

Entwicklung eines eigenen neuronalen Netzes.

Fedor Romanov, MTS-B31

Januar 2025

# Contents

<b>1</b>	<b>Aufgabe</b>	<b>3</b>
<b>2</b>	<b>Wahl des Algorithmus</b>	<b>3</b>
<b>3</b>	<b>Computer-Merkmale</b>	<b>3</b>
<b>4</b>	<b>Funktionsprinzip des Programms mit mathematischen Erklärungen</b>	<b>3</b>
4.1	Einlesen der Daten . . . . .	3
4.2	Kurzbeschreibung der Methode . . . . .	4
<b>5</b>	<b>Unterschied zwischen ReLU und Sigmoid</b>	<b>6</b>
5.1	Vergleich von ReLU und Sigmoid . . . . .	7
<b>6</b>	<b>Modellprüfung</b>	<b>7</b>
6.1	Trainingsprozess mit ReLU Aktivierungsfunktion . . . . .	7
6.2	Trainingsprozess mit Sigmoid Aktivierungsfunktion . . . . .	8
6.3	Vergleich von Sigmoid- und ReLU-Funktionen . . . . .	10
<b>7</b>	<b>Auswahl der Ausgangsbedingungen.</b>	<b>11</b>
<b>8</b>	<b>Benutzerhandbuch</b>	<b>12</b>
<b>9</b>	<b>Interessante Beispiele</b>	<b>12</b>
9.1	Erfolgreiche komplizierte Beispiele . . . . .	12
9.2	Fehlende Beispiele . . . . .	13

# 1 Aufgabe

Erstellung eines eigenen Faltungsneuronalen Netzes (CNN) zur Erkennung von Ziffern aus der MNIST-Projektdateiabank ohne Verwendung von Ressourcen und Bibliotheken Dritter. Erläuterung der Prinzipien der Funktionsweise des Modells, Speicherung der Arbeitsergebnisse und Möglichkeiten der weiteren Nutzung.

## 2 Wahl des Algorithmus

Wir werden CNNs verwenden, um dieses Problem zu lösen. Faltungsneuronalen Netze (CNNs) sind aus folgenden Gründen besonders effektiv für die Ziffernerkennung.

- Verarbeitung hochdimensionaler Daten. Bilder (z. B. 28x28 Pixel für MNIST) sind hochdimensionale Daten. CNNs können solche Daten effektiv verarbeiten, indem sie den Lernprozess in Faltungs- und Verschmelzungsoperationen aufteilen und so die Dimensionalität reduzieren, ohne dass wichtige Informationen verloren gehen.
- Verallgemeinerung. Aufgrund ihrer Architektur lassen sich CNNs gut über verschiedene Datensätze hinweg verallgemeinern, selbst wenn sie mit einer begrenzten Datenmenge trainiert wurden.
- Effektive Merkmalsextraktion. CNNs lernen automatisch relevante Merkmale wie Kanten, Formen und Texturen, die für die Unterscheidung zwischen verschiedenen Ziffern wichtig sind.

## 3 Computer-Merkmale

Name	MacBook Pro 13" 2018
CPU	Intel Core i5 (4 Kerne, 8 Threads) mit 2,3 GHz (Turbo Boost bis zu 3,8 GHz)
RAM	8 GB LPDDR3 (2133 MHz Frequenz)
SSD	256 GB
GPU	Intel Iris Plus Graphics 655

Table 1: Computer-Merkmale

Unter diesen Merkmalen sind die Informationen über den Prozessor am wichtigsten. Multitasking-Aufgaben, einschließlich Berechnungen, werden mit vier Kernen schneller ausgeführt. Je höher die Prozessorfrequenz, desto schneller können die einzelnen Schritte von Algorithmen ausgeführt werden.

Mehr Arbeitsspeicher ermöglicht es dem Computer, große Datenmengen zu verarbeiten, wodurch eine Verlangsamung durch die Nutzung des virtuellen Speichers vermieden wird. Für einfache Berechnungen wie die Suche nach einer Primzahl sind 8 GB jedoch mehr als ausreichend, da die Datenmenge gering ist.

## 4 Funktionsprinzip des Programms mit mathematischen Erklärungen

### 4.1 Einlesen der Daten

Die Eingabedaten bestehen aus vier Files. Zwei Bilddateien, von denen eine Datei mit 60000 Dateien zum Trainieren des Modells und die andere mit 10000 Dateien zum Testen verwendet wird. Außerdem zwei Dateien mit 60000 bzw. 10000 Labels, die zur Überprüfung der tatsächlichen Zahl im Bild verwendet werden.

Zunächst brauchen wir eine Methode, um die Bilddaten und Beschriftungen aus den Dateien zu lesen. Die ubyte-Dateien haben eine spezielle Struktur, die das Auslesen der Daten erleichtert. Zuerst kommt der MagicNumber-Parameter, der dafür verantwortlich ist, welche Art von Datei wir vor uns haben - Bilder oder Etiketten, dann kommen die Parameter, die für die Anzahl und Größe der Bilder verantwortlich sind, und dann die Bilddaten selbst im Format eines Arrays von

Zahlen mit Werten von 0 bis 255, abhängig von der Grauschattierung jedes Pixels. Der Einfachheit halber werden wir die Zahldaten in ein Array zwischen 0 und 1 schreiben und den Wert durch 255 teilen.

Für die Labels verwenden wir die "one-hot encoded representation". Das bedeutet, dass die Beschriftung einer Zahl als Array mit 1 an der Stelle, die dieser Zahl entspricht, und 0 an den übrigen Stellen gesetzt wird. Dies ist besonders praktisch bei der Berechnung des Losses.

