Künstliche neuronale Netze

M. Heckel

12.12.2017

1 Motivation — Warum neuronale Netze?

Durch die Programmierung von Maschinen lassen sich Algorithmen in Software implementieren. Dieses Programm kann dann eine Klasse von Problemen lösen. Bei Problemen, die sich einfach durch einen Algorithmus lösen lassen (z.B. Berechnungen) ist dieses Vorgehen sehr effizient: Ein einmal entwickelter Algorithmus kann immer wieder verwendet werden, um auch ähnliche Problemstellungen lösen zu können.
Allerdings gibt es auch Problemstellungen, die sich nicht, bzw. nur sehr schwer, durch einen Algorithmus lösen lassen. Dazu zählen z.B. das Erkennen von Inhalten auf Bildern oder die Erkennung menschlicher Sprache, sowie viele weitere Anwendungsgebiete. Wie komplex wäre ein Algorithmus, der erkennen soll, ob auf einem Bild ein Hund oder eine Katze dargestellt ist? Diese Aufgabe ist mit klassischen Algorithmen nur sehr schwer lösbar, da ein Computer ein Bild (Rastergrafik) nur als eine Folge von Bildpunkten mit einer gewissen Farbe/Helligkeit interpretiert. Wie soll er anhand dieser Informationen entscheiden, ob es sich um einen Hund oder um eine Katze handelt?
Die Lösung dieses Problems ist mit künstlichen neuronalen Netzen ohne Schwierigkeit möglich, da diese die Prinzipien der Gehirne von Lebewesen verwenden. Somit können auch Aufgaben, die scheinbar nur von ‘intelligenten Lebewesen’ erfüllbar sind, auch von Computern erfüllt werden.
Zusammengefasst stellen künstliche neuronale Netze eine Möglichkeit dar, mit der Computer ‘das Denken lernen können’. Außerdem können künstliche neuronale Netze aus ihren eigenen Fehlern lernen, sodass das Netz mit der Zeit immer besser wird, wenn es Informationen zur Qualität der gelieferten Ergebnisse erhält.

2 Neuronale Netze — Theoretische Grundlagen

2.1 Was ist ein Neuron?

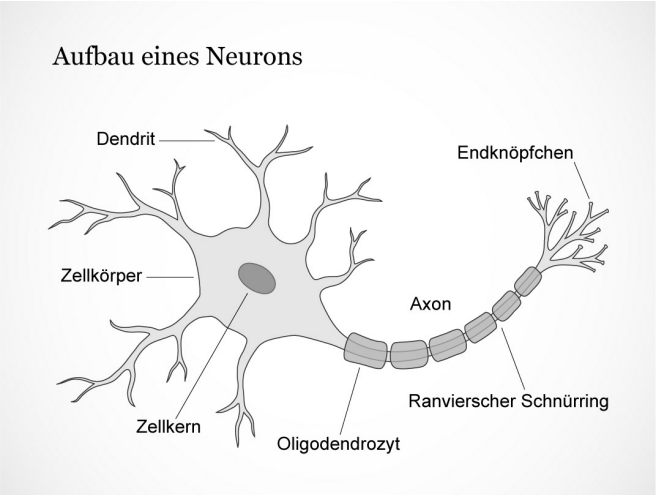


Abbildung 0.1: Nervenzelle

Ein Neuron, auch Nervenzelle genannt, ist eine Zelle, die über mehrere ‘Eingänge’ (Dendriten), sowie einen ‘Ausgang’ (Axon) verfügt. Dieser Ausgang wird wiederum auf mehrere Endknöpfchen verteilt, die alle den gleichen Signalwert an weitere (nachgeschaltete) Neuronen geben.
Diese Zellen verfügen über ein sogenanntes Schwellpotential. Das Axon gibt nur ein Signal weiter, wenn die Potentiale der Eingänge das Schwellpotential übersteigen. Anderenfalls gibt das Axon kein Signal weiter.
Natürlich ist es nun kein Problem, ein solches Neuron ‘künstlich’, also als Datenstruktur darzustellen.

