CNNs ein sogenanntes Overfitting (Überanpassung) geben kann, ist es mit einer Regularisierungsmethode, dem *Dropout* möglich, bestimmte Verbindungen der Eingangsdaten nicht mehr weiterzugeben. Das heißt, beim Training des Netzes wird eine vorher spezifische Anzahl von Neuronen in jedem Layer des Netzes ausgeschaltet, daher *Dropout*, und für die weitere Berechnung nicht berücksichtigt. Im Beispiel wird aber auf die Regularisierungsmethode verzichtet.

10.3.5 Fully Connected Layer

Bei diesem Layer, der auch als *Dense Layer* bezeichnet wird, handelt es sich um eine Standardschicht, bei der alle Neuronen mit sämtlichen Inputs und Outputs verbunden sind.

Listing 10.14 Fully Connected Layer

```
public double FullyConnected(double[] input, double[] weights)
{
    double sum = 0;

    for (int i = 0; i < input.Length; i++)
    {
        sum = sum + (input[i] * weights[i]);
    }

    return sum;
}
```

Diese Schicht stellt ein vollständig verbundenes Netz dar, in dem die endgültige Klassifizierung stattfindet.

10.3.6 Methoden

Die Methode *RectifiedLinearUnit* implementiert die Aktivierungsfunktion ReLU. Die Rectified Linear Unit (gleichgerichtete Lineareinheit) ist eine nicht-lineare Aktivierungsfunktion und wird wie folgt dargestellt:

$$f(x) = \text{Max}(0, x)$$

Der Wert x stellt den Input dar. Gemäß der Gleichung ist die Ausgabe von ReLU der maximale Wert zwischen null (0) und dem Eingabewert. Die Funktion gibt 0 zurück, wenn sie eine negative Eingabe erhält, aber für jeden positiven Wert x gibt sie diesen Wert zurück.

Listing 10.15 Rectified Linear Unit (ReLU)

```
public double[, ,] RectifiedLinearUnit(double[, ,] input)
{
    double[, ,] output = new double[input.GetLength(0), input.GetLength(1),
                                    input.GetLength(2)];

    for (int j = 0; j < input.GetLength(0); j++)
    {
        for (int k = 0; k < input.GetLength(1); k++)
        {
            for (int l = 0; l < input.GetLength(2); l++)
            {
                output[j, k, l] = input[j, k, l] < 0 ? 0 : input[j, k, l];
            }
        }
    }
}
```

```

        }
    }

    return output;
}

```

Die Methode *Filter* definiert die benötigte Filtermatrix und die Methode *RandomWeights* sorgt für die zufällige Vorinitialisierung der Gewichte für die Neuronen.

Listing 10.16 Filter und RandomWeights

```

public double[,,,] Filter(int filter, int nooffilters, int pixelsize)
{
    double[,,,] doubleFilter = new double[filter, nooffilters, pixelsize,
                                              pixelsize];
    Random random = new Random();
    for (int i = 0; i < filter; i++)
    {
        for (int j = 0; j < nooffilters; j++)
        {
            for (int k = 0; k < pixelsize; k++)
            {
                for (int l = 0; l < pixelsize; l++)
                {
                    doubleFilter[i, j, k, l] = random.NextDouble();
                }
            }
        }
    }

    return doubleFilter;
}

public double[] RandomWeights(int count)
{
    double[] weights = new double[count];
    Random random = new Random();

    for (int i = 0; i < count; i++)
    {
        weights[i] = random.NextDouble();
    }

    return weights;
}

```

10.3.7 Training

Die Methode *TrainingForConvolutionalNeuralNetwork* lädt das Testbild in eine Bitmap und zerlegt dieses in die benötigte Struktur für das Netz. Danach werden die entsprechenden Layer im

Netzwerk aktiviert. Das Ergebnis des Fully Connected Layer finden Sie dann in der Variablen *fullyConnectedOutput*. Sie können es dann entsprechend auswerten bzw. weiterverarbeiten.

Listing 10.17 Trainings Routine

```
public void TrainingForConvolutionalNeuralNetwork()
{
    Bitmap img = new Bitmap(@"C:\temp\Test.JPG", true);
    double[,] pixelvalues = new double[3, img.Width, img.Height];

    for (int i = 0; i < img.Width; i++)
    {
        for (int j = 0; j < img.Height; j++)
        {
            Color pixel = img.GetPixel(i, j);
            pixelvalues[0, i, j] = pixel.R;
            pixelvalues[1, i, j] = pixel.G;
            pixelvalues[2, i, j] = pixel.B;
        }
    }

    var filters = this.Filter(1, 3, 3);
    var convolutionOutput = Convolutional(pixelvalues, filters);
    var activationOutput = RectifiedLinearUnit(convolutionOutput);

    var maxPoolingOutput = MaxPooling(activationOutput, 2);
    var flattenOutput = Flatten(maxPoolingOutput);
    double[] weights = this.RandomWeights(flattenOutput.Length);

    var fullyConnectedOutput = FullyConnected(flattenOutput, weights);

    //TODO
    //Evaluation
}
```

In der Klasse *Program* bilden Sie nur noch eine Instanz der Klasse *ConvNet* und rufen die Trainingsmethode auf.

Listing 10.18 Trainingsaufruf

```
class Program
{
    static void Main(string[] args)
    {
        ConvNet convNet = new ConvNet();
        convNet.TrainingForConvolutionalNeuralNetwork();
    }
}
```

Die Übung zeigt nur den grundlegenden Aufbau eines CNN. Sie beinhaltet jedoch sehr viele Anregungen für weitere Anpassungen bzw. Ausarbeitungen und Verbesserungen.

Sie zeigt aber gleichzeitig auch, wie schnell ein einfaches Lernprojekt zur Veranschaulichung umgesetzt werden kann. Soll das Beispiel für größere Aufgaben, wie die Verarbeitung von mehreren Bildern, Filtern und Klassifizierung, umgesetzt werden, so steigt die Komplexität enorm und ist somit für die Darstellung in einem Buch nicht mehr wirklich geeignet. Für Lernzwecke und das Verständnis von neuronalen Netzen ist ein einfaches Lernprojekt jedoch unverzichtbar.

■ 10.4 Objekterkennung

Durch die rasante Entwicklung im Bereich der optischen Sensoren stehen heute auch Daten für die Objekterkennung kostengünstig und in Echtzeit (das heißtt, dass z. B. Sensordaten direkt ausgelesen werden können) für Berechnungen zur Verfügung. Mit der steigenden Leistungsfähigkeit im Bereich Deep Learning und entsprechender Algorithmen und Modelle für die Bild- und Objekterkennung wird es möglich, Objekte zu klassifizieren und z. B. mit 3D-Sensoren auch deren Position im Raum zu bestimmen. Das Institut für mobile Maschinen und Nutzfahrzeuge der TU Braunschweig erforscht auf der Basis von 3D-Sensoren und Deep Learning ganz neue Assistenzsysteme.

Auch für die Analyse von Bilddaten bei Krankheitsdiagnosen werden Deep-Learning-Algorithmen verwendet. So können zum Beispiel auf Röntgenbildern oder CT-Aufnahmen Anomalien erkannt werden. Selbst Google Deepmind verwendete in seinem AlphaGo-System ein CNN, um die aktuelle Spielposition auf dem Go-Board zu evaluieren.

Weitere Anwendungsbeispiele findet man in der Logistik bei der Produkterkennung im Kommissionierbereich mit Kommissionierrobotern oder auch bei autonomen Flurförderzeugen wie Hubwagen und Gabelstaplern. Auch für die Objekterkennung ohne Barcodes, zum Zählen und Messen (Bestimmung der Größe, Feststellen von Abweichungen) von Produkten werden inzwischen Deep-Learning-Technologien eingesetzt. So kann eine schnelle KI-basierte Barcode-Lokalisierung und Objektidentifikation manuelles Scannen ersetzen. Durch das gleichzeitige Zählen vieler Objekte mithilfe von Computer-Vision-Technologien kann man einen sicheren und problemlosen Verpackungsprozess gewährleisten. Somit lässt sich zum Beispiel auch mit der Objekterkennung die Visualisierung eines 3D-Packschemas realisieren.

Das heißtt, in der Praxis sind hier die besonderen Herausforderungen für den Einsatz von Bild- und Objekterkennung mit Deep Learning die benötigten Trainingsdaten, die hohe Rechenleistung für die Echtzeitanwendung und die ausreichende Erkennungsgenauigkeit.

Inzwischen versucht man mit den heute zur Verfügung stehenden Deep-Learning-Technologien den nächsten Evolutionsschritt in der Automatisierung von logistischen Prozessen zu gehen. Hierzu zählt auch beispielsweise die Sortierung von Packstücken unterschiedlichster Art. Für eine angeschlossene Anlage mit Greifrobotern stellt dies potenzielle Sortierungenauigkeiten, Fehlleitungen, Anlagenausfälle, Wartezeiten und eine extrem aufwendige manuelle Nachbearbeitung dar. Mithilfe von gesteuerten Anlagen auf Basis von neuronalen Netzen ist es inzwischen möglich, anhand des gelernten Klassifikationsschemas die Pakete und deren Form exakt zu erkennen, sodass wiederum der Roboter die Sortievorgänge vollständig automatisiert durchführen kann.

Die hierfür auf Deep Learning basierende Technologie ist ein rechenaufwendiger Prozess, der Daten über mehrere Stufen separiert und so die Klassifizierung vornimmt. In solchen Anwendungsbereichen kann es sinnvoll sein, auf entsprechende vortrainierte und erweiterbare Modelle von Microsoft Cognitive Services bzw. Computer Vision oder auf TensorFlow zurückzugreifen, um nicht selbst von Grund auf so ein hochkomplexes Modell entwickeln zu müssen. Somit kann es sehr nützlich sein, auf ein bereits vollständig trainiertes Objekterkennungsmodell zuzugreifen und mithilfe von ML.NET ein bereits trainiertes neuronales Netz zu hosten und auszuführen. Die schnellste Vorgehensweise wäre, ein neuronales vortrainiertes TensorFlow-Modell für die sofortige Verwendung in ML.NET zu implementieren.

10.4.1 Transferlernen mit ML.NET

Im nachfolgenden Beispiel soll ein TensorFlow-Modell, das auf den *ImageNet*-Datensatz trainiert wurde, Verwendung finden.



ImageNet

Bei ImageNet handelt es sich um eine Bilddatenbank [59], die eine nützliche Ressource für Forscher, Dozenten, Studenten und Entwickler darstellt. Die Bilddatenbank ist nach der *WordNet*-Hierarchie [60] organisiert und jeder definierte Knoten, spezifiziert über Substantive wie z. B. Haus, Hund, Katze etc., wird durch Hunderte von Bildern dargestellt und kann somit hervorragend für das Training eines neuronalen Netzes verwendet werden.

TensorFlow erleichtert über seinen Portal TensorFlow Hub [61] (Bild 10.8) die Verwendung von vortrainierten Modellen für das sogenannte Transferlernen. Unter Transferlernen (Transfer-Learning) versteht man das Übertragen der Ergebnisse eines fertig trainierten neuronalen Netzes auf eine neue Aufgabe. Das heißt, Sie beginnen zum Beispiel mit der Verwendung eines vorab trainierten ImageNet-Klassifikator-Modells, um ein Bild zu prognostizieren, somit wäre in diesem Fall keine Schulung des neuronalen Netzes erforderlich. Möchten Sie den Klassifikator für einen Datensatz mit verschiedenen Klassen trainieren, so verwenden Sie das vortrainierte Modell, indem Sie dann benutzerdefiniert die oberste Ebene des Modells neu trainieren, um die Klassen in Ihrem Datensatz zu erkennen. In diesem Fall spricht man von einfachem Transferlernen.

Beim Transferlernen werden die fertig trainierten Layer entweder konstant gehalten und nur am Output Layer nachtrainiert, oder es werden einige bzw. alle Layer auf Basis des neuen aktuellen Trainingsdatensatzes trainiert.

Somit kann es sehr hilfreich sein, mit einfachem Transferlernen auf die bereits gelernten Features eines vortrainierten neuronalen Netzes aufzubauen. Dabei werden zum Beispiel die fertig trainierten Layer eines CNNs übernommen und nur der Output Layer wird auf die Anzahl der zu erkennenden Objektklassen des neuronalen Netzes angepasst und nachtrainiert. Man spricht in diesem Fall auch vom Finetuning des neuronalen Netzes. Der große Vorteil hierbei ist, dass das neuronale Netz Formen und Strukturen schon sehr gut erkennen und unterscheiden kann, es muss nur noch eine neue Zuordnung der Objektklassen lernen.

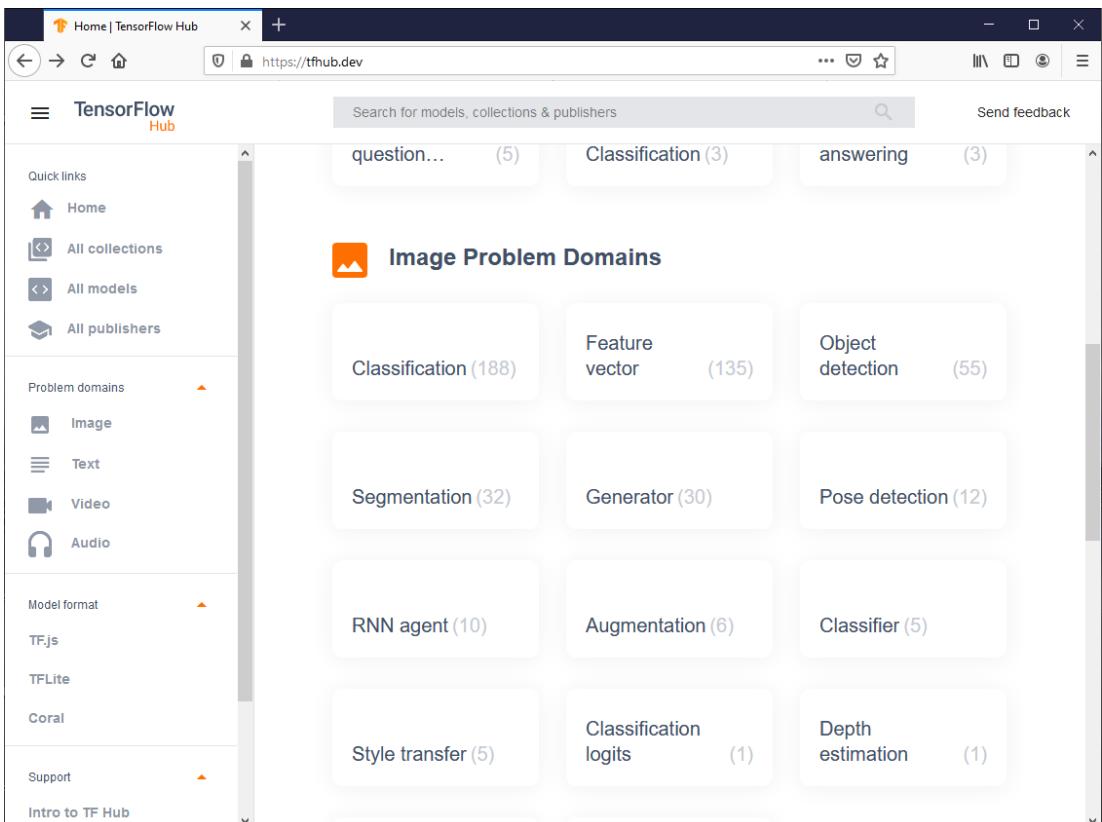


Bild 10.8 Das TensorFlow-Hub-Portal

10.4.2 Neue Bilddaten vorbereiten

Das nachfolgende Beispiel nutzt zur Bildklassifizierung ein vortrainiertes TensorFlow-Modell, das auf dem ImageNet-Klassifikator basiert.

Neben TensorFlow Hub gibt es auch unter GitHub eine weitere Vielzahl von TensorFlow-Modellen. Unter [62] finden Sie für die Verwendung in ML.NET zum Beispiel das passende TensorFlow-ImageNet-Klassifikator-Modell mit der Bezeichnung *tensorflow_inception_graph.pb* unter Apache License Version 2.0.

Möchten Sie ein entsprechendes TensorFlow-Modell, egal ob von GitHub oder TensorFlow Hub, als Modellgraph visualisieren, so können Sie dies über das TensorBoard tun. Um das TensorBoard zu verwenden, gibt es zwei Möglichkeiten, entweder die Ausführung in Google Colab [63] oder die Installation der Quellen von GitHub [64]. Des Weiteren benötigen Sie noch ein Jupyter Notebook mit einem kurzen Python Snippet, um ein vorhandenes Modell zu importieren. Listing 10.19 zeigt das benötigte Python-Skript, um die Modelldatei in TensorBoard zu visualisieren.

Listing 10.19 Import Model to TensorBoard

```

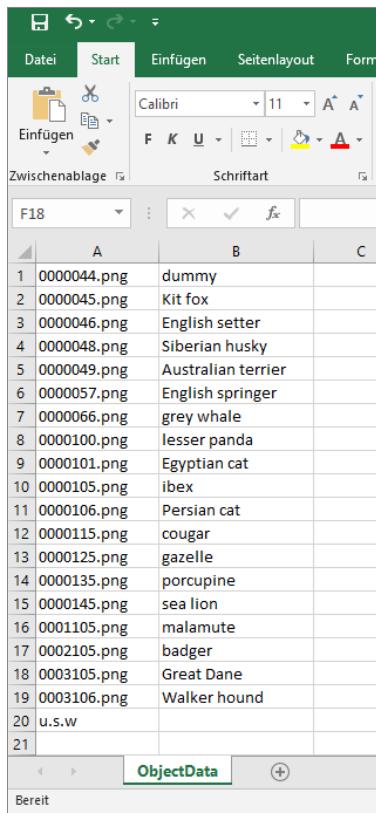
LOG_DIR = 'logs' # Der Pfad, in dem Sie Tensorboard-Ereignisse speichern möchten

with tf.Session() as sess:
    model_filename = 'model.pb' # your model path
    with gfile.FastGFile(model_filename, 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(f.read())
        g_in = tf.import_graph_def(graph_def)
    train_writer = tf.summary.FileWriter(LOG_DIR)
    train_writer.add_graph(sess.graph)

```

Das Beispiel soll es Ihnen ermöglichen, eigene Bilder in einem Ordner mit dem vortrainierten TensorFlow-Modell zu klassifizieren. Dazu legen Sie die Bilder in einen lokalen Ordner auf Ihrem PC und erhalten so ein eigenes Bilder-Set mit *.jpg- oder *.png-Dateien. Sie müssen nur darauf achten, dass die Bilder auch im ImageNet Dataset enthalten sind und Sie auch die englische Bezeichnung für die Merkmale (Labels) benutzen.

Die Merkmale (Labels) der Bilder werden dann in einer CSV-Datei für das Modell erfasst und mit dem Dateinamen des zu testenden Bildes bereitgestellt (Bild 10.9). Das heißt, die Bilddatei *0000046.png* entspricht dem Merkmal (Label) „*English setter*“.