## 4.2 Kurzbeschreibung der Methode

Unser neuronales Netz wird aus den folgenden Elementen bestehen: Eine Eingabeschicht, die mit einem Bild in Form einer  $1 \times 724 (28 \times 28)$ -Matrix gefüttert wird, dann eine innere Schicht (eine oder mehrere) und eine Ausgabeschicht von  $1 \times 10$ , in der die Wahrscheinlichkeiten, welchen Wert die in der Eingabe angegebene Zahl annimmt, aufgezeichnet werden. Jedes der Elemente der Eingabeschicht wird mit jedem Element der ersten inneren Schicht durch Gewichte verbunden. Die inneren Schichten werden ebenfalls durch Gewichte miteinander verbunden und die letzte innere Schicht mit der Ausgabeschicht. Außerdem wird jedes Element der inneren Schicht mit einer Vorspannung ("bias") versehen.

Aus offenen Quellen ist bekannt, dass neuronale Netze mit mehreren internen Schichten besser abschneiden als Netze mit einer. Um die Aufgabe zu vereinfachen, setzen wir die inneren Schichten auf zwei Schichten der Größe  $1 \times 16$ . Somit beträgt die Gesamtzahl der Gewichte und Abweichungen für unser Netz etwa 13000.

Unsere Code implementiert ein einfaches Convolutional Neural Network (CNN) zur Erkennung von Ziffern. Im Folgenden Text (der ist aus offenen Quellen übernommen) wird Schritt für Schritt erklärt, wie das Netzwerk die Daten verarbeitet, sich selbst trainiert und Vorhersagen generiert, unterstützt durch relevante mathematische Formulierungen.

### Schritt 1: Initialisierung von Gewichten und Biases

Die Gewichte  $W$  und Biases  $b$  für alle Schichten werden zufällig wie folgt initialisiert:

- Die Gewichte werden mit kleinen Zufallswerten aus einer Normalverteilung mit Mittelwert 0 und einer kleinen Standardabweichung (z.B. 0,01) initialisiert:

$$W_{ij} \sim \mathcal{N}(0, 0.01). \quad (1)$$

- Die Biases werden auf 0 gesetzt:

$$b_i = 0. \quad (2)$$

**Schritt 2: Forward Propagation** Im Forward-Pass werden die Eingaben durch die Schichten des Netzwerks geleitet, und die Aktivierungen werden berechnet. Die Schritte sind wie folgt:

**2.1: Lineare Transformation** Für jede Schicht  $l$  wird die Ausgabe berechnet als:

$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}, \quad (3)$$

wobei:

- $W^{(l)}$  die Gewichtsmatrix für Schicht  $l$  ist.
- $a^{(l-1)}$  die Aktivierung aus der vorherigen Schicht (oder die Eingabe für die erste Schicht) ist.
- $b^{(l)}$  der Bias-Vektor für Schicht  $l$  ist.

**2.2: Aktivierungsfunktionen** Das Netzwerk wird mit zwei verschiedenen Aktivierungsfunktionen getestet: ReLU und Sigmoid. Jede dieser Funktionen hat ihre spezifischen Eigenschaften und Auswirkungen auf das Training und die Konvergenzgeschwindigkeit.

**ReLU-Aktivierungsfunktion** Die ReLU (Rectified Linear Unit)-Aktivierungsfunktion ist definiert als:

$$\text{ReLU}(x) = \max(0, x). \quad (4)$$

Diese Funktion wird elementweise auf  $z^{(l)}$  angewendet, um die Aktivierungen  $a^{(l)}$  zu erhalten:

$$a^{(l)} = \text{ReLU}(z^{(l)}). \quad (5)$$

ReLU hat den Vorteil, dass sie sparsames Aktivieren fördert (nur positive Werte bleiben erhalten) und keine Sättigungsprobleme wie die Sigmoid-Funktion aufweist, was zu einer schnelleren Konvergenz führt.

**Sigmoid-Aktivierungsfunktion** Die Sigmoid-Aktivierungsfunktion ist definiert als:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}. \quad (6)$$

Diese Funktion transformiert jeden Eingangswert  $x$  in einen Bereich zwischen 0 und 1, wodurch sie häufig für Wahrscheinlichkeitsvorhersagen genutzt wird. Die Aktivierungen  $a^{(l)}$  werden durch folgende Gleichung berechnet:

$$a^{(l)} = \text{Sigmoid}(z^{(l)}). \quad (7)$$

**Eigenschaften der Sigmoid-Funktion:**

- **Sättigung:** Für sehr große oder sehr kleine Werte von  $x$  nähert sich die Ausgabe der Sigmoid-Funktion asymptotisch 1 oder 0. Dies kann während des Backpropagationsprozesses zu einem sogenannten "Vanishing Gradient"-Problem führen.
- **Nichtlinearität:** Die Sigmoid-Funktion führt Nichtlinearität in das Netzwerk ein, was es ermöglicht, komplexe Muster zu lernen.
- **Differenzierbarkeit:** Die Ableitung der Sigmoid-Funktion ist einfach zu berechnen und ergibt:

$$\text{Sigmoid}'(x) = \text{Sigmoid}(x) \cdot (1 - \text{Sigmoid}(x)). \quad (8)$$

Im Rahmen dieses Experiments werden beide Aktivierungsfunktionen, ReLU und Sigmoid, verwendet, um die jeweiligen Auswirkungen auf das Training und die Leistung des Modells zu untersuchen. Besonderes Augenmerk wird dabei auf die Konvergenzgeschwindigkeit und die endgültige Genauigkeit gelegt.

**2.3: Ausgabeschicht mit Softmax** Die Ausgabeschicht verwendet die Softmax-Funktion, um Wahrscheinlichkeiten für jede Klasse zu berechnen:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}. \quad (9)$$

**Schritt 3: Verlustfunktion** Die Kreuzentropie-Verlustfunktion wird verwendet, um den Unterschied zwischen den vorhergesagten Wahrscheinlichkeiten und den wahren Labels zu quantifizieren:

$$\mathcal{L} = - \sum_i y_i \log(\hat{y}_i), \quad (10)$$

wobei:

- $y_i$  das wahre Label (one-hot codiert) ist.
- $\hat{y}_i$  die vorhergesagte Wahrscheinlichkeit für Klasse  $i$  ist.