2.2 Künstliche Neuronen vernetzen

Ein einzelnes Neuron hat nicht besonders viel mit Intelligenz zu tun. Es schaltet einfach in Abhängigkeit der Signale am Eingang ein Signal am Ausgang — oder auch nicht. Die eigentliche Fähigkeit des Denkens entsteht aus einer Zusammenschaltung vieler solcher Neuronen. Da die Zusammenschaltung der Neuronen im Gehirn sehr komplex ist, wird dieser Zusammenhang für künstliche Neuronen vereinfacht:
Die Neuronen werden in Schichten eingeteilt, wobei es eine Eingangsschicht gibt, die eingehende Signale aufnimmt, eine — oder auch mehrere — ‘versteckte’ Schichten, mit denen das Netz nicht mit der Umwelt interagiert, und einer Ausgangsschicht, die dann die Ergebnisse liefert.

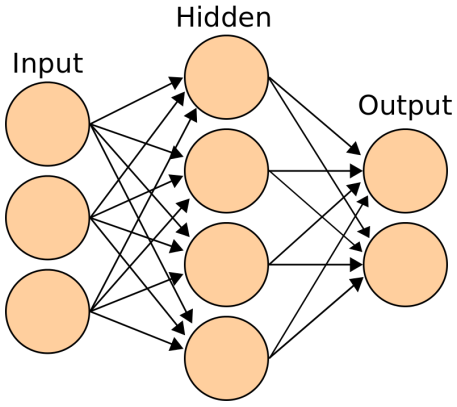


Abbildung 0.2: Einfaches neuronales Netz

2.3 Gewichte

In Punkt 2.2 wurde gezeigt, dass künstliche Neuronen zusammen geschaltet werden können. Allerdings wird noch eine Möglichkeit benötigt, um die Auswirkung eines Neurons auf ein anderes Neuron anzupassen. Dies geschieht mit sog. Gewichten.
Diese Gewichte sind Zahlen zwischen 0 (es kommt kein Signal an) und 1 (das maximale Signal kommt an). Durch die Gewichte wird die Dämpfung des Signals zwischen zwei Neuronen definiert.
Im Neuron werden dann die gedämpften Werte addiert.

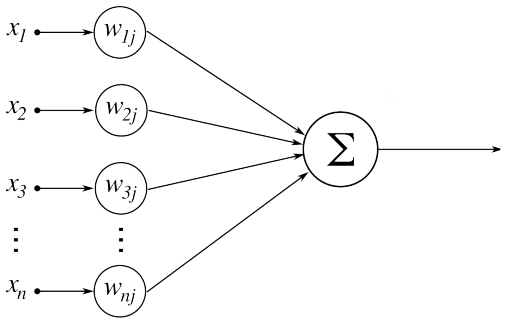


Abbildung 0.3: Neuron mit Gewichten

Da Gewichte immer zwischen zwei Knoten stehen, werden sie auch mit zwei Zahlen bezeichnet. Die erste Zahl steht dabei für die Nummer des Vorgängerknotens, die zweite Zahl für die Nummer des Nachfolgerknotens.
Wenn der Einfluss eines künstlichen Neurons auf ein anderes künstliches Neuron also zu groß ist, so muss das entsprechende Gewicht verringert werden. Ist der Einfluss hingegen zu gering, muss das Gewicht erhöht werden. Diese Anpassung wird im Abschnitt zur Backpropagation genauer behandelt.

2.4 Aktivierungsfunktion

Nun besteht das Problem, dass der Wertebereich des neuronalen Netzwerks nach oben nicht direkt begrenzt ist, weshalb die Auswertung der Ergebnisse schwierig ist. So lässt sich z.B. nicht sagen, ob ein Wert von 100 auf einem Ausgangsknoten viel oder wenig ist.
Das Problem daran ist, dass das Netzwerk bei großen Eingangswerten übersteuern würde. Außerdem würden ‘unwichtige’ Anteile nicht herausgefiltert werden.
Wie bereits im Abschnitt über biologische Neuronen beschrieben, verfügen diese über ein Schwellpotential. Erst wenn die Summe der Eingangspotentiale diesen Wert übersteigt, ‘schaltet’ das Neuron.
Wir könnten diesen Ansatz nun auf unsere Neuronen anwenden, und bekämen folgende (oder eine ähnliche) Aktivierungsfunktion:

