

# Neuronales Netz

Zwischenstand 1. Halbjahr

---

## Inhalt

Theorie des neuronalen Netzes.....	2
Grundidee des Neuronalen Netzes .....	2
Berechnung der gewichteten Summe mittels Matrizenmultiplikation .....	2
Fehlerberechnung & Backpropagierung.....	3
Aktualisierung der Gewichte .....	3
Implementierung – Zwischenstand 18.01.2024 .....	3
Einlesen der Bilder.....	3
Gewichtsmatrizen.....	4
Zusätzliche Funktionalitäten.....	4
Noch fehlend .....	5
Quellen .....	5

# Theorie des neuronalen Netzes

## Grundidee des Neuronalen Netzes

Das Neuronale Netz besteht zunächst aus mehreren Schichten, wobei jede Schicht eine bestimmte Anzahl an Neuronen besitzt. Alle Neuronen einer Schicht sind dabei mit allen Neuronen der vorhergehenden und der nachfolgenden Schicht verbunden.

Die erste Schicht wird dabei als Eingabeschicht und die letzte Schicht dabei als Ausgabeschicht bezeichnet. Die Schichten dazwischen werden als versteckte Schichten bezeichnet.

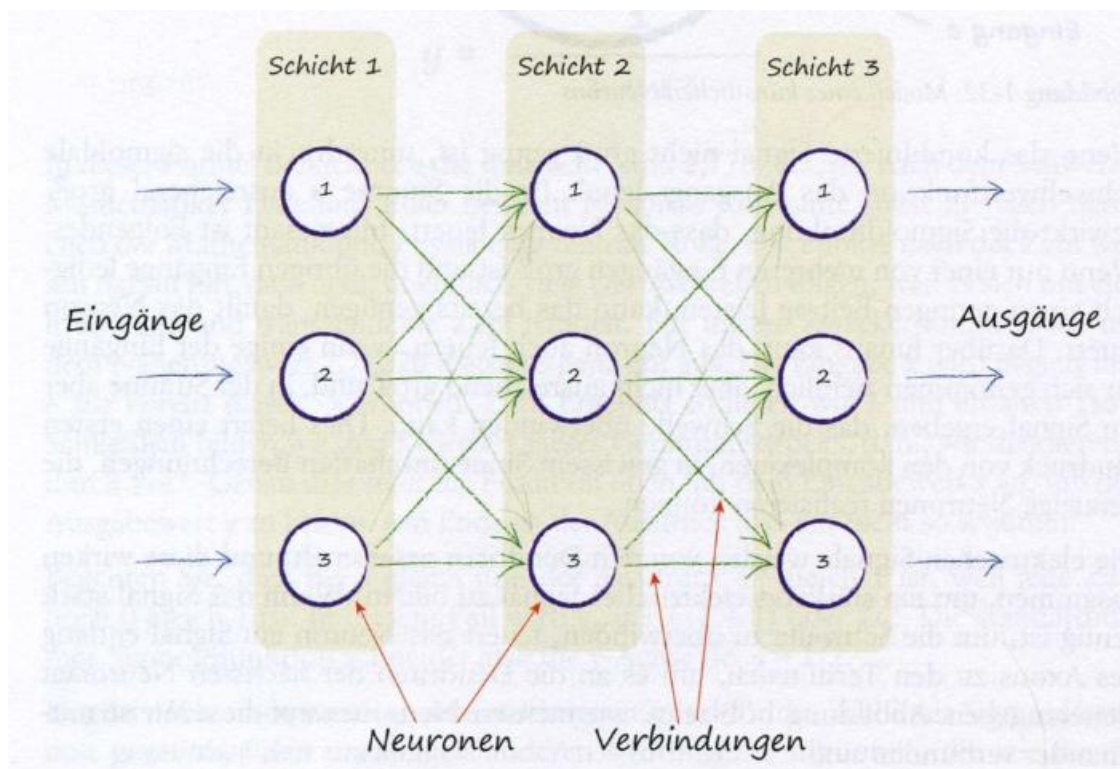


Abbildung 1: Aufbau eines neuronalen Netzes [1]

