Implementierung eines neuronalen Netzwerkes zur Zeichenerkennung in SetIX

Studienarbeit

Studiengang Angewandte Informatik

Duale Hochschule Baden-Württemberg Mannheim

von

Lucas Heuser und Johannes Hill

Bearbeitungszeitraum:	05.09.2016 - 29.05.2017
Matrikelnummer, Kurs:	9706659, TINF14AI-BI
Matrikelnummer, Kurs:	9705747, TINF14AI-BI
Ausbildungsfirma:	Roche Diagnostics GmbH, Mannheim
Abteilung:	Scientific Information Services
Betreuer der DHBW-Mannheim:	Prof. Dr. Karl Stroetmann

Unterschrift des Betreuers

Eidesstattliche Erklärung

Hiermit erklären wir, dass wir die vorliegende Arbeit mit dem Thema

Implementierung eines neuronalen Netzwerkes zur Zeichenerkennung in SetlX

selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlich-ten Schriften entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Mannheim, den 22. Mai 2017

Lucas Heuser	Johannes Hill

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Ziel der Arbeit . . . 1
- 1.2 Aufgabe des Neuronales Netzwerks . . . 1
- 1.3 Verfügbarkeit des Programmcodes auf GitHub . . . 2
- 1.4 Aktuelle Relevanz von neuronalen Netzen . . . 2
- 1.5 Aufbau der Arbeit . . . 2

2 Theorie **3**
- 2.1 Das Neuron . . . 3
- 2.2 Neuronales Netzwerk . . . 4
- 2.3 Backpropagation . . . 7
- 2.4 Stochastic Gradient Descent . . . 9

3 Implementierung **11**
- 3.1 Laden und Aufbereitung der MNIST Daten . . . 11
- 3.2 Implementierung des neuronalen Netzes . . . 12
- 3.3 Animation . . . 19

4 Fazit und Ausblick **25**
- 4.1 Auswertung des Ergebnisses . . . 25
- 4.2 Performance der SetIX Implementierung . . . 26

Abbildungsverzeichnis

1.1 Handgeschriebene Ziffer 5 [2] 1

2.1 Neuron mit Eingabevektor $\mathbf{x}$, Gewichtungsvektor $\mathbf{w}$ und Ausgabe y. [1] 3
2.2 Die Sigmoid-Funktion. [2] 4
2.3 Aufbau des neuronalen Netzwerks hinsichtlich der einzelnen Schichten. [2] 5

3.1 Eingabe in das neuronale Netzwerk. 20
3.2 Default Animation, welche über die Eingabe mit dem Wert 0 aufgerufen wird. 21
3.3 Untersuchungsbereich eines einzelnen Neurons der verborgenen Schicht. 22
3.4 Untersuchungsbereich aller Neuronen der verborgenen Schicht. 23
3.5 Untersuchungsbereich aller Neuronen der verborgenen Schicht hinsichtlich einer Eingabe. 24

Kapitel 1

Einleitung

1.1 Ziel der Arbeit

Diese Arbeit wurde im Rahmen einer Studienarbeit an der Dualen Hochschule Baden-Württemberg unter der Leitung von Prof. Dr. Karl Stroetmann angefertigt. Die Arbeit dient zur Unterstützung und Erweiterung der von Herrn Stroetmann gehaltenen Vorlesung „Artificial-Intelligence“[3]. Ziel der Arbeit ist es die Vorlesung um ein praktisches Beispiel für Neuronale Netze zu erweitern. In den Vorlesungen von Herrn Stroetmann wird zur Veranschaulichung von Algorithmen und Methoden die, an mathematische Formulierungen angelehnte, Programmiersprache SetlX verwendet [4]. In dieser Programmiersprache sollte auch das Neuronale Netzwerk programmiert werden.
Als Basis des in dieser Studienarbeit implementierten Netzwerkes, dient die Python Implementierung einer Zeichenerkennung von Michael Nielsen [2].

1.2 Aufgabe des Neuronales Netzwerks

Ziel des in dieser Arbeit implementierten Neuronalen Netzwerkes ist es, handgeschriebene Zeichen zu erkennen und auszuwerten. Die eingelesenen Zeichen bestehen aus 28x28 Pixeln, welche in verschiedenen Graustufen dargestellt werden. Die Ziffern bestehen aus Werten zwischen 0 und 9. Abb. 1.1

Abbildung 1.1: Handgeschriebene Ziffer 5 [2]

zeigt ein Beispiel einer solchen Ziffer. Mit Hilfe des menschlichen Auges und Gehirns ist es für die meisten Menschen ohne Probleme möglich, zu erkennen, dass es sich hierbei um eine Ziffer mit dem Wert „5“handelt. Eine Erkennung mittels herkömmlicher Computeralgorithmen hingegen stellt sich allerdings als sehr komplex und schwierig heraus. Gründe hierfür sind, dass beispielsweise verschiedene Ziffern durch unterschiedliche Handschriften signifikante Unterschiede aufweisen. Auch können beim Schreibvorgang einzelne Linien durch den Druck des Stiftes schwächer oder gar nicht abgebildet werden, was die gezeichnete Zahl ebenso variieren lässt. Diese und viele weitere Faktoren führen dazu, dass eine solche Zeichenerkennung mit Hilfe von einfachen Auswertealgorithmen zu hohen Fehlerraten führt.

Mit Hilfe eines Neuronalen Netzwerkes ist es bei solch einem Problem möglich, das Netzwerk automatisch mit Hilfe von Traningsdaten zu trainieren. Das bedeutet, dem Netzwerk wird eine möglichst

große Menge an Testdaten übergeben und das Netzwerk lernt automatisch mit Hilfe dieser Daten. Um dies bewerkstelligen zu können, müssen die Trainingsdaten aus folgenden Komponenten bestehen:

1. Eingabedaten (hier: Pixel des auszuwertenden Zeichens)
2. Erwartetes Ergebnis zu jeder Eingabe (hier: 5)

1.3 Verfügbarkeit des Programmcodes auf GitHub

Der in dieser Studienarbeit entwickelte Programmcode, sowie sämtliche Dokumentation sind in GitHub unter folgender Adresse zu finden:

https://github.com/lucash94/Neural-Network-in-SetlX

Im Verzeichnis „Studienarbeit“befindet sich diese Arbeit und das Verzeichnis „setlx“beinhaltet die eigentliche Implementierung in SetlX. Das dritte Verzeichnis „res“dient zur Aufbewahrung aller sonstigen Dateien und Aufzeichnungen der Studienarbeit.

1.4 Aktuelle Relevanz von neuronalen Netzen

Die Relevanz neuronaler Netzwerke nimmt im privaten Alltag immer mehr zu. Mittlerweile bieten große IT-Unternehmen Produkte für den Massengebrauch an, welche sich der Hilfe Neuronaler Netzwerke bedienen. Einige populäre Beispiele dieser Projekte sind:

1. Verbesserung der Übersetzungsergebnisse des Google Translaters wurden mittels Neuronalen Netzen und einer hohen Anzahl an Trainingsdaten ermöglicht. Am 26.09.2016 wurde das Google Neural Machine Translation system (GNMT) in das Online-Tool eingeführt. [5]
2. Das Programm AlphaGo des Unternehmens Google DeepMind ist spezialisiert auf das aus China stammende Brettspiel Go. Mit Hilfe eines neuronalen Netzes war AlphaGo das erste Computerprogramm, welches einen professionellen Go-Spieler schlagen konnte. [6]
3. Die Foto- und Videobearbeitungsapplikation Prisma nutzt ein neuronales Netz um Fotos und Videos von Nutzern mit Effekten und Filtern basierend auf berühmten Kunstwerken zu versehen. [7]

1.5 Aufbau der Arbeit

Diese Arbeit ist in drei wesentliche Kategorien unterteilt. Zu Beginn der Arbeit wird ein Überblick über das theoretische Wissen sowie den allgemeinen Aufbau und die Funktion Neuronaler Netze gegeben. Anschließend wird die konkrete Umsetzung des Projektes in SetlX erläutert. Hierbei wird kurz die Beschaffung der Datensätze gefolgt von der Hauptimplementierung besprochen. Ebenso wird es einen Abschnitt über ein weiteres Programm geben, welches die Ausgabe des neuronalen Netzwerkes grafisch darstellt.
Der letzte Abschnitt der Arbeit befasst sich mit der Auswertung des Ergebnisses. Hierbei wird die Performance des finalen Programmes diskutiert sowie ein Fazit über den Erfolg oder Misserfolg der Arbeit gezogen.

Kapitel 2

Theorie

Dieses Kapitel beschäftigt sich mit den Grundlagen für das Entwickeln und Implementieren eines neuronalen Netzwerks. Die Notationen für die mathematischen Ausdrücke orientieren sich an dem Skript zur Vorlesung *Artifical Intelligence* bei Prof. Dr. Karl Stroetmann [3].

2.1 Das Neuron

Ein grundlegender Bestandteil des menschlichen Gehirns ist das Neuron. Bereits ein kleiner Ausschnitt in der Größe eines Reiskorns enthält über 10000 Neuronen, wobei jedes Neuron durchschnittlich 6000 Verbindungen mit anderen Neuronen bildet [1]. Dieses biologische Netzwerk ermöglicht dem Menschen, die Welt um ihn herum zu erleben. Das Ziel in diesem Abschnitt ist es, diese natürliche Struktur zu nutzen, um maschinelle Lernmodelle zu entwickeln, die Probleme auf analoge Weise lösen. Hierbei ist es nicht notwendig zu wissen wie das biologische Neuron funktioniert, noch wie ein ein Netzwerk aus biologischen Neuronen arbeitet. Stattdessen wird eine mathematische Abstraktion eines Neurons formuliert, welches die Grundlage für unser neuronales Netzwerk bildet.
Ein Neuron mit n Eingaben wird als Paar $\langle w, b\rangle$ definiert, wobei der Vektor $\mathbf{w} \in \mathbb{R}^m$ den Gewichtungsvektor und $b \in \mathbb{R}$ die Vorbelastung repräsentieren. Konzeptionell gesehen, ist das Neuron eine Funktion p, welche den Eingabevektor $\mathbf{x} \in \mathbb{R}^m$ auf das Intervall $[0, 1]$ abbildet. Diese Funktion ist definiert als

$$p(\mathbf{x}; \mathbf{w}, b) := a(\mathbf{x} \cdot \mathbf{w} + b),$$

wobei a als die sogenannte Aktivierungsfunktion bezeichnet wird (siehe Abb. 2.1).

