HANSER

Leseprobe

zu

Neuronale Netze mit C# programmieren

von Daniel Basler

Print-ISBN: 978-3-446-46229-8
E-Book-ISBN: 978-3-446-46426-1
E-Pub-ISBN: 978-3-446-46635-7

Weitere Informationen und Bestellungen unter
https://www.hanser-kundencenter.de/fachbuch/artikel/978-3-446-46229-8
sowie im Buchhandel

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 Stoppwörter 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

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.

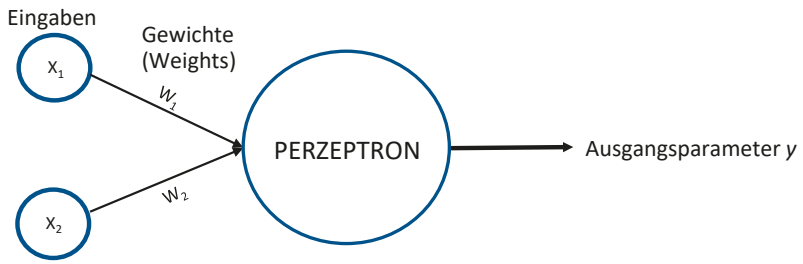


Bild 2.9 Das einstufige Perzeptron

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 $[X_1, X_2, ..., X_n]$ in einen binären Ausgabeparameter Y_i umgerechnet. Die Gewichte $[W_1, W_2, ..., W_n]$ 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.

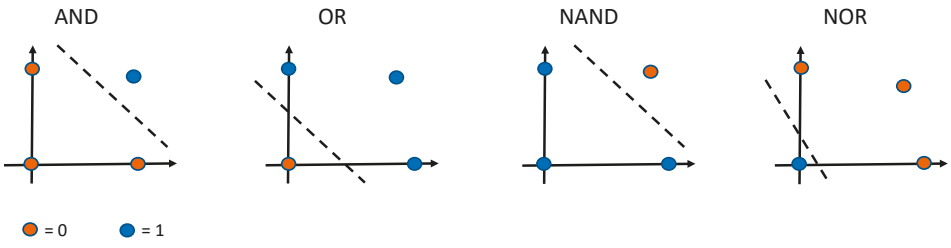


Bild 2.10 Logische Operatoren für das Perzeptron

Das NotAnd-Perceptron-Beispiel

Alle im Bild 2.10 aufgeführten logischen Operatoren lassen sich in C# als Perzeptron abbilden. Zur besseren Veranschaulichung implementieren wir den *NotAnd*-Operator als Perzeptron mithilfe von Visual Studio 2019. Das Programmbeispiel verzichtet hierbei auf entsprechende Kommentare im Code sowie auf den normalen Fehlerprüfcode, um das Perzeptron 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*-Perzeptron ist exemplarisch durch die C#-Klasse in Listing 2.1 implementiert.

Tabelle 2.1 Wahrheitstabelle für das NotAnd-Perzeptron

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).