**Schritt 4: Backpropagation** Die Gradienten des Verlusts in Bezug auf die Gewichte und Biases werden mit der Kettenregel berechnet. Für Schicht  $l$  werden die folgenden Schritte ausgeführt:

**4.1: Gradienten der Ausgabeschicht** Der Fehler in der Ausgabeschicht wird berechnet als:

$$\delta^{(L)} = \hat{y} - y. \quad (11)$$

**4.2: Gradienten der versteckten Schichten** Für eine versteckte Schicht  $l$  wird der Fehler rückwärts propagiert. Dabei hängt die Berechnung des Gradienten von der gewählten Aktivierungsfunktion ab.

**ReLU-Aktivierung:** Bei der ReLU-Aktivierungsfunktion ergibt sich der Fehler wie folgt:

$$\delta^{(l)} = \left( W^{(l+1)T} \delta^{(l+1)} \right) \odot \text{ReLU}'(z^{(l)}), \quad (12)$$

wobei  $\text{ReLU}'(z^{(l)})$  die Ableitung der ReLU-Aktivierungsfunktion ist:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{falls } x > 0, \\ 0 & \text{falls } x \leq 0. \end{cases} \quad (13)$$

ReLU ist besonders vorteilhaft, da sie keine Sättigung aufweist und somit die Gradienten während des Trainings erhalten bleiben.

**Sigmoid-Aktivierung:** Bei der Sigmoid-Aktivierungsfunktion wird der Fehler wie folgt berechnet:

$$\delta^{(l)} = \left( W^{(l+1)T} \delta^{(l+1)} \right) \odot \text{Sigmoid}'(z^{(l)}), \quad (14)$$

wobei  $\text{Sigmoid}'(z^{(l)})$  die Ableitung der Sigmoid-Aktivierungsfunktion ist:

$$\text{Sigmoid}'(x) = \text{Sigmoid}(x) \cdot (1 - \text{Sigmoid}(x)). \quad (15)$$

**Eigenschaften der Sigmoid-Ableitung:**

- Die Ableitung der Sigmoid-Funktion ist maximal bei  $x = 0$  (entspricht  $\frac{1}{4}$ ) und nimmt bei extremen Werten von  $x$  gegen 0 ab.
- Dies führt dazu, dass die Gradienten in tiefen Schichten oft verschwinden (Vanishing Gradient Problem), was das Training erschwert.

**Vergleich:** Während die ReLU-Funktion in der Regel eine schnellere Konvergenz und stabilere Gradienten während des Trainings ermöglicht, bietet die Sigmoid-Funktion eine glatte Ausgabe, die für bestimmte Aufgaben, wie Wahrscheinlichkeitsmodellierung, nützlich ist. Im Rahmen dieses Experiments werden beide Aktivierungsfunktionen verwendet, um ihre Auswirkungen auf die Gradientenberechnung und das Training zu analysieren.

**4.3: Aktualisierung von Gewichten und Biases** Die Gradienten der Gewichte und Biases werden berechnet als:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \cdot a^{(l-1)T}, \quad (16)$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}. \quad (17)$$

Die Gewichte und Biases werden mit Stochastic Gradient Descent (SGD) aktualisiert:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad (18)$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}, \quad (19)$$

wobei  $\eta$  die Lernrate ist.

**Schritt 5: Training des Modells** Der Trainingsprozess umfasst:

1. Zufälliges Mischen der Trainingsdaten.
2. Aufteilen der Daten in Mini-Batches der Größe  $B$ .
3. Durchführen von Forward- und Backpropagation für jedes Mini-Batch.
4. Aktualisierung der Gewichte und Biases mit den akkumulierten Gradienten.

**Schritt 6: Evaluation** Nach dem Training wird das Modell mit Testdaten evaluiert. Die Genauigkeit wird berechnet als:

$$\text{Genauigkeit} = \frac{\text{Anzahl der korrekten Vorhersagen}}{\text{Gesamtanzahl der Testdaten}} \times 100\%. \quad (20)$$

## 5 Unterschied zwischen ReLU und Sigmoid

### ReLU (Rectified Linear Unit)

Die ReLU-Aktivierungsfunktion ist definiert als:

$$\text{ReLU}(x) = \max(0, x). \quad (21)$$

ReLU setzt alle negativen Werte auf 0 und lässt positive Werte unverändert. Ihre Eigenschaften sind:

- Einfachheit der Berechnung.
- Vermeidung des "Vanishing Gradient"-Problems, da die Ableitung für  $x > 0$  konstant 1 ist.
- Schnelle Konvergenz während des Trainings.

## Sigmoid

Die Sigmoid-Aktivierungsfunktion ist definiert als:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}. \quad (22)$$

Die Sigmoid-Funktion gibt Werte im Bereich  $[0, 1]$  aus und eignet sich gut für Wahrscheinlichkeiten. Ihre Eigenschaften sind:

- Glatte Kurve, die sich gut für Wahrscheinlichkeitsinterpretationen eignet.
- Neigung zum "Vanishing Gradient"-Problem, da die Ableitung bei extremen Eingabewerten ( $x \gg 0$  oder  $x \ll 0$ ) gegen 0 geht:

$$\text{Sigmoid}'(x) = \text{Sigmoid}(x) \cdot (1 - \text{Sigmoid}(x)). \quad (23)$$

### 5.1 Vergleich von ReLU und Sigmoid

Eigenschaft	ReLU	Sigmoid
Aktivierungsbereich	$[0, \infty)$	$[0, 1]$
Ableitung	Konstant für $x > 0$ (1)	Werte nahe 0 für große oder kleine $x$
Vanishing Gradient	Vermeidet das Problem	Anfällig für das Problem
Berechnungskomplexität	Sehr einfach	Aufwändiger (exponentielle Funktion)

Table 2: Vergleich von ReLU und Sigmoid

## 6 Modellprüfung

Um zu testen, wie unser Modell funktioniert, werden wir die verschiedenen Parameter vergleichen und prüfen, wie sie sich auf das Endergebnis auswirken. Wir werden 50 Epochen Training verwenden und die Parameter dafür ändern: Aktivierungsfunktion (Sigmoid, ReLU), Stichprobengröße für das Training (batch-size 10,50,100) und Lernrate (learning-rate 0.01,0.025,0.1).

