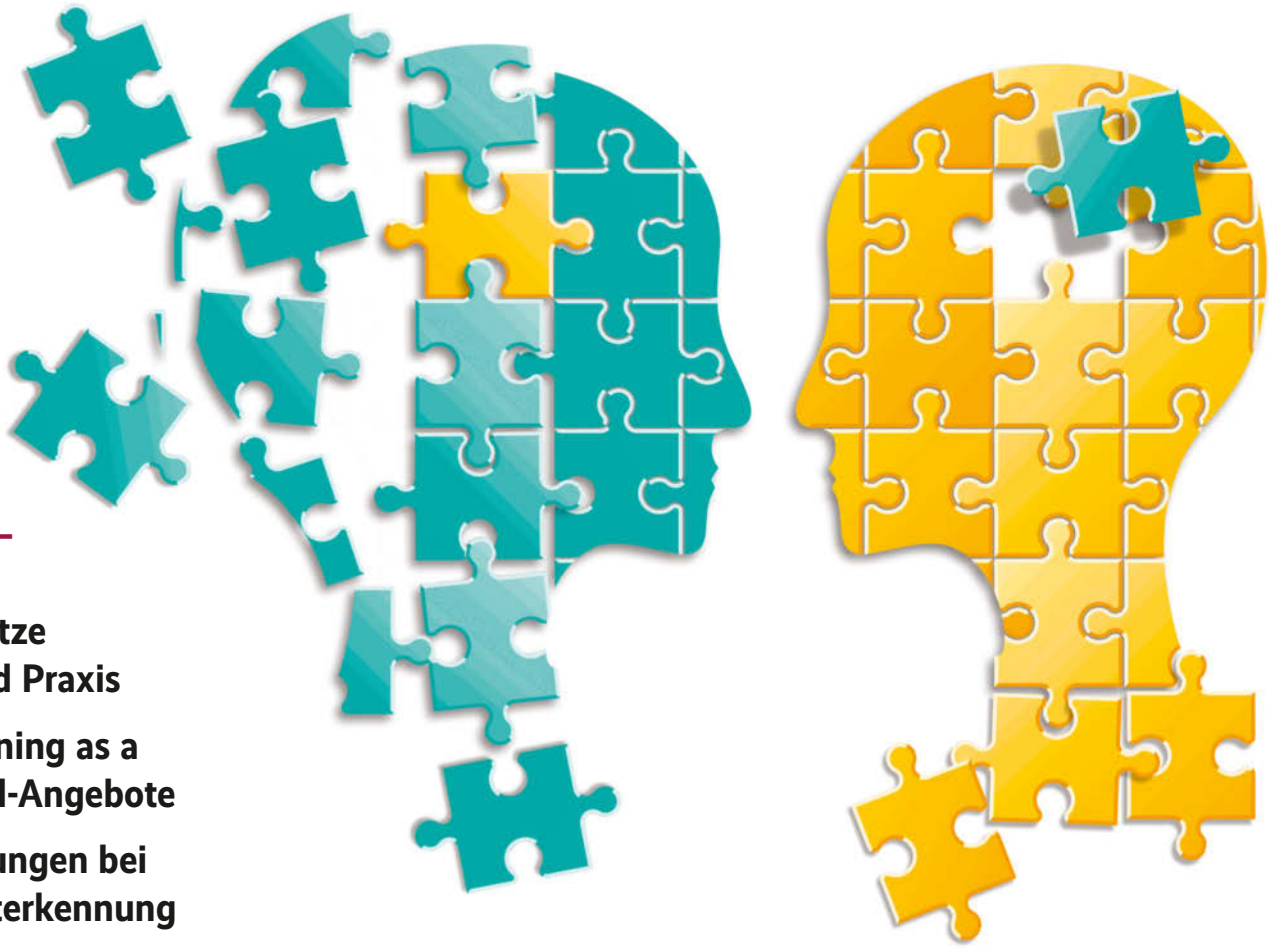
iX DEVELOPER

Winter 2018

Machine Learning

Verstehen, verwenden, verifizieren



Data Science – Status quo

Neuronale Netze in Theorie und Praxis

Machine Learning as a Service: Cloud-Angebote

Herausforderungen bei Bild- und Texterkennung

Blick in die Blackbox

TensorFlow, Keras & Co.