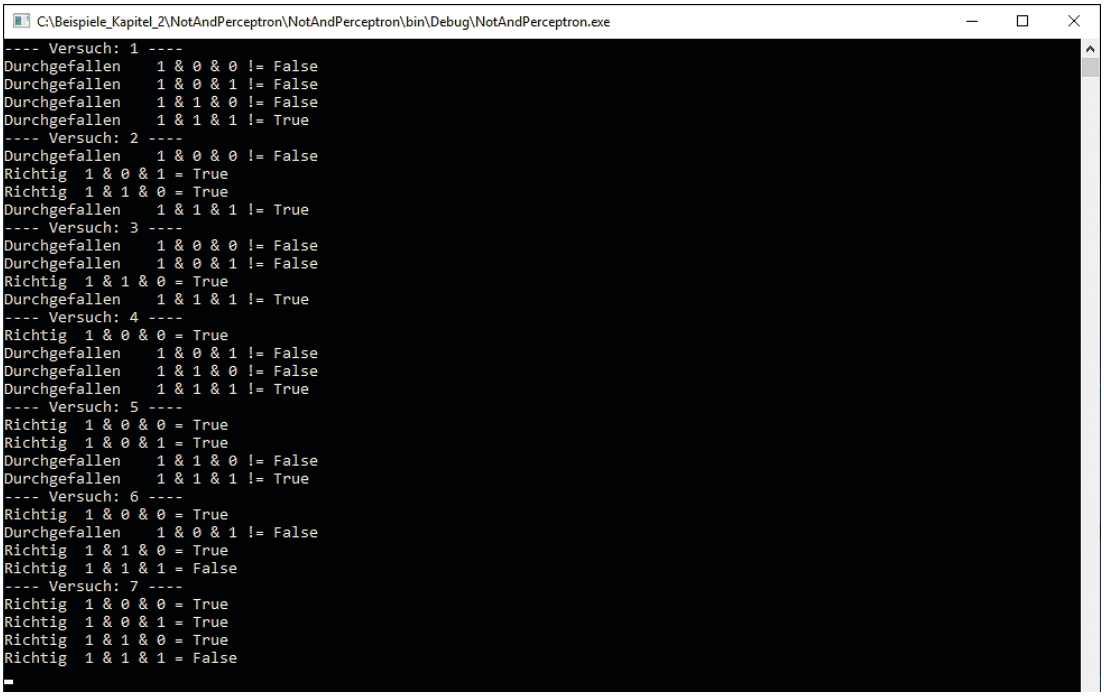


Bild 2.11 Ergebnis des Lernalgorithmus

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.

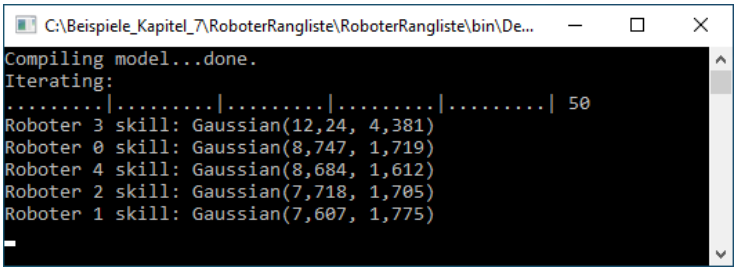


Bild 7.10 Die Anwendung wird ausgeführt.

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.

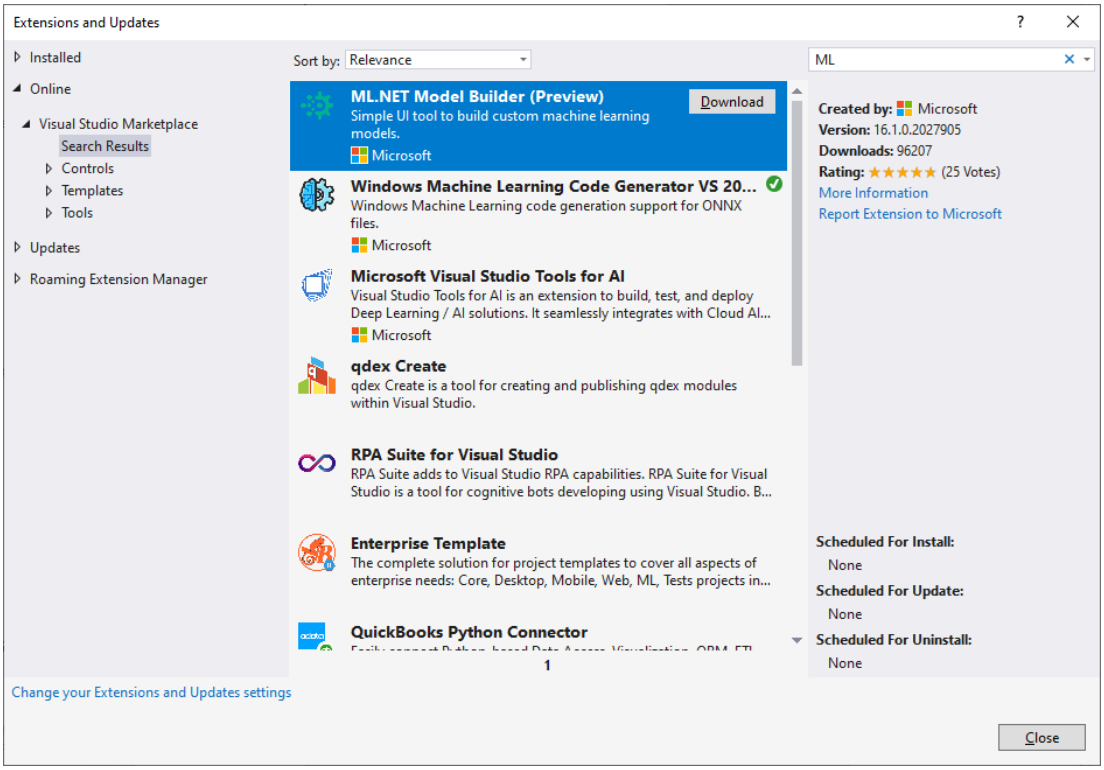


Bild 7.11 Das Model Builder Tool als Extension

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.