The screenshot shows a Microsoft Excel spreadsheet titled "ObjectData". The table has three columns: A, B, and C. Column A contains file names ending in ".png", and column B contains their corresponding labels. The data is as follows:

	A	B
1	0000044.png	dummy
2	0000045.png	Kit fox
3	0000046.png	English setter
4	0000048.png	Siberian husky
5	0000049.png	Australian terrier
6	0000057.png	English springer
7	0000066.png	grey whale
8	0000100.png	lesser panda
9	0000101.png	Egyptian cat
10	0000105.png	ibex
11	0000106.png	Persian cat
12	0000115.png	cougar
13	0000125.png	gazelle
14	0000135.png	porcupine
15	0000145.png	sea lion
16	0001105.png	malamute
17	0002105.png	badger
18	0003105.png	Great Dane
19	0003106.png	Walker hound
20	u.s.w	
21		

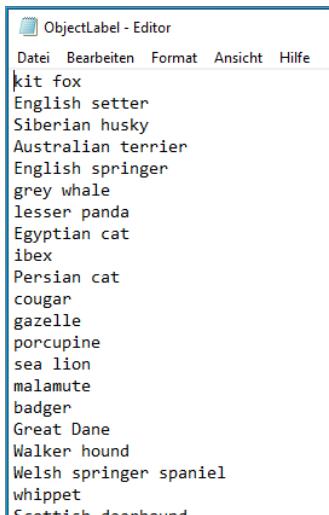
Bild 10.9

Erstellung eines Bilddataset für das Modell

Speichern Sie die Datei als CSV-Datei durch Semikolon getrennt ab. Unter Excel können Sie die Datei über *Datei|Speichern unter* und dann mit der Auswahl des Datentyps *CSV(MS-DOS) (*.csv)* abspeichern.

Weiterhin benötigen Sie für das Beispiel noch eine einfache Textdatei (**.txt*), die das Setzen des Merkmals (Label) für die Bilder übernimmt. Erstellen Sie also eine Textdatei mit dem Namen *ObjectLabel.txt*, in der in jeder Zeile das entsprechende Merkmal für das Bild hinterlegt ist (Bild 10.10).

Sind die gewünschten Bilder in den Ordner kopiert und ist die Bild- und Merkmalsdatei erstellt, können Sie mit der Implementierung des Beispiels fortfahren.



```
ObjectLabel - Editor
Datei Bearbeiten Format Ansicht Hilfe
kit fox
English setter
Siberian husky
Australian terrier
English springer
grey whale
lesser panda
Egyptian cat
ibex
Persian cat
cougar
gazelle
porcupine
sea lion
malamute
badger
Great Dane
Walker hound
Welsh springer spaniel
whippet
Scottish deerhound
```

Bild 10.10
Ausschnitt aus der ObjectLabel-Datei

10.4.3 Trainiertes TensorFlow-Modell verwenden

Sie verwenden das vortrainierte Modell *tensorflow_inception_graph.pb* und machen sich somit den bereits trainierten Identifikationsteil für Ihr eigenes Bild-Dataset zunutze. Die Beispieldaufgabe ist ähnlich gelagert wie die Zeitreihenvorhersage in Abschnitt 10.1.2, nur dass Sie jetzt auf ein Modell von TensorFlow zurückgreifen.

Für die Umsetzung erstellen Sie eine C#-.NET-Core-Konsolenanwendung über die Vorlage *Console App (.NET Core)* aus Visual Studio 2019 und vergeben als Projektnamen die Bezeichnung *ObjectIdentificationDemo*.

Da auch hier wieder das ML.NET Framework zum Einsatz kommt, müssen Sie über den *NuGet-Manager* von Visual Studio dem Projekt folgende Pakete hinzufügen:

- Microsoft.ML
- Microsoft.ML.ImageAnalytics
- Microsoft.ML.TensorFlow
- Microsoft.ML.TensorFlow.Redist
- Microsoft.ML.Vision

Bild 10.11 zeigt die eingebundenen Pakete im Visual-Studio-Projekt. Die Implementierung des Beispielcodes erfolgt auch hier in der *Program.cs*-Datei. Erweitern Sie als Erstes die Datei um die folgenden *using*-Anweisungen.

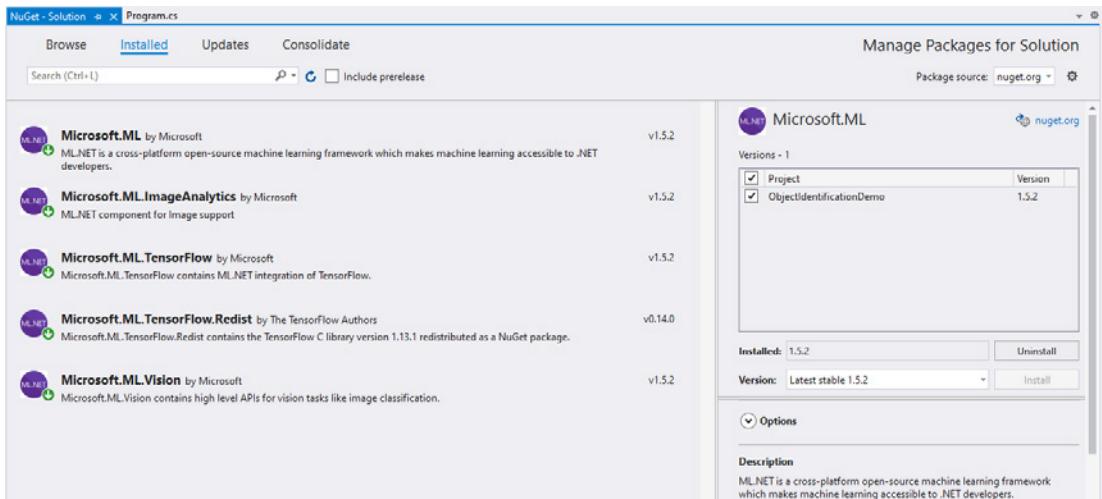


Bild 10.11 Benötigte ML-Pakete für das Projekt

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Microsoft.ML;
using Microsoft.ML.Data;
```

Als Nächstes erstellen Sie die *ImageData*-Klasse für die Übernahme der Daten aus der CSV-Datei. Fügen Sie unter der Klasse *Program* die neue Klasse hinzu.

```
public class ImageData
{
    [LoadColumn(0)] public string Path;
    [LoadColumn(1)] public string Label;

    public static IEnumerable<ImageData> ReadDataFromCsv(string file)
    {
        return File.ReadAllLines(file)
            .Select(x => x.Split(','))
            .Select(x => new ImageData
            {
                Path = x[0],
                Label = x[1]
            });
    }
}
```

Die *ImageData*-Klasse enthält die Spalte *Path* für die Angabe der Bilddatei (Name der Datei und entsprechender Dateityp) und die Spalte *Label* für die Merkmalsausprägung. Des Weiteren benötigen Sie noch eine *ObjectPrediction*-Klasse, welche die vorhergesagten Werte für die Prognose enthält.

```
public class ObjectPrediction
{
    [ColumnName("softmax2")]
    public float[] PredictedLabels;
}
```

10.4.4 MLContext, Pipeline und Prognose

Wie bekannt, ist die *MLContext*-Klasse der Ausgangspunkt für alle ML.NET-Vorgänge. Initialisieren Sie daher in der Main-Methode der Klasse *Program* eine neue *mlContext*-Variable.

Die Daten der Bilder werden dann über die Methode der Klasse *ML.Data.TextLoader* geladen. Das so erstellte Dataset enthält alle Daten aus der vorgegebenen CSV-Datei.

```
var m1Context = new MLContext();

var data = m1Context.Data.LoadFromTextFile<ImageData>(@"ObjectData.csv",
    hasHeader: false);
```

Das Modell für ML.NET wird über die Pipeline spezifiziert. Hier werden die Sequenzen für das Data-Loading, die Transformation und die Lernkomponenten festgelegt. Implementieren Sie die Pipeline in der Main-Methode wie folgt:

```
var pipeline = m1Context.Transforms
    .LoadImages(
        outputColumnName: "input",
        imageFolder: "objectImages",
        inputColumnName: nameof(ImageData.Path))
    .Append(m1Context.Transforms.ResizeImages(
        outputColumnName: "input",
        imageWidth: 224,
        imageHeight: 224,
        inputColumnName: "input"))
    .Append(m1Context.Transforms.ExtractPixels(
        outputColumnName: "input",
        interleavePixelColors: true,
        offsetImage: 117)).Append(m1Context.Model.LoadTensorFlowModel
        ("tensorflow_inception_graph.pb")
    .ScoreTensorFlowModel(
        outputColumnNames: new[] { "softmax2" },
        inputColumnNames: new[] { "input" },
        addBatchDimensionInput: true));
```

Die Pipeline wird in diesem Beispiel mit folgenden Methoden versehen:

- Die Methode *LoadImage* legt das Laden der Bilder fest. Hier wird der Name der Eingabespalte (*inputColumnName*) mit dem Dateinamen und dem Ordner (*imageFolder*), in dem das Bild vorhanden ist, sowie dem Namen der Ausgabespalte (*outputColumnName*) definiert.
- Über *ResizeImages* wird die Größe der Bilder auf das Modell angepasst, da die Bilder im Modell mit einer Größe von 224 * 224 Pixel trainiert wurden.
- Die Methode *ExtractPixels* sorgt für die Verflachung zu einem 1-dimensionalen Float-Array.
- *LoadTensorFlowModel* lädt das angegebene TensorFlow-Modell.
- Die Methode *ScoreTensorFlowModel* speist die Bilddaten in das Modell und sammelt die Scores aus dem Klassifikator auf der Ausgabeseite.

ScoreTensorFlowModel erfordert den Namen des Eingangsknotens, der die Bilddaten empfängt, und den Namen des Ausgabeknotens, der das Ergebnis der Softmax-Aktivierungsfunktion enthält. Die Knotennamen werden im Beispiel mit *input* und *softmax2* bezeichnet.

Als Nächstes führen Sie die *Fit*-Methode aus. Allerdings bewirkt der Aufruf in diesem Fall kein Training, da alle Modellparameter eingefroren sind und das Modell schon vollständig trainiert vorliegt. Die *Fit*-Methode setzt einfach nur die Pipeline zusammen und gibt die benötigte Modellinstanz zurück.

```
Console.WriteLine("Auf die Pipleine wird Fit aufgerufen...");
var model = pipeline.Fit(data);
Console.WriteLine("Fit ausgeführt!");
```

Über die Methode *CreatePredictionEngine* wird die Prognose-Engine erstellt. Die beiden Typ-Argumente sind die Datenklasse der Bildinformation *ImageData* und die Klasse *ObjectPrediction* für die Vorhersage. Zudem wird die Merkmalsliste (Label) aus der Textdatei geladen.

Über die *foreach*-Schleife wird die Prognose-Engine ausgewertet und über eine LINQ-Abfrage der höchste Wert für eine entsprechende Kategorisierungsbezeichnung gesucht und auf der Konsole ausgegeben.

```
var engine = mlContext.Model.CreatePredictionEngine<ImageData,
                                         ObjectPrediction>(model);
var labels = File.ReadAllLines(@"c:/temp/ObjectLabel.txt");
Console.WriteLine("Vorhersage (Prognose der Objekterkennung)...");
var images = ImageData.ReadDataFromCsv(@"ObjectData.csv");
foreach (var image in images)
{
    Console.Write($" [{image.Path}]: ");
    var prediction = engine.Predict(image).PredictedLabels;

    var i = 0;
    var best = (from p in prediction
                select new { Index = i++, Prediction = p })
        .OrderByDescending(p => p.Prediction).First();
    var predictedLabel = labels[best.Index];

    Console.WriteLine($"{predictedLabel} {(predictedLabel != image.Label ?
        "Objekt konnte nicht erkannt werden" : "")}");
}
```

Bild 10.12 zeigt das Ergebnis der Auswertung. Wie Sie am Laufzeitverhalten feststellen können, erfolgt die Auswertung innerhalb von Sekunden. Hier zeigt das verwendete vortrainierte Modell seine Wirkung.



The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is as follows:

```
Auf die Pipeline wird Fit aufgerufen....  
Fit ausgeführt!  
Vorhersage (Prognose der Objekterkennung)....  
[0000044.png]: Sporttasche  
[0000045.png]: Pilz  
[0000046.png]: --> Objekt konnte nicht erkannt werden  
[0000048.png]: Pinscher  
[0000049.png]: --> Objekt konnte nicht erkannt werden  
[0000057.png]: --> Objekt konnte nicht erkannt werden  
[0000066.png]: Grauwal  
[0000100.png]: Panda  
[0000101.png]: Aktienindex  
[0000105.png]: Katze  
  
C:\Entwicklung_Kapitel_10\ObjectIdentificationDemo\ObjectIdentificationDemo\bin\Debug  
\netcoreapp3.1\ObjectIdentificationDemo.exe (process 2760) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debug  
ging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

Bild 10.12 Ausschnitt der vorgenommenen Prognose

Das kurze Beispiel sollte verdeutlichen, dass es durchaus sinnvoll sein kann, vortrainierte neuronale Netze einzusetzen. Die Implementierung ist immer recht übersichtlich und die Verarbeitungsgeschwindigkeit sehr hoch. Schauen Sie also im Vorfeld, ob für Ihren Anwendungsbereich nicht schon ein passendes Modell existiert.

■ 10.5 Natural Language Processing

Das Konzept des Natural Language Processing, kurz NLP, also die Verarbeitung natürlicher Sprache, ist seit einiger Zeit durch Systeme wie Alexa und Siri in den Mittelpunkt der Aufmerksamkeit gerückt. Dabei handelt es sich auch bei NLP um eine alte Forschungsdisziplin, deren Grundlagen auf Arbeiten aus den 1940er- und 1950er-Jahren zurückgehen. Seitdem hat sich Dank Deep Learning vor allem die Sprachanalyse mithilfe von NLP durch den Einsatz von Convolutional Neural Networks (CNN) und Recurrent Neural Networks (RNN) bzw. der erweiterten Form als Long Short-Term Memory Network (LSTM), siehe Abschnitt 5.4, rasant weiterentwickelt.

Heute wird NLP für Aufgaben wie die Standpunktanalyse, Textgegenstandserkennung, Spracherkennung, Schlüsselbegriffserkennung und Dokumentkategorisierung eingesetzt. NLP muss Sprache in Form von gesprochenen Wörtern oder Texten in Form von zusammenhängenden Sätzen oder komplexen Schriftstücken erfassen und den Sinn extrahieren.

Hierfür nutzt NLP eine Vielzahl verschiedenster Techniken, die schrittweise bis zur vollständigen Erfassung der Bedeutung eines Textes durchlaufen werden. In folgenden Teilbereichen wird NLP genutzt:

- Spracherkennung
- Segmentierung zuvor erfasster Sprache in einzelne Wörter und Sätze
- Erkennen der Grundformen der Wörter und Erfassung grammatischer Informationen
- Erkennen der Funktionen einzelner Wörter im Satz, handelt es sich bei dem Wort um ein Subjekt, Verb, Adjektiv, Objekt oder um einen Artikel oder um einen Firmennamen o. Ä.
- Extraktion der Bedeutung von Sätzen und Satzteilen
- Erkennen von Satzzusammenhängen und Satzbezeichnungen

Aufgrund dessen stellt NLP für die folgenden wichtigen Anwendungen eine Kernfunktionalität dar:

- Übersetzen von Sprachen mit zum Beispiel Google Translate oder DeepL
- Sprachdialogsysteme wie z. B. Navigationsgeräte mit Sprachsteuerung oder Chatbots in Call-Centern
- Persönliche Assistenten wie Google, Amazon Alexa und Apple Siri
- Im Bereich Textmining zur Klassifikation eines Problems bei Kundenbeschwerden oder der Sentiment-Analyse zur Vorhersage von Aktienkursen

NLP funktioniert heute auf der Grundlage von Machine Learning bzw. Deep Learning. Hierbei verwenden die Systeme die gespeicherten Wörter und ihren Aufbau wie jede andere Form von Daten. Die Modelle werden mit Redewendungen, Sätzen und ganzen Büchern gefüttert und verarbeiten die Daten dann auf Grundlage grammatischer Regeln und sprachlicher Gepflogenheiten. Diese Modelle werden dann genutzt, um Datenmuster zu erkennen und um zu prognostizieren, welches Wort als Nächstes folgt.

Daher ist vor allem das Transferlernen für die Nutzung von NLP sehr wichtig. Daher werden vielfach bereits vortrainierte Modelle für verschiedene Sprachen eingesetzt. Diese Modelle haben dann schon viele Zusammenhänge gelernt und können so schnell auf einen neuen Anwendungsfall trainiert werden. Durch diese aufgeführten Möglichkeiten ergeben sich inzwischen auch ganz allgemeine Anwendungsbereiche von NLP im Alltag. Dazu zählen das sinnhafte Zusammenfassen von langen Texten, das automatisierte Schreiben von Texten, das Erkennen von Stimmungen des Sprechenden und auch das Erfassen von Stilmitteln wie Ironie, rhetorischen Fragen und Sarkasmus.

10.5.1 Textklassifikation

Im Rahmen der steigenden Flut von Informationen im akademischen Bereich, aber auch in der Industrie und den Medien, ist es erforderlich geworden, Möglichkeiten zu entwickeln, um bestimmte Informationen in sogenannten Sachtexten, hierzu zählen z. B. Kommentare in Zeitungen und Publikationen, schriftliche Interviews oder auch Gebrauchsanweisungen, zu sortieren, zu filtern und zu klassifizieren.

Die Textklassifikation beschäftigt sich mit der Zuordnung von Texten in vordefinierte Klassen. Das heißt, über einen Text-Klassifikator ordnet man einen Text einer Menge von inhaltlichen Kategorien zu. Die Aufgabe der Textklassifikation besteht also darin, Dokumente in Abhängigkeit ihres semantischen Inhalts einer oder mehreren Kategorien oder Klassen zuzuordnen (Bild 10.13). Hierbei unterscheidet man binäre- und mehrwertige Klassifikation:

- Die binäre Klassifikation unterscheidet nur zwei Zustände; das heißt, es gibt nur zwei mögliche Klassen.
- Die Mehrklassen-Klassifikation unterscheidet eine Vielzahl von Klassen (siehe auch Abschnitt 3.6.1.2). Für die Textklassifikation wäre dies zum Beispiel die Zuordnung von Dokumenten zu verschiedenen Bereichen wie Politik, Wirtschaft, Sport usw.
- Unter mehreren disjunkten Klassen versteht man zum Beispiel die Einordnung von E-Mails in verschiedene Ordner. Aber auch das Prinzip „*One against all*“, hierbei trennt jeder Klassifikator Beispiele einer Klasse gegen alle anderen Klassen. Des Weiteren gibt es auch noch die paarweise Diskrimination, hier steht ein Klassifikator für jedes Paar von Klassen bzw. Abstimmungen (diese Einteilung wird für die Codierung sprachlicher Diskriminierung benötigt). Diskrimination bezeichnet die Feststellung von Unterschieden, womit eine Differenzierung von Wörtern möglich ist. So nutzt man dieses Verfahren zum Beispiel bei der Differenzierung von Wörtern mit „ch“ und „sch“.