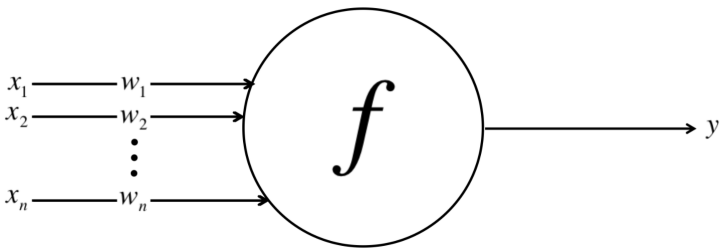

Abbildung 2.1: Neuron mit Eingabevektor $\mathbf{x}$, Gewichtungsvektor $\mathbf{w}$ und Ausgabe y. [1]

In dieser Arbeit wird die Sigmoid-Funktion für die Aktivierung eines Neurons verwendet. Die Sigmoid-Funktion $S : \mathbb{R} \to [0, 1]$ ist definiert als

$$S(t) := \tfrac{1}{1+\exp(-t)}.$$

Fällt die Betrachtung auf die Definition der Sigmoid-Funktion, lassen sich auf Basis der folgenden Überlegungen

$$\lim_{x\to-\infty}\exp(-x)=\infty,\qquad \lim_{x\to+\infty}\exp(-x)=0,\qquad \text{and}\qquad \lim_{x\to\infty}\frac{1}{x}=0,$$

die folgenden Eigenschaften ableiten:

$$\lim_{t\to-\infty}S(t)=0\qquad\text{and}\qquad\lim_{t\to+\infty}S(t)=1.$$

Die Sigmoid-Funktion S konvergiert somit bei der Grenzwertbetrachtung gegen 0 bzw. 1. Eine weitere Eigenschaft der Sigmoid-Funktion besteht in deren Symmetrie (siehe Abb. 2.2).

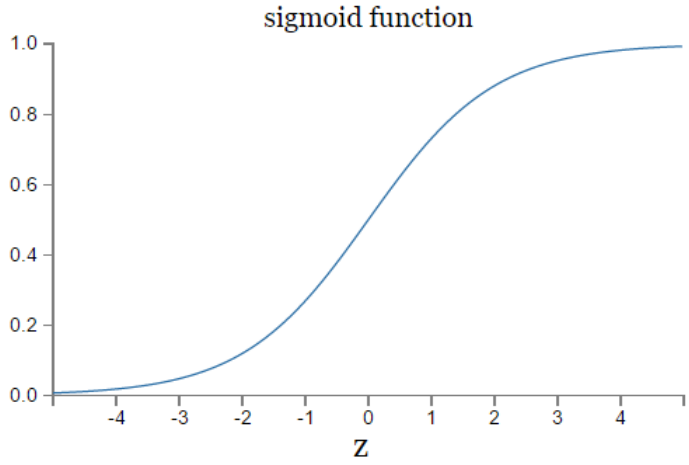

Abbildung 2.2: Die Sigmoid-Funktion. [2]

Bei einer Verschiebung der Funktion um $\frac{1}{2}$, liegt eine zentral symmetrische Funktion vor.

$$S(-t)-\tfrac{1}{2}=-\left(S(t)-\tfrac{1}{2}\right).$$

Die Addition von $\frac{1}{2}$ auf beiden Seiten der Gleichung liefert

$$S(-t)=1-S(t).$$

Fällt die Betrachtung zurück auf auf die Funktion p zur Beschreibung des Neurons, liefert die Indexnotation die folgende Schreibweise. Mit

$$\mathbf{w}=\langle w_1,\cdots,w_m\rangle^T$$

für den Gewichtungsvektor und

$$\mathbf{x}=\langle x_1,\cdots,x_m\rangle^T$$

für den Eingabevektor, ergibt sich

$$p(\mathbf{x};\mathbf{w},b)=S\left(\left(\sum_{i=1}^{m}x_i\cdot w_i\right)+b\right).$$

2.2 Neuronales Netzwerk

Das in dieser Arbeit angewandte Netzwerk nennt sich hierbei *feedforward neural network* und beschreibt ein Netzwerk aus Neuronen, deren Informationsfluss keine Schleifen durchläuft. Die Topologie des neuronalen Netzwerk ist gegeben durch eine Zahl $L\in\mathbb{N}$ und einer Liste $[m(1),\cdots,m(L)]$ mit L natürlichen Zahlen. Hierbei bezeichnet L die Anzahl der Schichten im neuronalen Netzwerk und für $i\in\{2,\cdots,L\}$ gibt der Wert von $m(i)$ die Anzahl der Neuronen der l-ten Schicht an. Die erste Schicht wird in diesem Modell als Eingabeschicht bezeichnet. Sie enthält im Vergleich zu anderen Schichten

keine Neuronen sondern Eingabeknoten. Die letzte Schicht (mit Index L) wird als Ausgabeschicht bezeichnet, wohingegen alle restlichen Schichten als verborgene Schichten bezeichnet werden. Liegen dem Netzwerk mehr wie nur eine verborgene Schicht vor, so bezeichnet man dieses als *deep neural network* (siehe Abb. 2.3).

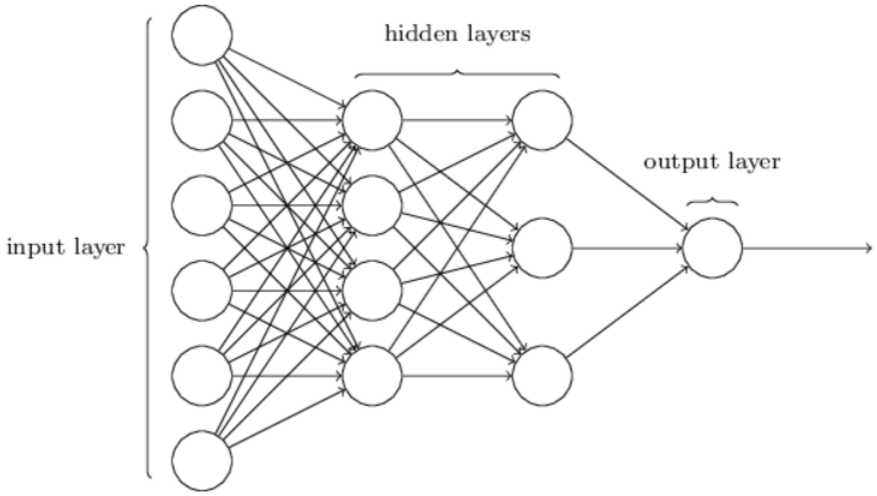

Abbildung 2.3: Aufbau des neuronalen Netzwerks hinsichtlich der einzelnen Schichten. [2]

Für die erste Schicht, die Eingabeschicht, ist die Eingabedimension definiert durch $m(1)$. Analog ist die Ausgabedimension durch $m(L)$ definiert. Jeder Knoten der l-ten Schicht ist zu jedem Knoten der $(l+1)$-ten Schicht über eine Gewichtung verbunden. Weiterhin ist die Gewichtung des k-ten Neuron der l-ten Schicht zu dem j-ten Neuron in der $(l+1)$-ten Schicht gegeben durch $w_{j,k}^{(l)}$. Alle Gewichtungen in Schicht l sind über die Gewichtungsmatrix $W^{(l)}$ zusammengefasst. Die Matrix ist eine $m(l) \times m(l-1)$ Matrix mit $W^{(l)} \in \mathbb{R}^{m(l)\times m(l-1)}$ und ist definiert als

$$W^{(l)} := \left(w_{j,k}^{(l)}\right).$$

Das j-te Neuron in Schicht l hat ebenfalls noch eine Vorbelastung $b_j^{(l)}$. Die Vorbelastungen der Schicht l werden ebenfalls über den Vorbelastungsvektor $\mathbf{b}^{(l)}$ zusammengefasst mit

$$\mathbf{b}^{(l)} := \langle b_1^{(l)}, \cdots, b_{m(l)}^{(l)} \rangle^\top.$$

Für die Aktivierungsfunktion $a_j^{(l)}$ des j-ten Neurons in Schicht l ergibt sich hierbei die folgende rekursive Definition:

1. Für die erste Schicht ergibt sich

$$a_j^{(1)} := x_j. \tag{2.1}$$

   Dies bedeutet, dass der Eingabevektor $\mathbf{x}$ die Aktivierung der Eingangsknoten darstellt.

2. Für alle anderen Knoten ergibt sich

$$a_j^{(l)}(\mathbf{x}) := S\left(\left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)}(\mathbf{x})\right) + b_j^{(l)}\right) \quad \forall l \in \{2, \cdots, L\}. \tag{2.2}$$

Der Aktivierungsvektor der l-ten Schicht ist somit definiert durch

$$\mathbf{a}^{(l)} := \langle a_1^{(l)}, \cdots, a_{m(l)}^{(l)} \rangle^\top.$$

Des Weiteren ist die Ausgabe des neuronalen Netzwerks für eine Eingabe $\mathbf{x}$ über die Neuronen der Ausgabeschicht gegeben. Der Ausgabevektor $\mathbf{o}(\mathbf{x}) \in \mathbb{R}^{m(L)}$ ist definiert über

$$\mathbf{o}(\mathbf{x}) := \langle a_1^{(L)}(\mathbf{x}), \cdots, a_{m(L)}^{(L)}(\mathbf{x}) \rangle^\top = \mathbf{a}^{(L)}(\mathbf{x}).$$

Mit den zuvor definierten Gleichungen 2.1 und 2.2 kann nun betrachtet werden, wie Informationen durch Netzwerk verbreitet werden (**feedforward**).

1. Zu Beginn ist der Eingabevektor $\mathbf{x}$ gegeben und gespeichert in der Eingabeschicht des neuronalen Netzwerks:
$$\mathbf{a}^{(1)}(\mathbf{x}) := \mathbf{x}.$$
2. Die erste Schicht von Neuronen, welche die zweite Schicht mit Knoten darstellt, wird aktiviert und berechnet über den Aktivierungsvektor $\mathbf{a}^{(2)}$ nach der Formel
$$\mathbf{a}^{(2)}(\mathbf{x}) := S\big(W^{(2)} \cdot \mathbf{a}^{(1)}(\mathbf{x}) + \mathbf{b}^{(2)}\big) = S\big(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}\big).$$
3. Die zweite Schicht von Neuronen, welche die dritte Schicht mit Knoten darstellt, wird aktiviert und berechnet über den Aktivierungsvektor $\mathbf{a}^{(3)}$ nach der Formel
$$\mathbf{a}^{(3)}(\mathbf{x}) := S\big(W^{(3)} \cdot \mathbf{a}^{(2)}(\mathbf{x}) + \mathbf{b}^{(3)}\big) = S\Big(W^{(3)} \cdot S\big(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}\big) + \mathbf{b}^{(3)}\Big)$$
4. Dies wird solange weitergeführt bis die Ausgabeschicht erreicht wird und die Ausgabe
$$\mathbf{o}(\mathbf{x}) := \mathbf{a}^{(L)}(\mathbf{x})$$
berechnet wurde.