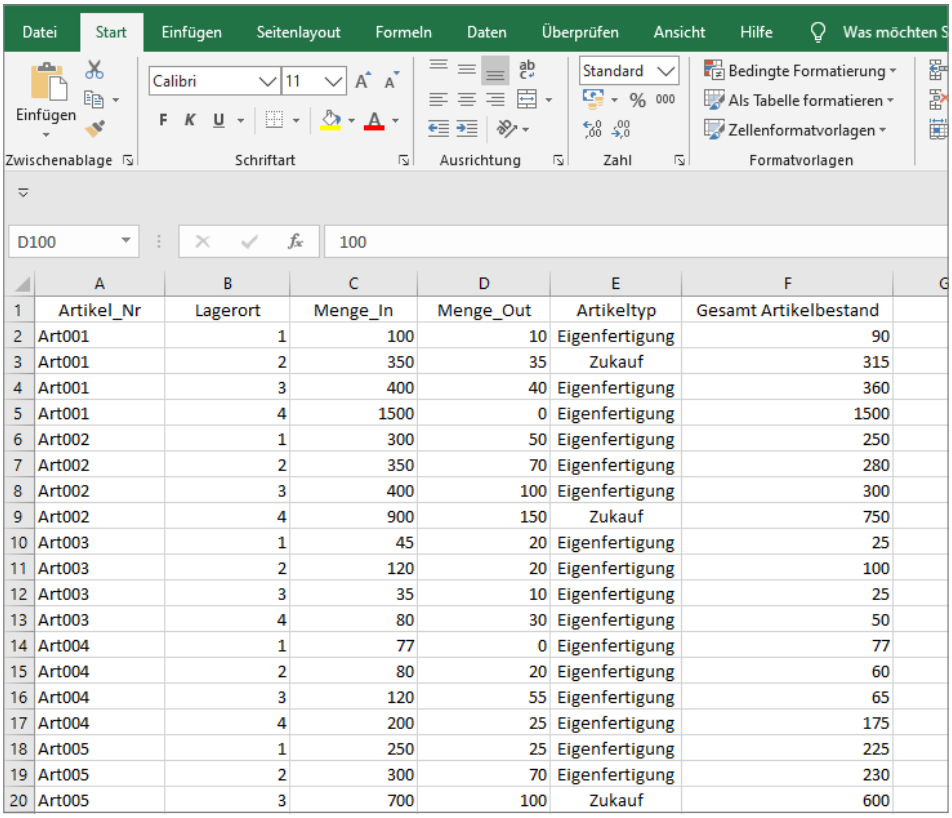
7.5.6 Daten

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
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.

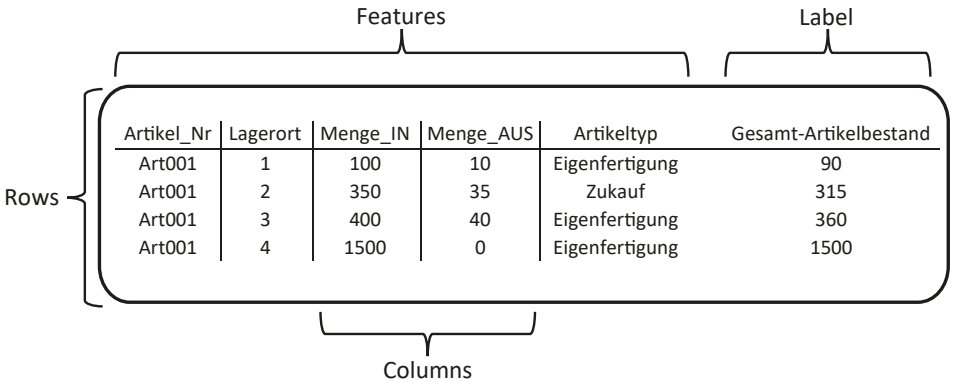
Das Dataset stellt immer eine Tabelle mit Zeilen (Rows) als Trainingsbeispiele und Spalten (Columns) mit Attributen dar. Jede Zeile enthält somit:

- 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.