ML-Frameworks und -Bibliotheken

Programmiersprachen: Python, Scala, C++

Hardware pusht ML

GPUs und CPUs für Machine Learning

Neuromorphe Chips für neuronale Netze

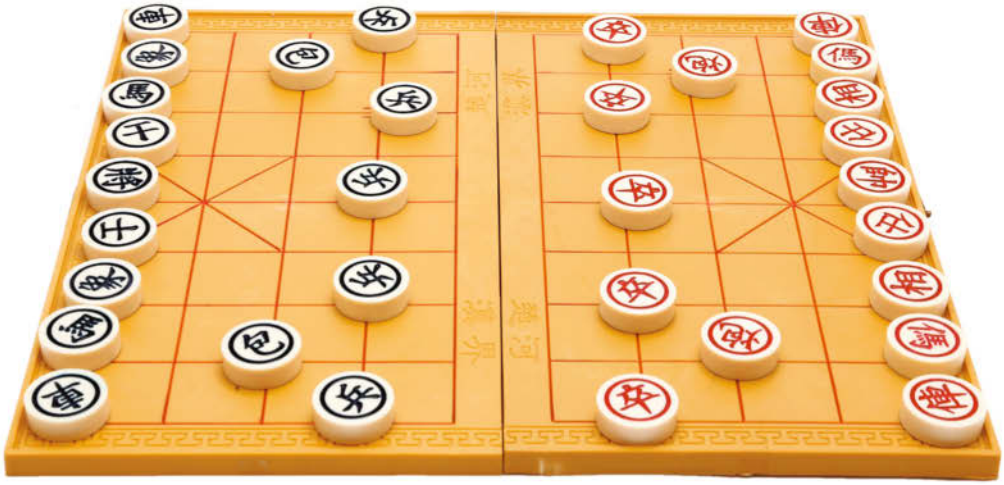
Mensch trifft KI

Rechtsfragen bei Künstlicher Intelligenz

Verantwortungsvoller Einsatz und Ethik

Künstliche neuronale Netze in Theorie und Praxis

Das Gehirn des Rechners



Marcel Tilly

Ein Blick hinter die Kulissen offenbart, dass KNNs und Deep Learning weit weniger komplex sind als viele glauben mögen. Das Verständnis für die Arbeitsweise und Methoden hilft dabei, Machine-Learning-Frameworks für eigene Projekte passend einzusetzen.

Künstliche Intelligenz ist in der Realität und in der allgemeinen Wahrnehmung angekommen. Die Erfolge von DeepMind mit AlphaGo, die Berichte zu Bild- und Gesichtserkennung oder smarten Bots sind in aller Munde. Die meisten Ansätze nutzen Machine Learning im Allgemeinen oder Deep Learning im Speziellen. Das mag anfangs nach höchst komplexer Mathematik klingen und etwas Mystisches besitzen, aber eigentlich sind die Grundlagen ziemlich trivial und lassen sich auf ein paar einfache Konzepte herunterbrechen.

Im Grunde geht es bei Deep Learning darum, ein künstliches neuronales Netz (KNN) zu bilden, das aus verschiedenen Schichten mit sogenannten Neuronen besteht. Dabei übernimmt jedes Neuron eine einfache mathematische Berechnung und gibt das Ergebnis an die Neuronen der nächsten Schicht weiter.

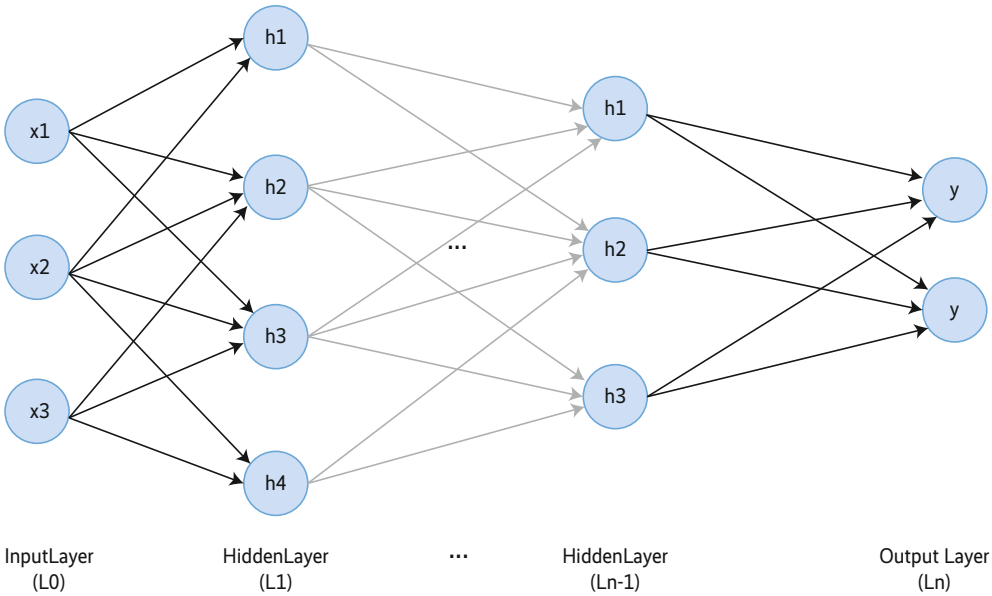
Durch die jüngsten Erfolge mag es überraschend klingen, dass die Ideen und Konzepte hinter neuronalen Netzen eigentlich schon relativ alt sind. Ein Blick auf die Geschichte und Konzepte von KNNs, die bereits in den 1940er Jahren ihren Anfang nahm, entmystifiziert die Themen KI, Deep Learning und neuronale Netze. Siehe dazu den Artikel "Am Anfang war das Neuron" auf Seite 21.