Die Eingabeschicht erhält nun zunächst Startwerte. In unserem Fall sind es die 784 (28\*28) Pixelwerte eines Bildes. Jeder dieser Werte wird nun an jede Neuronen der nächsten Schicht weitergeleitet. Anzumerken dabei ist, dass jede Verbindung eine Gewichtung hat mit dem der entsprechende Werte multipliziert wird, das heißt, dass die Neuronen entsprechend die gewichtete Summe der vorherigen Neuronen erhalten. Anschließend wird auf diese Summe ein Schwellenwert-Funktion, wie beispielsweise die Sigmoid-Funktion ( $y = \frac{1}{1+e^{-x}}$  mit  $x \triangleq \text{gewichtete Summe}$ ), um die neuen Output-Werte dieser Schicht zu berechnen. Dies wird nun so lange wiederholt, bis man die Output-Werte der Ausgabeschicht berechnet hat.

## Berechnung der gewichteten Summe mittels Matrizenmultiplikation

Das Berechnen der oben genannten gewichteten Summen lässt sich als Matrizenmultiplikation darstellen:

$$\begin{pmatrix} w_{1,1} & \cdots & w_{n,1} \\ \vdots & \ddots & \vdots \\ w_{1,n} & \cdots & w_{n,n} \end{pmatrix} * \begin{pmatrix} \text{output}_1 \\ \cdots \\ \text{output}_n \end{pmatrix} = \begin{pmatrix} \text{input}_1 \\ \cdots \\ \text{input}_n \end{pmatrix}$$

## Fehlerberechnung & Backpropagierung

In unserem Beispiel müsste die Ausgabeschicht im besten Falle 9-mal den Wert 0 und einmal den Wert 1 (entsprechend die Neurone deren Position gleich dem Label ist) ausgeben. Da dies unrealistisch ist, müssen die Ausgabefehler berechnet werden als die Differenz zwischen der Sollausgabe  $t_n$  und der tatsächlichen Ausgabe  $o_n$ :  $e_n = (t_n - o_n)$

Diese Fehler müssen im Folgendem auf die versteckten Schichten und die Eingabeschicht zurückgeführt werden. Dabei ist zu beachten, dass die Fehler anteilig der Gewichtung aufgeteilt werden. Dies kann ebenfalls als Matrizenmultiplikation dargestellt werden:

$$error_{Vorgänger} = \begin{pmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,n} \end{pmatrix} * \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}$$

Dabei ist anzumerken, dass die Gewichtsmatrix transponiert wird, um die Fehler der Vorgängerschicht zu berechnen. Diesen Vorgang wiederholen wir jetzt so lange, bis wir den Fehler der Eingabeschicht berechnet haben.

## Aktualisierung der Gewichte

Im Folgendem ist noch zu klären, wie die Gewichte mit den berechneten Fehlern aktualisiert werden. Mittels Gradientenverfahren erhalten wir für folgende Gleichung für die Gewichtsaktualisierung:

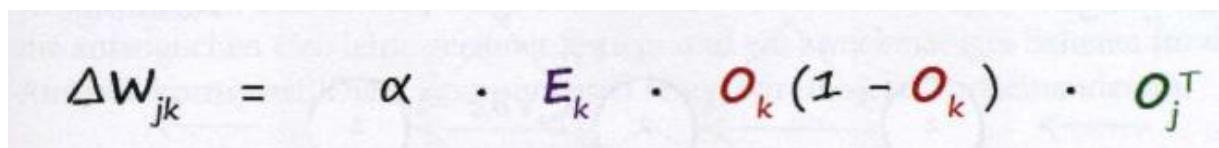

$$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k (1 - O_k) \cdot O_j^T$$

Abbildung 2: Matrixform der Gewichtsaktualisierung

$\Delta w_{jk}$	$\triangleq$	Änderung der Gewichtsmatrix zwischen der vorhergehenden Schicht $j$ und der nachfolgenden Schicht $k$
$\alpha$	$\triangleq$	Modifizierung der Änderung (Lernrate)
$E_k$	$\triangleq$	Fehler der nachfolgenden Schicht $k$
$O_k$	$\triangleq$	Werte der nachfolgenden Schicht $k$
$O_j^T$	$\triangleq$	Werte der vorhergehenden Schicht $j$ (transponiert)

Die genaue Herleitung ist in diesem Fall weggelassen wurden, um das Dokument nicht zu lang zu machen.