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.

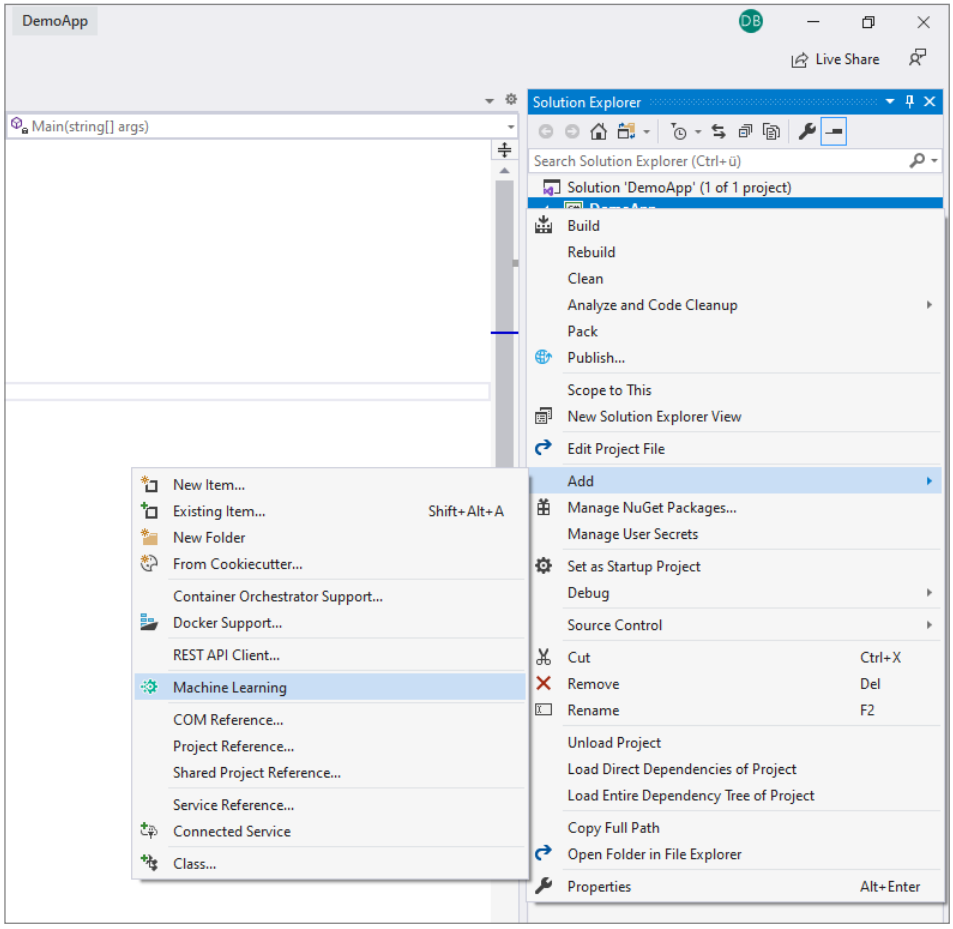


Bild 7.14 Model Builder zum Projekt hinzufügen

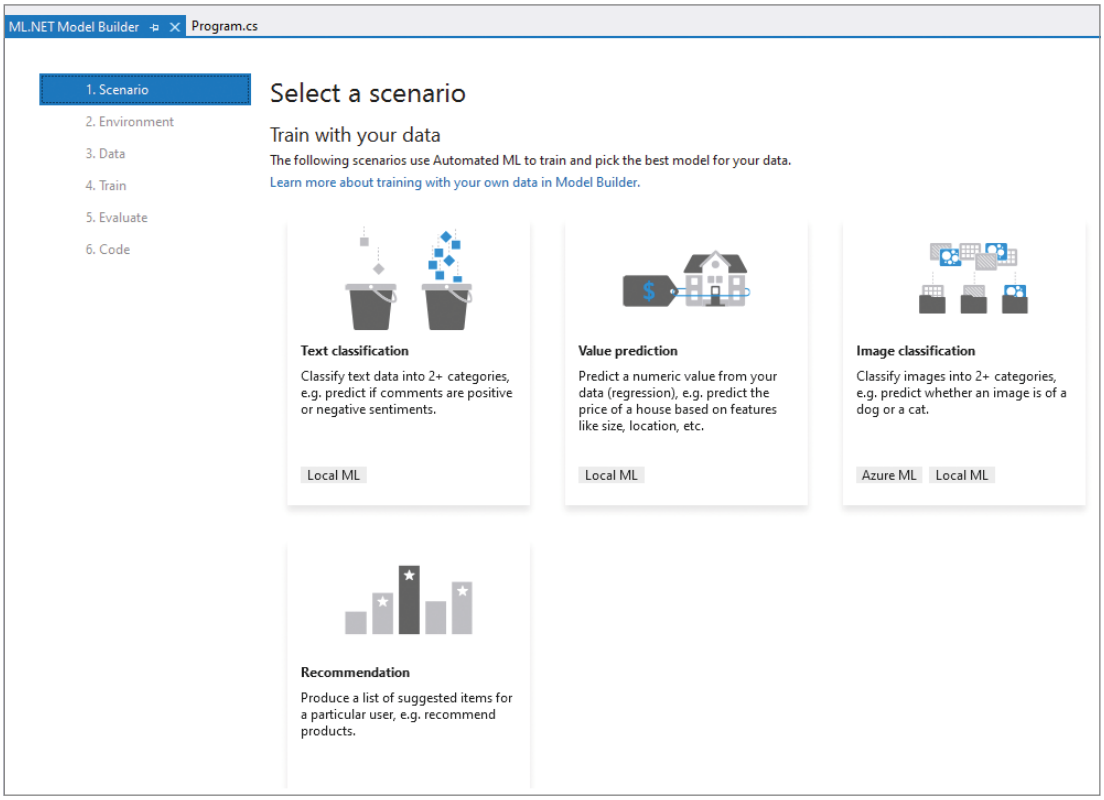


Bild 7.15 Model Builder in Visual Studio 2019

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* (Merkmalen) 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*.