Vor allem in den letzten Jahren haben KNNs und Deep Neural Networks (DNNs) immer mehr an Popularität gewonnen und brachten große Erfolge in den Bereichen Bild- und Spracherkennung, Textverständnis und Übersetzung. Gründe dafür sind sicherlich auch die riesige Anzahl an verfügbaren Daten und die immensen, verfügbaren Rechenkapazitäten. Denn zum (An-)Lernen eines DNN mit vielen Layer und Neuronen sind viele Daten von guter Qualität und viel Rechenleistung notwendig – das gab es in den 1940er Jahren halt noch nicht!

Das Gehirn des Rechners

Immer wieder fällt im Zusammenhang mit künstlichen neuronalen Netzen der Vergleich zum menschlichen Gehirn. Das bedeutet, dass das KNN dessen Funktionsweise oder besser der des Nervensystems nachempfunden ist. Letzteres verarbeitet,

Ein KNN besteht aus einem Input- und einem Output-Layer, zwischen denen zahlreiche Hidden Layer zu finden sind – hier: ein Feed-Forward NN (Abb. 1)

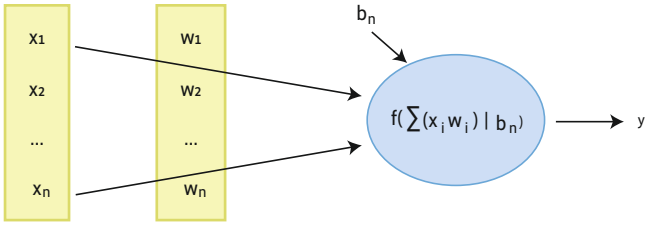


vereinfacht gesprochen, Informationen, und im menschlichen Gehirn gibt es Neuronen, die über Synapsen, Dendriten und Transmitter miteinander verbunden sind und so eingehende Signale in Informationen umwandeln können. Jedes Neuron fungiert dazu als ein Schalter mit Informationsein- und -ausgang, dessen Aktivierung erfolgt, wenn genügend Reize anliegen, sodass es dann einen Impuls an andere Neuronen senden kann. Dadurch bilden sich schließlich Erinnerungen und Intelligenz.

Die genaue Erklärung des Vorgangs ist das Gebiet der Gehirnforscher, aber in der Informatik hat sich folgender Ansatz festgesetzt: Ein künstliches, neuronales Netz besteht aus Neuronen mit gerichteten und gewichteten Verbindungen. Sie sind in Schichten (Layer) organisiert, wobei die Neuronen von der Schicht L1 mit allen Neuronen der Schicht L2 verbunden sind (s. Abb. 1). Die erste Ebene heißt Eingangsschicht (Input Layer, LI) und die letzte Ausgangsschicht (Output Layer, LO). Die Anzahl der Neuronen im Input Layer entspricht der Anzahl der Eingangswerte, und die Zahl der Neuronen im Output Layer der Anzahl der möglichen Ergebnisse. Zwischen Eingangs- und Ausgangsschichten kann es beliebig viele Zwischenschichten (Hidden Layer) geben. Letztere haben dem gesamten Konzept den Namen Deep Neural Networks eingebracht.

Das Neuron unter der Lupe

Um die Arbeitsweise der Neuronen und die Bedeutung der Verbindungen zu verstehen, kommt Mathematik ins Spiel. Jede Verbindung kennzeichnet den Datenfluss von einem Neuron zu einem anderen. Die übermittelten Daten sind einfache Zahlwerte, die unterschiedlich gewichtet sind. Das Neuron verknüpft die eingehenden Werte *x1 … xn* und die Wichtungen *w1 … wn* mathematisch. Hierzu führt das Neuron eine einfache Berechnung durch: Es bildet die gewichtete Summe über die Eingangswerte (s. Abb. 2).