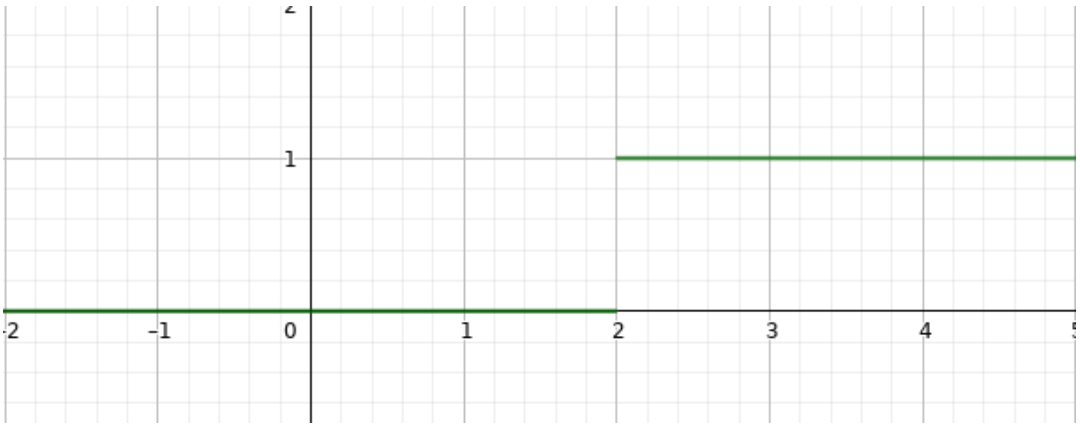


Abbildung 0.4: Einfache Aktivierungsfunktion

Betrachten wir zunächst eine einfache Aktivierungsfunktion (Abbildung 0.4):
Die x-Achse stellt dabei die Summe der gewichteten Eingänge, und die y-Achse das Signal am Ausgang dar. Somit ist der Wertebereich begrenzt. Eine 1 am Ausgang ist also viel, eine 0 ist wenig.
Dabei gibt es allerdings das Problem, dass die Anpassung der Gewichte, um ein bestimmtes Ergebnis zu erreichen, nur sehr schwer möglich ist, da der Eingangswert der Funktion am Ausgang nur an genau einer Stelle eine Veränderung hervorruft, in diesem Fall bei x=2.
Um dieses Problem zu beheben, wird in der Praxis häufig eine logistische Funktion, meist die Sigmoid-Funktion verwendet: $y = \frac{1}{1+e^{-x}}$. Graphisch sieht diese Funktion folgendermaßen aus:

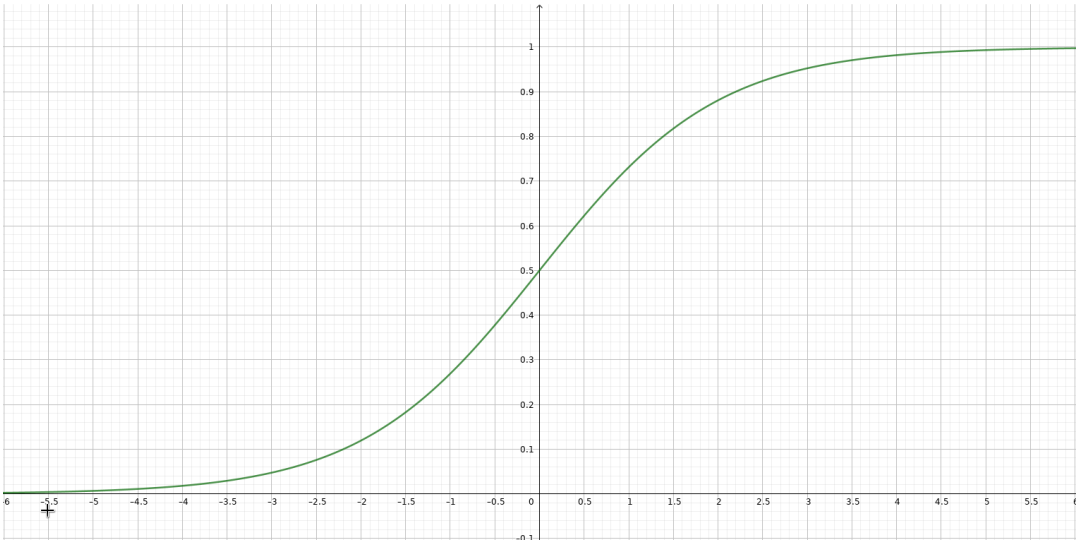


Abbildung 0.5: Logistische Aktivierungsfunktion

Diese Funktion beschränkt die Möglichen Werte auf einen Bereich zwischen 0 und 1, wobei ein größerer Eingangswert aber immer einen größeren Ausgangswert zur Folge hat. Somit ist die Anpassung der Gewichte deutlich einfacher, da ein Neuron nicht mehr binär ist, d.h. nur die Zustände 1 oder 0 ausgeben kann, sondern nun auch jeden beliebigen Zwischenwert annehmen kann.