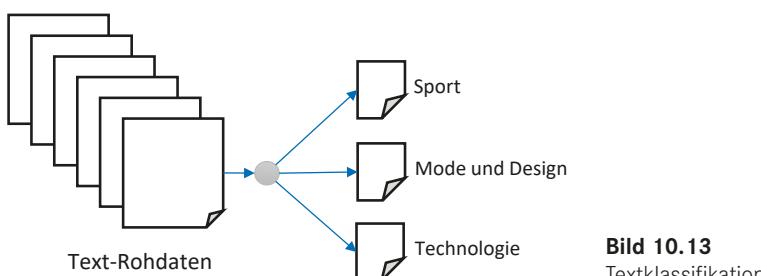


Bild 10.13
Textklassifikation

Der Aufbau eines Textklassifizierers erfolgt als End-to-End-Textklassifikations-Pipeline, die aus folgenden Komponenten besteht (Bild 10.14):

- **Trainingstext:** Hierbei handelt es sich um den Eingabetext (Schulungstext), durch den das Modell in der Lage ist, zu lernen und die erforderliche Klasse vorherzusagen.
- **Label:** Unter dem Label befinden sich die vordefinierten Kategorien/Klassen, die das Modell prognostizieren soll.
- **Machine-Learning-Algorithmus:** Stellt den benötigten Algorithmus, also auch das verwendete neuronale Netz bereit, durch den das Modell in der Lage ist, die Klassifizierung durchzuführen.
- **Prognose-Modell:** Stellt den eigentlichen Klassifikator als Modell dar. Es ist das Modell, auf dem der Datensatz trainiert wurde und welches die Vorhersage ausführt.

Durch die korrekte Markierung (Labeling) der Informationen in den Dokumenten kann das Modell lernen. Bei der Textklassifikation wird sehr häufig auf das Supervised Learning (überwachtes Lernen, Abschnitt 2.5.1) zurückgegriffen. Hier greift vielfach ein Mensch ein, der im Vorfeld die entsprechenden Markierungen im Dokument vornimmt.

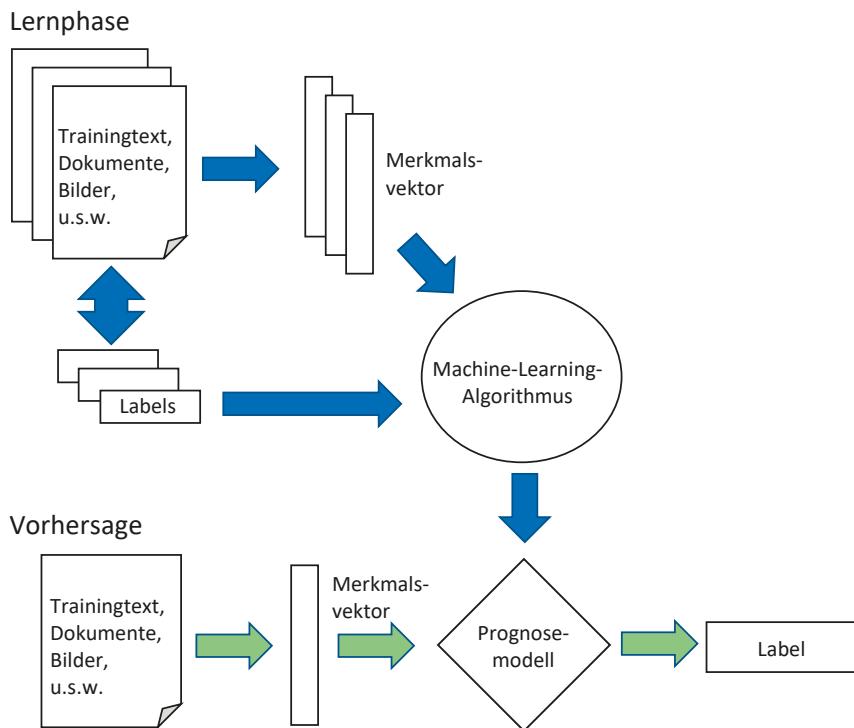


Bild 10.14 Supervised Learning mit Textklassifikator

10.5.2 Merkmalsvektoren (Feature Vectors)

Für die Klassifikation einer Text- bzw. Mustererkennung ist der Merkmalsvektor ein n-dimensionaler Vektor mit numerischen Merkmalen, die ein Objekt beschreiben. Bei der Darstellung von Bildern haben die Merkmale den Pixeln eines Bildes entsprochen, während bei der Darstellung von Texten die Merkmale die Häufigkeit des Auftretens von Textbegriffen sein können. Auch bei der Verwendung einer kurzen Sprachaufnahme zur Textanalyse werden die beschriebenen Merkmale in kleine überlappende Zeitfenster mit einer Länge von ca. 20–60 ms aus dem Audiosignal extrahiert und in einen Vektor fester Länge transformiert.

Wie lassen sich jetzt aber Texte bzw. Wörter in Vektoren mit Fließkommazahlen übernehmen. Hier gibt es unterschiedliche Vorgehensweisen. Das Verfahren hängt stark von der Problemstellung ab. Die bekanntesten Verfahren sind Bag-Of-Words und Word2Vec.

Bag-Of-Words (BOW)

Der Bag-Of-Words, kurz BOW, ist der einfachste Ansatz zur Übersetzung von Text in einen Vektor. Hier wird nach der Bereinigung von Sonderzeichen ein Wörterbuch für den gesamten Text-Korpus erstellt. Danach wird das Wörterbuch absteigend nach der Zahl der Wortvorkommen sortiert. So wird jedem Wort eine Zahl zugeordnet. Die Zahl Null (0) wird dabei nicht verwendet, sie spezifiziert, welche Wörter nicht im Wörterbuch vorhanden sind.

Man spricht diesbezüglich auch von „*Out-Of-Vocabulary*“. Somit repräsentiert die 1 das Wort, welches am häufigsten vorkommt, 2 am zweithäufigsten, 3 am dritthäufigsten und immer so weiter. Der am weitesten verbreitete Ansatz der Berechnung ist die *Term Frequency - Inverse Document Frequency* (TF-IDF)-Technik.

- *Term Frequency* (TF): Anzahl der Auftritte des Terms t in einem Dokument / Anzahl der Terme im Dokument.
- *Inverse Document Frequency* (IDF): $\log(N/n)$, wobei N die Anzahl der Dokumente und n die Anzahl der Dokumente ist, in denen ein Begriff t aufgetreten ist. Die IDF eines seltenen Wortes ist hoch, während die IDF eines häufigen Wortes niedrig ist. Dies hat den Effekt, dass Wörter, die sich voneinander unterscheiden, hervorgehoben werden.

Zum Schluss wird der TF-IDF-Wert eines Terms als $= TF * IDF$ berechnet.

Dieses Vorgehen kann für eine kleine und sehr einfach gestaltete Aufgabe ausreichend sein, der große Nachteil von BOW ist die Anzahl der Wörter im Wörterbuch. Als Input für ein neuronales Netz kann daher jedes Wort nur als *One-Hot-Encoded Vector* [65] verwendet werden, der genauso lang ist wie die Anzahl der Wörter, die das Wörterbuch enthält. Verfügt das Wörterbuch also über 50.000 Wörter, so besteht auch der Vektor aus dieser Länge, der jeweils nur eine 1 und sonst nur Nullen enthält. Diese Art von Vektor nennt man *Sparse Vector* [66]. Bei der Modellierung einer Sparse-Vector-Multiplikation kann es durch die aktuelle CPU-Architektur zu unregelmäßigen Speicherzugriffen und indirekter Speicherreferenz kommen. Dadurch können neuronale Netze bei der Berechnung fehleranfällig werden.

Word2Vec

Das Word2Vec-Verfahren hat sich inzwischen durchgesetzt und BOW in vielen Bereichen der Textklassifizierung abgelöst. Word2Vec kann Wörter über ein ausgefeiltes Verfahren auf Vektoren mit einer hohen Zahl von Dimensionen (gebräuchlich sind ca. 100–300 Dimensionen) abbilden.

Der Word2Vec-Algorithmus wurde 2013 von Tomas Mikolov bei Google entwickelt. Er analysiert die in den Texten vorhandenen Wortzusammenhänge und erstellt dann für jedes Wort, das in den Texten vorkommt, einen Vektor. Wenn Sie also den Satz „Have, a, good, great, day“ betrachten, so lässt sich mit diesen Wörtern ein einstufiger codierter Vektor erstellen. Die Länge des einstufigen codierten Vektors wäre gleich der Anzahl der Wörter, im Beispiel also 5. Somit hätten Sie einen Vektor aus Nullen mit der Ausnahme des Elements im Index, das das entsprechende Wort im Vokabular repräsentiert.

Dieses Element wäre dann eins. Das heißt, die Codierung für den Satz würde wie folgt aussehen: Have = [1,0,0,0,0]; a = [0,1,0,0,0]; good = [0,0,1,0,0]; great = [0,0,0,1,0]; day = [0,0,0,0,1]. Dieses Vorgehen ermöglicht dem Algorithmus, alle im Text-Korpus vorhandenen Wörter anhand ihres Vorkommens und der sie umgebenden Wörter so in einem multidimensionalen Raum anzutragen, dass Wörter, die häufig in ähnlichen Kontexten auftauchen, auch einen ähnlichen Vektor haben. Somit repräsentiert jeder Wort-Vektor zumindest ansatzweise die semantische Bedeutung des Wortes im Kontext.

In der Praxis ist dieser Algorithmus sehr rechenintensiv und benötigt einen sehr großen und möglichst fehlerfreien Text-Korpus, um eine sinnvolle Vektorstruktur zu erzeugen. Das hier aufgezeigte Verfahren, das für Wörter möglichst geeignete Vektoren findet, wird als *Word Embedding* bezeichnet.

Character-level Convolutional Networks for Text Classification

Ein weiteres Verfahren, das Character-level Convolutional Networks for Text Classification, arbeitet mit Folgen von Buchstaben und Zeichen und damit nicht wie die schon vorgestellten Algorithmen mit Folgen von Wörtern. Vorgestellt wurde es von den KI-Forschern Xiang Zhang, Junbo Zhao und Yann Le-Cun von der Universität New York.

Der große Vorteil hierbei ist, dass das Modell Rechtschreibfehler und Emoticons lernen kann. Außerdem kann dasselbe Modell für verschiedene Sprachen verwendet werden, auch für solche, bei denen eine Segmentierung in Wörtern nicht möglich ist.

Bei diesem Verfahren werden Zeichen entsprechend ihrer Position in einem Alphabet durch Vektoren ersetzt. Ist das erste Zeichen ein a, so gilt $a = [1,0,0,0,0\dots]$, wenn b das zweite Zeichen ist, dann gilt $b = [0,1,0,0,0\dots]$ usw. Die Dimension der Vektoren entspricht hierbei immer der Länge des Alphabets, das für die jeweilige Aufgabenstellung zum Einsatz kommt. Neben den normalen Alpha-Zeichen werden auch Ziffern, Leer- und Sonderzeichen in das Alphabet aufgenommen. Ausführlichere Informationen über die Funktionsweise und den Aufbau des Character-Level-CNNs finden Sie in der PDF-Datei unter [67].

Durch die Möglichkeit der Übersetzung von Texten in Vektoren kann man jetzt natürlich die so generierten Vektoren auch als Input für ein Convolutional Neural Network (CNN) oder alternativ für ein Long Short-Term Memory Network (LSTM) verwenden, um so die gewünschte Klassifikation durchzuführen.

10.5.3 Texterkennung mit CNN

Durch den Vorstoß der drei KI-Forscher werden CNNs heute auch, neben der Bildverarbeitung, vermehrt für verschiedene NLP-Aufgaben eingesetzt. Allerdings hat die CNN-Architektur für Sequenzanalysen einige Probleme mit langen Sequenzen. Das führte dann auch zu dem bekannten Problem der Vanishing Gradients [68]. Daraufhin entwickelte man die sogenannten Temporal CNNs [69], indem man CNN-Schichten mit Lücken verbindet (*dilated connections*). Dadurch wächst das Rezeptionsfeld und man benötigt nur noch einen Bruchteil der Schichten, um lange Sequenzen extrahieren zu können. Alternativ können Sie bei sehr kurzen Sequenzen das CNN im Vergleich zu dem von Zhong et al. verkleinern, so zum Beispiel in der Kernel-Größe im ersten und zweiten Layer sowie bei der Anzahl der Filter. Nutzt man dann noch das Verfahren von Character-Level-CNNs, so werden die relevanten Merkmale (Features) automatisch gelernt und müssen nicht aufwendig konstruiert und ausgewählt werden.

Bild 10.15 zeigt die Verarbeitung von Textdaten durch ein CNN. Das Ergebnis jeder Faltung wird ausgelöst, wenn ein spezielles Muster erkannt wird. Variiert man dann noch die Größe der Kerne und die Verkettung ihrer Ausgaben, so ist es möglich, Muster von mehrfacher Größe, das heißt 2, 3 oder 5 benachbarter Wörter, zu erkennen. Die Muster können hierbei auch Textfragmente, sogenannte N-Gramme sein, die vom CNN aber im Satz unabhängig von ihrer Position identifiziert werden können.

Die Schichten in der Abbildung des CNN mit Convolutions, Max-Pooling und Fully-Connected Layer sind stark verkleinert dargestellt.

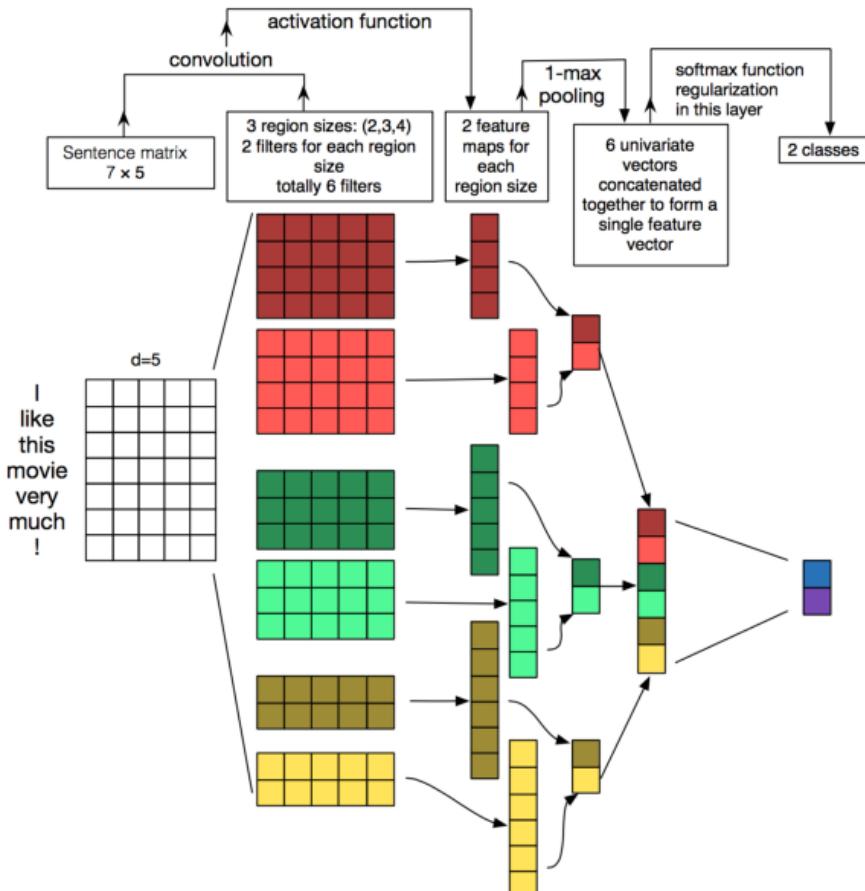


Bild 10.15 CNN für die Textklassifizierung (Quelle: Wildml.com)

10.5.4 Textklassifikation mit RNN

Auch Recurrent Neural Networks eignen sich durch die Verbindungen zwischen den Knoten eines gerichteten Graphen entlang einer Sequenz sehr gut für die Verarbeitung von Texten. Vor allem Long Short-Term Memory (LSTM) Networks sind durch ihren speziellen Speicher geeignet, langfristige Abhängigkeiten zu erlernen. LSTMs bilden eine Sequenz von Blöcken neuronaler Netze, die wie eine Kette miteinander verbunden sind (siehe Abschnitt 5.4). Bei den Textdaten, die verarbeitet werden sollen, muss es sich dann aber auch um einen entsprechenden Sequenztyp handeln. Die Reihenfolge der Wörter ist sehr wichtig für deren Bedeutung. Hierüber kann dann das RNN bzw. LSTM die langfristige Abhängigkeit erfassen.

Bild 10.16 zeigt, wie die Verwendung eines RNN Encoder alle Textinformationen codiert, bevor in einem Feedforward-Netzwerk die Klassifizierung durchgeführt wird.

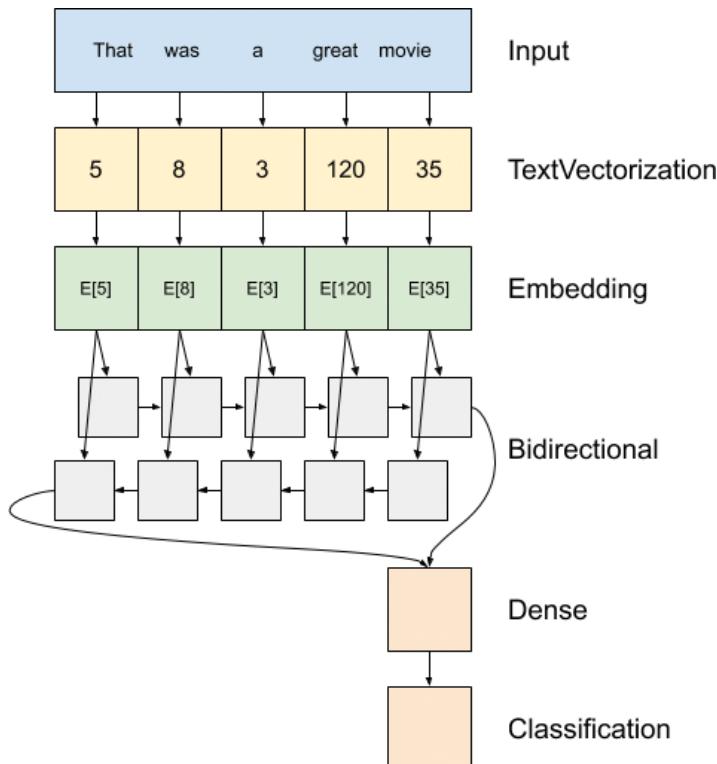


Bild 10.16 Das RNN-Modell als Diagramm (Quelle: TensorFlow.org)

Der erste Layer stellt den Encoder dar, der den Text in eine Folge von Token-Indizes konvertiert. Dann folgt mit Embedding die sogenannte Einbettungsschicht. Diese speichert einen Vektor pro Wort. Beim Aufruf konvertiert sie die Sequenzen von Wortindizes in Sequenzen von Vektoren.