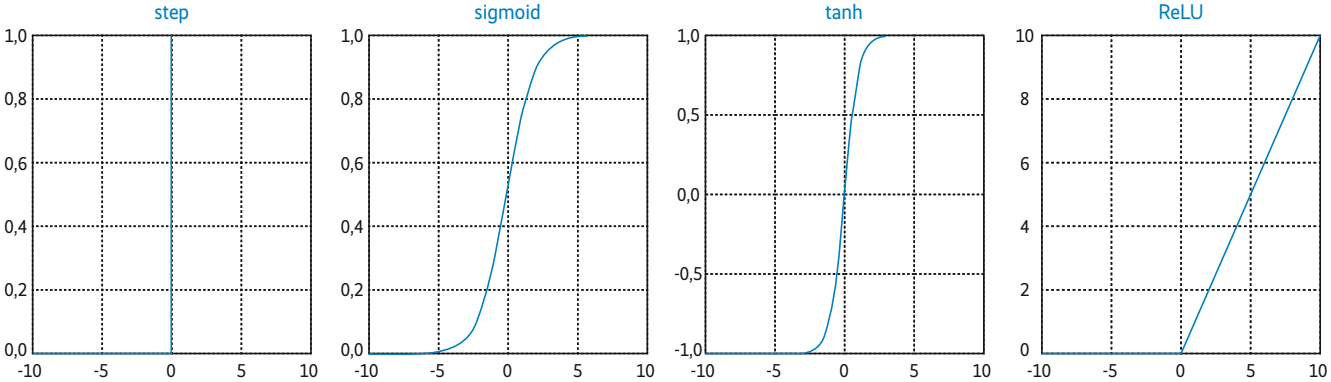


Schematische Darstellung eines Neurons (Abb. 2)

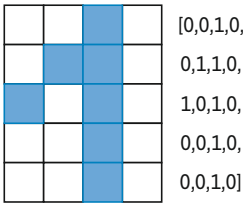
Listing 1: Dot-Produkt von X und W in Python

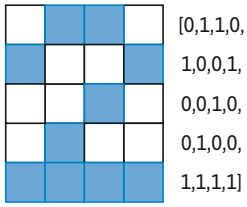
```
import numpy as np

def sigmoid(x):
    return 1/(1+np.exp(-x))

def layer(I,W):
     return sigmoid(np.dot(I,W))
```

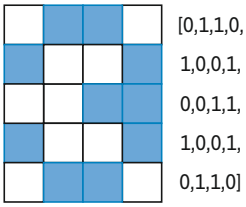
Hinzu kann noch ein Initialwert (Bias) b0 kommen. Das Resultat entscheidet darüber, ob das Neuron aktiviert ist und somit zum Ergebnis beitragen soll. Es gilt die einfache Regel, dass ein Wert v <= 0 einer Nicht-Aktivierung und ein Wert v > 0



Aktivierungsfunktionen *sigmoid*, *tanh* und *ReLU* (Abb. 3)







Ziffern 1, 2 und 3 als Trainingsdaten als Bitmap- und Array-Darstellung (Abb. 4)

einer Aktivierung entspricht. Mathematisch deckt das eine sogenannte Aktivierungsfunktion *f* ab. Eigentlich käme eine step-Funktion zum Einsatz, aber aus mathematischer Sicht haben sich die *sigmod*-, die *tanh*- oder auch die *ReLU*-Funktionen als sinnvoll erwiesen (s. Abb. 3). Sie bilden die *step*-Funktion (0 für v <= 0 und 1 für v > 0) relativ exakt nach, sind aber stetig differenzierbar. Denn zum Anlernen des Netzes ist die Ableitung der Aktivierungsfunktion notwendig. Da die *step*-Funktion an der Stelle x=0 leider nicht stetig ist, kommen die Ersatzfunktionen ins Spiel.

Nun kann das Neuron den Ausgabewert y liefern, wie in folgendem Beispiel:

$$y = sigmoid(\sum_{i=1}^{n}(x_i w_i) + b_0)$$

Von der Theorie zum Code

Für die Umsetzung der Theorie in Code bietet sich Python an. Hinzu kommt die NumPy-Bibliothek, die bei der Matrizen- und Vektor-Rechnung hilft. Im Grunde lassen sich die Eingangswerte sehr schön als Vektor *X* (Großbuchstaben im Artikel und den Listings stehen für Vektoren oder Matrizen) mit *i*-Werten und die Wichtung als Vektor *W* mit *i*-Gewichten repräsentieren – der Bias bleibt zunächst außen vor.