2.5 Aus Fehlern lernen mit Backpropagation

Bis jetzt wurde immer davon ausgegangen, ein bereits trainiertes neuronales Netzwerk vorliegen zu haben, allerdings ist dies in der Realität meist nicht gegeben. Beim Erstellen des Netzwerkes werden die Gewichte vorerst zufällig initialisiert (zwischen 0 und 1).
Danach wird das Netzwerk mit Testdaten trainiert. Diese Testdaten bestehen sowohl aus der 'Aufgabenstellung' als auch aus der dazugehörigen 'Lösung'.
Das Training des Netzes findet in mehreren Phasen statt, die sich für jeden Testdatensatz wiederholen:

- 'Aufgabenstellung' an das Netzwerk übergeben
- Lösung des Netzwerks mit gegebener Lösung der Aufgabe vergleichen
- Den Fehler durch Backpropagation auf die einzelnen Gewichte zurückführen

Wie funktioniert Backpropagation?

- Fehler, der an einem künstlichen Neuron aufgetreten ist, anteilig auf die Gewichte der Verbindungen verteilen
- Summe der Fehler aller Ausgänge ergibt Fehler der Knoten in der vorherigen Schicht
- Die Gewichte anhand einer sog. Lernrate korrigieren (je größer die Lernrate, desto größer die Korrektur)
  Bei einer zu geringen Lernrate verändert sich das Netz zu langsam, und hält zu stark an falschen Ergebnissen fest. Bei einer zu hohen Lernrate verwirft das Netz vorher Gelerntes, und lernt nur aus den letzten Beispielen.

2.6 Zusammenfassung: Künstliches Neuron

Es ist somit möglich, die Nervenzellen von Lebewesen technisch ‘nachzubauen’:

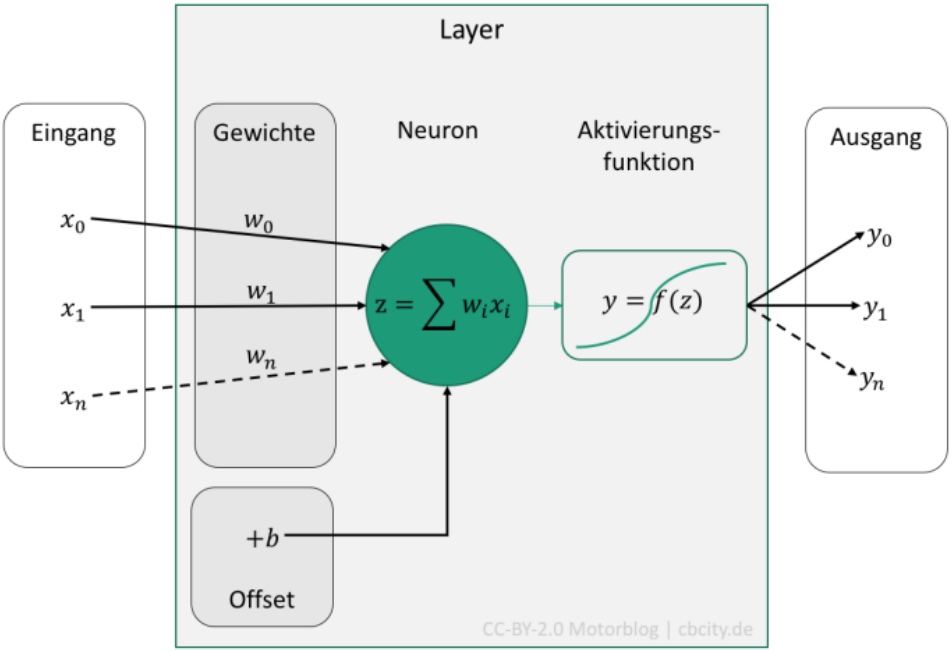


Abbildung 0.6: Technisches Modell eines Neuron

Dieses künstliche Neuron besitzt zwischen 1 und n Eingänge (x_n), die jeweils mit einem Gewicht (w_n) versehen sind. Diese werden von dem künstlichen Neuron addiert (ggf. kann auch ein zusätzlicher Offset addiert werden). Daraufhin wird diese Summe durch die Aktivierungsfunktion normalisiert. Dieser normalisierte Wert wird dann an alle Ausgänge 1 bis n (y_n) weitergegeben. Dabei ist wichtig, dass die Anzahl der Ein- und Ausgänge nicht gleich sein müssen.