### 6.1 Trainingsprozess mit ReLU Aktivierungsfunktion

Beim Training eines Modells mit einer Ausgangsfunktion. Die folgenden, in den Diagrammen dargestellten Ergebnisse wurden erzielt. Die Diagramme zeigen das Verhältnis (Epoche)/(Erfolgsrate) für verschiedene Werte der Learning-Rate und Batch-Size.

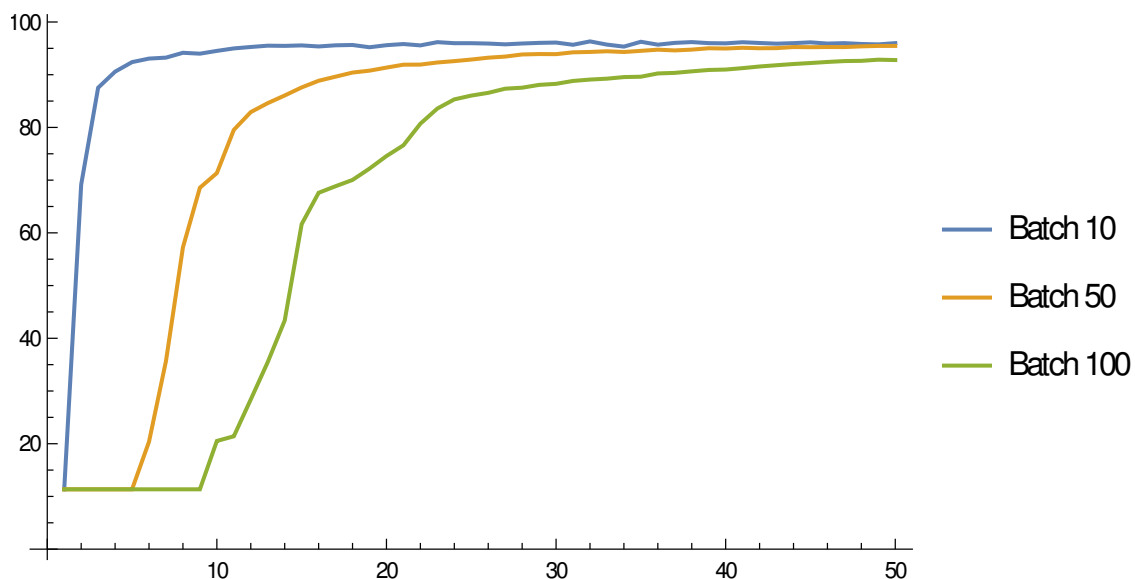


Figure 1: Aktivierungsfunktion ReLU, Learning-Rate 0.01

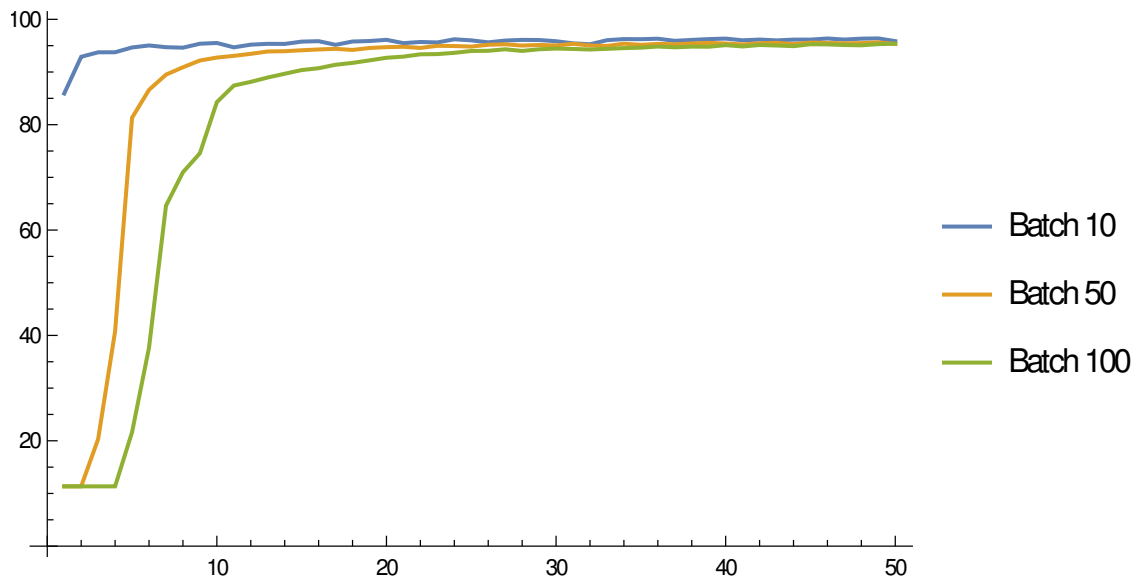


Figure 2: Aktivierungsfunktion ReLu, Learning-Rate 0.025

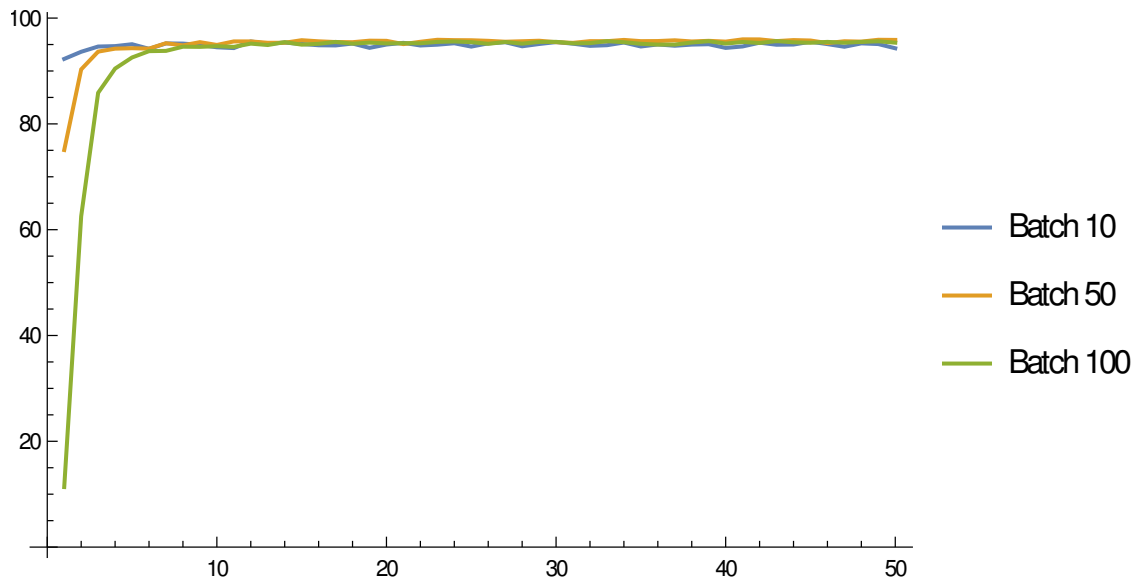


Figure 3: Aktivierungsfunktion ReLu, Learning-Rate 0.1

Man kann folgende Schlussfolgerung ziehen. Unabhängig von den Anfangsparametern liegt die durchschnittliche Erkennungsrate bei 95%. Die schnellsten Spitzenwerte werden bei kleinen Werten erreicht. Dies ist höchstwahrscheinlich auf die Tatsache zurückzuführen, dass die Gradientenaktualisierungen bei einem kleinen Stapel eher "verrauscht" sind, da sie aus weniger Stichproben berechnet werden. Dieses "Rauschen" kann dem Modell helfen, lokale Minima oder Plateaus zu Beginn des Trainingsprozesses zu vermeiden, was zu einem schnelleren Anfangslernen führt. Allerdings ist die Streuung der Werte in späteren Phasen bei einem hohen Batch-Wert geringer.

## 6.2 Trainingsprozess mit Sigmoid Aktivierungsfunktion

Beim Training eines Modells mit einer Ausgangsfunktion. Die folgenden, in den Diagrammen dargestellten Ergebnisse wurden erzielt. Die Diagramme zeigen das Verhältnis (Epoche)/(Erfolgsrate) für verschiedene Werte der Lerning-Rate und Batch-Size.