In der folgenden Betrachtung wird angenommen, dass dem neuronalen Netzwerk n Trainingsdaten mit

$$\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle \quad \text{for } i = 1, \cdots, n$$

vorliegen, sodass

$$\mathbf{x}^{(i)} \in \mathbb{R}^{(1)} \text{ und } \mathbf{y}^{(i)} \in \mathbb{R}^{m(L)}.$$

Das Ziel ist eine Belegung der Gewichtungsmatrix $W^{(l)}$ und dem Vorbelastungsvektor $b^{(l)}$ zu finden, damit

$$\mathbf{o}\big(\mathbf{x}^{(i)}\big) = \mathbf{y}^{(i)} \quad \text{for all } i \in \{1, \cdots, n\}.$$

In der Regel wird es nicht möglich sein die Gleichungen für alle $i \in \{1, \cdots, n\}$ zu erfüllen, weshalb es gilt den Fehler zu minimieren. An dieser Stelle fällt die Betrachtung auf die quadratische Fehlerkostenfunktion C, die definiert ist als

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \Big(\mathbf{o}\big(\mathbf{x}^{(i)}\big) - \mathbf{y}^{(i)}\Big)^2.$$

An dieser Stelle ist zu berücksichtigen, dass die Kostenfunktion hinsichtlich der Trainingsdaten $\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle$ additiv ist. Eine Vereinfachung der Notation kann vorgenommen, wenn die Betrachtung bei der Kostenfunktion auf ein Trainingsbeispiel $\langle \mathbf{x}, \mathbf{y} \rangle$ fällt. Dazu wird

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big) := \frac{1}{2} \cdot \Big(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}\Big)^2$$

definiert. Weiterhin ergibt sich für die allgemeine Kostenfunktion

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{n} \cdot \sum_{i=1}^{n} C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}\Big(W^{(2)}, \cdots W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big).$$

Für den weiteren Verlauf dieser Arbeit die Notation $C_{\mathbf{x},\mathbf{y}}$ für den Ausdruck

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big)$$

eingeführt.

2.3 Backpropagation

Der Backpropagation-Algorithmus ist das meist genutzte neuronale Modell. Grund ist die universelle Einsetzbarkeit für beliebige Approximationsaufgaben im Hinblick auf die effiziente Berechnung der partiellen Ableitung der Kostenfunktion C nach der Gewichtung $w_{j,k}^{(l)}$ und der Vorbelastungen $b_j^{(l)}$. Ziel des Algorithmus ist die Minimierung der Abweichung zwischen erwartetem Ausgabewert und dem durch das neuronale Netzwerk ermitteltem Wert. Hierzu wird in dem folgenden Kapitel das Werkzeug für dessen Berechnung gegeben.

Für die Backpropagation-Gleichungen werden zunächst Hilfsgrößen definiert, damit.
Die Definition von $z_j^{(l)}$, soll die Eingabe der Aktivierungsfunktion S des j-ten Neuron der l-ten Schicht darstellen:

$$z_j^{(l)} := \left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)} \right) + b_j^{(l)} \quad \forall j \in \{1, \cdots, m(l)\} \text{ und } \forall l \in \{2, \cdots, L\}.$$

Im Wesentlichen ist $z_j^{(l)}$ die Eingabe der Sigmoid-Funktion, damit die Aktivierung $a_j^{(l)}$ mit

$$a_j^{(l)} = S(z_j^{(l)}).$$

berechnet werden kann. Angenommen im j-ten Neuron der l-ten Schicht erfahren die Berechnungen eine zusätzliche Änderung $\Delta z_j^{(l)}$. Dadurch erfährt die Aktivierung des nachgelagerten Neuron den folgenden Zusatz:

$$a_j^{(l)} = S(z_j^{(l)} + \Delta z_j^{(l)}).$$

Diese Änderungen wird durch das neuronale Netzwerk weitergeleitet und kann entsprechende Auswirkungen auf die korrekte Bestimmung der Eingaben haben. Damit Fehler vermieder werden empfiehlt sich im Hinblick auf die partielle Ableitung der Kostenfunktion $C_{\mathbf{x},\mathbf{y}}$ nach der Gewichtung $w_{j,k}^{(l)}$ und den Vorbelastungen $b_j^{(l)}$ zuerst die Berechnung der partiellen Ableitung von $C_{\mathbf{x},\mathbf{y}}$ nach $z_j^{(l)}$ durchzuführen. Hierzu sei folgende Definition gegeben:

$$\varepsilon_j^{(l)} := \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(l)}} \quad \forall j \in \{1, \cdots, m(l)\} \text{ und } \forall l \in \{2, \cdots, L\},$$

An dieser Stelle ist zu berücksichtigen, dass die partielle Ableitung $\varepsilon_j^{(l)}$ abhängig von $\mathbf{x}$ und $\mathbf{y}$ ist. Weiterhin werden die Größen $\varepsilon_j^{(l)}$ in dem Vektor

$$\boldsymbol{\varepsilon}^{(l)} := \begin{pmatrix} \varepsilon_1^{(l)} \\ \vdots \\ \varepsilon_{m(l)}^{(l)} \end{pmatrix}.$$

zusammengefasst und als *Fehler in Schicht l* bezeichnet.
Mit den definierten Hilfsgrößen ist es nun möglich die vier Gleichung für die Backpropagation zu beschreiben.

1. Die erste Gleichungen berechnet $\varepsilon_j^{(L)}$ für $l = L$, wobei $S'(x)$ die Ableitung der Sigmoid-Funktion bezeichnet. Mit dieser Gleichung wir der Fehler in der Ausgabeschicht berechnet.

$$\varepsilon_j^{(L)} = (a_j^{(L)} - y_j) \cdot S'\left(z_j^{(L)}\right) \quad \forall j \in \{1, \cdots, m(L)\}. \tag{BP1}$$

   mit

$$S'(x) = (1 - S(t)) \cdot S(t). \tag{2.3}$$

   Die Gleichung BP1 kann auch als vektorisierte Schreibweise mittels dem Hardamard-Produkt

angegeben werden. Hierfür ergibt sich:

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'(\mathbf{z}^{(L)}), \tag{BP1v}$$

wobei $S'(\mathbf{z}^{(\mathbf{L})})$ wie folgt definiert ist:

$$S' \begin{pmatrix} z_1^{(L)} \\ \vdots \\ z_{m(L)}^{(L)} \end{pmatrix} := \begin{pmatrix} S'(z_1^{(L)}) \\ \vdots \\ S'(z_{m(L)}^{(L)}) \end{pmatrix}.$$

2. Die zweite Gleichung berechnet $\varepsilon_j^{(L)}$ für alle $l < L$. Der Fehler $\boldsymbol{\varepsilon}^{(l+1)}$ von Schicht $l+1$ fließt hierbei in die Berechnung von $\boldsymbol{\varepsilon}^{(l)}$ ein.

$$\varepsilon_j^{(l)} = \sum_{i=1}^{m(l+1)} w_{i,j}^{(l+1)} \cdot \varepsilon_i^{(l+1)} \cdot S'(z_j^{(l)}) \quad \forall j \in \{1, \cdots, m(l)\} \text{ und } \forall l \in \{2, \cdots, L-1\} \tag{BP2}$$

Auch diese Gleichung kann in der vektorisierten Schreibweise angegeben werden:

$$\boldsymbol{\varepsilon}^{(l)} = \left(\left(W^{(l+1)}\right)^\top \cdot \boldsymbol{\varepsilon}^{(l+1)} \right) \odot S'(z^{(l)}) \quad \forall l \in \{2, \cdots, L-1\}. \tag{BP2v}$$

3. Die nächste Gleichung ist durch die partielle Ableitung von $C_{\mathbf{x},\mathbf{y}}$ nach der Vorbelastung $b_j^{(l)}$ des j-ten Neuron in Schicht l gegeben. Diese Gleichung gibt somit die Änderungsrate der Kosten hinsichtlich der Vorbelastungen an.

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{b_j^{(l)}} = \varepsilon_j^{(l)} \quad \forall j \in \{1, \cdots, m(l)\} \text{ und } \forall l \in \{2, \cdots, l\} \tag{BP3}$$

Die vektorisierte Schreibweise nimmt hierbei die folgende Form an:

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \forall l \in \{2, \cdots, l\} \tag{BP3v}$$

Mit $\nabla_{\mathbf{b}^{(l)}}$ wird in dieser Arbeit der Gradient von $C_{\mathbf{x},\mathbf{y}}$ im Hinblick auf $b^{(l)}$ bezeichnet.

4. Die letzte Gleichung ist durch die partielle Ableitung von $C_{\mathbf{x},\mathbf{y}}$ nach der Gewichtung $w_{j,k}^{(l)}$ gegeben. Diese Gleichung gibt somit die Änderungsrate der Kosten hinsichtlich der Gewichtungen an.

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \cdot \varepsilon_j^{(l)} \quad \forall j \in \{1, \cdots, m(l)\}, \forall k \in \{1, \cdots, m(l-1)\}, \forall l \in \{2, \cdots, l\} \tag{BP4}$$

Die vektorisierte Schreibweise liefert auch in diesem Fall die folgende Form:

$$\nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^\top \quad \forall l \in \{2, \cdots, l\} \tag{BP4v}$$

Der Ausdruck $\boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^\top$ bezeichnet hierbei die Matrixmultiplikation zwischen dem Spaltenvektor $\boldsymbol{\varepsilon}^{(l)}$ als $m(l) \times 1$ Matrix und dem Reihenvektor $\left(\mathbf{a}^{(l-1)}\right)^\top$ als $1 \times m(l-1)$ Matrix.

Bei Anbetracht der Gleichungen BP3 und BP4v wird nun ersichtlich, weshalb die Einführung des Vektors $\boldsymbol{\varepsilon}^{(l)}$ von Vorteil war. Dieser Vektor ermöglicht die partielle Ableitung nach den Gewichtungen sowie den Vorbelastungen für die Kostenfunktion. Ebenfalls wurde die vektorisierte Schreibweise für die einzelnen Gleichungen in diesem Abschnitt aus Effizienzgründen für die spätere Implementierung eingeführt. Dies liegt in der schnelleren Ausführung von Matrix-Vektor-Multiplikationen bei Interpretersprachen wie z.B. SetlX begründet.

Der Backpropagation Algorithmus durchläuft die folgenden Schritte:

1. **Feedforward-Berechnung:**
   (a) Zuweisung des Eingabevektors **x** der Eingabeschicht mit über die Aktivierung $a^{(1)}$
   (b) Für jede weitere Schicht $l \in \{2, \cdots, L\}$ wird die Aktivierung
   $$a^{(l)}(\mathbf{x}) := S(W^{(l)} \cdot a^{(l-1)}(x) + \mathbf{b}^{(l)}$$
   berechnet.
2. **Backpropagation-Berechnung:**
   (a) Berechnung des Fehler in Schicht L (Ausgabeschicht) mit
   $$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'\big(\mathbf{z}^{(L)}\big)$$
   (b) Berechnung der Fehler mittels *backpropagate* für die Schichten $l = L-1, L-2, \cdots, 2$
   $$\boldsymbol{\varepsilon}^{(l)} = \Big(\big(W^{(l+1)}\big)^{\top} \cdot \boldsymbol{\varepsilon}^{(l+1)}\Big) \odot S'\big(z^{(l)}\big)$$
   (c) Ausgabe der Gradienten der Kostenfunktion
   $$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \text{und} \quad \nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \big(\mathbf{a}^{(l-1)}\big)^{\top}$$

Wie bereits beschrieben berechnet der Backpropagation Algorithmus den Gradienten der Kostenfunktion für die einzelnen Trainingsbeispiele. In der Praxis ist es üblich den Backpropagation mit einem Lernalgorithmus zu kombinieren. Hierzu wird im folgenden Abschnitt auf den *Stochastic Gradient Descent* eingegangen.