3 Neuronale Netze — Technische Grundlagen

3.1 Berechnung des Netzwerks

Dieses technische Neuron ist bereits ein guter Ansatz. Allerdings besteht die Zielsetzung darin, Netze aus solchen Neuronen zu erstellen, die dann von einem Computer berechenbar sind, und in der Konsequenz dafür sorgen, dass der Rechner ‘intelligent’ ist. In diesem Abschnitt wollen wir uns anschauen, wie solche Netze einfach und effizient von Computern berechnet werden können.

Sicher wäre es möglich, ein Programm zu schreiben, dass künstliche Neuronen als Objekte verwaltet, und diese dann vernetzt. Allerdings wäre die Umsetzung eines neuronalen Netzes damit verhältnismäßig kompliziert. Aus diesem Grund wird in Folge eine bessere Methode betrachtet.

3.2 Grundlagen: Matrizenmultiplikation

Definition 1 (*Matrix*)

Eine Matrix von Typ (m; n) ist ein rechteckiges Schema von $m \cdot n$ Elementen, die in m Zeilen und n Spalten angeordnet sind.

Das ist ein Beispiel für eine 2x3-Matrix:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

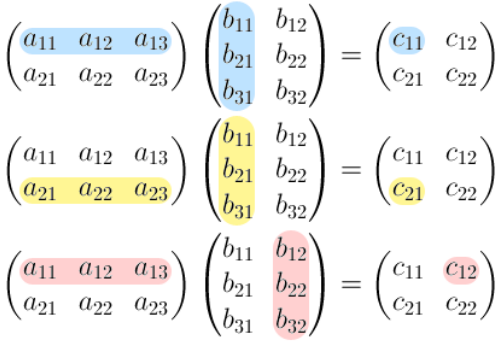


Abbildung 0.7: Matrizenmultiplikation

Um das Produkt zweier Matrizen berechnen zu können, müssen diese beiden Matrizen inverse Dimensionen haben, d.h. wenn die erste Matrix die Dimensionen 2x3 hat, dann muss die zweite Matrix die Dimensionen 3x2 haben.
Nun werden (wie in der Grafik zu sehen) die erste Zeile der ersten Matrix und die erste Spalte der zweiten Matrix benötigt. Das Ergebnis in der dritten Matrix wird durch Multiplikation der jeweils ersten, zweiten und dritten Elemente, und deren Addition gebildet:

$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31}$

Analog dazu werden die weiteren Elemente der Zielmatrix berechnet:

$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32}$

$c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31}$

$c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32}$

3.3 Berechnung des Netzwerks mit Matrizen

Angenommen, es sei ein sehr einfaches neuronales Netz mit vier Neuronen gegeben: Die Ausgänge berechnen sich folgendermaßen:

$b_1 = (a_1 \cdot w_{11}) + (a_2 \cdot w_{21})$
$b_2 = (a_1 \cdot w_{12}) + (a_2 \cdot w_{22})$

Danach muss jeweils die Aktivierungsfunktion angewendet werden, um diese Ausgänge zu normalisieren.

Das Aufstellen dieser Formeln ist für kleine Netze noch ohne Schwierigkeiten möglich. Werden die Netze allerdings größer, wird das Aufstellen dieser Formeln komplex. Außerdem sollte die Anzahl der Knoten des neuronalen Netzes möglichst als Parameter bei der Erstellung des Netzes angegeben werden können.