## Implementierung – Zwischenstand 18.01.2024

Im Folgendem erkläre ich, welche der obigen Punkte ich bisher implementiert habe und wie ich dies getan habe. Dabei nenne & erkläre ich nur die Funktionen, anstatt den kompletten Code als Text einzufügen, damit das Dokument nicht zu lang wird. Weiterhin verlinke ich zu jedem Punkt die Datei im meinem [Repo](#), auf die ich mich beziehe.

### Einlesen der Bilder

Für die Bilder wurde eine Klasse *Image* erstellt, welche in der Datei [images.py](#) gespeichert ist.

Als Attribute besitzt sie 28x28 (784) Pixel des Bildes (*pixels*) sowie das Label, also die eigentliche Zahl (*actual\_number*). Der Datentyp der Pixel ist ein eindimensionales Array aus Floats. Dabei anzumerken ist, dass sie in der IDX-Datei ganzzahlige Werte im Intervall [0; 255] annehmen. Da das neuronale Netz jedoch Floats im Intervall [0; 1] ausgibt, wurden die Pixelwerte mit 255 dividiert, sodass sie auch Werte im Intervall [0; 255] annehmen. Der Datentyp des Labels ist ein Integer.

Die Klasse verfügt über Getter der beiden Attribute. Weiterhin verfügt sie über zwei statische Methoden (*read\_image\_pixels\_from\_idx* & *read\_image\_labels\_from\_idx*), die die Labels und die Pixels aus den IDX-Dateien auszulesen und als Array des entsprechenden Datentyps zurückzugeben. Die letzteren beiden Methoden werden dabei von einer anderen statischen Methode aufgerufen, die die Labels und Pixels dann in einer CSV-Datei zusammen speichert (*save\_image\_bytes\_and\_labels*). Dabei ist dann anzumerken, dass jede Zeile in der CSV-Datei einem Bild entspricht. Abschließend besitzt die Klasse noch eine statische Methode, mit welcher *Image*-Objekte aus einer CSV-Datei erstellt werden können. (*create\_images\_from\_csv*)

Zudem existiert ein Skript namens [convert\\_idx\\_to\\_csv.py](#), welches die IDX-Dateien einliest die Bildinformationen als CSV-Dateien einspeichert. Die CSV-Dateien mit den Bildinformationen befinden sich nicht im Repo, da sie zu groß sind. (Testdaten 58,2 MB, Trainingsdaten: 348 MB)

## Gewichtsmatrizen

Zunächst existiert eine Klasse *Matrix*, welche in der Datei [matrix.py](#) gespeichert ist.

Als Attribut besitzt sie die Werte der Matrix, welche als ein zweidimensionales Array aus Floats repräsentiert wird (*values*).

Für das Attribut ist ein Getter und ein Setter vorhanden, sowie einer Methode, die einen bestimmten Wert innerhalb der Matrix ausgibt. Zudem gibt es noch Methoden, die die Größe der Matrix ausgeben können (*get\_number\_of\_rows* & *get\_number\_of\_columns*). Weiterhin verfügt die Klasse über eine Methode, die überprüft, ob alle Reihen gleich groß sind (*matrix\_all\_rows\_same\_length*). Zudem verfügt sie über eine Methode, die überprüft, ob die übergebenen Werte einer Matrix entsprechen, das heißt ob ein zweidimensionales Array aus Floats sind. Die beiden letzteren Methoden werden beim Aufrufen des Konstruktors mit aufgerufen und werfen bei einem Fehler eine Exception. Des Weiteren verfügt die Klasse über eine Methode, die die inverse Matrix erstellt (*inverse*). Abschließend verfügt die Klasse noch über eine statische Funktion, welche zwei Matrizen miteinander multipliziert und das Ergebnis als neues Matrix-Objekt zurückgibt.