Das RNN verarbeitet jetzt die Sequenzeingabe durch die Iteration durch die Elemente. Die Eingabe wird in diesem Beispielmodell vorwärts und rückwärts durch die RNN-Schicht verarbeitet, bevor sie dann zur Ausgabe verkettet wird. Der *Dense* Layer führt eine endgültige Verarbeitung durch und konvertiert die Vektordarstellung in ein einzelnes Protokoll als Klassifizierungsausgabe.

Wie Sie an dem oben gezeigten kurzen theoretischen Ausflug erkennen konnten, ist die Verarbeitung von Text mit neuronalen Netzen ein sehr komplexes Thema. Bis vor einigen Jahren war es für .NET-Entwickler noch sehr schwierig, C# für NLP-Aufgaben zu verwenden. Viele der angebotenen Frameworks wie *NLTK* (Natural Language Toolkit), *CoreNLP* oder auch *SpaCy* sowie die vortrainierten Modelle von TensorFlow waren Python oder C++ vorbehalten und konnten daher nicht einfach so benutzt werden. Inzwischen können aber dank ML.NET und den Microsoft Cognitive Services auch .NET-Entwickler NLP nutzen. Des Weiteren findet man im Bereich NLP auch immer mehr Open-Source-Projekte, die sich auch mit .NET nutzen lassen, so zum Beispiel das *Standford CoreNLP für .NET*. Beachten Sie aber, dass viele dieser Projekte nur die englische Sprache unterstützen.

Bei der Nutzung von NLP trifft man immer wieder auf Techniken, die bei der Verarbeitung von natürlicher Sprache verwendet werden. Diese sollten Sie auf jeden Fall kennen:

- **Tokenizer:** Die Aufteilung des Textes in Wörtern oder Ausdrücke.
- **Wortstammerkennung und Lemmatisierung:** Hierunter versteht man die Normalisierung von Wörtern, sodass unterschiedliche Formen dem kanonischen Wort mit der gleichen Bedeutung entsprechen. Beispielsweise werden „Ausführung“ und „ausgeführt“ dem Wort „Ausführen“ zugeordnet.
- **Entitätsextraktion:** Beschreibt das Identifizieren von Themen im Text.
- **Wortarterkennung:** Erkennen von Text als Verb, Nomen, Partizip usw.
- **Erkennung von Satzgrenzen:** Erkennung von vollständigen Sätzen innerhalb von Textabschnitten.

Wie man an dieser Aufzählung ersehen kann, stellt die Verwendung von Natural Language Processing die Entwickler vor einige Herausforderungen, da sie die verschiedenen Techniken wie Tokenizer, Wortstammerkennung, Entitätsextraktion etc. für ihre Aufgabe einsetzen müssen. Möchten Sie z. B. eine Sammlung von Freitextdokumenten verarbeiten, so ist dieser Vorgang sehr rechenintensiv. Daher sollten Sie für solche Vorgänge ein standardisiertes Dokumentenformat einsetzen, um gute Ergebnisse bei der Extrahierung bestimmter Fakten aus dem Dokument zu erzielen. Ein Beispiel für die Freitextverarbeitung ist die Verwendung einer Rechnungsnummer und des Rechnungsdatums in einem Dokument. So kann es sich als schwierig erweisen, einen Prozess zu erstellen, der beide Textangaben korrekt extrahiert. Sie sollten bei der Entwicklung von Lösungen im Bereich der Textklassifizierung bzw. Spracherkennung auf vorbereitete Datensätze und Dokumente zurückgreifen, die schon von ML.NET oder TensorFlow verwendet wurden, sodass aufbereitete und vortrainierte Modelle zur Verfügung stehen.

10.5.5 Word Embedding mit ML.NET

Wie schon ausgeführt, müssen Sie, um für Wörter Vorhersagen treffen zu können, diese als Merkmalsvektor (Feature Vector) darstellen, dieses Verfahren bezeichnet das Word Embedding.

Word Embeddings stellen eine Repräsentation von Wörtern basierend auf der Semantik dar und versuchen, die Bedeutung der Wörter mit einer Menge von Zahlen zu erfassen. Durch das Übersetzen in einen Vektor, der dadurch Informationen bereithält, bevor Berechnungen im neuronalen Netz stattgefunden haben, bieten Word Embeddings die Möglichkeit, eine Vielzahl von Methoden zu ergänzen. Gerne nutzt man das Konzept, mit Vektoren ebenso Filme, Webshop-Produkte oder Musiktitel darzustellen. Beachten Sie aber, dass Word Embeddings immer durch die Daten, aus denen sie Informationen gewinnen, begrenzt werden.

Die Methode zur Erzeugung von Word Embeddings, wie zum Beispiel das schon vorgestellte Word2Vec, das von TensorFlow unterstützt wird, lässt sich meist programmiertisch in ein paar übersichtlichen Codezeilen implementieren. ML.NET bietet für die Unterstützung der Natural-Language-Processing-Szenarien durch die Verwendung von Word-Embedding-Transform-Methoden eine schnelle und effektive Umsetzung für das Erzeugen und Benutzen von Merkmalsvektoren an.

Die Transformationsmethoden von ML.NET ermöglichen die Verwendung von vortrainierten Word-Embedding-Modellen in der MLContext-Pipeline. Somit greifen Sie direkt auf ein Modell zu, das Sie für Ihre Zwecke verwenden können, anstatt ein eigenes mühsam erstellen zu müssen. Unter ML.NET stehen die vortrainierten Modelle *GLoVe* [70], *fastText* [71] und *SSWE* [72] zur Verfügung.

Die Verwendung der Methode erfolgt im bekannten MLContext. Listing 10.20 zeigt die Umsetzung des Wortes „example“ in einen Merkmalsvektor mithilfe des *GLoVe*-Modells.

Listing 10.20 WordEmbeddingDemo

```
using System;
using System.Collections.Generic;
using Microsoft.ML;
using Microsoft.ML.Transforms.Text;

namespace WordEmbeddingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var context = new MLContext();
            var emptyData = context.Data.LoadEnumerable(new List<TextInput>());

            var pipeline = context.Transforms.Text.NormalizeText("Text", null,
                keepDiacritics: false, keepPunctuations: false,
                keepNumbers: false)
                .Append(context.Transforms.Text.TokenizeIntoWords("Tokens", "Text"))
                .Append(context.Transforms.Text.ApplyWordEmbedding("Features",
                    "Tokens", WordEmbeddingEstimator.PretrainedModelKind.
                    SentimentSpecificWordEmbedding));

            var transformer = pipeline.Fit(emptyData);
            var predictionEngine = context.Model.CreatePredictionEngine<TextInput,
                TextFeatures>(transformer);

            var wordOfEmbeddingOne = new TextInput { Text = "example" };
            var predictionOne = predictionEngine.Predict(wordOfEmbeddingOne);

            Console.WriteLine("Merkmalsvektor: ");
            foreach (var feature in predictionOne.Features)
            {
                Console.Write($"{feature:F4} ");
            }

            Console.ReadLine();
        }
    }

    public class TextInput
    {
        public string Text { get; set; }
    }
}
```

```

public class TextFeatures
{
    public float[] Features { get; set; }
}

```

Über die Methode *NormalizeText* wird ein *TextNormalizingEstimator* erstellt, der eingehenden Text in eine *inputColumnName*-Spalte normalisiert, indem er optional die Groß-/Kleinschreibung ändert, diakritische Zeichen (bezeichnet einen Zusatz wie Strich, Punkt oder Häkchen über einem Buchstaben), Interpunktionszeichen und Zahlen entfernt und den neuen Text als *outputColumnName*-Spalte ausgibt.

Der nachfolgende *Tokenizer* zerlegt den Text in seine einzelnen Wörter und die Methode *AppyWordEmbedding* erstellt einen *WordEmbeddingEstimator*, der einen Textvektor in einen numerischen Vektor unter Verwendung von vordefinierten Modellen umwandelt.

Der *WordEmbeddingTransformer* erzeugt im Beispiel eine neue Spalte, die wie in den Parametern für den Namen der Ausgabespalte angegeben benannt wird, wobei jeder Eingangsvektor einem numerischen Vektor mit einer Größe der dreifachen Dimensionalität des verwendeten Modells zugeordnet wird. Somit wird die Länge eines Vektors vom Modell festgelegt und bestimmt, wie viele Kontextinformationen ein Vektor darstellen kann.

Im Beispiel wird das Modell GloVe50D verwendet, das selbst 50-dimensional ist, somit ist dann die Ausgabespalte ein Vektor mit einer Größe von 150 Dimensionen. Das Wort „example“ wird somit durch einen Vektor mit 150 Dimensionen dargestellt. Das erste Drittel im Vektor enthält die Mindestwerte, die jeder Zeichenfolge im Eingangsvektor entsprechen. Das zweite Drittel enthält den Durchschnitt der Einbettungen und das letzte Drittel enthält die Maximalwerte der angetroffenen Einbettungen. Durch die Verwendung eines anderen Modells können die Vektoren auch unterschiedliche Informationen beinhalten. Dazu zählen zum Beispiel:

- Wortart
- Geschlecht
- Alter
- Nationalität

Im Beispiel liefert der Min/Max-Wert ein begrenztes Hyper-Rechteck für die Wörter im Vektorraum. Verwendet man Wörter wie „one“, so stellt es von der Wortsemantik eine Anzahl dar, ebenso wie das Wort „many“. In diesem Fall befinden sich deren Vektoren auch näher beieinander als Vektoren von Wörtern, die grundsätzlich einer anderen Wortsemantik angehören. Das heißt, je ähnlicher sich Vektoren sind, desto ähnlicher ist ihre Bedeutung.

Über die *foreach*-Schleife im Beispiel werden dann die Dimensionen des Vektors auf der Konsole ausgegeben. Bild 10.17 zeigt das Ergebnis des übersetzten Wortes in einen entsprechenden Merkmalsvektor.

```
C:\Program Files\dotnet\dotnet.exe
Merkmalesvektor:
0,6867 -2,6825 2,2488 -3,1602 -2,5139 2,3222 0,0583 -2,1370 0,2874 -0,0870 0,8875 -1,4869 -3,11
87 -1,3329 -1,2295 -2,5673 -1,3468 0,9097 -1,3606 -3,4350 -2,3986 -1,5953 0,9694 -1,9100 0,1236
-1,1664 -2,9487 1,4209 0,7973 3,2967 -0,6297 -0,3774 0,7982 0,7299 -0,2727 -0,2719 -3,7419 0,7
111 1,4746 -1,3001 -1,4286 0,3028 2,3492 2,2013 -0,8765 -0,5800 1,6002 -1,1503 2,8612 1,4308 0,
6867 -2,6825 2,2488 -3,1602 -2,5139 2,3222 0,0583 -2,1370 0,2874 -0,0870 0,8875 -1,4869 -3,1187
-1,3329 -1,2295 -2,5673 -1,3468 0,9097 -1,3606 -3,4350 -2,3986 -1,5953 0,9694 -1,9100 0,1236 -
1,1664 -2,9487 1,4209 0,7973 3,2967 -0,6297 -0,3774 0,7982 0,7299 -0,2727 -0,2719 -3,7419 0,711
1 1,4746 -1,3001 -1,4286 0,3028 2,3492 2,2013 -0,8765 -0,5800 1,6002 -1,1503 2,8612 1,4308 0,68
67 -2,6825 2,2488 -3,1602 -2,5139 2,3222 0,0583 -2,1370 0,2874 -0,0870 0,8875 -1,4869 -3,1187 -
1,3329 -1,2295 -2,5673 -1,3468 0,9097 -1,3606 -3,4350 -2,3986 -1,5953 0,9694 -1,9100 0,1236 -1,
1664 -2,9487 1,4209 0,7973 3,2967 -0,6297 -0,3774 0,7982 0,7299 -0,2727 -0,2719 -3,7419 0,7111
1,4746 -1,3001 -1,4286 0,3028 2,3492 2,2013 -0,8765 -0,5800 1,6002 -1,1503 2,8612 1,4308
```

Bild 10.17 Das Ergebnis als Merkmalsvektor

Die so erzeugten Vektoren können dann zur Lösung verschiedenster Problemstellungen aus dem Bereich NLP beitragen, da die Vektoren im Vektorraum die Wörter nach ihrer Bedeutung anordnen und ähnliche Worte nahe beieinanderliegen und die Bedeutung berechnet werden kann. Das heißt, Sie können mit den Vektoren eine Berechnung in Form von [Berlin] – [Deutschland] + [Italien] = [Rom] durchführen. Sie errechnen in diesem Beispiel einen neuen Vektor, indem Sie eine Berechnung mit dem Vektor für das Wort [Berlin] minus den Vektor für das Wort [Deutschland] plus den Vektor für das Wort [Italien] ausführen. Als Ergebnis erhalten Sie einen neuen Vektor mit dem Wert [Rom], da dieser der dem Ergebnis am nächsten liegenden Vektor ist und somit als Ergebnis ermittelt werden kann. Sie können auch mithilfe der linearen Algebra eine Fragestellung wie „Frau verhält sich zu Mann wie Mädchen zu?“ definieren. Durch diese Möglichkeiten stellt Word Embedding ein wichtiges Werkzeug im Umfeld von NLP dar.

10.5.6 Stopwörter

Unter dem Begriff Stopwort (stopword) versteht man im Bereich NLP ein Wort ohne wirklichen Informationsgehalt bzw. welches nur sehr wenige nützliche Informationen enthält. Im Deutschen sind das vor allem Wörter wie „und“, „weil“, „der“, „solche“ oder auch „als“. Stopwörter im Englischen sind „a“, „the“, „is“, „are“ und so weiter.

Stopwörter werden unter NLP überall dort eingesetzt, wo Texte inhaltlich ausgewertet werden. Vielfach haben Wörter, wie „der“ oder „als“ im Text keine entsprechende Gewichtung und machen eine Analyse des Textes nur unnötig rechenintensiv, da sie im Text überproportional oft vorkommen. Auch für den Einsatz von Stopwörtern verfügt ML.NET über die passenden Methoden im MLContext. Listing 10.21 zeigt ein einfaches Beispiel am Satz „Es dauert ziemlich lange, bis die Seite gedruckt wird.“

Listing 10.21 WorkWithStopWords

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.ML;
```

```

namespace WorkWithStopWords
{
    class Program
    {
        static void Main(string[] args)
        {
            var context = new MLContext();
            var emptyData = new List<TextData>();
            var data = context.Data.LoadFromEnumerable(emptyData);

            var pipeline = context.Transforms.Text.TokenizeIntoWords("Tokens",
                "Text", separators: new[] { ' ', '.', ',' })
                .Append(context.Transforms.Text.RemoveDefaultStopWords("Tokens",
                    "Tokens",
                    Microsoft.ML.Transforms.Text.StopWordsRemovingEstimator.Language.
                    German));

            var stopWords = pipeline.Fit(data);

            var engine = context.Model.CreatePredictionEngine<TextData,
                TextTokens>(stopWords);

            var newText = engine.Predict(new TextData { Text = "Es dauert ziemlich
                lange bis die Seite gedruckt wird." });

            var sb = new StringBuilder();

            foreach (var token in newText.Tokens)
            {
                sb.AppendLine(token);
            }

            Console.WriteLine(sb.ToString());
            Console.ReadLine();
        }
    }

    public class TextTokens
    {
        public string[] Tokens { get; set; }
    }

    public class TextData
    {
        public string Text { get; set; }
    }
}

```

Hier erstellt der resultierende *StopWordsRemovingTransform* eine neue Spalte, die wie im Parameter für den Namen der Ausgabespalte angegeben benannt ist und füllt diese mit einem Vektor von Wörtern. Der Vektor enthält dann alle Wörter in der Eingabespalte mit Ausnahme der vordefinierten Liste von Stopwörtern für die angegebene Sprache.

The screenshot shows a Visual Studio code editor with the file 'Program.cs' open. The code demonstrates how to use ML.NET to tokenize text and remove stop words. A dropdown menu is open at the bottom right, showing language options: Danish, Dutch, English, French, German, Italian, Japanese, Norwegian_Bokmal, and Polish. 'German' is selected.

```

10  static void Main(string[] args)
11  {
12      var context = new MLContext();
13      var emptyData = new List<TextData>();
14      var data = context.Data.LoadFromEnumerable(emptyData);
15
16      var pipeline = context.Transforms.Text.TokenizeIntoWords("Tokens", "Text", separators: new[] { ' ', '.', ',' });
17      .Append(context.Transforms.Text.RemoveDefaultStopwords("Tokens", "Tokens",
18          Microsoft.ML.Transforms.Text.StopwordsRemovingEstimator.Language.German));
19
20      var stopwords = pipeline.Fit(data);
21
22      var engine = context.Model.CreatePredictionEngine<TextData, TextTokens>(stopwords);
23
24      var newText = engine.Predict(new TextData { Text = "Das ist ein Testsat" });
25
26      var sb = new StringBuilder();
27
28      foreach (var token in newText.Tokens)
29      {
30          sb.AppendLine(token);
31      }

```

Bild 10.18 Auswahl der Sprache für die Stopwörterliste bei der Codierung

Führt man das Beispiel aus, so werden nur noch die Wörter ausgegeben, die thematisch relevant sind und nicht der Stopwörterliste entsprechen (Bild 10.19).

The terminal window shows the output of the executed code. It prints the words 'dauert', 'ziemlich', 'Seite', and 'gedruckt' on separate lines, indicating that stop words have been removed from the input text.

Bild 10.19
Ausgabe des Satzes ohne Stopwörter

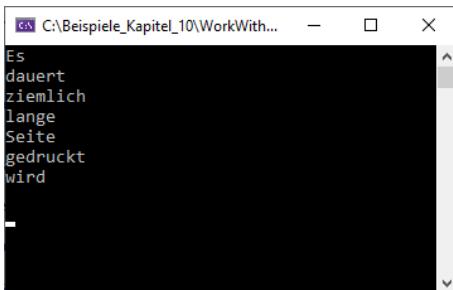
Über die Methode *RemoveStopWords* können Sie aber auch selber eine Liste von Stopwörtern definieren.