Dieses Problem lässt sich mit Matrizenmultiplikation lösen (Produkt):

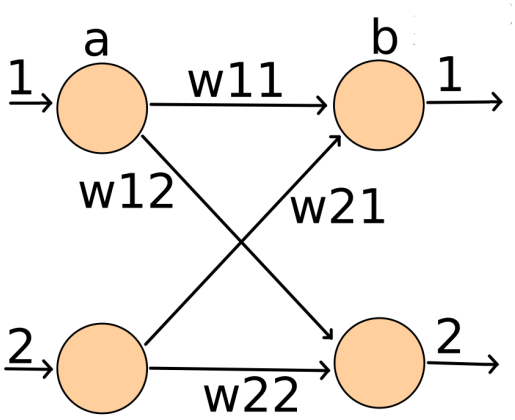


Abbildung 0.8: Einfaches neuronales Netz

$$\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{pmatrix} = \begin{pmatrix} (a_1 \cdot w_{11}) + (a_2 \cdot w_{21}) \\ (a_1 \cdot w_{12}) + (a_2 \cdot w_{22}) \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Somit entspricht die obere Zeile der Ergebnismatrix b_1 und die untere Zeile b_2. Um danach weiterrechnen zu können muss nun nur noch die Aktivierungsfunktion auf alle Elemente der Ergebnismatrix angewendet werden.

3.4 Backpropagation

Wie in Punkt 2.5 erklärt, muss das neuronale Netz nach der Erstellung trainiert werden. Vorerst wird dem Netz eine entsprechende ‘Aufgabe’ gestellt. Danach wird diese mit der ‘Lösung’ verglichen. Nun muss der Fehler berechnet werden:

- $(soll - ist)$
  Da diese Methode linear ist, wird ein doppelt so großer Fehler doppelt so stark korrigiert. Der Nachteil besteht darin, dass sich positive und negative Fehler aufheben. Wenn das Netz also insgesamt so weit über den ‘Lösungen’ liegt, wie es bei anderen ‘Aufgaben’ darunter liegt, wird nicht korrigiert (bzw. nur ‘hin und her’), sodass das Netz am Ende nichts ‘gelernt’ hat.

- $|soll - ist|$
  Mit dieser Fehlerfunktion kann das Problem mit den negativen Fehlern behoben werden, allerdings besteht hier das Problem darin, dass ein Fehlerminimum ggf. nicht erreicht wird, da der Fehler immer ‘überkorrigiert’ wird, d.h. wenn der Wert

des Netzes jetzt etwas zu groß ist, wird das Netz so korrigiert, dass der Wert nächstes Mal kleiner ist. Dann ist er aber ggf. zu klein und wird wieder nach oben korrigiert. Das eigentliche Minimum wird somit nicht erreicht.

- $(soll - ist)^2$
  Diese Fehlerfunktion ist immer positiv, d.h. gegenläufig Fehler heben sich nicht auf. Außerdem wird der Fehler bei kleinen Abweichungen sehr klein (kleiner als bei einer linearen Funktion) und bei größeren Abweichungen sehr groß. Somit ist es unwahrscheinlicher, dass das Fehlerminimum immer übersprungen wird.
- ...

Im weiteren Verlauf des Handouts wird aus Gründen der Einfachheit die Funktion $soll - ist$ verwendet.

Wie funktioniert nun die Backpropagation?

Zuerst wird der Fehler auf die einzelnen Gewichte verteilt. Danach wird die notwendige Änderung der Gewichte berechnet:
$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k(1 - O_k) \cdot O_j$
j und k geben dabei die Positionen an, d.h. j die Position des Knotens vor dem Gewicht, und k die Position des Knotens nach dem Gewicht. Das wird deutlicher, wenn wir die Gleichung für $j = 1, k = 2$ aufstellen:
$\Delta W_{1,2} = \alpha \cdot E_2 \cdot O_2(1 - O_2) \cdot O_1$
Dabei ist E der berechnete Fehler, O_k der vom Netz berechnete (tatsächliche) Wert, und O_j der Eingangswert des Knotens, bzw. das Ergebnis des vorherigen Knotens. α gibt die Lernrate des Netzes an.
Nun wird ΔW_{jk} noch zu W_{jk} addiert. Danach ist die Anpassung dieses Gewichtes abgeschlossen.
Dieser Vorgang wird für alle Gewichte zwischen der versteckten Schicht und der Ausgangsschicht wiederholt. Danach werden die Fehler auf die Gewichte der vorherigen Schichten (in diesem Beispiel gibt es nur eine vorherige Schicht) verteilt, und auch diese Gewichte angepasst, bis das gesamte Netz korrigiert wurde.

Geht das auch mit Matrizen?