2.4 Stochastic Gradient Descent

Für eine zuverlässige Klassifizierung der Eingaben wird eine Algorithmus benötigt, welcher die Bestimmung von Gewichtungen und Vorbelastungen bestmöglich gewährleistet. Hierzu wird in dieser Arbeit auf die Methode des *Stochastic Gradient Descent* zurückgegriffen. Im Bereich des maschinellen Lernen ist es notwendig das Minimum oder Maximum einer Funktion

$$f : \mathbb{R}^n \to \mathbb{R}.$$

zu ermitteln. Die im vorangegangenen Kapitel vorgestellten Gleichungen der Backpropagation beschreiben den Gradienten der Kostenfunktion für ein einzelnes Trainingsbeispiel $\langle \mathbf{x}, \mathbf{y} \rangle$. Besteh die Absicht unser neuronales Netzwerk zu trainieren, müssen alle Trainingsdaten berücksichtigt werden. Für n Trainingsdaten

$$\langle \mathbf{x}^{(1)}, \mathbf{y}^{(1)} \rangle,\ \langle \mathbf{x}^{(2)}, \mathbf{y}^{(2)} \rangle,\ \cdots,\ \langle \mathbf{x}^{(n)}, \mathbf{y}^{(n)} \rangle,$$

wurde die quadratische Fehlerkostenfunktion bereits wie folgt definiert:

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \Big(\mathbf{o}\big(\mathbf{x}^{(i)}\big) - \mathbf{y}^{(i)}\Big)^2.$$

Wenn die Berechnung der Gradienten für die quadratische Fehlerkostenfunktion hinsichtlich einer Gewichtungsmatrix $W^{(l)}$ oder einer Vorbelastung $b^{(l)}$ vorgenommen werden soll, müssen die Summen

$$\frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \quad \text{and} \quad \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial b_{j}^{(l)}}$$

über alle Trainingsdaten für einen Iterationsschritt des *Gradient Descent* berechnet werden. Dies kann

mit einer großen Anzahl von Trainingdaten kostenintensive Auswirkungen haben, weshalb beim *Stochastic Gradient Descent* für die Berechnung der Summen eine zufällige Teilmenge aus den Trainingsdaten für die Abschätzung herangezogen wird. Bei einer Teilmenge mit m Trainingsdaten, liegt die folgende Abschätzung zugrunde:

$$\frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \approx \frac{1}{2 \cdot m} \cdot \sum_{i=1}^{m} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \quad \text{und} \quad \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial b_{j}^{(l)}} \approx \frac{1}{2 \cdot m} \cdot \sum_{i=1}^{m} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial b_{j}^{(l)}}.$$

Die Namensgebung für diese Methode, ist durch die zufällige Trainingdatenauswahl zu erklären. Im Vergleich zur *Gradient Descent* Methode, kann sie das Lernen in neuronalen Netzwerken erheblich beschleunigen, da nicht alle Trainingsdaten in den Prozess einfließen.

Kapitel 3

Implementierung

3.1 Laden und Aufbereitung der MNIST Daten

Für das Neuronale Netzwerk zur Erkennung von hangeschriebenen Zeichen zwischen werden Test- und Trainingsdaten der MNIST Datenbank genutzt. Die Datensätze können unter folgender Adresse gefunden werden:

http://yann.lecun.com/exdb/mnist/

Da der MNIST Datensatz lediglich in Form von Binärdateien vorliegt und es in der aktuellen Version von SetlX nicht möglich ist Binärdateien zu lesen, wurde statt dem original Datensatz der umgewandelte Datensatz in Form einer CSV-Datei verwendet. Die Dateien können hier heruntergeladen werden:

https://pjreddie.com/projects/mnist-in-csv/

Die Verwendung des CSV-Formats führt dazu, dass die Größe der Datensätze auf Grund fehlender Komprimierungen ansteigt. Ebenso wird das Einlesen der Datensätze langsamer, was an der eigentlichen Funktion des neuronalen Netzes allerdings nichts ändert und somit für dieses Projekt vertretbar ist. Eine Option das erste Problem zu umgehen wäre die Komprimierung in das Zip-Format und die Dekomprimierung zu Beginn des Startens des Programms. Da SetlX keine Unzip-Funktion für Dateien bietet, müsste hierbei allerdings Kenntnis über das jeweils vom Benutzer verwendete Betriebssystem gegeben sein und ebenso ob und wenn ja welches Programm hierfür zur Verfügung steht. Bei der Festlegung auf ein ein Kommandozeilen-Befehl (z.B. "gunzip" oder "unzip" für Linux-basierte PCs) würde somit die Betriebssystemunabhängigkeit verloren gehen.
Verwendet werden die CSV-Dateien `mnist_test.csv` und `mnist_train.csv`. Die Traingsdaten umfassen insgesamt 60.000 Datensätze und die Testdaten 10.000 Datensätze. Die einzelnen Datensätze, also die handgeschriebenen Zeichen, sind in den Dateien in folgendem Format gespeichert:

`label, pixel1, pixel2, pixel3, ..., pixel784`
`label, pixel1, pixel2, pixel3, ..., pixel784`
...

Das heißt, in jeder Zeile befinden sich alle Daten zu einer Ziffer. Der erste Wert gibt den jeweiligen Wert an (z.B. 5) und darauf folgend befinden sich alle Pixel der Ziffer mit deren jeweiligen Graustufenwerten. Die Pixel werden der Reihe nach abgespeichert, wobei die "Leserichtung" einer Ziffer von links nach rechts und dann von oben nach unten ist.

Um die Datensätze nun in SetlX importieren zu können, wird die Datei `csv_loader.stlx` verwendet. Wird die Datei im SetlX-Interpreter ausgeführt, so liest sie die CSV-Dateien der Test- und Trainingsdaten (die Dateien müssen im selben Verzeichnis liegen und den oben erwähnten Namen haben) und speichert die Daten in den Variablen `test_data` sowie `training_data`. Die Testdaten sind hierbei als Liste von Paaren in Form folgender Form abgelegt:

```
[
    [pixels, label],
    [pixels, label],
    ...
]
```

Hierbei ist `pixels` eine Liste mit Integer Werten zwischen 0 und 255. Der Wert des Zeichens wird in `label` als Integer gespeichert.
Die Trainingsdaten sind prinzipiell nach dem gleichen Prinzip aufgebaut, allerdings wird hier für spätere Auswertungszwecke der Wert der Ziffer nicht als konkrete Zahl gespeichert, sondern in vektorisierter Form. Der vektorisierte Wert einer Zahl wird hier durch einen Vektor dargestellt, dessen Inhalt immer 0 ist, außer an der `label` + 1-ten Stelle. Dies entspricht dann genau der Form der Ausgabe des Netzwerkes. Beispielhaft würde eine Ziffer mit dem Wert 7 als folgender Vektor dargestellt werden:
$<< 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 >>$

Auf eine genaue Beschreibung der Implementierung des Ladevorgangs wird in dieser Studienarbeit verzichtet, da hierbei keine komplexen Funktionen angewandt wurden und das Verfahren nicht relevant für das Verständnis neuronaler Netze an sich ist.

3.2 Implementierung des neuronalen Netzes

Dieser Abschnitt beschreibt die eigentliche Implementierung des neuronalen Netzwerkes zur Erkennung von handgeschriebenen Ziffern in SetlX. Um den Code möglichst kompakt zu halten, wurden die in den Originaldateien enthaltenen Kommentarzeilen in dieser Seminararbeit zum größten Teil entfernt. Bei der Umsetzung des Netzwerkes in SetlX wird der SGD-Algorithmus als Lernmethode des Netzwerkes benutzt. Die im vorherigen Kapitel importierten Daten des MNIST-Datensatzes dienen als Grundlage der Ziffernerkennung. Das Netzwerk wird als Klasse in SetlX angelegt und enthält die folgenden Membervariablen:

1. `mNumLayers`: Anzahl der Schichten des aufzubauenden Netzwerkes
2. `mSizes`: Aufbau des Netzwerkes in Listenform. Bsp.: $[784, 30, 10]$ beschreibt ein Netzwerk mit 784 Eingabe-Feldern, 30 Neuronen in der zweiten Schicht und 10 Ausgabe-Neuronen
3. `mBiases`: Alle Vorbelastungen des Netzwerkes (genauer Aufbau wird im Folgenden erläutert)
4. `mWeights`: Alle Gewichte des Netzwerkes (genauer Aufbau wird im Folgenden erläutert)

Die Initialisierung des Netzwerkes zur Ziffernerkennung erfolgt durch folgende Befehle:

```
1  net := network([784, 30, 10]);
2  net.init();
```

Als Übergabeparameter bei der Erstellung eines Netzwerk-Objektes wird die Struktur des Netzwerkes in Form einer Liste übergeben. Diese wird dann lediglich `mSizes` zugeordnet und basierend hierrauf wird `mNumLayers` ermittelt. Die `init()`-Funktion der `network`-Klasse wird verwendet um die Gewichte und Vorbelastungen des Netzwerkes initial zufällig zu belegen. Hiermit werden Ausgangswerte gesetzt, welche später durch das Lernen des Netzwerkes angepasst werden. Im Folgenden sind die verwendeten Funktionen, welche während der Gewichts- und Vorbelastungs-Initialisierung verwendet werden, zu sehen.

```
1  init := procedure() {
2      computeRndBiases();
3      computeRndWeights();
4  };
5  computeRndBiases := procedure() {
6      this.mBiases := [
7          computeRndMatrix(1, mSizes[i]) : i in [2..mNumLayers]
8      ];
9  };
10 computeRndWeights := procedure() {
11     this.mWeights := [
12         computeRndMatrix(mSizes[i], mSizes[i+1]) : i in [1..mNumLayers-1]
13     ];
14 };
15 computeRndMatrix := procedure(row, col) {
16     return la_matrix([
17         [ ((random()-0.5)*2)/28 : p in [1..row] ] : q in [1..col]
18     ]);
19 };
```

1. `init`(): In der Funktion werden die Vorbelastungen und Gewichtungen des Netzwerkes zu Beginn initialisiert

2. `computeRndBiases`(): Die Funktion befüllt die Variable mBiases mit zufälligen Werten. Der für das Netzwerk benötigte Aufbau der Vorbelastungen entspricht folgender Form:
   [
   << << Bias_Schicht2_Neuron1 >> << Bias_Schicht2_Neuron2 >> ... >>,
   << << Bias_Schicht3_Neuron1 >> ... >>,
   ...]
   Das heißt es kann auf die Vorbelastungen mit folgendem Schema in SetlX zugegriffen werden:

   $$\texttt{mBiases}[\texttt{Schicht}][\texttt{Neuron}][1]$$

   Hierbei ist zu beachten, dass der letzte Index immer 1 ist, da jedes Neuron nur eine einzige Vorbelastung besitzt und die Vorbelastungen als Matrix abgelegt werden. Die Verwendung des Matrix-Datentyps wurde bewusst, auf Grund späterer Berechnungen mit Hilfe der `la_hadamard`()-Funktion, gewählt. Da es sich bei der Eingabe-Schicht des Netzwerkes nicht um Sigmoid-Neuronen handelt, sondern lediglich um Eingabewerte, werden hierfür keine Vorbelastungen benötigt. Deshalb wird bei der Erstellung der zufälligen Vorbelastungen nur $[2..\texttt{mNumLayers}]$ (also alle Schichten außer der Ersten) betrachtet.