```

var pipeline = context.Transforms.Text.TokenizeIntoWords("Tokens", "Text",
separators: new[] { ' ', '.', ',' })
    .Append(context.Transforms.Text.RemoveStopWords("Tokens", "Tokens",
new[] { "bis", "die" }));

```

Stopwörter werden im Allgemeinen als ein einzelner Satz von Wörtern betrachtet. Sie können für verschiedene Anwendungen auch wirklich unterschiedliche Bedeutung haben. In vielen domänenspezifischen Fällen, wie zum Beispiel in medizinischen, technikwissenschaftlichen oder literarischen Texten, kann es sein, dass Sie einen eigenen Satz von Stopwörtern benötigen. In solchen Fällen sollten Sie immer eine spezifische Stopwörterliste erstellen und diese anstelle der ML.NET-Stopwörterliste verwenden. Bild 10.20 zeigt die Ausgabe des Satzes unter Verwendung einer eigenen Stopwörterliste.

**Bild 10.20**

Einsatz der eigenen Stopwortliste

■ 10.6 Stanford CoreNLP für .NET

Neben den umfangreichen ML-Frameworks und KI-Services von Microsoft, TensorFlow, Google und Amazon stellt die Universität Stanford ein kleines und übersichtliches Analysewerkzeug für natürliche Spracherkennung (NLP) unter der GNU General Public License für .NET zur Verfügung.

Bei *CoreNLP* [73] handelt es sich um ein Java-basiertes Open-Source-Projekt, das zurzeit sechs Sprachen unterstützt. Dazu zählen: Arabisch, Chinesisch, Englisch, Französisch, Deutsch und Spanisch. CoreNLP bietet die Analyse aus einer Texteingabe in Bezug auf Wörter, Wortteile, und ob es sich um Namen von Firmen, Personen usw. handelt. Des Weiteren ist es möglich, Daten, Zeiten und numerische Größen zu normalisieren und die Struktur von Sätzen in Form von Phrasen und Wortabhängigkeiten zu kennzeichnen und anzugeben, welche Substantivphrasen sich auf dieselben Entitäten beziehen.

Mit CoreNLP lassen sich Texte, einschließlich Token- und Satzgrenzen, Koreferent [74], Sentiment, Zitate, Attributionen und Beziehungen analysieren. Das Framework stellt einzelne Tools für die gewünschte Analyse bereit, wie zum Beispiel ein Stimmungsanalyse-Tool oder einen Part of Speech Tagger (POS-Tagger), der jeden Token seiner syntaktischen Wortart zuordnet. Die hierfür benötigten Modeldateien für die Sprachanalyse stehen als separater Download [75] bereit. Sie können bei CoreNLP ganz komfortabel über eine Option wählen, welche Tools aktiviert und welche deaktiviert werden sollen.

CoreNLP macht es Ihnen sehr einfach, eine Reihe von Sprachanalyse-Tools auf einen Text anzuwenden. Ausgehend von einem einfachen Text können Sie alle Tools mit nur zwei Codezeilen ausführen. Das Framework wird dabei ganz einfach über den NuGet-Manager von Visual Studio in Ihr Projekt integriert.

```
PM > Install-Package Stanford.NLP.CoreNLP
```

Das Kernstück von CoreNLP ist auch eine Pipeline, äquivalent zu ML.NET. Die Pipeline nimmt den Text auf, führt eine Reihe von gewünschten *NLP-Annotatonen* über den Text aus und erzeugt einen endgültigen Satz von Text-Anmerkungen, die dann als *Labeling*-Text verwendet werden können. Listing 10.22 zeigt ein einfaches Beispiel für die Annotation von Text.

Listing 10.22 CoreNLP Demo

```

using System;
using System.IO;
using java.util;
using java.io;
using edu.stanford.nlp.pipeline;

namespace CoreNLP_Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            var jarRoot = @"..\..\..\..\..\data\paket-files\nlp.stanford.edu\
                           stanford-corenlp-full-2016-10-31\models";

            var text = "This is a long text for the CoreNLP .NET Framework";

            var props = new Properties();
            props.setProperty("annotators", "tokenize, ssplit, pos, lemma, ner,
                               parse, dcoref");
            props.setProperty("ner.useSUTime", "0");

            var curDir = Environment.CurrentDirectory;
            Directory.SetCurrentDirectory(jarRoot);
            var pipeline = new StanfordCoreNLP(props);
            Directory.SetCurrentDirectory(curDir);

            var annotation = new Annotation(text);
            pipeline.annotate(annotation);

            using (var stream = new ByteArrayOutputStream())
            {
                pipeline.prettyPrint(annotation, new PrintWriter(stream));
                System.Console.WriteLine(stream.ToString());
                stream.Close();
            }
        }
    }
}

```

Über die Variable *jarRoot* wird der Pfad zum Ordner mit den benötigten extrahierten Modellen von CoreNLP angelegt. Die Variable *text* definiert den Rohtext für das Modell.

Dann wird über die Variable *props* die Konfiguration für die Pipeline definiert und anschließend werden über das aktuelle Verzeichnis die Modeldateien für die Pipeline erstellt. Über die *Annotate*-Methode erfolgt das Ausführen der CoreNLP-Modelle und eine entsprechende Ausgabe in der Windows-Konsole.

CoreNLP for .NET lässt sich mit sehr überschaubarem Aufwand auch in ein kleines .NET-Projekt implementieren. Vor allem im Bereich von Schulung und Forschung kann es sehr hilfreich sein, den Weg der Text-Klassifizierung an einem einfachen Beispiel zu zeigen. CoreNLP kann den Einstieg in das Thema Natural Language Processing und die Prozesse dahinter ungemein erleichtern. Hier lohnt sich auf jeden Fall ein Blick auf das Projekt der Stanford Universität.

■ 10.7 Sentiment-Analyse

Eine besondere Herausforderung stellt unter NLP die Sentiment-Analyse (Stimmungsanalyse) dar. Die Sentiment-Analyse wird eingesetzt, um Stimmungen aus einer Benutzeräußerung zu ermitteln. Es werden Methoden der automatischen Textkategorisierung für das Klassifizieren von Rezensionen hinsichtlich deren Stimmung (positiv oder negativ) untersucht. Für diesen Anwendungsfall benötigt man im NLP Techniken wie das sogenannte *Tagging*, die Bestimmung von Wortklassen, das *Parsing*, die Bestimmung von Satz-Konstituenten (*Parsing* bezeichnet eine sprachliche Einheit, die Teil einer größeren komplexen Einheit ist) und die Entitäten-Erkennung, die als *Named Entity Recognition* (NER) bezeichnet wird und deren Methoden es erlauben, Angaben wie Namen, Orte und Produkte im Textkorpus zu erkennen und zu kennzeichnen.

10.7.1 Sentiment

Das Wort Sentiment stammt aus dem Französischen und bedeutet so viel wie Gefühl oder Empfindung. Bei NLP versteht man unter der Sentiment-Analyse ein Verfahren, bei dem aus einem Text die emotionale Aussage identifiziert und quantifiziert wird.

Das heißt, bei der Sentiment-Analyse versucht man, den emotionalen Ton hinter den Wörtern, wie zum Beispiel „wow“ oder „beautiful“, zu verstehen. Wobei diese Wörter ja auch negativ besetzt sein können. Daher ist es wichtig, Wörter für die Sentiment-Analyse in einem entsprechenden Kontext zu prüfen und zu bewerten.

Prinzipiell können über eine Sentiment-Analyse alle Arten von Text untersucht werden, in den meisten Fällen bezieht man sich aber auf den Bereich des Social-Media-Monitorings wie Facebook und Twitter aber auch auf YouTube, Web-Shops und Web-Portale, um entsprechende Produkt- und Dienstleistungsbewertungen auszuwerten.

Beim Einsatz von NLP wird dem Modell mittels Beispieldaten beigebracht, ob es sich um positive oder negative Äußerungen handelt. Somit werden die Texte anhand der verwendeten Wörter oder Ausdrücke als positiv oder negativ klassifiziert. Für die Quantifizierung dient der Wertebereich von 1 und 0, wobei 1 als positiv und 0 als negativ bewertet wird. Es gibt auch Datasets, bei denen es genau umgekehrt ist und somit die 1 negativ und die 0 als positiv ausgewertet werden. Durch den Einsatz von Beispieldaten existieren bereits vorhandene Wertzuweisungen und man kann somit auch unbekannte Texte mit dem Modell analysieren und bewerten.

10.7.2 Sentiment-Analyse mit ML.NET

Das nachfolgende Beispiel führt eine einfache Sentiment-Analyse mithilfe von ML.NET durch. Als vorbereitete Daten nutzt das Beispiel ein Dataset aus einer Textdatei von „From Group to Individual Label using Deep Features“ von Kotzias et. al., KDD2015. Laden Sie die ZIP-Datei des Datasets *Sentiment Labelled Sentences* [76] herunter und entpacken Sie die Datei. Kopieren Sie die Datei *yelp.labelled.txt* in Ihr für das Beispiel verwendetes Arbeitsverzeichnis.

Bei dem Dataset handelt es sich um eine fiktive Benutzermeinung für das Empfehlungsportal des Internetunternehmens Yelp, Inc. Das Dataset beinhaltet die Meinung als Text, wobei eine positive Meinung mit 1 und eine negative mit 0 bezeichnet wird. Bild 10.21 zeigt den Aufbau der Textdatei.

```

yelp_labelled.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe
Now... Loved this place. 1
Crust is not good. 0
Not tasty and the texture was just nasty. 0
Stopped by during the late May bank holiday off Rick Steve recommendation and loved it. 1
The selection on the menu was great and so were the prices. 1
Now I am getting angry and I want my damn pho. 0
Honestly it didn't taste THAT fresh.) 0
The potatoes were like rubber and you could tell they had been made up ahead of time being kept under a warmer. 0
The fries were great too. 1
A great touch. 1
Service was very prompt. 1
Would not go back. 0
The cashier had no care what so ever on what I had to say it still ended up being wayyy overpriced. 0
I tried the Cape Cod ravoli, chicken,with cranberry...mmmm! 1
I was disgusted because I was pretty sure that was human hair. 0
I was shocked because no signs indicate cash only. 0
Highly recommended. 1
Waitress was a little slow in service. 0
This place is not worth your time, let alone Vegas. 0
did not like at all. 0
The Burritos Blah! 0
The food, amazing. 1
Service is also cute. 1
I could care less... The interior is just beautiful. 1
So they performed. 1
That's right....the red velvet cake....ohhh this stuff is so good. 1
- They never brought a salad we asked for. 0
This hole in the wall has great Mexican street tacos, and friendly staff. 1
Took an hour to get our food only 4 tables in restaurant my food was Luke warm, Our sever was running around like he was
The worst was the salmon sashimi. 0
Also there are combos like a burger, fries, and beer for 23 which is a decent deal. 1
This was like the final blow! 0
I found this place by accident and I could not be happier. 1
seems like a good quick place to grab a bite of some familiar pub food, but do yourself a favor and look elsewhere. 0
Overall, I like this place a lot. 1

```

Bild 10.21 Die verwendete Textdatei für das Dataset

Liegen die Daten für das Beispiel vor, können Sie mit dem Erstellen einer .NET Core Console App mit dem Namen *SentimentAnalyseDemo* beginnen. Ist das Projekt erstellt, installieren Sie das ML.NET Framework über den NuGet-Manager von Visual Studio.

Die Implementierung des Beispielcodes erfolgt in der *Program.cs*-Datei. Erweitern Sie als Erstes die Datei um die folgenden *using*-Anweisungen.

```

using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.ML;
using Microsoft.ML.Data;

```

Als Nächstes benötigen Sie eine Klasse für die Übernahme der Daten aus der Textdatei und eine weitere Klasse für die Werte, die bei der Prognose benötigt werden.

```

public class SentimentData
{
    [LoadColumn(0)]
    public string SentimentText;

    [LoadColumn(1), ColumnName("Label")]
    public bool Sentiment;
}

public class SentimentPrediction : SentimentData
{

    [ColumnName("PredictedLabel")]
    public bool Prediction { get; set; }
    public float Probability { get; set; }
    public float Score { get; set; }
}

```

Die Klasse *SentimentData* enthält die Kommentare als *SentimentText* und einen *bool*-Wert für die Quantifizierung, entweder 1 für positiv oder 0 für negativ. Die *LoadColumn*-Attribute beschreiben die Reihenfolge der Felder aus dem Dataset. Für das Modelltraining wird die *SentimentPrediction*-Klasse als Prognoseklasse benutzt. Der *boolsche* Wert *Prediction* ist der Wert, den das Modell vorhersagt.

Nachdem Sie die Prognose-Klassen erstellt haben, können Sie in der Main-Methode die *MLContext*-Klasse für die ML.NET-Umgebung in Ihrem Projekt implementieren. Listing 10.23 zeigt die komplette Implementierung der Stimmungsanalyse.

Listing 10.23 SentimentAnalyseDemo

```

class Program
{
    private static readonly string DataPath = @"C:/temp/yelp_labelled.txt";
    static void Main(string[] args)
    {
        var m1Context = new MLContext(seed: 1);
        var sentimentModel = BuildSentimentModel(m1Context);

        var opinions = new List<SentimentData>
        {
            new SentimentData {SentimentText = "This is an awful!"},
            new SentimentData {SentimentText = "This is excellent!"},
            new SentimentData {SentimentText = "Service was very prompt."},
            new SentimentData {SentimentText = "This was like the final
                                         blow!"},
        };
        PredictSentiment(m1Context, sentimentModel, opinions);

        Console.ReadLine();
    }

    private static ITransformer BuildSentimentModel(MLContext m1Context)

```

```

    {
        var data = mlContext.Data.LoadFromTextFile<SentimentData>
            (path: DataPath, hasHeader: true, separatorChar: '\t');

        var dataProcessPipeLine = mlContext.Transforms.Text
            .FeaturizeText(outputColumnName: "Features", inputColumnName:
                nameof(SentimentData.SentimentText));

        var trainingPipeLine = dataProcessPipeLine
            .Append(mlContext.BinaryClassification.Trainers.
                SdcaLogisticRegression(labelColumnName: "Label",
                featureColumnName: "Features"));

        var cvResults = mlContext.BinaryClassification
            .CrossValidate(data, estimator: trainingPipeLine);

        var accs = cvResults.Select(r => r.Metrics.Accuracy);
        var auc = cvResults.Select(r => r.Metrics.AreaUnderRocCurve);
        var f1s = cvResults.Select(r => r.Metrics.F1Score);
        var model = trainingPipeLine.Fit(data);

        Console.WriteLine("Bewertung der Qualitätsmetriken des Modells:");
        Console.WriteLine($"Accuracy: {accs.Average():P2}");
        Console.WriteLine($"Auc: {auc.Average():P2}");
        Console.WriteLine($"F1Score: {f1s.Average():P2}");
        Console.WriteLine("=====");

        return model;
    }

    private static void PredictSentiment(MLContext mlContext,
        ITransformer sentimentModel, List<SentimentData> opinions)
    {
        var predEngine = mlContext.Model.CreatePredictionEngine
            <SentimentData, SentimentPrediction>(sentimentModel);

        Console.WriteLine("\nText | Prognose | Wahrscheinlichkeit positiv");
        foreach (var item in opinions)
        {
            var resultprediction = predEngine.Predict(item);
            var predSentiment = Convert
                ..ToBoolean(resultprediction.Prediction)

                ? "positiv" : "negativ";

            Console.WriteLine("{0} | {1} | {2}",
                item.SentimentText, predSentiment,
                resultprediction.Probability);
        }
    }
}

```

In der *static*-Methode *ITransformer BuildSentimentModel* laden Sie das Beispieldataset und konvertieren über die *FeaturizeText*-Methode die Textspalte (*SentimentText*) in eine Spalte vom

Typ *Features* mit einem numerischen Schlüssel, der vom ML-Algorithmus verwendet wird. Das Beispiel verwendet einen Klassifizierungsalgorithmus, der Elemente oder Datenzeilen kategorisiert. Die Anwendung führt die binäre Klassifizierungsaufgabe auf die Kommentare aus und teilt diese entweder in positiv oder negativ ein.

SdcaLogisticRegressionBinaryTrainer ist der verwendete Klassifizierungsalgorithmus für das Training. Dieser wird an die Pipeline angefügt und akzeptiert den mit Features ausgestatteten *SentimentText* (Features) und die Label-Eingabeparameter. Um jetzt das Modell mit Kreuzvalidierung [77] zu trainieren, verwenden Sie die *CrossValidate*-Methode.

Das Evaluieren des Modells erfolgt auf Metriken und kann über die *Metrics*-Eigenschaft der einzelnen *CrossValidationResult*-Objekte durchgeführt werden. Die *Accuracy*-Metrik ermittelt die Genauigkeit eines Modells, das heißt den Anteil der korrekten Prognosen im Beispieldatensatz. Über die Metrik *AreaUnderRocCurve* ermitteln Sie, wie sicher das Modell die positiven und negativen Klassen korrekt klassifiziert. Der Wert der Metrik sollte so nah wie möglich bei 100 % (1) liegen. Mit der *F1Score*-Metrik ermitteln Sie das Maß für das Gleichgewicht zwischen Genauigkeit und Wiedererkennung. Auch der F1Score-Wert sollte so nah wie möglich bei 100 % (1) liegen.

Zum Schluss übernimmt die Methode *PredictSentiment* die Vorhersage für die in der Liste übergebenen Kommentare.

Die *PredictionEngine* ermöglicht die Prognose für eine einzelne Instanz der Daten. In der Methode *PredictSentiment* übergeben Sie die Kommentare an die *PredictionEngine* und die *Predict*-Methode trifft eine Vorhersage für die einzelnen Datenzeilen. Bild 10.22 zeigt das Ergebnis der Sentiment-Analyse.