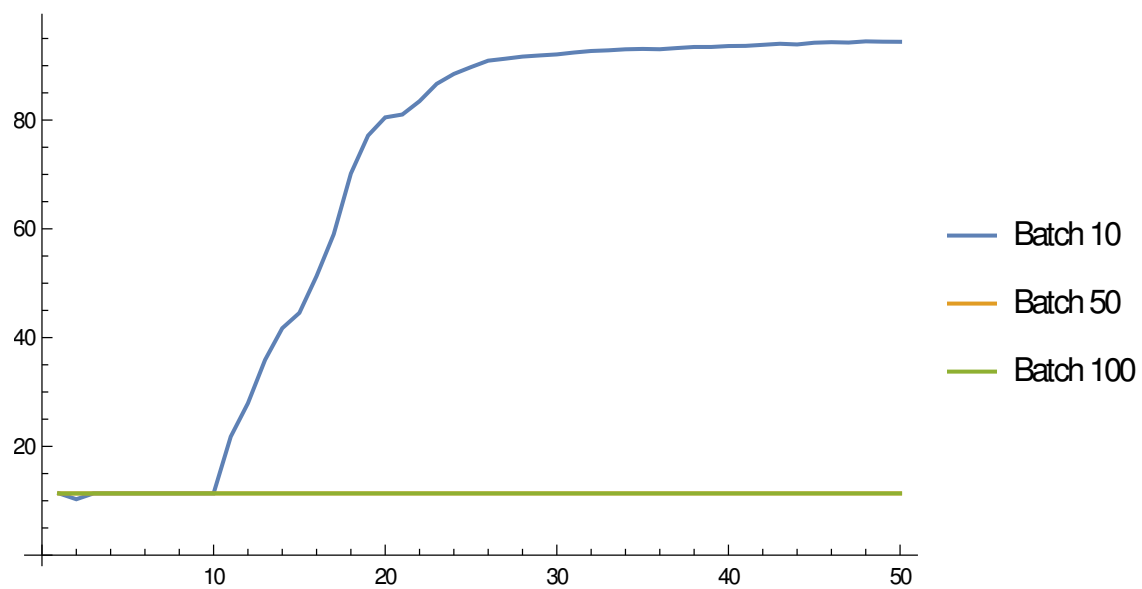


Figure 4: Aktivierungsfunktion Sigmoid, Learning-Rate 0.01

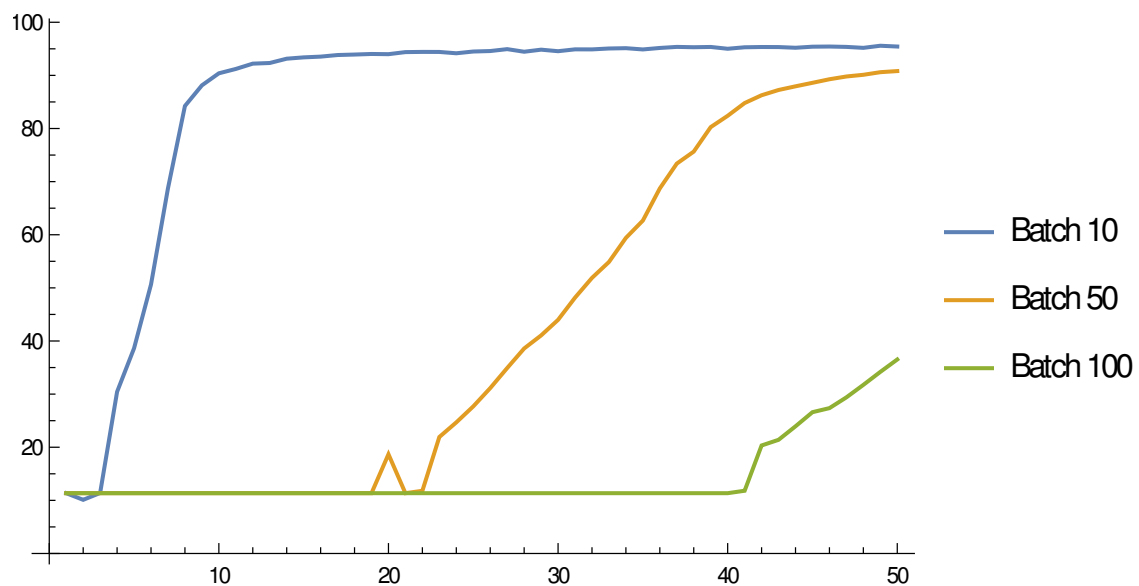


Figure 5: Aktivierungsfunktion Sigmoid, Learning-Rate 0.025

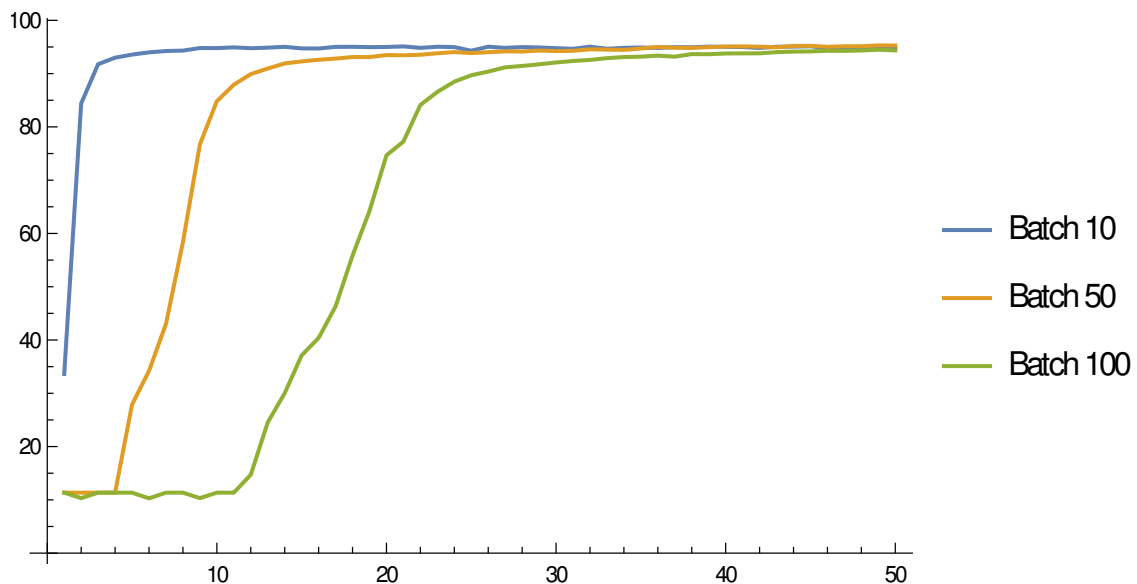


Figure 6: Aktivierungsfunktion Sigmoid, Learning-Rate 0.1

Man kann folgende Schlussfolgerung ziehen. Bei großen Batch-Size-Werten beginnt die Funktion sehr langsam zu lernen, weil die sigmoidale Aktivierungsfunktion Werte im Bereich  $[0,1]$  ausgibt und ihr Gradient sehr klein wird, wenn die Eingabewerte groß sind (positiv oder negativ). Infolgedessen werden während der Backpropagation die Gradienten für die Gewichte und Offsets sehr klein, was die Aktualisierung der Gewichte verlangsamt oder ganz stoppt. Auch eine Lernrate von 0,01-0,025 kann in Kombination mit einer größeren Stapelgröße zu klein sein. Größere Stapelgrößen berechnen die Gradienten genauer, sind aber weniger empfindlich gegenüber verrauschten Aktualisierungen, was die Konvergenz verzögern könnte, wenn die Lernrate nicht ausreichend hoch ist.