3. `computeRndWeights`(): Diese Funktion ist equivalent zu der Vorbelastungs-Funktion, lediglich wird die Struktur der Gewichte mit folgenden Zugriffmsmöglichkeiten angelegt:

   $$\texttt{mWeights}[\texttt{Schicht}][\texttt{Neuron}][\texttt{Gewicht}]$$

4. `computeRndMatrix`(): Diese Hilfsfunktion dient zur Erstellung der Struktur der Gewichte und Vorbelastungen in den zuvor vorgestellten Funktionen. Auf Grund der Verwendung der `la_hadamard`()-Funktion, welche im weiteren Programm benötigt wird, wurde sich für die Verwendung des Matrix-Datentyps statt eines Vektors entschieden. Die Funktion enthält als Parameter eine Matrix-Struktur mittels der Anzahl von Reihen und Spalten. Zurückgegeben wird die zugehörige Matrix mit zufälligen Werten zwischen $-1/28$ und $1/28$. Der Wert 28 ergibt sich aus der Größe

des Eingabevektors (28x28 Pixel). Die übergebende Struktur hat die Form $[\mathrm{x}, \mathrm{y}]$, wobei x die Anzahl der Spalten und y die Anzahl der Zeilen angibt.
Bsp.: $s := [1, 2] \ \rightarrow << << x >> << y >> >>$ und $s := [2, 1] \ \rightarrow << << x \ y >> >>$

Sei nun W die Matrix der Gewichte und B die Matrix aller Vorbelastungen und $\mathbf{a}$ bezeichnet den Aktivierungsvektor der vorherigen Schicht, also deren Ausgabe (zu Beginn also die Pixel der Eingabe). Nach Gleichung 2.2 zur Berechnung einer Sigmoid-Ausgabe lässt sich nun folgende Formel aufstellen:

$$\mathbf{a}^{(l)} = S(W^{(l)} \cdot \mathbf{a}^{(l-1)} + B^{(l)}) \tag{3.1}$$

Hierbei bezeichnet $\mathbf{a}^{(l)}$ den Ausgabe-Vektor der aktuellen Schicht, welcher dann der nächsten Schicht weitergeleitet wird (feedforwarding). Nachfolgend sind die Implementierungen der Sigmoid-Funktionen sowie dem Feedforwarding zu sehen.

```
1  feedforward := procedure(a) {
2      for( i in {1..#mBiases} ) {
3          a := sigmoid( mWeights[i]*a + mBiases[i] );
4      }
5      return a;
6  };
7  sigmoid := procedure(z) {
8      return la_vector([ 1.0/(1.0 + exp(- z[i] )) : i in [1..#z] ]);
9  };
10 sigmoid_prime := procedure(z) {
11     s := sigmoid(z);
12     return la_matrix([ [ s[i] * (1 - s[i]) ] : i in [1..#s] ]);
13 };
```

1. `feedforward(a)`: Anwendung der Gleichung (3.1) auf alle Schichten des Netzwerkes angewandt. Zurückgegeben wird die resultierende Ausgabe jedes Neurons der letzten Schicht in vektorisierter Form.

2. `sigmoid(z)`: Diese Funktionen nimmt einen Vektor z und berechnet mit Hilfe der Sigmoid-Formel (siehe Kapitel 2.1) die Ausgabe der Neuronen in vektorisierter Form.

3. `sigmoid_prime(z)`: Für einen gegebenen Vektor z wird die Ableitung der Sigmoid-Funktion (nach Formel (2.3)) berechnet und in Form einer Matrix (Matrix-Form auf Grund späterer Berechnung mit `la_hadamard()`) zurückgegeben.

Die Feedforward-Funktion dient also dazu, die Eingabewerte durch das gesamte Netzwerk durchzureichen und die daraus resultierende Ausgabe zu ermitteln. Als nächstes wird der Algorithmus diskutiert, durch welchem es dem Netzwerk ermöglicht wird zu „lernen“. Hierfür wird der SGD-Algorithmus verwendet. Die Implementierung des SGDs in SetlX ist nachfolgend aufgezeigt und wird nun im Detail erläutert.

```
1  sgd := procedure(training_data, epochs, mini_batch_size, eta, test_data) {
2      if(test_data != null) {
3          n_test := #test_data;
4      }
5      n := #training_data;
6      for(j in {1..epochs}) {
7          training_data := shuffle(training_data);
8          mini_batches := [
```

```
9               training_data[k..k+mini_batch_size-1] : k in [1,mini_batch_size..n]
10          ];
11          for(mini_batch in mini_batches) {
12              update_mini_batch(mini_batch, eta);
13          }
14          if(test_data != null) {
15              ev := evaluate(test_data);
16              print("Epoch $j$: $ev$ / $n_test$");
17          }
18          else {
19              print("Epoch $j$ complete");
20          }
21      }
22  };
```

1. Zeile 1: Übergabeparameter der Funktion sind die Trainingsdatensätze (Liste von Tupeln $[\mathtt{x}, \mathtt{y}]$ mit `x` als Eingabewerten und `y` als gewünschtem Ergebnis), die Anzahl der Epochen (Integer-Wert), die Größe der Mini-Batches (Integer-Wert), die gewünschte Lernrate (Fließkomma-Wert) und den optionalen Testdatensätzen (äquivalenter Aufbau zu Trainingsdaten).
2. Zeile 6: Der nachfolgende Programmcode wird entsprechend der übergebenen Epochenanzahl mehrfach ausgeführt.
3. Zeile 7-10: Zuerst werden alle Trainingsdaten zufällig vermischt und anschließend Mini-Batches (also Ausschnitte aus dem Gesamtdatensatz) der vorher festgelegten Größe aus den Trainingsdaten extrahiert. Somit wird eine zufällige Belegung von Mini-Batches garantiert. Alle Mini-Batches werden in Listenform in der Variablen `mini_batches` gespeichert.
4. Zeile 11-13: Anschließend wird für jeden Mini-Batch aus `mini_batches` eine Iteration des Gradient Descent Algorithmus angewendet. Dies geschieht mit Hilfe der Funktion `update_mini_batches`, welche im nächsten Schritt ausführlicher erläutert wird. Zweck der Funktion ist es die Gewichte und Vorbelastungen des Netzwerkes mit Hilfe einer Iteration des SGD-Algorithmus anzupassen. Die Basis für diese Anpassung liefert der übergebene Mini-Batch und die Lernrate.
5. Zeile 14-20: Dieser Programmcode dient zur Ausgabe auf der Konsole und teilt dem Benutzer die aktuelle Anzahl an korrekt ermittelten Datensätzen der Trainingsdaten nach jeder Epoche mit. Hierfür wird die Hilfsfunktion `evaluate` verwendet, welche unter Berücksichtigung des aktuellen Netzwerkzustandes die Outputs ermittelt, welcher bei Eingabe der Testdaten durch das Netzwerk errechnet wurden (genaue Implementierung folgt). Sollten der `sgd`-Funktion keine Testdaten übergeben worden sein, so entfällt diese Ausgabe.

Die in der SGD-Funktion erwähnte Hilfsfunktion `update_mini_batches` dient dazu, auf einem gegebenen Testdatensatz (Mini-Batch) eine Iteration des Gradient Descent Algorithmus anzuwenden. Zur Berechnung des Gradienten wird Backpropagation genutzt.

```
1   update_mini_batch := procedure(mini_batch, eta) {
2       nabla_b := [ 0*mBiases[i] : i in {1..#mBiases}];
3       nabla_w := [ 0*mWeights[i] : i in {1..#mWeights}];
4       for([x,y] in mini_batch) {
5           [delta_nabla_b, delta_nabla_w] := backprop(x,y);
6           nabla_b := [ nabla_b[i] + delta_nabla_b[i] : i in {1..#nabla_b} ];
7           nabla_w := [ nabla_w[i] + delta_nabla_w[i] : i in {1..#nabla_w} ];
8       }
9       this.mWeights := [
```

```
10          mWeights[i] - eta/#mini_batch * nabla_w[i] : i in {1..#mWeights}
11      ];
12      this.mBiases := [
13          mBiases[i] - eta/#mini_batch * nabla_b[i] : i in {1..#mBiases}
14      ];
15  };
```

1. Zeile 1: Der Funktion wird ein Mini-Batch aus der SDG-Funktion in Listenform mitgegeben. Die jeweiligen Datensätze der Liste bestehen bestehen aus Tupeln der Form $[x, y]$, wobei x die Pixel des jeweiligen Zeichens darstellt und y der erwartete Wert des Zeichens ist.
2. Zeile 2-3: Hier werden die Variablen `nabla_b` und `nabla_w` als Listen mit 0-en initialisiert. Die Länge der Listen entspricht jeweils der Zeilenanzahl der Gewichts- und Vorbelastungs-Matrizen. `nabla_b` und `nabla_w` stehen für die Gradienten der Gewichte und Vorbelastungen des Netzwerkes.
3. Zeile 5: Auf jedes Tupel $[x, y]$ der mitgegebenen Testdaten wird nun der Backpropagation-Algorithmus angewendet. Dieser dient dazu den Gradienten der Kostenfunktion möglichst schnell und effizient zu berechnen. Die Implementierung von Backpropagation folgt im Anschluss.
4. Zeile 6-7: Die durch die Backpropagation ermittelten Gradienten für die Gewichte und Vorbelastungen werden in den entsprechenden Variablen gespeichert.
5. Zeile 9-14: Nachdem die Gradienten durch jeden Datensatz des Mini-Batches angepasst wurden, werden am Ende der Funktion nun die Gewichte und Vorbelastungen des Netzwerkes entsprechend des Ergebnisses angepasst. Hierfür werden folgende Formeln des SGD-Algorithmus verwendet:

$$W' = W - \frac{\eta}{m} \cdot \nabla W \qquad\qquad B' = B - \frac{\eta}{m} \cdot \nabla B \tag{3.2}$$

   Hierbei bezeichnet W die Gewichtsmatrix und B die Vorbelastungsmatrix des Netzwerkes. Die Lernrate wird durch η dargestellt und m bezeichnet die Größe der betrachteten Testdaten. Die Lernrate wird durch den Benutzer vorgegeben und der Funktion als Parameter übergeben. m kann durch die Größe des Mini-Batches ermittelt werden.