```

C:\Beispiele_Kapitel_10\SentimentAnalyseDemo\Sentiment...
Bewertung der Qualitätsmetriken des Modells:
Accuracy: 79,63 %
Auc: 88,20 %
F1Score: 79,84 %
=====
Text      | Prognose | Wahrscheinlichkeit positiv
This is an awful! | positiv | 0,93246675
This is excellent! | positiv | 0,98606426
Service was very prompt. | positiv | 0,8853948
This was like the final blow! | negativ | 0,2528711

```

Bild 10.22 Auswertung der Stimmungsanalyse

Sie können das Modell noch verbessern, indem Sie die Anzahl der Trainingsdaten erhöhen und auch eine Trennung zwischen Trainings- und Testdatensätzen vornehmen. Alternativ ist es auch möglich, eine Sentiment-Analyse mithilfe von AutoML zu erstellen und das Modell dann in einer Web-App zu verwenden.

10.7.3 Sentiment-Analyse mit AutoML

In Abschnitt 7.5 haben Sie schon AutoML und den Model Builder kennengelernt. Auch mit dem Model Builder lässt sich das Beispiel für Sentiment-Analyse sehr schnell umsetzen.

Da Beispieldatensätze aus der Textdatei zur Verfügung stehen, können Sie zunächst mit dem Model Builder ein ML-Modell aufbauen und anhand der vorhandenen Daten trainieren und bewerten, wie gut das Modell ist. Anschließend wird das Modell über eine ASP.NET Core Web-App konsumiert, sodass Sie eine Quantifizierung von neuen Kommentaren oder Rezensionen vornehmen können.

Sind alle Voraussetzungen für den Model Builder installiert, können Sie das Beispiel ganz einfach mit Visual Studio umsetzen. Sollten Sie das Model Builder Tool noch nicht installiert haben, finden Sie die benötigten Hinweise dazu in Abschnitt 7.5.4, „Einbindung ins Projekt“.

10.7.4 Modell erstellen mit dem Model Builder

Als Erstes erstellen Sie über Visual Studio ein entsprechendes Projekt. Die ASP.NET-Core-Web-Application-Vorlage ist ein Template für das Entwickeln von Enterprise-Web-Apps. Wählen Sie in Visual Studio *Create a new project* aus und anschließend das Projekt *ASP.NET Core Web Application*. Klicken Sie auf *Next* und markieren Sie die Vorlage *Web Application (Model-View-Controller)* wie in Bild 10.23 dargestellt.

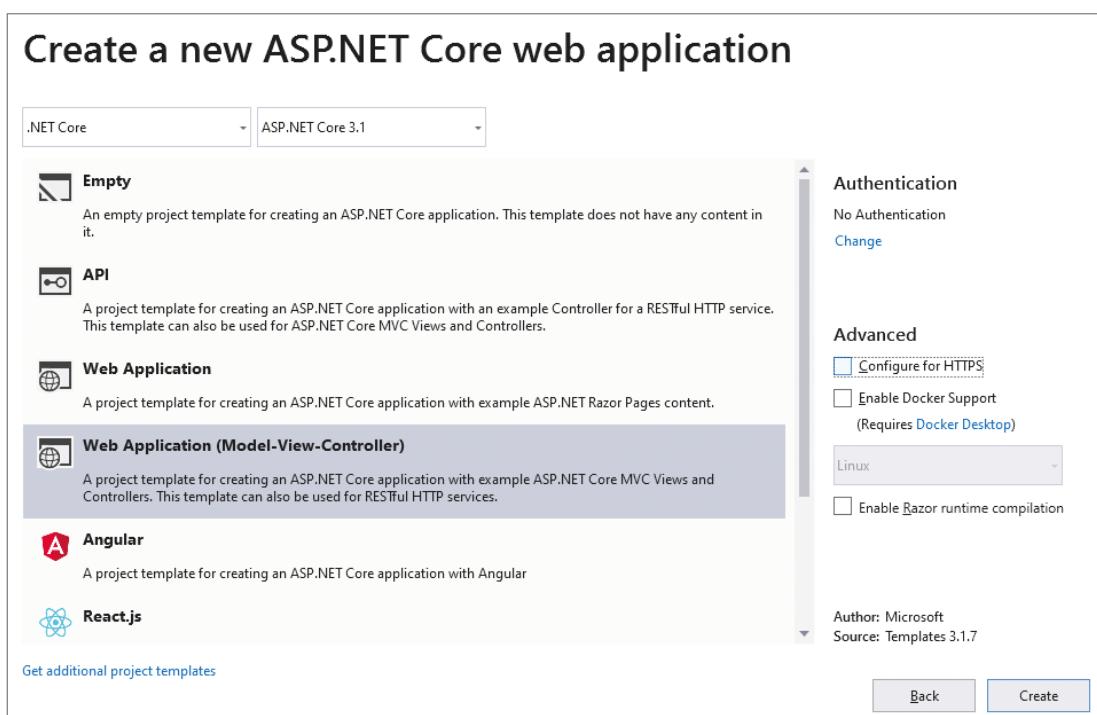


Bild 10.23 Installation der Vorlage in Visual Studio

Vergeben Sie als *Project name* den Namen *SentimentAnalyseWebApp*. Visual Studio erstellt jetzt automatisch ein Grundgerüst mit einer entsprechenden Projektstruktur. Die Vorlage verfügt schon über eine vollständig lauffähige Web-App, die als Single-Page-Application (SPA) für eigene Weiterentwicklungen bereitgestellt wird.

Nachdem die Vorlage installiert ist, können Sie mit dem Aufbau des ML-Modells beginnen. Wählen Sie hierfür das Kontextmenü im Solution-Explorer aus und klicken Sie auf *Add | Machine Learning*. Jetzt wird der ML Model Builder in Ihrem Projekt gestartet.

Als Szenario wählen Sie Textklassifizierung (*Text classification*) aus (Bild 10.24). Das ausgewählte Szenario dient der Unterstützung von Daten in Kategorien und Sie können damit prüfen, ob Kommentare positiv oder negativ sind.

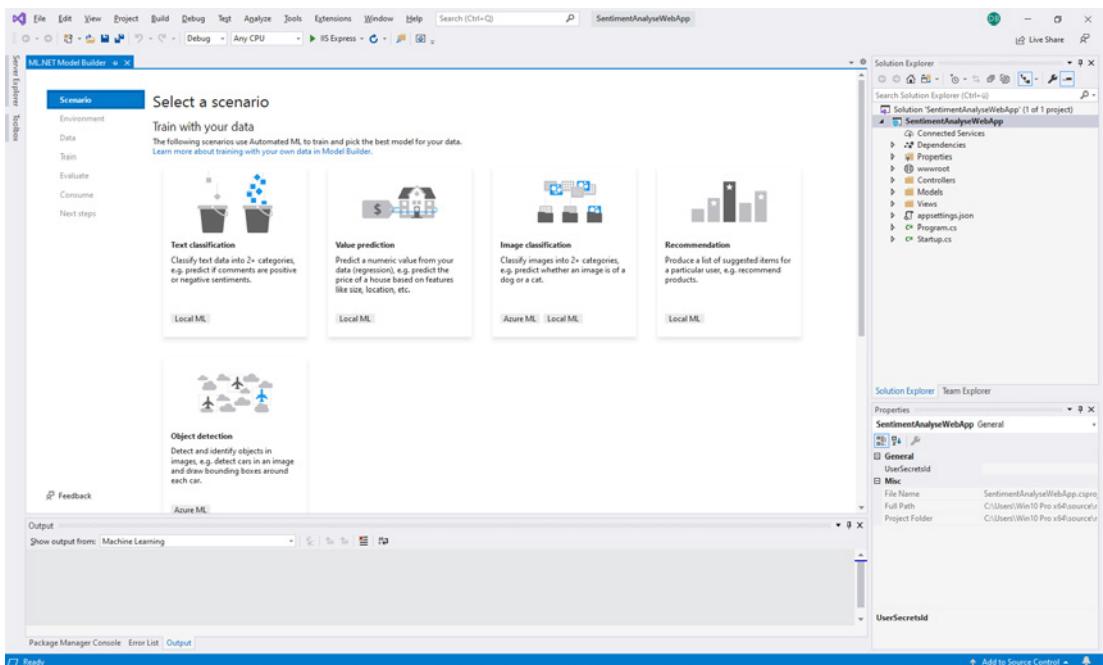


Bild 10.24 Mögliche Szenarien zur Auswahl im Model Builder

Im nächsten Schritt fügen Sie dem Model Builder die Textdatei *yelp_labelled.txt* hinzu und wählen als Label, die Sie vorhersagen möchten, die Spalte *col1* für *Column to predict (Label)* aus (Bild 10.25). Klicken Sie danach auf *Next step*.

ML.NET Model Builder ▾ X SentimentAnalyseWebApp

Scenario

Environment

Data (selected)

Train

Evaluate

Code

Next steps

Add data

In order to build a model, you must add data and choose your column to predict.
[How do I get sample datasets and learn more?](#)

Input

Choose input data source from either SQL Server or File:

File C:\temp\yelp_labelled.txt ...

Supported file formats: .csv, .tsv or .txt.

Column to predict (Label): col1

Input Columns (Features): 1 of 1 column selected

Data Preview

10 of 1.000 rows and 1 of 2 columns.

col1 (Label)	col0
1	Wow... Loved this place.
0	Crust is not good.
0	Not tasty and the texture was just nasty.
1	Stopped by during the late May bank holiday off Rick Steve recommendation and loved it.
1	The selection on the menu was great and so were the prices.
0	Now I am getting angry and I want my damn pho.
0	Honestly it didn't taste THAT fresh.)
0	The potatoes were like rubber and you could tell they had been made up ahead of time being kept under a warmer.
1	The fries were great too.
1	A great touch.

Feedback Next step

Bild 10.25 Hinzufügen der Daten und Auswahl der Label-Spalte

Im Dialog *Train* geben Sie jetzt die Zeit vor, für die Sie das Modell trainieren möchten. Geben Sie hier 200 Sekunden ein und klicken Sie auf *Train*.

Das Training im Model Builder ist vollständig automatisiert und liefert, nachdem das Modell erfolgreich trainiert wurde, im Output-Fenster von Visual Studio eine entsprechende Zusammenfassung (Bild 10.26) (siehe auch Abschnitt 7.5.7, „Training und Auswertung“).

The screenshot shows the 'ML.NET Model Builder' interface with the title bar 'ML.NET Model Builder' and 'SentmentAnalyseWebApp'. The main area is titled 'Train'.

Train

Specify a time to train for evaluating various models.
How long should I train for?

Train (highlighted)

Evaluate Time to train (seconds):

Code Train again ✓ Training complete

Next steps

Training results

	Best accuracy:	81,83%
Feedback	Best model:	SgdCalibratedOva
	Training time:	179,08 seconds
	Models explored (total):	8

Output

Show output from: Machine Learning

Model	MicroAccuracy	MacroAccuracy	Duration	#Iteration
AveragedPerceptronOva	0,8029	0,8042	5,3	1
SdcaMaximumEntropyMulti	0,7941	0,7950	12,7	2
LightGbmMulti	0,8003	0,8015	33,2	3
SymbolicSgdLogisticRegressionOva	0,7486	0,7412	4,8	4
FastTreeOva	0,8019	0,8046	102,8	5
LinearSvmOva	0,7846	0,7862	4,9	6
LbfgsLogisticRegressionOva	0,8037	0,8050	8,9	7
SgdCalibratedOva	0,8183	0,8212	6,5	8

=====Experiment Results=====

Summary

```
|ML Task: multiclass-classification
|Dataset: C:\temp\yelp_labelled.txt
|Label : coll
|Total experiment time : 179,0835988 Secs
|Total number of models explored: 8
```

Top 5 models explored

Trainer	MicroAccuracy	MacroAccuracy	Duration	#Iteration
SgdCalibratedOva	0,8183	0,8212	6,5	1
LbfgsLogisticRegressionOva	0,8037	0,8050	8,9	2
AveragedPerceptronOva	0,8029	0,8042	5,3	3
FastTreeOva	0,8019	0,8046	102,8	4

Bild 10.26 Auswertung des Trainings

Nachdem das Modell erfolgreich trainiert wurde, klicken Sie auf *Next*. Bei der nachfolgenden Auswertung (*Evaluate*) können Sie ermitteln, wie gut das Modell ist. Sie können in diesem Dialogfenster das Modell mit Testdaten versehen und anschließend messen, wie gut die Prognose ist (Bild 10.27).

The screenshot shows the 'Evaluate' step in the ML.NET Model Builder. The left sidebar has buttons for Scenario, Environment, Data, Train, Evaluate (which is selected and highlighted in blue), Code, and Next steps. The main content area has a title 'Evaluate' and sub-sections 'Best model:' (Accuracy: 81,83%, Model: SgdCalibratedOva) and 'Try your model'. Under 'Try your model', there's a 'Sample data' section with a pre-filled row: col0 contains 'Wow... Loved this place.' and a 'Predict' button. To the right is a 'Results' table:

	Results	
1	81%	
0	19%	

At the bottom are 'Next step' and 'Feedback' buttons.

Bild 10.27 Prüfen anhand von Testdaten

Als letzten Schritt bei der Erstellung des Modells klicken Sie auf *Next step* und fügen über den Button *Add to solution* das finale Modell und den dazugehörigen Code in Form von zwei neu generierten Projekten Ihrer Visual Studio Solution hinzu.

10.7.5 Das Modell als Web-App

Nachdem das finale Modell für die Sentiment-Analyse mit dem Model Builder erstellt worden ist, müssen Sie vor der Verwendung in Ihrem Web-App-Projekt noch über den NuGet-Manager Microsoft ML.NET einbinden. Sonst können Sie in Ihrem Projekt das Modell nicht konsumieren.

Des Weiteren benötigen Sie über die *Dependencies* (Abhängigkeiten) noch einen Verweis auf die Projekt-Referenz des Modells. Binden Sie daher das Modell wie in Bild 10.28 gezeigt in Ihr Web-App-Projekt ein.

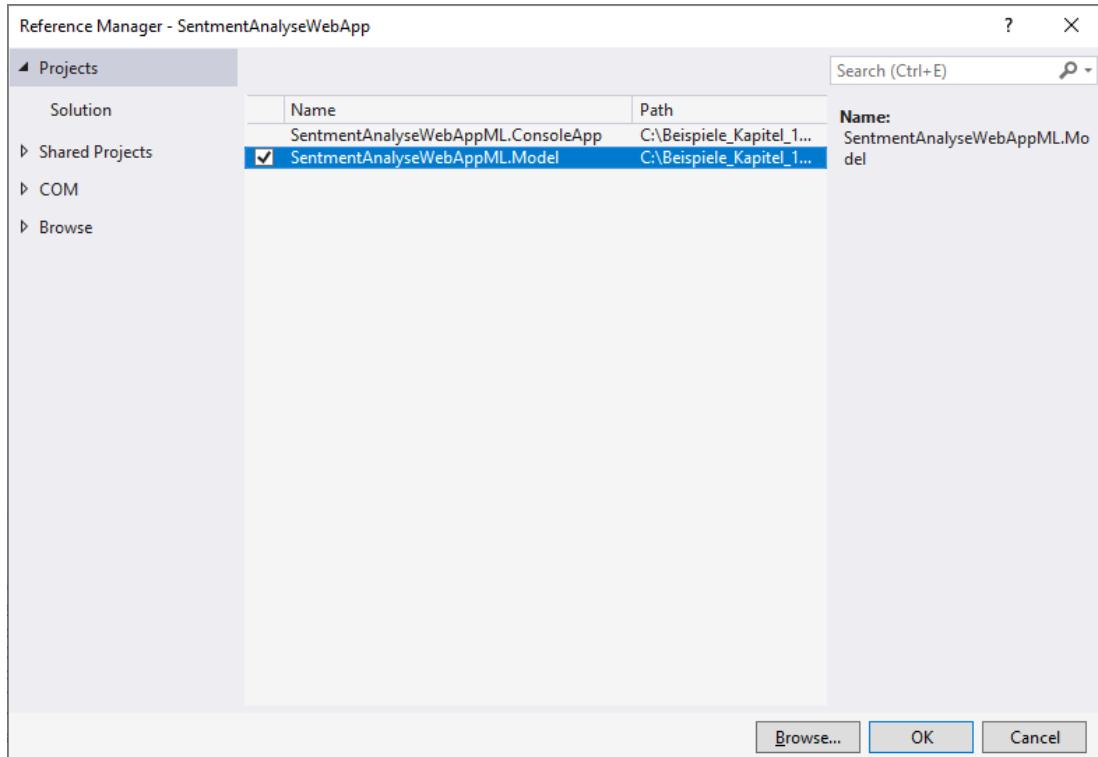


Bild 10.28 Das Modell als Referenz einbinden

Die weitere Projektstruktur funktioniert nach dem Prinzip des MVC-Entwurfsmusters [78]. Somit benötigen Sie nur noch einen Controller zum Konsumieren des Modells und eine View für die Eingabe und Auswertung der Kommentare.

Erstellen Sie dazu einen neuen leeren MVC-Controller mit dem Namen *SentimentController*. Die Implementierung für das Machine-Learning-Modell erfolgt wie in Listing 10.24 dargestellt.

Listing 10.24 SentimentController

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.ML;
using SentimentAnalyseWebAppML.Model;

namespace SentimentAnalyseWebApp.Controllers
{
    public class SentimentController : Controller
    {
        [HttpGet]
        public IActionResult SentimentAnalyse()
        {
            return View();
        }

        [HttpPost]
        public IActionResult SentimentAnalyse(ModelInput input)
        {

            MLContext mlContext = new MLContext();

            ITransformer mlModel = mlContext.Model.Load(@"..\SentimentAnalyseWebAppML\Model\MLModel.zip", out var modelInputSchema);
            var predEngine = mlContext.Model.CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);

            ModelOutput result = predEngine.Predict(input);
            ViewBag.Result = result;
            return View();
        }
    }
}

```

Auch hier wird wieder über die *MLContext*-Klasse eine ML.NET-Umgebung für die Web-App erstellt. Die Prognose erfolgt über eine *PredictionEngine* und wird über ein *Result*-Objekt der View zur Verfügung gestellt.

Die Eingabe des Kommentars und die Ausgabe der Vorhersage finden in der View *SentimentAnalyse* statt. Legen Sie hierfür unter *Views* einen neuen Ordner *Sentiment* an und implementieren Sie dort die View wie in Listing 10.25 aufgeführt.

Listing 10.25 SentimentAnalyse

```

@model SentimentAnalyseWebAppML.Model.ModelInput

@{ ViewData["Title"] = "Sentiment-Analyse"; }
<h2>Sentiment-Analyse with ASP.NET Core Web-App</h2>
<hr />
@{ var result = ViewBag.Result; }
<div class="row">
    <div class="col-md-4">
        <form asp-action="SentimentAnalyse">

```

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Col0" class="control-label"></label>
    <input asp-for="Col0" class="form-control" />
    <span asp-validation-for="Col0" class="text-danger"></span>
</div>
<div class="form-group">
    <input type="submit" value="Check" class="btn btn-default" />
</div>
</form>
</div>
<div class="col-md-offset-4 col-md-4">
@if (result != null)
{
    <h4>Result</h4>
    <h4>Predection: @result.Prediction</h4>
    <h4>Score: @result.Score[0]</h4>
}
</div>
</div>
```

Zum Schluss müssen Sie nur noch in der Klasse *Startup* unter *UseEndpoints* den richtigen Controller und die richtige View für Ihre Anwendung aufrufen. Bild 10.29 zeigt das Ergebnis in der gestarteten Web-App.

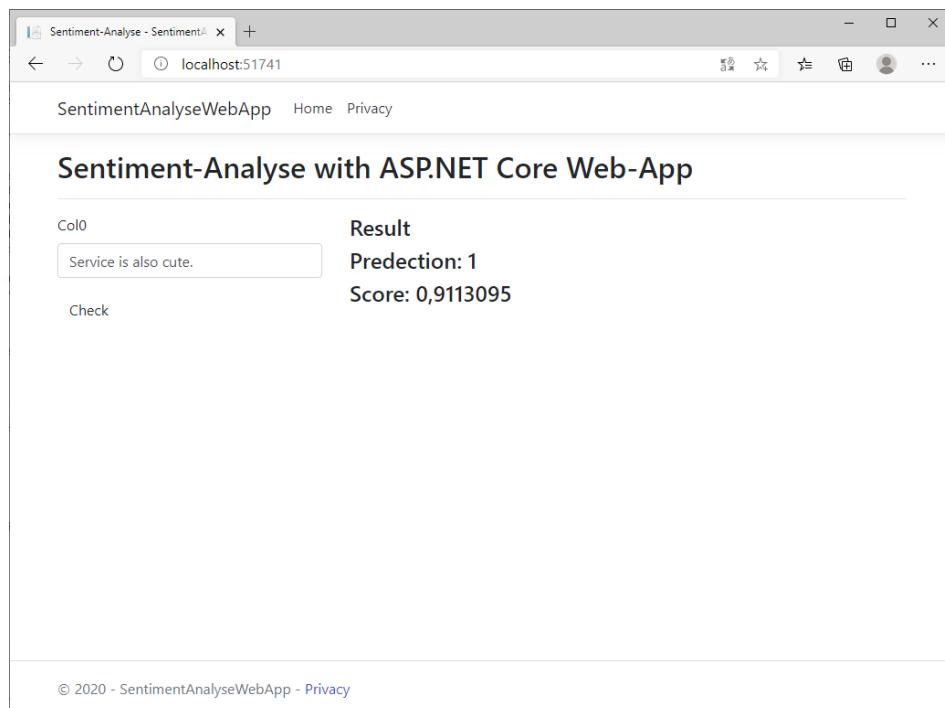


Bild 10.29 Die fertige Web-App mit passender Prognose

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Sentiment}/{action=SentimentAnalyse}/{id?}");
});
```

Die Beispielimplementierung verdeutlicht, wie schnell Sie ein zuvor erstelltes ML-Modell in einer Web-App per MVC-Entwurfsmuster aufbauen und bereitstellen können. Des Weiteren haben Sie mit dem Template ein gutes Arbeitswerkzeug für einen effektiven Einstieg in die Entwicklung einer Web-App.