Weiterhin gibt es eine Klasse *WeightMatrix*, welche von der Klasse *Matrix* erbt und in der Datei [weight\\_matrix.py](#) gespeichert ist.

Diese Kindklasse besitzt keine zusätzlichen Attribute.

Die Klasse besitzt jedoch zwei zusätzliche Methoden. Zum einen besitzt sie eine statische Methode, mit welcher man Gewichtsmatrizen in eine CSV-Datei schreiben kann (*write\_weights*). Dabei ist jede Zeile in der CSV-Datei eine *WeightMatrix*-Objekt. Zudem gibt es dann noch eine Methode, mit welcher *WeightMatrix*-Objekte aus einer CSV-Datei erstellt werden können (*create\_weights\_from\_csv*).

Abschließend existiert noch ein Skript [initialize\\_weight\\_matrices.py](#), welches zwei Gewichtsmatrizen mit zufälligen Werten im Intervall [0; 1] erstellt. Sollten bereits Gewichtsmatrizen existieren, wird der Benutzer gefragt, ob diese überschrieben werden sollen. Ist dem nicht der Fall, wird das Skript beendet.

## Zusätzliche Funktionalitäten

Es existiert ein Skript [transfer\\_functions.py](#), in welchem verschiedene Transferfunktionen gespeichert werden sollen, die das neuronale Netz nutzen kann. Zum aktuellen Zeitpunkt ist jedoch nur die Sigmoid-Funktion implementiert.

## Noch fehlend

Zum aktuellen Zeitpunkt muss das neuronale Netz selbst implementiert werden, also die Eingabe der Startsignale, das Berechnen der gewichteten Summen und die Ausgabe der Werte in der Ausgabeschicht. Weiterhin muss dann noch die Fehlerfindung & die Backpropagierung implementiert werden.

Sobald dies implementiert wurde, muss das neuronale Netz nur noch trainiert werden, bis die Erfolgswerte den Ansprüchen (>90%) entsprechen.

Weiterhin ist zu überlegen, ob eine graphische Ausgabe der Bilder nicht sinnvoll wäre, also das die 28 mal 28 Pixel zusammen mit der dem Label und dem Ergebnis des neuronalen Netzes dargestellt werden, sodass es beispielsweise so aussehen könnte:

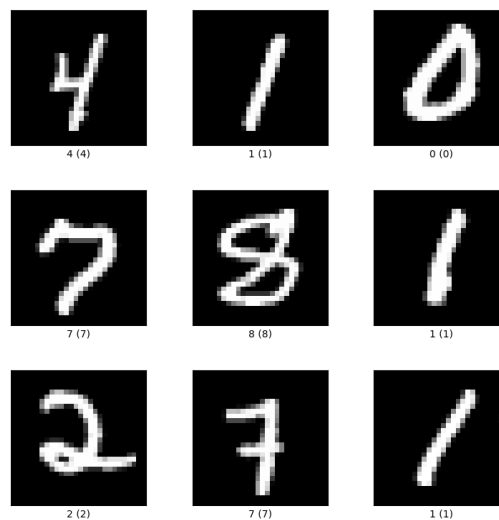


Abbildung 3: Idee für graphische Darstellung

## Quellen

- [1] Tariq Rashid, dpunkt (Hrsg.), *Neuronale Netze selbst programmieren – Ein verständlicher Einstieg mit Python* (2019), übersetzt von Frank Langenau, Seite 36
- [2] Tariq Rashid, dpunkt (Hrsg.), *Neuronale Netze selbst programmieren – Ein verständlicher Einstieg mit Python* (2019), übersetzt von Frank Langenau, Seite 87
- [3] <https://storage.googleapis.com/tfds-data/visualization/fig/mnist-3.0.1.png> (15.01.2024, 17:30)