Analog zum Berechnen der 'normalen' Ergebnisse lassen sich auch diese Berechnungen durch Matrizen ausdrücken, was das Ganze deutlich vereinfacht. Allerdings muss dafür vorher eine weitere Operation mit Matrizen betrachtet werden: die **Transposition**. Dabei werden die Dimensionen der Matrix vertauscht, indem die Matrix an der Hauptdiagonale gespiegelt wird. Bei einer $1 \times n$-Matrix werden die Elemente, die von oben nach unten standen, nun von lins nach rechts geschrieben, oder anders herum.

Beispiel:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Die Transposition der Matrix M wird mit M^T bezeichnet. Das ist notwendig, da die

Matrix der Ausgänge der vorherigen Schicht transponiert werden muss, um das Produkt bilden zu können. Die Berechnung sieht dann so aus:
$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k(1 - O_k) \cdot O_j^T$
Diese Formel kennen wir bereits von weiter oben. Wenn wir nun für $E_k * O_k * (1 - O_k)$ und O_j Matrizen verwenden, und O_j transponieren, kommen wir für je drei Ein- und Ausgänge auf folgende Berechnung:

$$\begin{pmatrix} E_1 \cdot S_1(1-S_1) \\ E_2 \cdot S_2(1-S_2) \\ E_3 \cdot S_3(1-S_3) \end{pmatrix} \times \begin{pmatrix} O_1 & O_2 & O_3 \end{pmatrix} = \begin{pmatrix} \Delta W_{1,1} & \Delta W_{2,1} & \Delta W_{3,1} \\ \Delta W_{1,2} & \Delta W_{2,2} & \Delta W_{3,2} \\ \Delta W_{1,3} & \Delta W_{2,3} & \Delta W_{3,3} \end{pmatrix}$$

Der Vorteil besteht darin, dass diese Formel für beliebig viele Ein- und Ausgänge funktioniert, ohne etwas am Programm zu ändern.

4 Neuronale Netze praktisch

4.1 Neuronale Netze in Python — Grundlagen

Das Netz für diesen Vortrag wurde in Python implementiert. Prinzipiell ist dies auch in jeder anderen Programmiersprache möglich, allerdings ist es förderlich, wenn die Programmiersprache z.B. durch Verwendung einer Bibliothek die Multiplikation und Transposition von Matrizen unterstützt. Das hier programmierte Netz unterscheidet Ziffern, die es auf Pixelgrafiken mit einer Auflösung von 28*28px erkennt. Im folgenden ist der Quellcode der Methoden der Klasse ‘neuronalNetwork’ zu sehen:

Methode für das Lernen

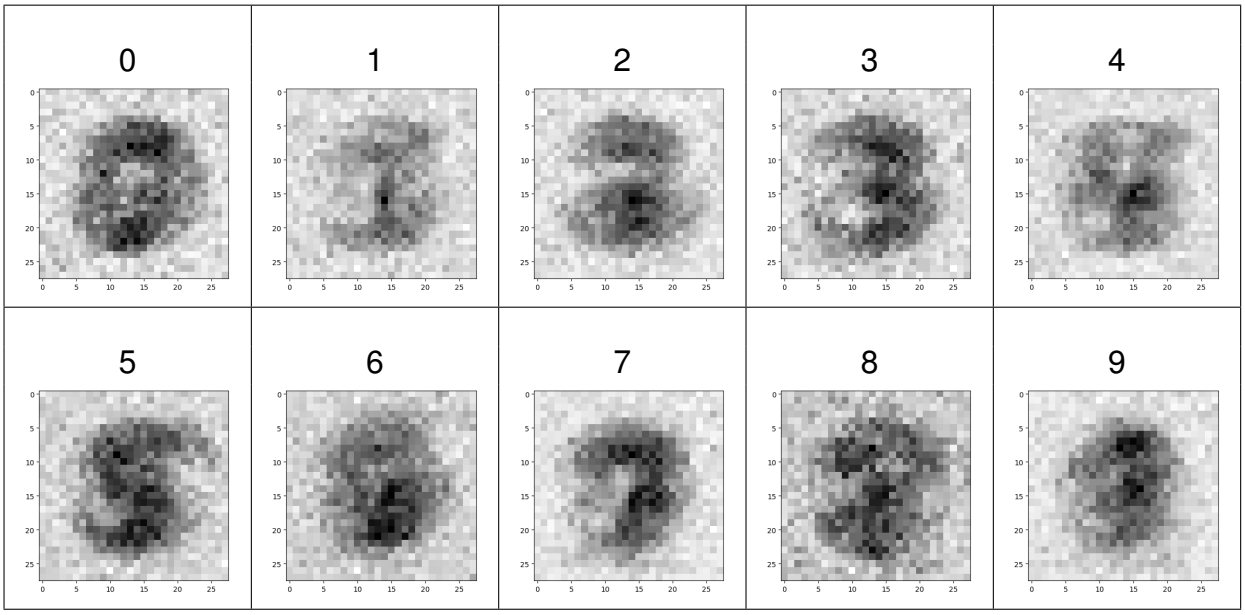
```
1  def train(self, inputs_list, targets_list):
2      inputs = numpy.array(inputs_list, ndmin=2).T
3      targets = numpy.array(targets_list, ndmin=2).T
4
5      hidden_inputs = numpy.dot(self.wih, inputs)
6      hidden_outputs = self.activation_function(hidden_inputs)
7      final_inputs = numpy.dot(self.who, hidden_outputs)
8      final_outputs = self.activation_function(final_inputs)
9
10     output_errors = targets - final_outputs
11     hidden_errors = numpy.dot(self.who.T, output_errors)
12
13     self.who += self.lr * numpy.dot((output_errors * final_outputs *
14             (1.0 - final_outputs)), numpy.transpose(hidden_outputs))
15     self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs *
16             (1.0 - hidden_outputs)), numpy.transpose(inputs))
```