Im nächsten Abschnitt wird die Implementierung des Backpropagation Algorithmus vorgestellt. Dieser dient dazu den Gradienten der Gewichte und Vorbelastungen zu berechnen, damit das Netzwerk anhand der Testdatensätze lernen kann. Zur Erinnerung sind hier noch einmal die vier grundlegenden Formeln des Algorithmus erwähnt (siehe auch Kapitel 2.3):

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'\big(\mathbf{z}^{(L)}\big) \tag{3.3}$$

$$\boldsymbol{\varepsilon}^{(l)} = \Big(\big(W^{(l+1)}\big)^{\top} \cdot \boldsymbol{\varepsilon}^{(l+1)}\Big) \odot S'\big(z^{(l)}\big) \quad \text{für alle } l \in \{2, \cdots, L-1\}. \tag{3.4}$$

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \text{für alle } l \in \{2, \cdots, l\} \tag{3.5}$$

$$\nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \big(\mathbf{a}^{(l-1)}\big)^{\top} \quad \text{für alle } l \in \{2, \cdots, l\} \tag{3.6}$$

Nachfolgend ist die eigentlichen Umsetzung in SetlX mit einigen Erläuterungen zu sehen.

```
1  backprop := procedure(x,y) {
2      nabla_b := [ 0 : i in {1..#mBiases}];
3      nabla_w := [ 0 : i in {1..#mWeights}];
4      activation := x;
5      activations := [ la_matrix(x) ];
6      len_act := #activations;
7      activations += [0 : i in [1..#mBiases]];
8      zs := [0 : i in [1..#mBiases]];
9      for(i in {1..#mBiases}) {
10         zs[i] := mWeights[i] * activation + mBiases[i];
11         activation := sigmoid(z[i]);
12         activations[i + len_act] := la_matrix(activation);
13     }
14     cdm := la_matrix( cost_derivative(activations[-1], y) );
15     epsilon := la_hadamard( cdm, sigmoid_prime(zs[-1]));
16     lb := #nabla_b;
17     lw := #nabla_w;
18     nabla_b[lb] := epsilon;
19     nabla_w[lw] := epsilon * activations[-2]!;
20     for( l in {2..mNumLayers-1} ) {
21         sp := sigmoid_prime(zs[-l]);
22         epsilon := la_hadamard( mWeights[-l+1]! * epsilon, sp );
23         nabla_b[lb-l+1] := epsilon;
24         nabla_w[lw-l+1] := epsilon * activations[-l-1]!;
25     }
26     return [nabla_b, nabla_w];
27 };
```

1. Zeile 1: Der Funktion werden Datensätze in Listenform mitgegeben. Die Datensätze bestehen aus Tupeln der Form $[x, y]$, wobei x die Pixel des jeweiligen Zeichens darstellt und y der tatsächliche Wert des Zeichens ist.

2. Zeile 2-3: Initialisierung der Gradienten-Variablen `nabla_b` und `nabla_w` mit 0-en.

3. Zeile 4-7: Die Variable `activiation` enthält den aktuellen Eingabevektor der vorherigen Schicht und wird für das Feedforwarding benötigt. Zu Beginn der Funktion entspricht `activiation` dem Pixel-Vektor der Eingabe, also x. `activiations` speichert die Aktivierungsvektoren aller Schichten. Der erste Wert der Liste wird mit dem Eingabevektor belegt. Aus Performance-Gründen wird die Variabel wieder mit 0-en initialisiert, um ein späteres Anhängen an die Liste zu vermeiden (einfügen, statt anhängen).

4. Zeile 8: `zs` bezeichnet die Lister aller z-Vektoren und wird mit 0-en initialisiert. Ein z-Vektor beinhaltet alle in der jeweiligen Schicht durch die entsprechenden Werte (Gewichte und Vorbelastungen) gewichteten Eingaben. Dies entspricht also der späteren Eingabewert der Sigmoid-Funktion. Zur Veranschaulichung der z-Vektoren und deren Bedeutung dient folgende Formel:

$$\mathbf{a}^{(l+1)} = \sigma(\mathbf{z}) \tag{3.7}$$

   Hierbei bezeichnet $\mathbf{a}^{(l+1)}$ den Aktivierungsvektor der nächsten Schicht.

5. Zeile 10-13: Für jede Schicht des Netzwerkes wird der entsprechende z-Vektor entsprechend der Gleichungen (3.1) und (3.7) berechnet und der Liste `zs` hinzugefügt. Mit Hilfe des aktuellen z-Vektors kann der Aktivierungsvektor jeder Schicht berechnet werden. Alle Aktivierungsvektoren

des Netzwerkes werden pro Schicht in `activiations` abgelegt. Um später mit den Aktivierungsvektoren besser rechnen zu können, werden die vektorisierten Aktivierungen in Matrixform in `activiations` abgelegt.

6. Zeile 15-16: Diese Zeilen stellen die Implementierung der ersten Gleichung des Backpropagation-Algorithmus (3.3) dar. Hierbei bezeichnet `epsilon` den Ausgabefehler $\varepsilon^{(L)}$ des Netzwerkes. Um diesen berechnen zu können, wird die Hilfsfunktion `cost_derivate` aufgerufen, welche den erwarteten Ausgabevektor y von dem letzten Aktivierungsvektor (also die Ausgabe des Netzwerkes) subtrahiert. Da die Hadamard-Funktion von SetlX lediglich Matrizen als Parameter akzeptiert und `cost_derivate` einen Vektor berechnet, muss dieser noch mittels `la_matrix` in eine Matrix umgewandelt werden.

7. Zeile 17-18: Die Variablen `lb` und `lw` bezeichnen jeweils die Länge der Gewichts- und Vorbelastungslisten. Diese Variablen werden im Anschluss benötigt, da es in SetlX zwar möglich ist eine Liste oder eine Matrix von hinten mittels negativem Index (z.B. $a[-1]$) zu lesen, allerdings nicht zu beschreiben.

8. Zeile 19-20: Berechnung der Gradienten der Gewichte und Vorbelastung der Ausgabeschicht mittels der Formeln (3.5) und (3.6).

9. Zeile 21-24: Dieser Code beschreibt die Berechnung der Gradienten für alle Schichten zwischen der zweiten und der Vorletzten in rückwärtiger Reihenfolge (also in unserem Netzwerkaufbau gilt für die Schleife: $l \in 2$). Zunächst wird wieder der Ausgabefehler $\varepsilon^{(L)}$ berechnet. Dies geschieht in Zeile 24 nach Formel (3.4). Da wir in der Schleife mit negativen Indizes arbeiten, entspricht `epsilon` in jeder Iteration der nächsthöheren Schicht. Zeile 25 und 26 entsprechen den Formeln (3.5) und (3.6) und passen die Gradientenvariablen entsprechend an. Hierbei ist zu beachten, dass der Ausdruck "`lb - l + 1`" dem Ausdruck "$-$`l`" entspricht. Da wie erwähnt ein Schreiben von Matrizen und Arrays mit negativen Indizes nicht möglich ist, musste auf die Werte mit einem positiven Index zugegriffen werden.

10. Zeile 28: Die Funktion liefert als Rückgabeparameter die entgültigen Gradienten der Netzwerkgewichte und -vorbelastungen, welche anschließend in der SGD-Funktion für den Gradientenabstieg verwendet werden.

Als Letztes wird die Funktion `evaluate` diskutiert, welche in der sgd-Funktion aufgerufen wurde und dazu dient die Anzahl der vom Netzwerk korrekt ermittelten Datensätze zu berechnen. Die Funktion ist durch folgenden Code gegeben:

```
1  evaluate := procedure(test_data) {
2      test_results := [[argmax(feedforward(x)) - 1, y]: [x, y] in test_data];
3      return #[1 : [x,y] in test_results | x == y];
4  };
5  argmax := procedure(x) {
6      [maxValue, maxIndex] := [x[1], 1];
7      for (i in [2 .. #x] | x[i] > maxValue) {
8          [maxValue, maxIndex] := [x[i], i];
9      }
10     return maxIndex;
11 };
```

1. Zeile 1: Der Funktion werden Datensätze in Listenform mitgegeben. Die Datensätze bestehen aus Tupeln der Form $[x, y]$, wobei x die Pixel des jeweiligen Zeichens darstellt und y der Wert des Zeichens ist.

2. Zeile 2: `test_results` speichert die vom Netzwerk ermittelte Ausgabe, sowie die tatsächliche Ausgabe in Tupelform für jeden Datensatz. Mit Hilfe der bereits besprochenen Feedforward-Funktion wird zunächst die vektorisierte Ausgabe des Netzwerkes für den jeweiligen Datensatz berechnet. Anschließend wird mit Hilfe der `argmax()`-Funktion der Index des maximalen Wertes im Vektor ermittelt. Die ermittelte Ziffer ergibt sich nun aus dem Index subtrahiert mit 1, da die Ziffern mit 0 beginnend im Ausgabevektor gespeichert sind. In die Variable `test_results` wird letztendlich der errechnete Wert sowie der tatsächliche Wert (y) gespeichert.
3. Zeile 3: Die Funktion gibt im Anschluss die Anzahl aller übereinstimmenden Ergebnisse in `test_results` zurück.
4. Zeile 5-11: Die Funktion `argmax()` ermittelt in einem Vektor oder einer Liste den Index des größten darin enthaltenen Wertes. Hierbei wird mit einer Schleife über die komplette Liste oder den kompletten Vektor iteriert und der aktuell höchste Wert mit entsprechendem Index in den Variablen `maxValue` und `maxIndex` gespeichert. Zurückgegeben wird der somit gefundene Index.

Eine vorgefertigte Prozedur zur Initialisierung des benötigten Netzwerkes mit Beispielparametern befindet sich in der Datei `start.stlx`, welche mit dem Befehl

```
setlx start.stlx
```

über die Konsole gestartet werden kann.

3.3 Animation

In diesem Abschnitt der Arbeit wird auf die Implementierung der grafischen Ausgabe des neuronalen Netzwerks zur Erkennung von handgeschriebenen Zahlen in SetIX eingegangen. Das Ziel ist die grafische Aufbereitung der Daten aus dem Hauptprogramm. Es bildet hierbei die folgenden Informationen ab:

- Aufbau des neuronalen Netzwerks.
- Betrachtung des Untersuchungsbereich eines Neurons in der verborgenen Schicht.
- Betrachtung des Untersuchungsbereich aller Neuronen der verborgenen Schicht.
- Betrachtung des Untersuchungsbereich aller Neuronen für eine Eingabe $\mathbf{x}$ aus den Testdaten.