In der Vektordarstellung wird aus der gewichteten Summe das *Dot*-Produkt von *X* und *W*. Listing 1 zeigt den passenden Python-Code, der sich mit einfachen Werten testen lässt:

```
X= np.array([1,2,3])
W = np.array([1,1,1])
print(layer(X, W))
```

Dabei ergibt sich folgendes Resultat:

```
0.9975273768433653.
```

Das wirkt zunächst nicht besonders spektakulär, da es sich lediglich um ein einzelnes Neuron handelt, obwohl die Funktion hier *layer* heißt. Das Schöne bei der Verwendung von NumPy ist, dass Entwickler Vektoren, Matrizen und Tensoren verarbeiten können, ohne die Syntax zu ändern. Aus diesem Grund ist die *layer*-Funktion in der Lage, einen kompletten Layer mit mehreren Neuronen in einem Rutsch zu berechnen, abhängig davon, welche Form die Parameter I und W haben. Wie es ausschaut, wenn es etwas komplizierter wird, soll ein Beispiel veranschaulichen.

Listing 2: Die Ziffern im Eingangsvektor X und Ausgangsvektor Y

```
X = np.array([[0,0,1,0,0,1,1,0,1,0,1,0,0,0,1,0,0,0,1,0],       # Ziffer '1'
              [0,1,1,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,1,1,1,1],   # Ziffer '2'
              [0,1,1,0,1,0,0,1,0,0,1,1,1,0,0,1,0,1,1,0]])      # Ziffer '3'

Y = np.array([[1,0,0],     # klassifiziert als 1
              [0,1,0],     # klassifiziert als 2
              [0,0,1]])     # klassifiziert als 3
```

Listing 3: Initialisierung der Neuronenverbindungen

```
np.random.seed(1)                # Initialisierung

# 20 Eingangswerte und 30 Neuronen im Hidden-Layer
W0 = np.random.random((20,30))

# 30 Neuroen vom Hidden-Layer werden mit den 3 Ausgangswerten verbunden
W1 = np.random.random((30,3)).
```

Listing 4: Berechnung und Minimierung der Differenz zwischen Ist und Soll

```
def sigmoid_d(x):   # Ableitung der sigmoid-Funktion
    return sigmoid(x)*(1-sigmoid(x))

for i in xrange(60000):
    L0 = X
    L1 = layer(L0, W0)
    L2 = layer(L1, W1)

    L2_err = Y - L2              # Fehler = Soll -Ist
    L2_delta = L2_err * sigmoid_d(L2)
    W1 += np.dot(L1, L2_delta)

    if np.mean(np.abs(L2_err)) < 0.05:
        break

    L1_err = np.dot(L2_delta, W1)
    L1_delta = L1_err * sigmoid_d(L1)
    W0 += np.dot(L0, L1_delta)
)).
```

Zeichenerkennung – Das „Hello World!“ des KNN

Eine griffige Beispielanwendung benötigt Daten, die idealerweise gelabelt sind. Der MNIST-Datensatz (Modified National Institute of Standards and Technology) enthält viele Trainingsdaten als Bilder mit handgeschriebenen Ziffern. Er eignet sich gut zum Erklären neuronaler Netze und von Deep Learning. Für diesen Artikel soll aber ein kleiner eigener Datensatz ausreichen. Dabei geht es darum, ein Modell für drei Ziffern (1, 2 und 3) zu trainieren (s. Abb. 4) und als Test zu prüfen, wie gut das Modell Ziffern mit kleinen Abweichungen erkennen kann.

Die Ziffern lassen sich auf vier mal fünf Feldern darstellen und damit in entsprechende Eingangsvektoren mit 4 × 5 = 20 Werten übertragen. Ein weißes Feld entspricht dabei dem Wert 0.0 und ein schwarzes Feld dem Wert 1.0. Bei dem MNIST-Datensatz handelt es sich um 28 × 28 Felder mit Graustufen. Dort wäre ein Eingangsvektor 28 × 28 = 784 Felder groß und die Graustufen entsprechende *Double*-Werte mit dem jeweili-

Listing 5: Korrektur der Wichtung

```
W1 += np.dot(L1, L2_delta)
       # mit L2_delta = L2_err * sigmoid_d(L2)
       # und L2_err = Y - L2
```

Listing 6: Wichtung der Werte

```
W1[(Y-L2)*sigmoid'(L2)]*sigmoid'(L0)*L0

 W0 += np.dot(L0, L1_delta)
       # mit L1_delta = L1_err * sigmoid_d(L1)
       # und L1_err = np.dot(L2_delta, W1)
```

gen Schwarzanteil (0.0 = weiß bis 1.0 = schwarz). Für das vereinfachte Beispiel ergibt sich ein recht simples Mapping auf das KNN (s. Abb. 5). Listing 2 zeigt den zugehörigen Eingangsvektor X und den Ausgangsvektor Y (das Sollresultat für den Input X).

Bei Vektor *Y* entspricht [1,0,0] einer 1, [0,1,0] einer 2 und [0,0,1] einer 3. Bei dem MNIST-Modell wäre der Ausgangsvektor ein Vektor mit 10 Einheiten für die Ziffern 0 bis 9. Listing 3 zeigt das Initialisieren der Verbindungen zwischen den Neuronen, der anfänglich zufällig gewählten Wichtungen.