Methode für die Vorhersage

```
1 def query(self, inputs_list):
2     inputs = numpy.array(inputs_list, ndmin=2).T
3     hidden_inputs = numpy.dot(self.wih, inputs)
4     hidden_outputs = self.activation_function(hidden_inputs)
5     final_inputs = numpy.dot(self.who, hidden_outputs)
6     final_outputs = self.activation_function(final_inputs)
```

4.2 Dem neuronalen Netz ins Gehirn schauen

Das ‘Gehirn’ des neuronalen Netzes besteht aus den Gewichten, welche ‘nur’ Zahlen zwischen 0 und 1 sind. Um trotzdem vorstellbar zu machen, wie es in dem ‘Gehirn’ aussieht, wurden die gewünschten Ergebnisse übergeben, und dann rückwärts gerechnet. Im Folgenden sind die vom Netzwerk berechneten Eingaben zu sehen:



Diese Bilder zeigen, wie sich das Gehirn die entsprechenden Zahlen ‘vorstellt’.

5 Zusammenfassung

Es ist möglich, einfache mathematische Operationen mit Matrizen zu verwenden, um neuronale Netze zu erstellen, zu trainieren, und abzufragen. Diese Netze sind der Funktionsweise biologischer Gehirne nachempfunden, weshalb sie Probleme auch so lösen, wie es ein biologisches Gehirn tun würde: Ein Problem wird nicht durch einen eindeutigen Algorithmus, sondern durch Lernen und Anwenden des Gelernten gelöst. Natürlich kann das Netz aus schlechten Daten auch nicht richtig lernen, und wird nicht richtig funktionieren.
Wie auch bei biologischen Gehirnen kommt es bei künstlichen 'Gehirnen' mitunter zu Fehlern. Das Ziel ist im Gegensatz zur Entwicklung von Algorithmen nicht die vollständige Vermeidung von Fehlern, sondern die Minimierung der Fehler, da eine vollständige Vermeidung nicht möglich ist.
Daraus ist bereits der Anwendungsbereich der künstlichen neuronalen Netze erkennbar: Das Lösen von Problemen, die algorithmisch nur schwer oder gar nicht gelöst werden können. Dazu gehören z.B. das Erkennen von Objekten auf Pixelgrafiken, Spracherkennung, und viele weitere Gebiete.
Der Vortrag, sowie dieses Handout basieren zu großen Teilen auf dem Buch 'Neuronale Netze selbst programmieren' von Tariq Rashid. Der Quellcode aller Beispiele aus diesem Buch liegt auf GitHub:
https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork

Ich hoffe, mit dem Vortrag und diesem Handout ein wenig Interesse an künstlichen neuronalen Netzen geweckt zu haben. Bei Fragen oder Anregungen kann gerne eine Mail an mich geschickt werden.