Die folgenden Befehle ermöglichen die einzelnen Funktionalitäten in der Anwendung anzusteuern:

Eingabebefehl	**Erklärung**
$\mathbf{0}$	Animation für den Aufbau des Netzwerks mit den unterschiedlichen Schichten z.B. $[784, 30, 19]$
$\mathbf{1 - 30}$	Animation für den Untersuchungsbereich des angesteuerten Neurons n im Hidden-Layer mit $n \in \{1..30\}$ des entsprechenden Netzwerk
$\mathbf{101} - (\texttt{mTestDataSize} + \mathbf{100})$	Animation für den Untersuchungsbereich aller Neuronen für eine gegeben Zahl x aus den Testdaten
$\mathbf{1000}$	Animation für den Untersuchbereichs der einzelnen Neuronen

Mit dem **Eingabebefehl 0** wird dynamisch der Aufbau des angewendeten neuronalen Netzwerks dem Anwender repräsentiert (siehe Abb. 3.2). Wie bereits im Theorieteil dieser Arbeit beschrieben, besteht das neuronale Netzwerk aus einer Eingabeschicht, einer oder mehrerer verborgener Schichten und einer

Ausgabeschicht.
Die Eingabe für das Netzwerk ist eine handgeschriebene Zahl mit der Bildgröße von 28×28 Pixeln. Jedes Pixel stellt eine Eingabe des neuronalen Netzwerks dar, wobei 784 Eingaben aus der Bildgröße resultieren (siehe Abb. 3.1). Diese werden für die Animation ebenfalls wie das Ursprungsbild in der Form eines Quadrats abgebildet. Weiterhin ist der Aufbau und die Verbindungen zwischen den Neuronen der verborgenen Schicht und der Ausgabeschicht dargestellt.

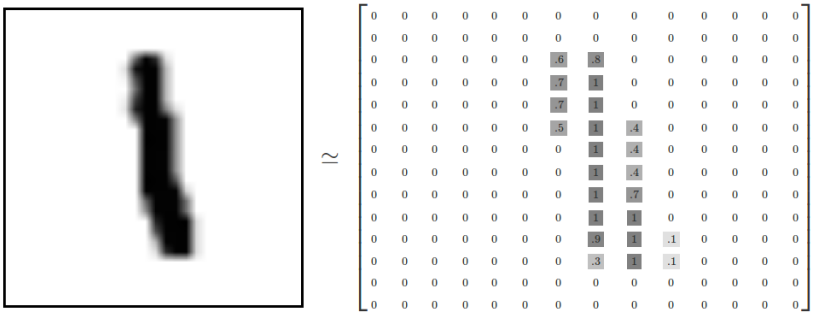

Abbildung 3.1: Eingabe in das neuronale Netzwerk.

Die **Eingabebefehle 1 - 30** betrachten den Untersuchungsbereich eines einzelnen Neurons der verborgenen Schicht, welches hierbei bei der Auswahl farblich hervorgehoben wird. Der Untersuchungsbereich ist farblich in rote und blaue Bereiche aufgeteilt. Pixel mit einer roten Färbung, haben eine hohe Gewichtung. Dahingegen weisen Pixel mit einer blauen Färbung einen geringen Gewichtungsfaktor auf. Blassere Farben haben hierbei eine geringere Merkmalsausprägung. Die Animation weist hierbei den folgenden Aufbau auf (siehe Abb. 3.3).
Über den **Eingabebefehl 1000** wird der Untersuchungsbereich von allen Neuronen in einer Übersicht abgebildet. Dies bedeutet, dass dieser Befehl die Ausgaben von den Eingabebefehlen 1-30 in sich zusammenfasst (siehe Abb. 3.4). Die grafische Ausgabe ermöglicht dem Anwender die Untersuchungsbereiche verschiedener Neuronen zu vergleichen und gegenüberzustellen. Konzentriert sich ein Neuron im Vergleich zu allen anderen nur auf einen bestimmten Bereich? Nimmt ein Neuron für die Erkennung einer handgeschrieben Zahl (z.B. 6) einen größeren Einfluss wie andere Neuronen?

Der letzte **Eingabebereich 101-Datensatzgröße** ermöglicht den Untersuchungsbereich der Neuronen hinsichtlich einer Eingabe aus den Trainingsdaten zu betrachten. Die farbliche Bedeutung für den Untersuchungsbereich ist wie folgt definiert:

Farbe	Erklärung
blau, grün	Bereich mit einer geringen Relevanz
rot, gelb	Bereich mit einer hohen Relevanz

Weiterhin ist wird das Untersuchungsobjekt, die Eingabe, sowie die Ausgabe mit dem berechneten Ergebnissen in der Animation abgebildet (siehe Abb. 3.5). Die berechneten Ergebnisse ermöglichen dem Nutzer zu überprüfen, ob die Klassifizierung durch das neuronale Netzwerk richtig vorgenommen wurde und welche Neuronen in bei der Ergebnisfindung höheren Einfluss nehmen. Die Berechnung unter jedem Untersuchungsbereich ist durch die Sigmoid-Funktion bestimmt.

An dieser Stelle soll angemerkt sein, dass aufgrund der Verwendung von nur einer verborgenen Schicht in diesem Versuchsaufbau und der Anzahl von nur 30 Iterationen keine Erkenntnisse anhand der grafischen Ausgaben getroffen werden können. Hierzu müsste ein *Multilayer-Network* mit einer wesentlichen höheren Anzahl von Iterationen implementiert werden.

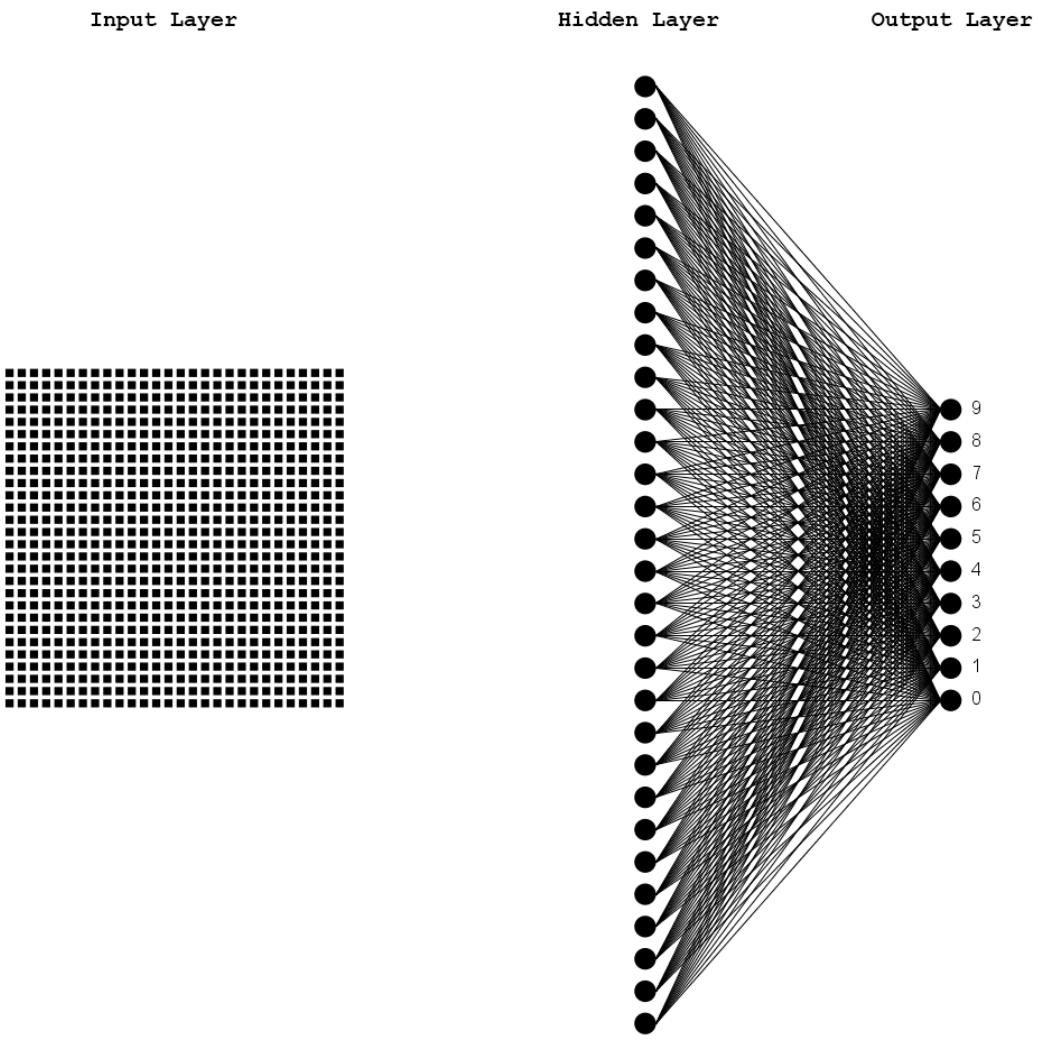

Abbildung 3.2: Default Animation, welche über die Eingabe mit dem Wert 0 aufgerufen wird.