Der Code legt die Verbindung zwischen dem Layer *L0* (20 Neuronen für die 20 Pixel der Zifferngrafiken) und dem Layer *L1* (Hidden Layer mit 30 Neuronen) in den Vektor *W0* und für die Verbindung zwischen *L1* und *L2* (Output Layer mit drei Neuronen für die drei Ziffern) in W1 ab.

Beim Aufbau des KNN mit

```
L0 = X             # Input Layer
L1 = layer(L0, W0) # Hidden Layer
L2 = layer(L1, W1) # Output Layer
print(L2)
```

ergibt sich etwa Folgendes:

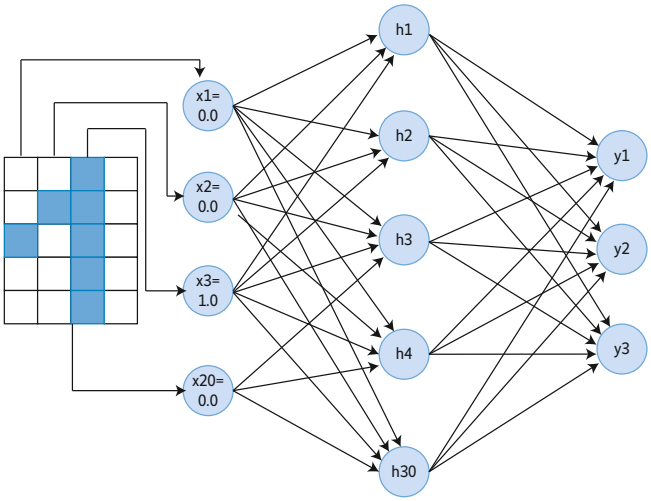
```
[[0.99999987 0.99999897 0.999999291
 [0.99999992 0.99999932 0.9999995 ]
 [0.99999992 0.99999931 0.9999995 ]]
```

Das ist sicherlich nicht das gewünschte Ergebnis, das eher so aussehen sollte wie Y:

```
[[1.0 0.0 0.0]
 [0.0 1.0 0.0]
 [0.0 0.0 0.1]]
```

Der Grund für die deutlichen Abweichungen ist, dass das KNN bisher noch nicht trainiert ist oder anders ausgedrückt: Es hat noch nichts gelernt.

Das wirft die Frage auf, wie ein künstliches neuronales Netz lernen kann. Die Antwort ist recht einfach: mit Trainingsdaten. Diesen Ansatz nennt man überwachtes Lernen (Supervised Learning). Der Trainingssatz besteht aus Werten und Labeln. Für jede Datenreihe ist bekannt, wie das Ergebnis aussehen muss. So lässt sich Soll und Ist miteinander vergleichen und das Modell mit jeder Iteration verbessern.



Mapping der Zahlen auf das KNN (Abb. 5)