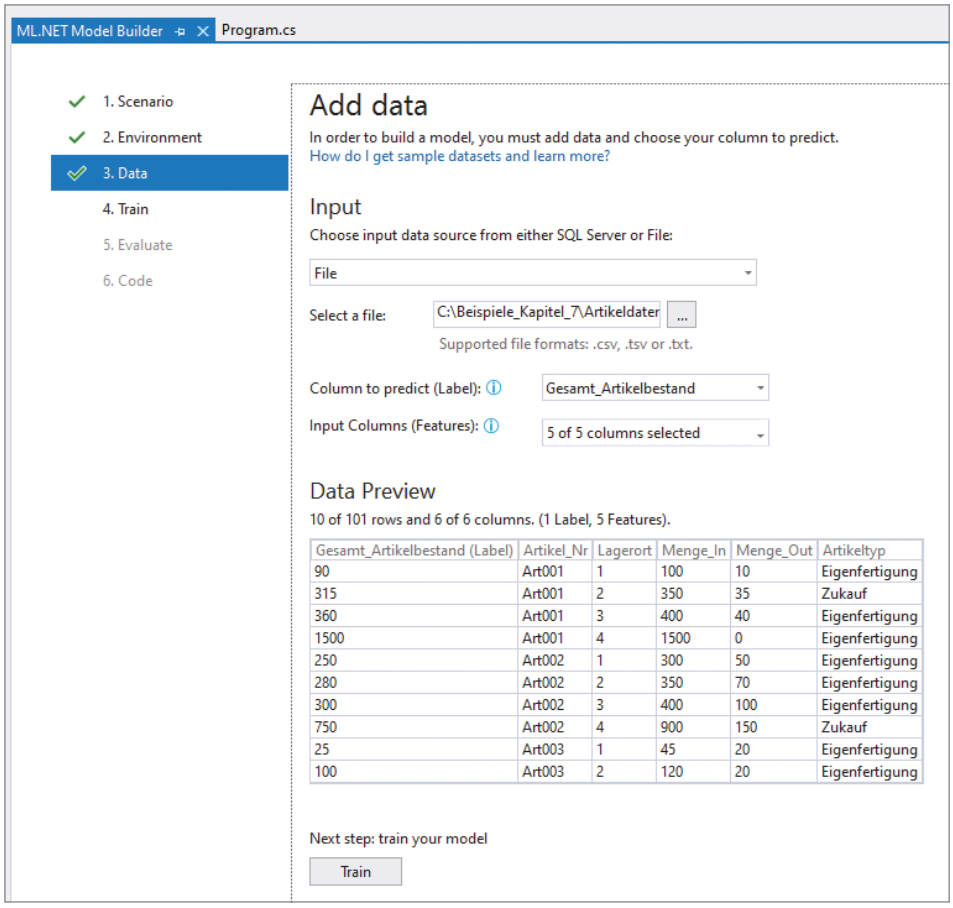


Bild 7.16 Einstellung der gewünschten Vorhersage

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).