### 6.3 Vergleich von Sigmoid- und ReLU-Funktionen

Am Ende wäre es interessant, die beiden Aktivierungsfunktionen Sigmoid und ReLU zu vergleichen. Wir nehmen eine größere Anzahl von Epochen und verdoppeln auch die versteckten Schichten, um die Erkennungsgenauigkeit aufgrund von mehr Gewichten und Abweichungen zu verbessern. Die Anfangsparameter lauten wie folgt:

```
// Parameter von Netzwerk
private static final int INPUT_SIZE = 784;
private static final int HIDDEN_SIZE_1 = 32;
private static final int HIDDEN_SIZE_2 = 32;
private static final int OUTPUT_SIZE = 10;
// Parameter von Lernprozess
private static final double LEARNING_RATE = 0.025;
private static final int EPOCHS = 100;
private static final int BATCH_SIZE = 50;
```

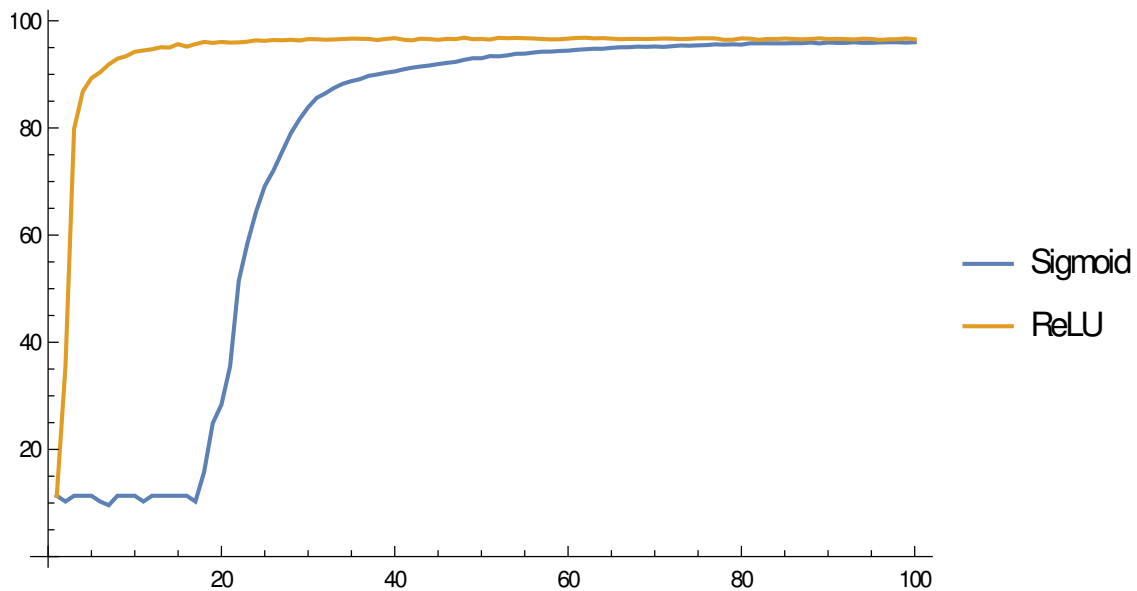


Figure 7: Vergleich von Sigmoid- und ReLU-Funktionen mit gleicher Anfangsparameter

Die Grafik zeigt, dass ReLU im Nahbereich schneller ist, aber im Fernbereich holt Sigmoid gegenüber ReLU auf. Auch bei der Entfernung hat Sigmoid kleinere Abweichungen zwischen den Epochen. Die Endwerte der erfolgreichen Zahlenerkennung für die Funktionen sind wie folgt:

- ReLU-Funktion 96,54%
- Sigmoid-Funktion 96,01%

Daraus können wir schließen, dass der endgültige Erfolgswert wahrscheinlich eher von der Anzahl der Neuronen im Netz als von der Aktivierungsfunktion abhängt. Allerdings ist das frühe Lernen mit der ReLU-Funktion schneller.

## 7 Auswahl der Ausgangsbedingungen.

Meiner Meinung nach und auf der Grundlage der durchgeführten Tests ist der folgende Satz von Ausgangsbedingungen für dieses Problem optimal.

Die Struktur des neuronalen Netzes selbst: Zwei versteckte Schichten mit je 16 Neuronen reichen aus, um eine Genauigkeit von 95 % und mehr zu erreichen. Eine größere Anzahl von Neuronen in einer Schicht kann eine Steigerung von einigen Prozent bewirken, erhöht aber die Trainingsdauer und die Datenmenge erheblich.

Als Aktivierungsfunktion würde ich ReLU wählen, da sie unter allen Bedingungen stabil ist, die Obergrenze der Erkennung schneller erreicht und im Allgemeinen einfacher zu berechnen ist.

Für die Batch-size würde ich einen kleinen Wert zwischen 10 und 50 wählen, 10 ist meiner Meinung nach völlig ausreichend und vereinfacht auch die Berechnung.

Die Lernrate muss nicht zu klein sein, 0,1 sorgt für die nötige Geschwindigkeit und Genauigkeit und vor allem Stabilität. Aber für eine große Anzahl von Epochen ist es besser, einen kleineren Wert zu verwenden, zum Beispiel 0,025.

Die Anzahl der Epochen hängt von der Aktivierungsfunktion ab. Für ReLU sind 10-20 Epochen ausreichend, während für Sigmoid je nach Bedingungen mehr als 50 Epochen erforderlich sein können.

Der endgültige Satz von Bedingungen lautet wie folgt:

```
private static final int INPUT_SIZE = 784;
private static final int HIDDEN_SIZE_1 = 16;
private static final int HIDDEN_SIZE_2 = 16;
private static final int OUTPUT_SIZE = 10;

private static final double LEARNING_RATE = 0.1;
private static final int EPOCHS = 30;
private static final int BATCH_SIZE = 10;
```

```
// Activation function 1-Sigmoid 2-ReLU
private static final int ACTIVATION_FUNCTION = 2;
```

## 8 Benutzerhandbuch

Dieser Code besteht aus drei Klassen: "SimpleCNN" , "MNISTLoader" und "NeuralNetworkGUI".

Wenn Sie nur Bilder aus der Trainingsdatenbank testen wollen, führen Sie das Programm "NeuralNetworkGUI.java" aus (es öffnet sich mit einer Verzögerung aufgrund des Ladens von Daten aus der Gewichtsdatei). Es gibt ein zufälliges Bild aus dem Trainingssatz aus, sowie eine wahrscheinliche und wahre Bildbezeichnung. Die Gewichte und Abweichungen für diese Klasse werden aus der Datei "data.txt" entnommen, die sich im Unterordner befindet. Es hat eine Genauigkeit von 95,58%.