Wie Sie an diesem Beispiel auch noch sehen können, ist die Erstellung eines erfolgreichen Modells immer ein iterativer Prozess. Das Modell im Beispiel besitzt keine so hohe Qualität, da es nur eine kleine Anzahl von Datensätzen verarbeitet, um eine schnelle Modellschulung zu ermöglichen. Versuchen Sie, die Qualität zu verbessern, indem Sie eine größere Menge an Trainingsdaten bereitstellen.

Referenzen und Quellen

Die Angabe von Internet-Verweisen unterliegt häufig Änderungen. Es kann daher sein, dass die angegebenen Links nach Drucklegung des Buches nicht mehr alle auffindbar sind oder der Inhalt variiert.

- [1] Alan Turing, https://de.wikipedia.org/wiki/Alan_Turing
- [2] John McCarthy, https://de.wikipedia.org/wiki/John_McCarthy
- [3] Big Data, https://de.wikipedia.org/wiki/Big_Data
- [4] Thomas Mitchell, <http://www.cs.cmu.edu/~tom/>
- [5] Arthur Samuel, https://de.wikipedia.org/wiki/Arthur_L._Samuel
- [6] Satz von Bayes, https://de.wikipedia.org/wiki/Satz_von_Bayes
- [7] Singulärwertzerlegung, <https://de.wikipedia.org/wiki/Singulärwertzerlegung>
- [8] Backpropagation Verfahren von Paul Werbos, https://de.wikipedia.org/wiki/Paul_Werbos
- [9] Mittlerer quadratischer Fehler, https://de.wikipedia.org/wiki/Mittlere_quadratische_Abweichung
- [10] Gers, Schraudolph, Schmidhuber: Learning Precise Timing with LSTM Recurrent Networks, <https://www.jmlr.org/papers/volume3/gers02a/gers02a.pdf>
- [11] Gated Recurrent Unit (GRU), https://en.wikipedia.org/wiki/Gated_recurrent_unit
- [12] Sobel- und Scharr-Operatoren, <https://de.wikipedia.org/wiki/Sobel-Operator#Scharr-Operator>
- [13] Technische Universität (TU) Eindhoven, <https://www.tue.nl>
- [14] Tokamak-Prinzip, https://de.wikipedia.org/wiki/Fusion_mittels_magnetischen_Einschlusses
- [15] Operatoren von TensorFlow Lite, https://www.tensorflow.org/lite/guide/ops_compatibility
- [16] TensorFlow Playground, <https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle®Dataset=reg-donut&nIn=2&nOut=1&w0=-0.1&b0=0.1&w1=0.3&b1=-0.7&x0=-0.8&y0=-1.0&x1=0.8&y1=1.0>
- [17] Herbrich, Graepel, Campbell: Bayes Point Machine, <https://www.jmlr.org/papers/volume1/herbrich01a/herbrich01a.pdf>
- [18] TrueSkill Ranking System, <https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/>
- [19] Hidden Markov Model, https://de.wikipedia.org/wiki/Hidden_Markov_Model
- [20] Resources and References Infer.NET, <https://dotnet.github.io/infer/userguide/Resources%20and%20References.html>
- [21] Das Projektteam von Infer.NET (Video), <https://dotnet.github.io/infer/default.html>
- [22] Zufallsvariable, <https://de.wikipedia.org/wiki/Zufallsvariable>
- [23] Wahrscheinlichkeitsverteilung, <https://mathepedia.de/Wahrscheinlichkeitsverteilungen.html>
- [24] Expectation Propagation, https://en.wikipedia.org/wiki/Expectation_propagation

- [25] Stanford University: Belief Propagation, <https://web.stanford.edu/~montanar/RESEARCH/BOOK/partD.pdf>
- [26] Variational Message Passing, https://en.wikipedia.org/wiki/Variational_message_passing
- [27] Gibbs Sampling, https://en.wikipedia.org/wiki/Gibbs_sampling
- [28] A-posteriori Wahrscheinlichkeit, <https://de.wikipedia.org/wiki/A-posteriori-Wahrscheinlichkeit>
- [29] User Guide Infer.NET, <https://dotnet.github.io/infer/userguide/>
- [30] Evidence Model (Beweis) unter Infer.NET, <https://dotnet.github.io/infer/userguide/Computing%20model%20evidence%20for%20model%20selection.html>
- [31] YouTube Video der Kiva Systems Lösung bei Amazon, <https://www.youtube.com/watch?v=6KRjuuEVEZs>
- [32] Infer.NET, <https://github.com/dotnet/infer>
- [33] Normalverteilung, <https://de.wikipedia.org/wiki/Normalverteilung>
- [34] Infer.NET 101, <https://dotnet.github.io/infer/InferNet101.pdf>
- [35] LightGbm-Algorithmus, <https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc>
- [36] UCI Machine Learning Repository, <https://archive.ics.uci.edu/ml/index.php>
- [37] Machine Learning Algorithmen ML.NET, <https://docs.microsoft.com/de-de/dotnet/machine-learning/how-to-choose-an-ml-net-algorithm>
- [38] Microsoft ML TensorFlow NuGet-Package, <https://www.nuget.org/packages/Microsoft.ML.TensorFlow/>
- [39] ML.NET ONNX Beispiel, <https://github.com/dotnet/machinelearning-samples/tree/master/samples/csharp/end-to-end-apps/ObjectDetection-Onnx>
- [40] Open Neural Network Exchange (ONNX) Spezifikation, <https://github.com/onnx/onnx>
- [41] SciSharp Technologie Stack, <https://scisharp.github.io/SciSharp/>
- [42] The Definitive Guide to TensorFlow.NET, <https://tensorflownet.readthedocs.io/en/latest/>
- [43] Beispiele TensorFlow.NET, <https://github.com/SciSharp/TensorFlow.NET>
- [44] Python-Unterstützung für Keras.NET, <https://www.python.org/downloads/>
- [45] Backend-Unterstützung für TensorFlow und Keras, https://keras.io/api/utils/backend_utils/
- [46] Keras, <https://keras.io/>
- [47] SciSharp Stack, <https://github.com/SciSharp>
- [48] Einführung und Beispiele NeuralNetwork.NET, <https://github.com/Sergio0694/NeuralNetwork.NET>
- [49] Methoden und Funktionen zu Amazon Lex, https://docs.aws.amazon.com/de_de/lex/latest/dg/lex-dg.pdf
- [50] Common Line Interface (CLI) von Amazon Lex, https://docs.aws.amazon.com/de_de/lex/latest/dg/gs-cli.html
- [51] Amazon SDK for .NET, <https://aws.amazon.com/de/sdk-for-net/>
- [52] AWS Toolkit für Visual Studio, <https://aws.amazon.com/de/visualstudio/>
- [53] Amazon-AWS-Konto einrichten, <https://aws.amazon.com/de/account/>
- [54] Children's Online Privacy Protection Act (COOPA), https://en.wikipedia.org/wiki/Children%27s_Online_Privacy_Protection_Act
- [55] AWS Chatbot for .NET, <https://github.com/aws-samples/aws-lex-net-chatbot>
- [56] Azure Cognitive Services, <https://azure.microsoft.com/de-de/services/cognitive-services/>

- [57] Bing Search SDK for .NET, <https://github.com/Azure-Samples/cognitive-services-dotnet-sdk-samples/tree/master/BingSearchv7>
- [58] Konfidenzintervall, <https://de.wikipedia.org/wiki/Konfidenzintervall>
- [59] ImageNet, <http://www.image-net.org/>
- [60] WordNet, <https://en.wikipedia.org/wiki/WordNet>
- [61] Hub Portal TensorFlow, <https://www.tensorflow.org/hub>
- [62] Martin Wicke: Vorlage ImageNet-Klassifikator Modelle,
<https://github.com/martinwicke/tensorflow-tutorial>
- [63] Google Colaboratory (Colab), <https://colab.research.google.com/notebooks/intro.ipynb>
- [64] TensorBoard, <https://github.com/tensorflow/tensorboard>
- [65] One-Hot-Encoded Vector, <https://en.wikipedia.org/wiki/One-hot>
- [66] Sparse Vector, <https://www.cs.umd.edu/Outreach/hsContest99/questions/node3.html>
- [67] Zhang, Zhao, LeCun: Character-level Convolutional Networks for Text Classification,
<https://papers.nips.cc/paper/2015/file/250cf8b51c773f3f8dc8b4be867a9a02-Paper.pdf>
- [68] Vanishing Gradient Problem, https://en.wikipedia.org/wiki/Vanishing_gradient_problem
- [69] Lea, Vidal, Reiter, Hagen: Temporal Convolutional Networks,
<https://openreview.net/pdf?id=SkIVpqHi>
- [70] Pennington, Socher, Manning: Global Vectors for Word Representation,
<https://nlp.stanford.edu/pubs/glove.pdf>
- [71] fastText, <https://en.wikipedia.org/wiki/FastText>
- [72] Tang, Wie, Yang, Zhou, Liu, Qin: Sentiment-Specific Word Embedding (SSWE),
<https://www.aclweb.org/anthology/P14-1146.pdf>
- [73] Stanford CoreNLP, <https://stanfordnlp.github.io/CoreNLP/>
- [74] Koreferenz, <https://de.wikipedia.org/wiki/Koreferenz>
- [75] Sprachdatei für CoreNLP, <https://stanfordnlp.github.io/CoreNLP/download.html>
- [76] Dataset für Sentment Labelled Sentences,
<https://www.kaggle.com/marklvl/sentiment-labelled-sentences-data-set>
- [77] Kreuzvalidierung,
<https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f>
- [78] MVC-Entwurfsmuster, https://de.wikipedia.org/wiki/Model_View_Controller

Stichwortverzeichnis

A

Adaline 52
Adaptive Linear Neuron *siehe* Adaline
Agglomerativ 39
Aktivierungsfunktion 17, 64, 85, 122, 145, 146, 150
Algorithmus 33, 34
Amazon
– Lex 8
– Polly 9
– Rekognition 9
– RoboMaker 9
– SageMaker 8
– Translate 9
– Web-Service (AWS) 8
Amazon Cognito 244
Amazon EC2 244
Amazon Lex 239, 244
– Automatic Speech Recognition 239
– Chatbot 244, 257
– Confirmation prompt 241, 248
– Dialog Action 243
– Intents 240, 247
– Natural Language Understanding 239
– Request Attributes 241
– Response 241, 249
– Session Attributes 241, 243
– Slots 255
– Utterances 240
Amazon Web Services 238
AND-Funktion 42
API *siehe* C#, Application Programming Interface
Asynchrone Netze 14
Ausgabefunktion 19
Automated Machine Learning 193
AutoML 200 *siehe auch* Automated Machine Learning
– Model Builder 195
– Sentiment-Analyse 338

Autonom fahrende Autos 49
AWS *siehe* Amazon Web Services
AWS Cognito Identity Pool 256
AWS Explorer 238, 244
AWS Lambda 242, 251, 253
– Codehook 242, 251
– Context 242
– Event 242
– Response 243
AWS SDK for .NET 238
Azure Cognitive Services 259
Azure Machine Learning Studio 269

B

Backpropagation 86, 95, 300
– Momentum 118
Backpropagation-Algorithmus 89, 92, 99, 108, 114
Backpropagation-Through-Time-Algorithmus 147, 149
Backward-Pass 95, 127
Bag-Of-Words 320
Batch Processing 156
Batch Size 92
Batch-Training 113
Bayes-Klassifikator 30
Bayessche Inferenz 180
Bayessches Netz 30
Beispiel
– Bildklassifikation 280
– Energie-Prognose 276
Berechnungsgraph 176
Bias 16, 17, 75
Big Data 2, 6
Bilderkennung 24
Binäre Klassifikation 319
Binäre Schwellenfunktion *siehe* Schrittfunktion
Bing
– Autosuggest 262

- Custom Search 261
- Image Search 261
- News Search 261
- Spell Check 262
- Video Search 261
- Visual Search 262
- Web Search API 261
- Bing Search API 259
- Bing-Websuche 260
- Boltzmann-Maschine 54
- BPTT *siehe* Backpropagation-Through-Time-Algorithmus

- C**
- C#, Application Programming Interface 174
- Character-level Convolutional Networks for Text Classification 322
- Children's Online Privacy Protection Act 246
- C#-Implementierung, RNN Zelle 146
- CLI 207
- Cloud-Services 2
- Clusteranalyse 38
- Cluster, hierarchisch 39
- Clustering 29, 38
 - Expectation-Maximization 39
 - K-Means 38
- CNN *siehe* Convolutional Neural Network
- Cognitive Computing 6
- Cognitive Services Bing Search API 260
- Computer Vision 159
- Compute Unified Device Architecture 236
- Conversational User Interface 240
- ConvNet *siehe* Convolutional Neural Network
- Convolutional Layer 303
- Convolutional Neural Network 14, 15, 159, 302, 322
 - 2D-Volumen 161
 - 3D-Volumen 161, 165
 - Aktivierungsfunktion 166
 - Architektur 161
 - Bilder 160
 - Dense 170
 - Dropout Layer 170
 - Filter 159
 - Filtertyp 164
 - Filterung 166
 - Flattening 170
 - Hyperparameter 165, 167
 - Kanten 162
 - Maximal-Pooling 168
- Overfitting 161
- Pooling 164, 165
- Pooling Layer 168
- Rectified Linear Unit 166
- Schrittweite 165
- Subsampling 168
- COPPA *siehe* Children's Online Privacy Protection Act
- CopyPixel 162
- CoreNLP 331
- CUDA *siehe* Compute Unified Device Architecture

- D**
- Data Science 6
- Datenaufbereitung 26
- Datenimport 26
- Decision Tree 38
- Deep Learning 5, 48
- Delta-Regel 53
- Dense Layer 324
- Dezimalskalierung 68
- Dimensionsreduzierung 112
- direct feedback 143
- Disjunkte Klassen 319
- Diskrimination 319
- Divisiv 39
- DL *siehe* Deep Learning

- E**
- Eingabesignale 16
- Einsatzgebiete 49
 - Finanzwesen 50
 - Gesundheitswesen 49
 - IT Security 50
 - Logistik 50
 - Marketing 50
 - Produktion 50
 - Versicherungswesen 50
 - Vertrieb 50
- Einschichtige Netze 21
- Entitätsextraktion 325
- Entscheidungsbaum 37
- Epoche (Epoch) 92, 93
- Exploding Gradients 149

- F**
- Faltungsschicht *siehe* Convolutional Layer
- Feature Vectors *siehe* Merkmalsvektor
- Feedforward-Algorithmus 79, 83
- Feedforward-Architektur 117

Feedforward-Mechanismus 73
 Feedforward-Netze 63, 70, 85, 141, 145, 287
 Feedforward Neural Network 14, 15
 Feedforward-Pass 127
 Fehlerfunktion 147 *siehe auch* Kostenfunktion
 Festbreitengruppierung 68
 Festhöhengruppierung 68
 File, Read All Lines 123
 Fisher-Yates-Shuffle-Algorithmus 126, 134
 Flatten Layer 305
 FNN-Mechanismus 94
 Forecast 138
 Forget-Gate 152, 155
 forget-gate layer 152
 Forward-Pass 91
 Fully Connected 15
 Fully Connected Layer 306, 308

G

Gate Recurrent Unit 155
 Gauß-Verteilung 190
 Gewichtsfaktoren 16
 GloVe50D 327
 Google 9
 - AutoML 10
 - AutoML Natural Language 10
 - Cloud AI Platform 10
 - Vision API 10
 Gradient-Boosting Tree 38
 Gradient Clipping 155
 Gradientenabstieg 88, 147
 Gradientenabstiegsverfahren 113
 Grafikprozessoren 2

H

Handgeschriebene Ziffern 280
 Hidden Layer 14, 21, 48
 Hold-Out Validation 112, 124
 Hopfield-Netze 53
 Hyperbolischer Tangens 149, 153, 154
 Hyperfläche 36, 37
 Hyperparameter 26, 112, 139, 160
 Hyper-Rechteck 327
 HyperTan 118

I

ImageNet 310, 312
 indirect feedback 143
 Inferenz-Engine 188
 Infer.NET 180, 186

- A-posteriori-Verteilung 186
- Architektur 184
- Bayesian Neural Network 180
- Bayes-Point-Machine-Klassifikator 180
- Bayessche Inferenz 186
- Belief Propagation 183
- Expectation Propagation 183
- Factors 184
- Gamma-Priorität 182
- Gaußsche Präzision 182
- Gibbs-Sampling 183
- Hidden-Markov-Modelle 180
- Inferenz-Algorithmus 184
- Message operators 184
- Mittelwert 182
- Plug-in-Architektur 183
- probabilistische Programmierung 181
- TrueSkill-Matchmaking 180
- Variational Message Passing 183

Inkrementelles Training 113
siehe auch Online Learning

Input Gate 152, 155
 Input Gate Layer 152
 Input Layer 14
 Iteration 92

K

- Keras 179
- Keras.NET 233
 - funktionales Modell 234
 - Installation 233
 - Modell erstellen 234
 - sequenzielles Modell 234
 - XOR-Problem 234
- Kernel-Trick 36
- Kettenartige Struktur 150
- Kettenregel 148
- Klassifikation 29
- Klassische Programmierung 23
- K-Means-Algorithmus 38
- k-Nearest-Neighbour 34
- KNN *siehe* Künstliche neuronale Netze
- Konfusionsmatrix 69
- Kostenfunktion 87
 - berechnen 137
- Kreuzvalidierung 112
- Künstliche Intelligenz
 - hybride 3
 - schwache 2
 - starke 3