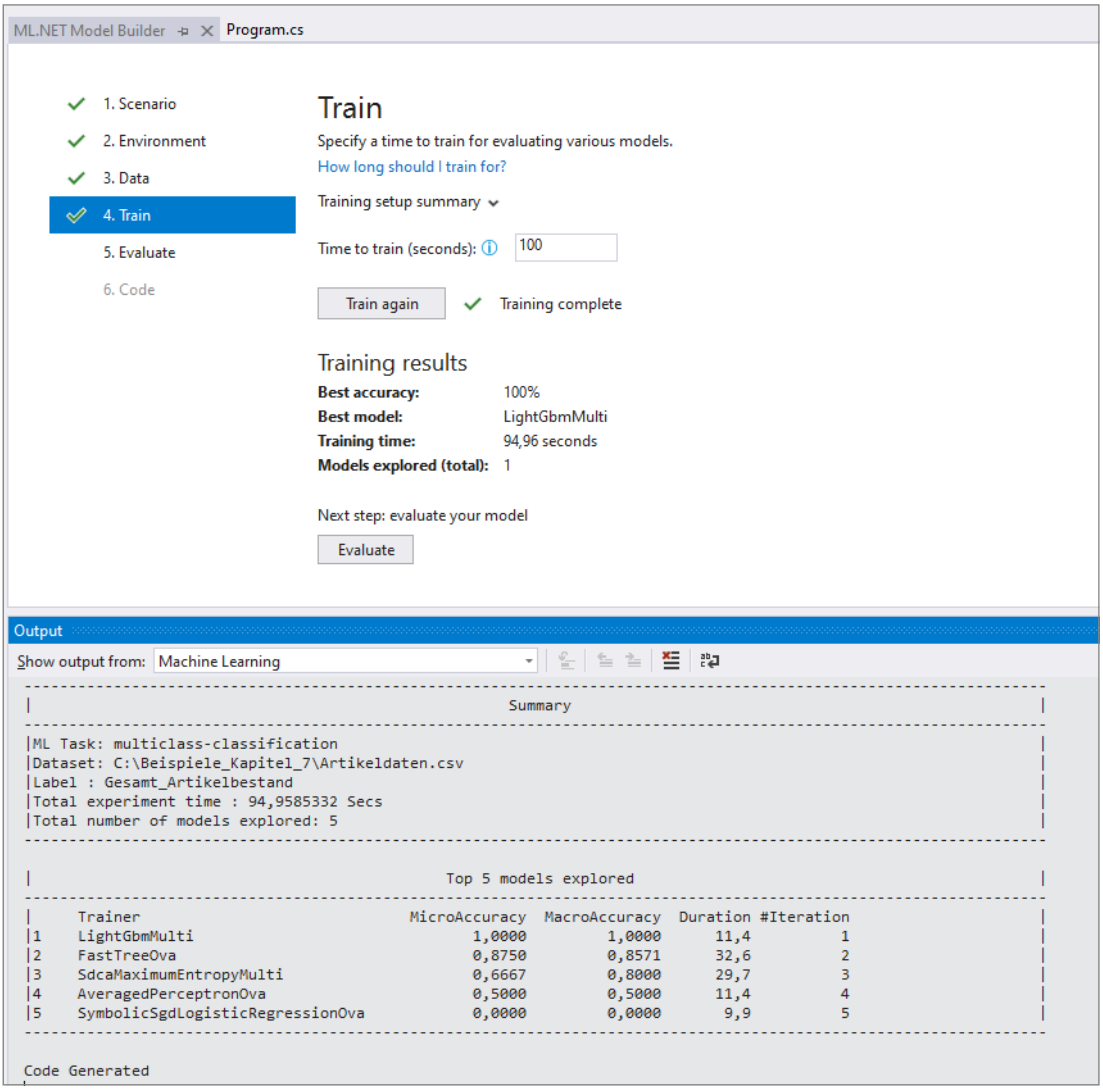


Bild 7.17 Das AutoML-Ergebnis im Model Builder

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.