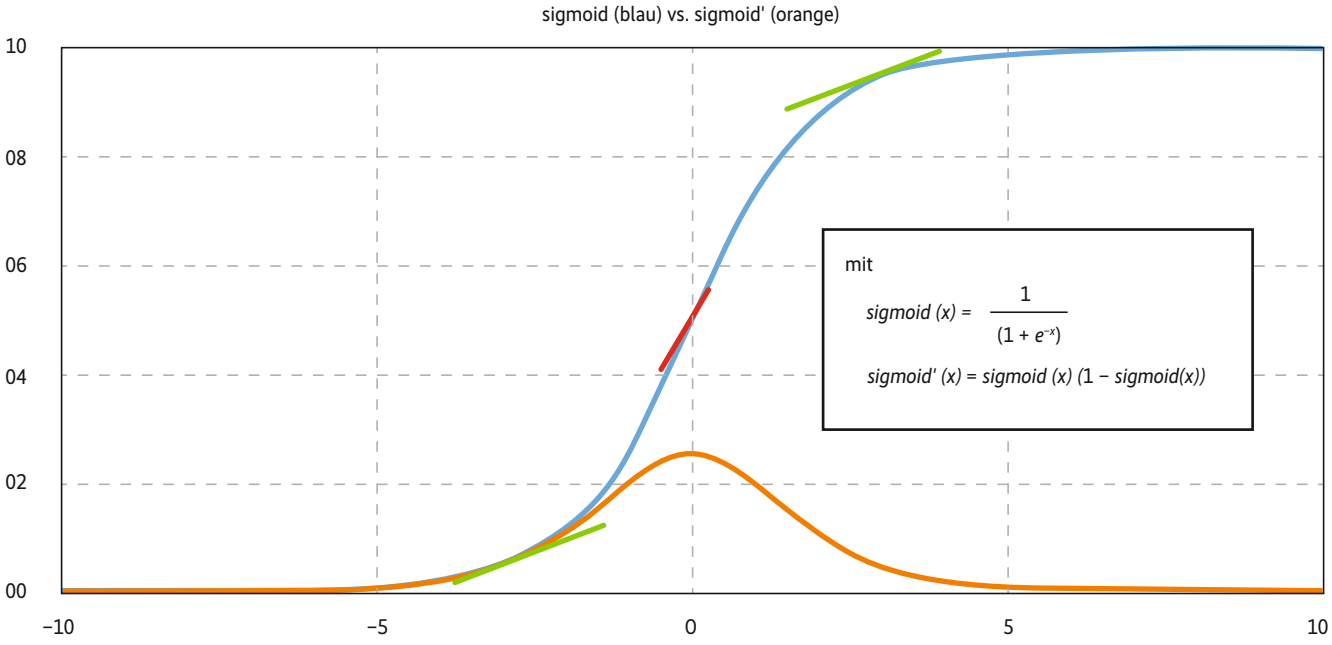
Training mit Backpropagation

Für das Beispiel ist bekannt, wie die Bitmaps beziehungsweise Arrays für die Ziffern 1,2 und 3 aussehen. Wenn am Eingang eine Ziffer 1 anliegt, soll somit *Y = [1,0,0]* herauskommen. Der Fehler *L2_err* ergibt sich durch den Vergleich des tatsächlichen (Ist-) (*L2*) mit dem Sollergebnis (*Y*):

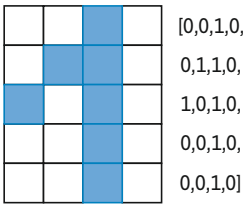
```
L2_err = Y -L2
```

Diese Abweichung müssen Entwickler anschließend auf die bisher rein zufälligen Wichtungen verteilen, um den Fehler zu verringern. Diesen Schritt wiederholen sie, bis er eine gewisse Grenze unterschreitet (s. Listing 4).

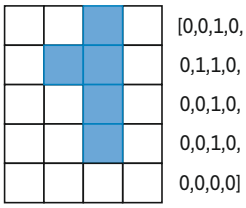
Die Verteilung des Fehlers auf die Wichtung ist entscheidend. Der erste Schritt ermittelt die Korrektur (*L2_delta*) zwischen *L2*



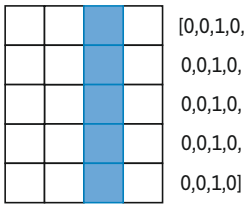
Sigmoid und Ableitung: Bei einem Wert um 0,5 (rote Steigung) ist eine Korrektur erforderlich. (Abb. 6)



Original



Test 1



Test 2

Beispiele für Abbildungen der Ziffer 1, die von den Trainingsdaten abweichen. (Abb. 7)

(Ausgabe) und *L1* (Hidden Layer). Das Delta ergibt sich aus dem Fehler (*L2_err*) und der Steigung im *sigmoid*-Graph an der Stelle des Werts. Wenn der Wert entweder groß (nahe 1) oder klein (nahe 0) ist, war das Modell sich sicher. Dabei ist die Fehlerkorrektur gering, da *sigmoid'* (Ableitung der *sigmoid*-Funktion) klein ist (s. Abb. 6: kleine, grüne Steigung). Sollte der Wert um y = 0.5 liegen, war das Modell recht unentschlossen und muss entsprechend mit einem großen Wert korrigiert werden (s. Abb. 6: große, rote Steigerung).

Mathematisch ist die Herleitung etwas komplexer und geht vom mittleren quadratischen Fehler

$$E = \frac{1}{2}||Y - L2||^2$$

mit Y = Sollwert und $L2$ = Output aus und deren Minimierung. Beim Verwenden des Gradienten-Abstiegs ergibt sich folgende Formel für die Korrektur der Wichtung, die kompliziert aussieht, aber im Prinzip die Kettenregel verwendet, da *L2* eine Funktion von *W1* ist:

$$\frac{\partial E}{\partial W1} = \frac{\partial E}{\partial L2} * \frac{\partial l2}{\partial W1} \quad \text{und} \quad \frac{\partial L2}{\partial w1} = sigmoid'(L2) * \frac{\partial (W1 * L1)}{\partial w1}$$

Und daraus folgt dann:

$$E'(W1) = \frac{\partial E}{\partial W1} = (Y - L2)\frac{\partial L2}{\partial W1} = (Y - L2) * sigmoid'(L2) * \frac{\partial W1L1}{\partial W1}$$
$$= (Y - L2) * sigmoid'(L2) * L1$$