Wenn Sie neue Gewichte und Abweichungen berechnen möchten, müssen Sie das Programm "SimpleCNN.java" ausführen und die erforderlichen Parameter darin ändern. Beachten Sie, dass die Parameter des neuronalen Netzes für NeuralNetworkGUI und SimpleCNN übereinstimmen müssen, um Fehler beim Laden der Datei zu vermeiden.

```
// Parameter von Netzwerk
private static final int INPUT_SIZE = 784;
private static final int HIDDEN_SIZE_1 = 16;
private static final int HIDDEN_SIZE_2 = 16;
private static final int OUTPUT_SIZE = 10;
```

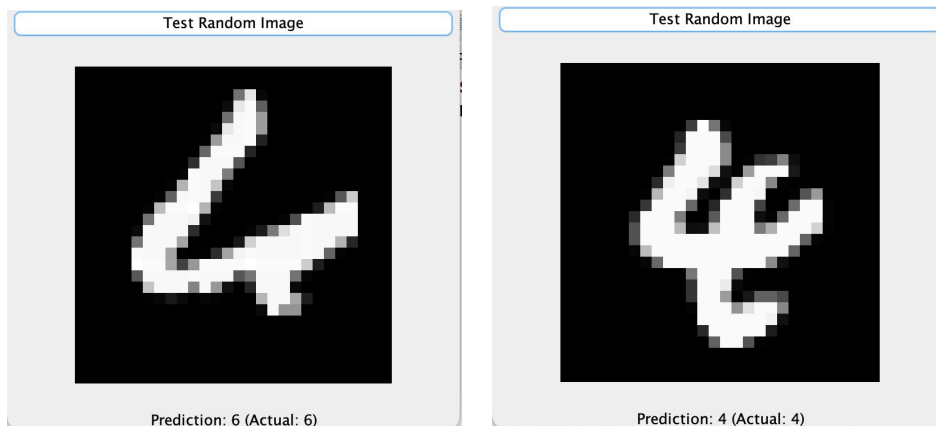
Das SimpleCNN-Programm gibt die Genauigkeit für jede Epoche sowie die endgültige Genauigkeit aus.

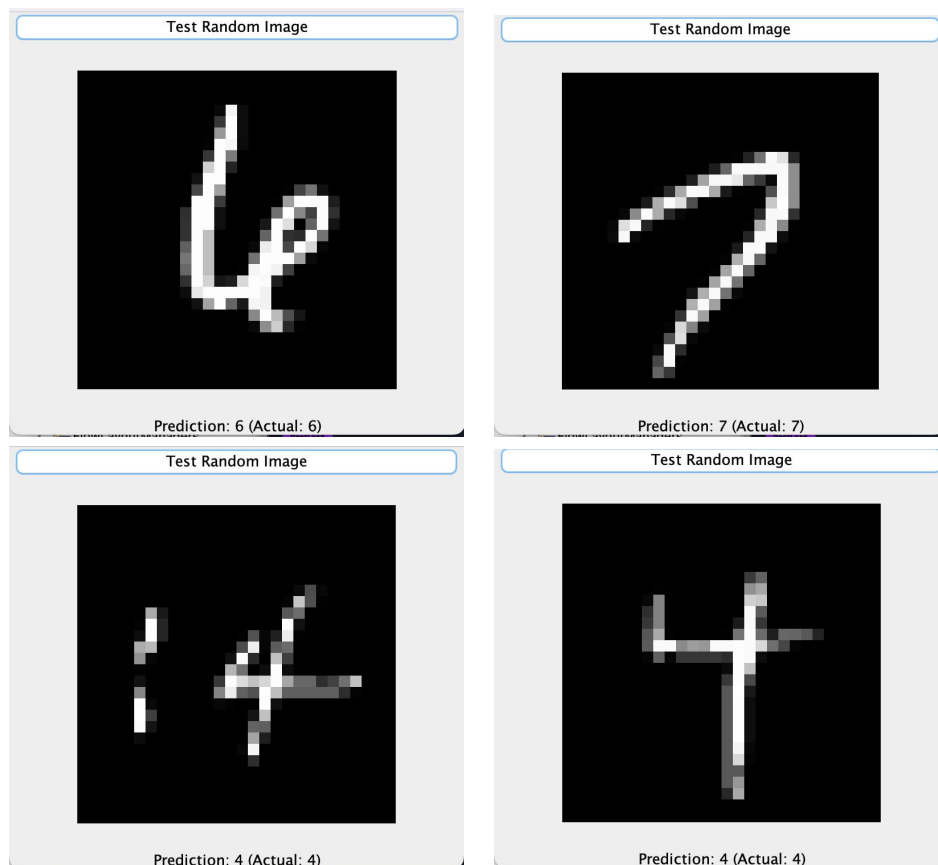
## 9 Interessante Beispiele

Ich möchte einige Beispiele zur Zahlenerkennung zeigen.

### 9.1 Erfolgreiche komplizierte Beispiele

In der ersten Gruppe sind die Zahlen untypische Bilder, die es manchmal sogar einem Menschen schwer machen zu verstehen, welche Zahl gezeigt wird. Aber gleichzeitig findet das neuronale Netz mit Hilfe seines "Neurons" in Form von Gewichten und Abweichungen die richtigen Lösungen.





## 9.2 Fehlende Beispiele

Die zweite Gruppe zeigt Fehler des neuronalen Netzes, wenn das falsche Ergebnis am wahrscheinlichsten ist. Es ist zu erkennen, dass einige Bilder tatsächlich mehrdeutig interpretiert werden können, aber die letzte zwei Bilde enthält eindeutig einen Erkennungsfehler und hat nichts mit Mehrdeutigkeit zu tun.