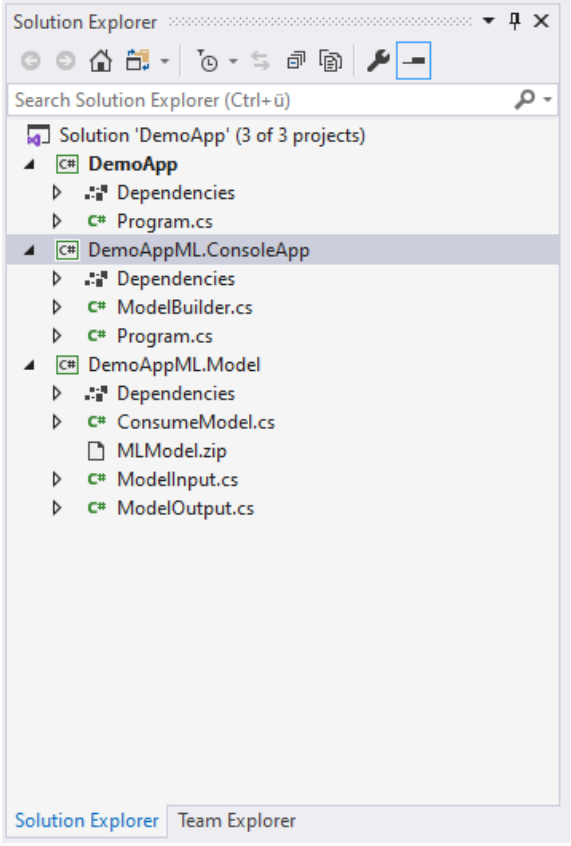


Bild 7.18 Erweiterung des Projekts durch den Model Builder

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 einsehen 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.

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 Verschwendung 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

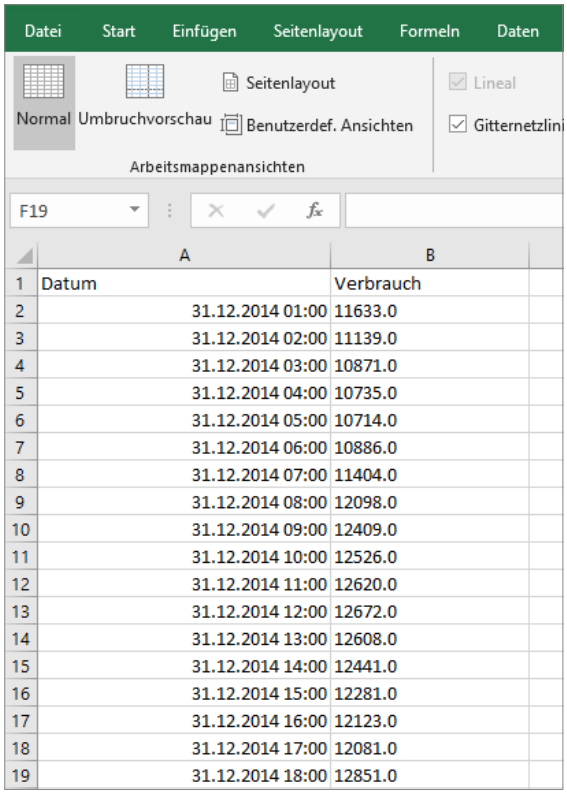
Zeitreihenanalyse gibt es in zahlreichen Anwendungen. Im Kern geht es darum, in vorbestimmten Sets von Daten Muster zu finden, die ein tiefergehendes Verständnis von vergangenen Ereignissen oder von vergangenen Verhaltensweisen von Kunden erlaubt. Diese Erkenntnisse werden dann als Grundlage für die 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 *TimesSeriesEngine*. 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 Vorhersagezeitraum 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 Modellerstellungsworkflows 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 *TimeSeriesPrediction-Engine*.

```
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; }
    }

}
```

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* Modified 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
Stoppwort 328, 330
stopword *siehe* Stoppwort
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
Teilsequenzen 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