Listing 5 zeigt die passende Codezeile dazu. Interessanter ist jedoch, wie sich der Fehler von *L2* auf *L1* auswirkt. Durch das Propagieren des Fehlers auf *L1* ergibt sich eine Wichtung aller Werte von *L1* in dem Maß, wie sie jeweils auf *L2* wirken. Hierfür gilt analog folgende Formel, die Listing 6 in Code umsetzt:

$$E'(W0) = \frac{\partial E}{\partial W0} = (Y - L2)\frac{\partial L2}{\partial W0} = (Y - L2) * sigmoid'(L2)\frac{\partial (W1L1)}{\partial W0} =$$
$$(Y - L2) * sigmoid'(L2) * W1 * sigmoid'(L1) * L0$$

Bei mehreren Hidden-Layer verwenden Entwickler diese Propagation auf weitere Layer. So ergibt sich bei 3 Layern:

```
E' (W2)=(Y-L3)*sigmoid' (L3)*L2
E' (W1)=(Y-L3)*sigmoid' (L3)*W2*sigmoid' (L2)*L1
E' (W0)=(Y-L3)*sigmoid' (L3)*W2*sigmoid' (L2)*W1*sigmoid' (L1)*L0
```

Dafür müssen Wichtungen und Werte angepasst und weiter propagiert werden. Dieses gibt dem Verfahren seinen Namen: Backpropagation.

Listing 7: Zwei Tests

```
# Test 1
L0 = np.array([[0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0,0,0,0]])
L1 = layer(L0, W0)
L2 = layer(L1, W1)
print(L2)

# Test 2
L0 = np.array([[0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0]])
L1 = layer(L0, W0)
L2 = layer(L1, W1)
print(L2)
```

Für weitere Optimierungen sei erwähnt, dass es noch weitere Parameter wie Schrittweiten gibt, mit denen sich verhindern lässt, dass man in einem lokalen Minimum landet.

Zeit zum Testen

Nun folgt die Frage, wie gut das Modell ist. Beim Testen mit einem neuen Input, also mit Ziffern, die nicht exakt den Gelernten entsprechen, sollte das Modell sie richtig klassifizieren. Abbildung 7 zeigt zwei von den Trainingsdaten abweichende Darstellungen der Ziffer 1. Test 1 aus Listing 7 ergibt

```
[[0.99997719 0.0000075  0.00001979]]
```

was einer Ziffer 1 mit einer Konfidenz von 99,9 Prozent entspricht. Test 2 aus demselben Listing ergibt

```
[[0.9994665  0.00000003 0.04031362]]
```

und entspricht ebenfalls einer 1 mit einer Konfidenz von 99,9 Prozent. Das korrekte Ergebnis überzeugt somit deutlich.

Natürlich sind die KNN-Modelle für echte Deep-Learning-Anwendungen deutlich komplexer. Dort finden sich KNNs mit bis zu 150 Layern, die jeweils eine Vielzahl Neuronen aufweisen. Diese Netze benötigten viele Daten und hohe Rechenleistung, um die Anpassung zu berechnen. Außerdem existieren spezielle Ausprägungen von KNNs. Data Scientists nutzen bei der Bildverarbeitung Convolutionals Neural Networks, um die Bilder zunächst mit Methoden wie Convolution, Cropping oder Filtern vorzuverarbeiten, bevor sie die resultierenden Bilddaten klassifizieren.

Fazit

Im Bereich Textverständnis und Übersetzung kommen Recurrent Neural Networks (RNNs) zum Einsatz, die mehrere KNNs in Reihe schalten, um den Kontext der Hidden-Layer weiterzureichen. Darüber hinaus gibt es zahlreiche Verfeinerungen und Optimierungen im Bereich Backpropagation wie Stochastic Gradient Descent oder Batches.

Im Alltag ist das genaue Verständnis für den Einsatz der Techniken gar nicht notwendig: Kaum jemand schreibt eigenen Code, um ein DNN abzubilden und zu verarbeiten. Dafür existieren Frameworks wie TensorFlow, Caffe, pyTorch, Keras oder CNTK, die viele Funktionen zur Optimierung, Hardware-Beschleunigung (GPU, FPGA), Visualisierung sowie zum Batch-Processing und Lesen von Daten anbieten. Im Grunde gehen aber alle KNNs auf die in diesem Artikel dargestellten Ansätze zurück. (rme@ix.de)

Marcel Tilly
beschäftigt sich Schwerpunkt mäßig bei Microsoft mit den Themen Artificial Intelligence und IoT. Nebenbei spricht er auf Konferenzen, schreibt Artikel oder programmiert irgendwas in den Bereichen AI, ML oder auch IoT.