Künstliche neuronale Netze 13, 30
 Kurzfristige Abhängigkeiten 143

L

Label 28
 Langfristige Abhängigkeiten 143
 lateral feedback 143
 Layer 13
 Lemmatisierung 325
 Lernalgorithmus 46
 Lernformen 31
 Lernmethode 25
 Lernrate 44, 127
 - anpassen 140
 - bestimmen 136
 Lernschritt *siehe* Epoche (Epoch)
 Lineare Algebra 55, 328
 Lineare Funktion 19
 Linear Threshold Unit 20
 Logische Operatoren
 - AND-Funktion 42
 - NAND-Funktion 42
 - NOR-Funktion 42
 - NOT-Funktion 42
 - OR-Funktion 42
 Long Short-Term Memory 149
 Long-Short-Term-Memory-Zelle 151
 LSTM *siehe auch* Long Short-Term Memory
 - Peephole 155
 LSTM-Architektur 155
 LSTM-Zelle 150, 158
 LTU *siehe* Linear Threshold Unit

M

Machine Learning 5, 23
 - Algorithmen 25
 - Lernform 25
 - Projekt 26
 Machine Learning as a Service 173, 237
 Madaline 52
 MakeMatrix 83
 Math.Exp 85
 Matrix 21, 58
 Matrix.Multiply 21
 Matrizen 55, 59
 Matrizenmultiplikation 20, 59
 Matrizenschreibweise 20
 Maximum-Entropie 215
 MaxIndex 138
 McCarthy, John 2

Mean Squared Error 118
 - berechnen 130, 137
 Mehrklassen-Klassifikation 68, 111, 319
 - Einer gegen den Rest 69
 - Mehrklassen-Feedforward-Netze 69
 - paarweise Klassifikation 69
 Mehrklassen-Klassifizierung 117
 Memory-Matrix 54
 Merkmalsvektor 320
 Methode, Training 134
 Microsoft Cognitive Services 11
 - Cognitive Toolkit 11
 - Freihanderkennungs-API 11
 - Gesichtserkennungs-API 11
 - Speech-Service 11
 - Video API 11
 Microsoft ML.NET 156
 Microsoft Research Cambridge-Team 192
 Mini-Batch-Training 114
 Min-Max-Normalisierung 67
 Mittlerer quadratischer Fehler 118, 231
 MLaaS *siehe* Machine Learning as a Service
 MLContext 315
 ML-Modell
 - Training 40
 - Validierung 40
 ML.NET 192
 - Bildklassifizierung 197
 - binäre Klassifikation 192
 - Clustering 192
 - Command Line Interface 207
 - DataLoader 210
 - DataTransforms 210
 - einbinden 193
 - Empfehlung 197
 - Fit-Methode 211, 215, 316
 - IDataView 209
 - ML-Algorithmen nachinstallieren 218
 - MLContext 213, 335
 - Model Builder 195
 - OneHotEncoding 214
 - Pipeline 211, 277, 315
 - PredictionEngine 210, 278, 337
 - Regression 192, 197
 - SdcaLogisticRegressionBinaryTrainer 337
 - Sentiment-Analyse 333
 - Singular Spectrum Analysis 275
 - SSA *siehe* Singular Spectrum Analysis
 - TensorFlow 218
 - Textklassifizierung 196

- Time-Series-Algorithmus 275
 - Transferlernen 310
 - Workflow 211
 - ML.NET Framework 278
 - MNIST *siehe* National Institute of Standards and Technology
 - Model Builder 195, 200
 - Evaluierungsphase 200
 - Features 199
 - Label 198
 - LightGbm-Algorithmus 205
 - Sentiment-Analyse 338
 - Text-Klassifikation 202
 - Modified National Institute of Standards and Technology 280
 - Momentum-Faktor 96, 115, 127, 138, 140
 - bestimmen 136
 - Momentum-Term 117
 - Multilayer Perceptron 47, 63, 67
 - Multiple Adaptive Linear Neuron *siehe* Madaline
 - Mustererkennung 280
 - MVC-Controller 343
 - MVC-Entwurfsmuster 346
- N**
- NAND-Funktion 42
 - Natural Language Processing 7, 317
 - Navigationsgeräte 49
 - Netzarchitektur 21
 - Netzparameter anpassen 140
 - Netztypen 14
 - Neural Architecture Search 208
 - NeuralNetwork.NET 236
 - Neural Turing Machine 54
 - Neuron 16
 - Beispiel 20
 - NLP *siehe* Natural Language Processing
 - NOR-Funktion 42
 - Normalverteilung 190
 - Normierung 67
 - NOT-Funktion 42
 - NTM *siehe* Neural Turing Machine
 - NumPy 225
 - NDArray 224
- O**
- Objekterkennung 309
 - Offline Learning 113
 - One against all 319
 - One Hot Vector 294

- Online Learning 113
 - ONNX *siehe* Open Neural Network Exchange
 - Open Neural Network Exchange 173, 174, 211, 219
 - Optimierungsmetrik 26
 - OR-Funktion 42
 - Ortsvektor 56
 - Output Gate 154, 155
 - Output Layer 14
 - Overfitting 27, 28
 - Overshooting 140
- P**
- Perceptron *siehe* Perzeptron
 - Perzeptron 41, 46 *siehe auch* Single-Layer Perceptron
 - NotAnd 43
 - Pooling Layer 303
 - Predictive Analytics 273
 - Predictive Maintenance 65, 115, 116
 - Predictive Policing 273
 - Primzahlenanalyse 94
 - Probabilistische Programmierung 181
 - Prozessübersicht, Machine Learning 40
- Q**
- Quadratischer Fehlerwert 138
 - Qualifikationsbewertungssystem 188
- R**
- Random-Forest 38
 - ReadingDataset 123, 124
 - Rectified Linear Unit 306
 - Rectifier-Funktion 19
 - Recurrent Neural Network 14, 15, 141, 323
 - entfaltetes 145
 - Struktur 144
 - Zelle 144
 - Regression 29
 - Reinforcement Learning *siehe* Verstärkendes Lernen
 - Rekurrente Netze *siehe* Recurrent Neural Network
 - Rekurrentes neuronales Netz *siehe* Recurrent Neural Network
 - ReLU 301 *siehe auch* Rectifier-Funktion
 - Representational State Transfer 262
 - REST *siehe* Representational State Transfer
 - RNN *siehe* Recurrent Neural Network
 - RNN-Zelle 146

Robotic Process Automation 4
 Rückkopplung 143
 - direkte 15, 143
 - indirekte 15, 143
 - seitliche 15, 143
 - vollständig verbunden 15, 143

S
 Satzgrenzen 325
 Satz von Bayes 30
 Schrittfunktion 18
 Schwellenwert 16
 SciSharp Stack 222
 Seed-Parameter 126
 Semi-Supervised Learning (Semi-überwachtes Lernen) 32
 Sentiment 333
 Sentiment-Analyse 333, 343
 Sequenzen 141, 143, 148
 Sequenzielle Daten 141, 143
 Sequenzlänge 143
 Serverless 239
 Service-Chatbot 50
 Session 224
 SetExpected 293
 Sigmoid 118
 Sigmoid-Aktivierungsfunktion 287
 Sigmoidfunktion 19
 Single-Layer Perceptron 14
 Singulärwertzerlegung 61, 275
 Skalar 55
 Skalarprodukt 58
 Smart Grid 274
 Softmax 78, 301
 Softplus-Funktion 19
 Spracherkennung 142
 SsaForecastingEstimator 277
 Stimmungsanalyse *siehe* Sentiment-Analyse
 Stopwort 328, 330
 stopword *siehe* Stopwort
 Stride *siehe* Convolutional Neural Network, Schrittweite
 Struktur, kettenartig 150
 Supervised Learning 111
 siehe auch Überwachtes Lernen
 Support Vector Machine 35
 SVM *siehe* Support Vector Machine
 Symmetrische Netze 14
 Symmetry Breaking 92

T
 Tangens-Hyperbolicus-Funktion 19
 tanh 85
 TBPTT *siehe* Truncated Backpropagation Through Time
 Teilesequenzen 148
 Tensor 56, 61, 175, 176
 TensorBoard 311
 TensorFlow 10, 156, 173, 175
 - Ablauf 176
 - Berechnungsgraph 178
 - Inferenz 178
 - Modell 178
 - placeholder *siehe* Platzhalter
 - Platzhalter 176
 - Playground 178
 - Session 178
 - Tensor 178
 - TensorBoard 177
 TensorFlow Hub 310
 TensorFlow-Modelle 311
 TensorFlow.NET 222
 - Berechnungsgraph 228
 - Gradient Descent 230
 - Installation 222
 - Konstante 227
 - lineare Regression 229
 - Platzhalter 225
 - Variable 226
 TensorFlow Probability 182
 Testdaten 27
 Testphase 112
 Textklassifizierung 339
 Tokenizer 325
 Toolkit for Visual Studio 238
 Topologie 14, 21
 TrainAllMnistImages 297
 Training
 - Batch 113
 - inkrementelles 113
 - Mini-Batch 114
 Trainingsdaten 27
 TrainTheNeuralNetwork 294
 Transponieren 61
 Transportfahrzeuge, fahrerlos 49
 Truncated Backpropagation Through Time 147
 Turing, Alan 1

U

Überwachtes Lernen 31
 UCI Machine Learning Repository 207
 Unsupervised Learning (Unüberwachtes Lernen)
 31

V

Validierungsdaten 27
 Vanishing Gradients 148, 155
 Vektor 56
 - Addition 57
 - Subtraktion 57
 Vektoraddition 150
 Vektorraum 327
 Verstärkendes Lernen 32
 - AlphaGo 33
 Vorausschauende Wartung 65
 Vortrainierte Modelle 10
 Vorwärtsgekoppelte Netze 14

W

Wahrheitsmatrix *siehe* Konfusionsmatrix
 WartungsDemo 71
 Watson Data Platform 12

- Natural Language Classifier 12

- Natural Language Understanding-Service 12

- Watson Assistant 12

Wetterdaten 143

Widrow-Hoff-Regel Delta-Regel

Word2Vec 321, 325

Word Embedding 325

Wortarterkennung 325

Wortstammerkennung 325

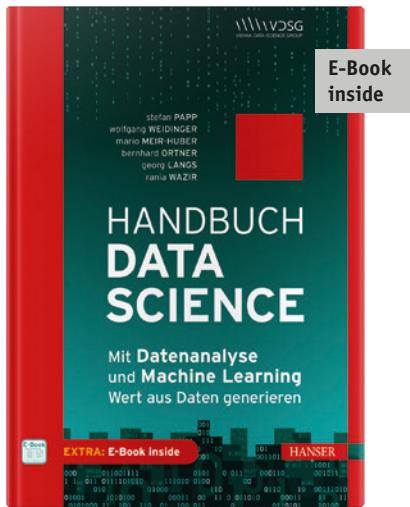
X

XOR 62
 XOR-Funktion 47
 XOR-Problem 158

Z

Zeitreihenanalyse 143, 274
 Zeitschritt 144
 Zellzustand 151
 Zufallsvariablen 182
 Zustandsvektor 154
 Zwei-Klassenproblem 68
 Z-Wert-Normalisierung 67
 Zyklische Netze 14

Data Science erfolgreich anwenden



Papp, Weidinger, Meir-Huber, Ortner, Langs, Wazir

Handbuch Data Science

Mit Datenanalyse und Machine Learning

Wert aus Daten generieren

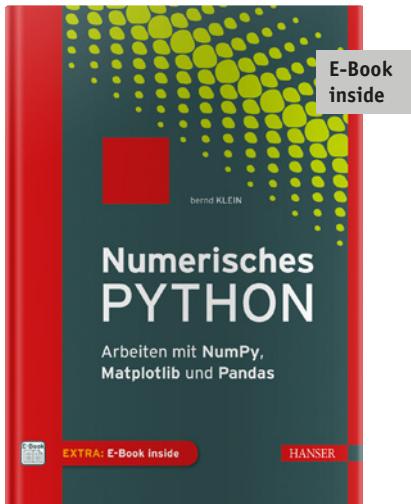
320 Seiten. Inklusive E-Book

€ 39,90. ISBN 978-3-446-45710-2

Auch einzeln als E-Book erhältlich

- Gibt einen umfassenden Überblick über die verschiedenen Anwendungsfelder von Data Science
- Fallbeispiele aus der Praxis machen die beschriebenen Konzepte greifbar
- Vermittelt das notwendige Wissen, um einfache Datenanalyse-Projekte durchzuführen
- Erklärt sowohl den Aufbau von Big Data-Plattformen und die Anwendung einzelner Tools als auch statistisch-mathematische und rechtliche Themen

Große Datenmengen mit Python verarbeiten



Klein

Numerisches Python

Arbeiten mit NumPy, Matplotlib und Pandas

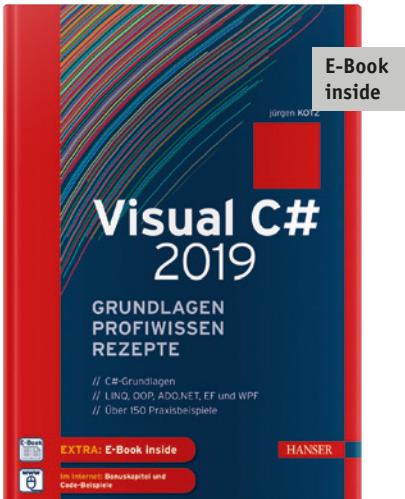
373 Seiten. Inklusive E-Book

€ 29,90. ISBN 978-3-446-45076-9

Auch einzeln als E-Book erhältlich

- Grundlagen der Lösung numerischer Probleme mit Python
- Verarbeitung großer Datenmengen (»Big Data«) mit NumPy, wie sie beispielsweise im maschinellen Lernen Anwendung finden
- Datenvisualisierung mit Matplotlib
- Für Ingenieure und Wissenschaftler, die große Datenmengen verarbeiten und visualisieren müssen
- Ideal zum Umstieg von Matlab auf Python

Der Klassiker der C#-Programmierung



Kotz

**Visual C# 2019 – Grundlagen, Profiwissen
und Rezepte**

1036 Seiten. Inklusive E-Book

€ 49,90. ISBN 978-3-446-45802-4

Auch einzeln als E-Book erhältlich

Mit diesem Buch haben Sie den idealen Begleiter für Ihre tägliche Arbeit und zugleich – dank der erfrischenden und unterhaltsamen Sprache – eine spannende Lektüre, die Lust macht, auch Projekte in der Freizeit umzusetzen.

Einsteiger erhalten ein umfangreiches Tutorial zu den Grundlagen der C#-Programmierung mit Visual Studio 2019, dem Profi liefert es fortgeschrittene Programmiertechniken zu allen wesentlichen Einsatzgebieten der Windows-Programmierung. Zum sofortigen Ausprobieren finden Sie am Ende eines jeden Kapitels hochwertige Lösungen für nahezu jedes Problem.

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

Hypethema Künstliche Intelligenz



Frochte

Maschinelles Lernen

Grundlagen und Algorithmen in Python

3., überarbeitete und erweiterte Auflage

616 Seiten. Inklusive E-Book

€ 39,99. ISBN 978-3-446-46144-4

Auch als E-Book erhältlich

- Liefert den Hintergrund, um zu verstehen, wie und warum welche Algorithmen eingesetzt werden
- Alle wesentlichen Teilgebiete des maschinellen Lernens werden in Python umgesetzt und an Beispielen demonstriert
- Verwendung zeitloser Ansätze und Tools
- Nutzung von freier, kostenloser Software
- Neu: Deep Q-Learning, Class Activation Maps und Grad-CAM, Pandas-Integration und -Einführung, OpenAI Gym integriert

Künstliche Intelligenz ganz praktisch



Lämmel, Cleve

Künstliche Intelligenz

Wissensverarbeitung – Neuronale Netze

5., überarbeitete Auflage

336 Seiten

€ 29,99. ISBN 978-3-446-45914-4

Auch als E-Book erhältlich

Das Buch gibt Ihnen eine leicht verständliche Einführung in die Künstliche Intelligenz (KI). Die Autoren zeigen, wie symbolverarbeitende KI in Form von Wissensnetzen oder Geschäftsregeln angewendet und wie künstliche neuronale Netze in der Mustererkennung oder auch im Data Mining eingesetzt werden können.

- Symbolverarbeitende künstliche Intelligenz und künstliche neuronale Netze in einem Buch
- Business Rules und Wissensnetze
- Convolutional Neural Networks und Deep Learning
- Übungen in PROLOG sowie mit JavaNNS und Python

Mehr Informationen finden Sie unter www.hanser-fachbuch.de

NEURONALE NETZE MIT C# PROGRAMMIEREN //

- Aufbau und Training von neuronalen Netzen
- Wichtige Machine-Learning-Algorithmen verstehen und einsetzen
- Arbeiten mit ML.NET und Infer.NET
- Vorstellung des Open Source Framework TensorFlow.NET
- Erstellen eines Lex-Chatbot für .NET
- Alle Beispiele sind mit Visual Studio und C# umsetzbar

Sie wollen neuronale Netze und Machine-Learning-Algorithmen mit C# entwickeln? Dann finden Sie in diesem Buch eine gut verständliche Einführung in die Grundlagen und es wird Ihnen gezeigt, wie Sie neuronale Netze und Machine-Learning-Algorithmen in Ihren eigenen Projekten praktisch einsetzen.

Mithilfe von Beispielen erstellen und trainieren Sie Ihr erstes neuronales Netz zur vorausschauenden Wartung einer Produktionsmaschine.

Im Praxisteil lernen Sie dann, wie Sie TensorFlow-Modelle in ML.NET benutzen oder Infer.NET direkt verwenden können. Des Weiteren nutzen Sie die Predictive- und Sentiment-Analyse, um sich mit Machine-Learning-Algorithmen vertraut zu machen.

Alle im Buch vorgestellten Projekte sind in C# programmiert und stehen als Download zur Verfügung.

Grundkenntnisse in C# werden für die Arbeit mit dem Buch vorausgesetzt. Alle Projekte lassen sich ohne größere Rechnerressourcen umsetzen.

Daniel **BASLER** arbeitet als Lead Developer und Softwarearchitekt. Seine Schwerpunkte liegen auf



Cross-Platform-Apps, Android, JavaScript und Microsoft-Technologien. Er entwickelt u.a. Software für Regal- und Flächenlagersysteme sowie Anlagenvisualisierung und setzt in diesem Umfeld verstärkt Machine-Learning-Methoden ein. Darüber hinaus schreibt er regelmäßig Artikel für die Fachzeitschriften dotnetpro und web&mobile Developer.

AUS DEM INHALT //

- Künstliche Intelligenz: Grundlagen
- Konzepte und Methoden von Machine Learning
- Neuronale Netze bauen und trainieren
- Maschinensimulation mit Multilayer Perceptron (MLP)
- Backpropagation
- Recurrent Neural Networks
- Convolutional Neural Networks
- Machine Learning as a Service
- Predictive Analytics
- Objekterkennung
- Sentiment-Analyse

HANSER