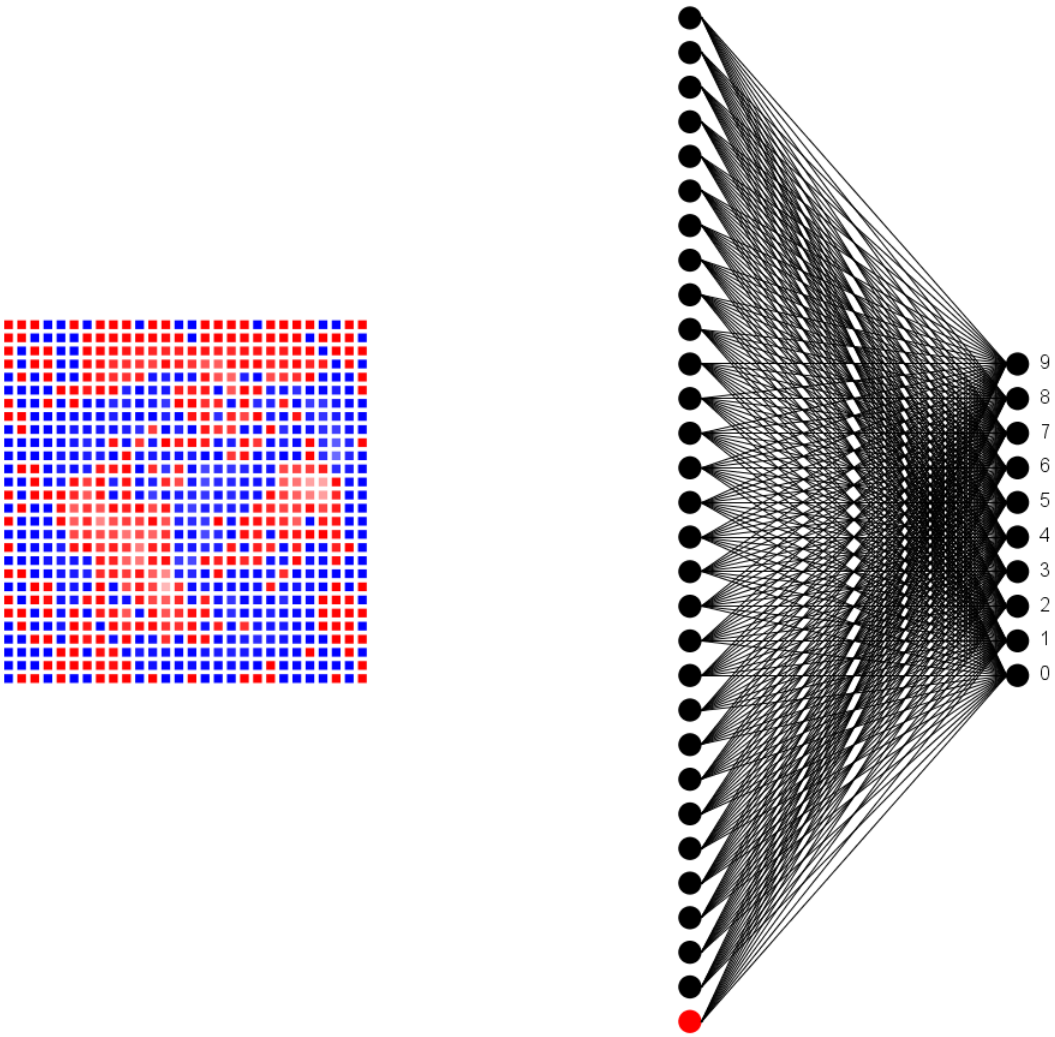

Abbildung 3.3: Untersuchungsbereich eines einzelnen Neurons der verborgenen Schicht.

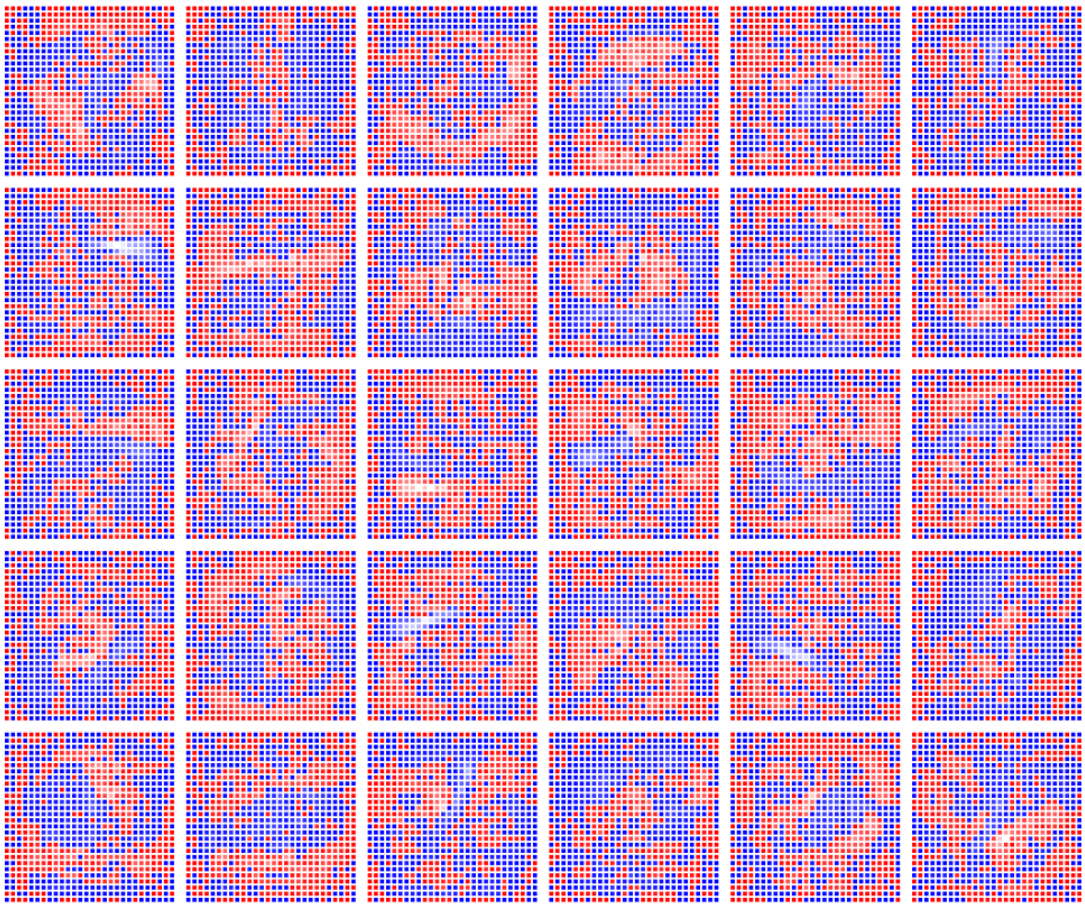

Abbildung 3.4: Untersuchungsbereich aller Neuronen der verborgenen Schicht.

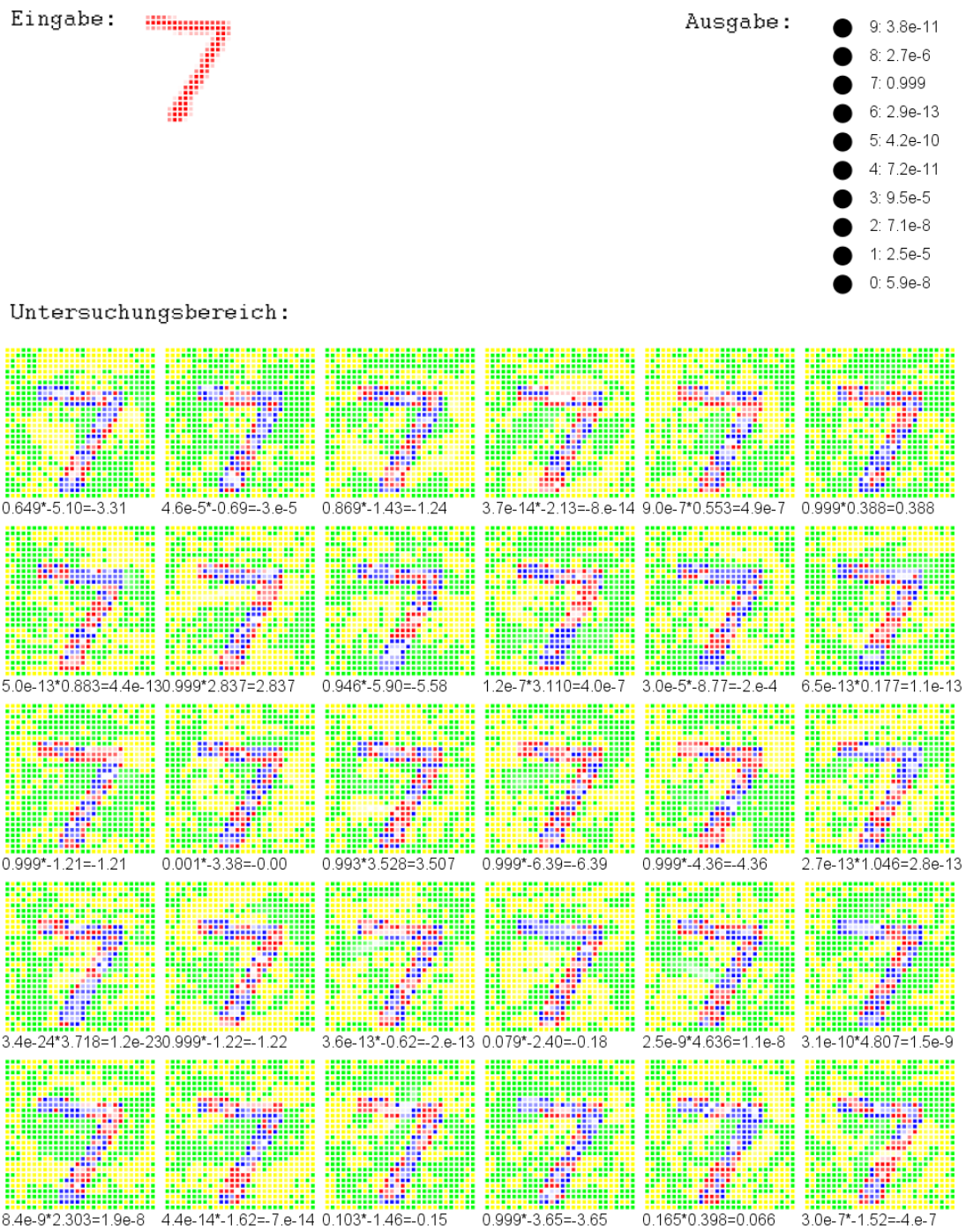

Abbildung 3.5: Untersuchungsbereich aller Neuronen der verborgenen Schicht hinsichtlich einer Eingabe.

Kapitel 4

Fazit und Ausblick

4.1 Auswertung des Ergebnisses

Die Umsetzung der Handschriftenerkennung von Ziffern mittels einem neuronalen Feedforward-Netz wurde erfolgreich in SetlX implementiert. Die Erkennungsrate des Netzwerkes liegt ungefähr zwischen 94 und 96 Prozent. Die nachfolgende Ausgabe entspricht einem Durchlauf der `start.stlx`-Datei. Hierbei wurden 10.000 Testdaten und 60.000 Trainingsdaten der MNIST-Datensätze verwendet. Die Epochenanzahl beträgt 30.

```
D:\DHBW\Neural-Network-in-SetlX\setlx>setlx start.stlx
Reading file:   mnist_test.csv
Image 10000 of 10000 imported
End reading:    mnist_test.csv
Reading file:   mnist_train.csv
Image 10000 of 60000 imported
Image 20000 of 60000 imported
Image 30000 of 60000 imported
Image 40000 of 60000 imported
Image 50000 of 60000 imported
Image 60000 of 60000 imported
End reading:    mnist_train.csv
Create Network
Init Network
Start SGD
Epoch 1: 9427 / 10000
Epoch 2: 9446 / 10000
Epoch 3: 9534 / 10000
Epoch 4: 9489 / 10000
Epoch 5: 9581 / 10000
Epoch 6: 9548 / 10000
Epoch 7: 9559 / 10000
Epoch 8: 9602 / 10000
Epoch 9: 9543 / 10000
Epoch 10: 9605 / 10000
Epoch 11: 9571 / 10000
Epoch 12: 9566 / 10000
Epoch 13: 9603 / 10000
Epoch 14: 9603 / 10000
Epoch 15: 9598 / 10000
```

```
Epoch 16: 9609 / 10000
Epoch 17: 9587 / 10000
Epoch 18: 9604 / 10000
Epoch 19: 9609 / 10000
Epoch 20: 9616 / 10000
Epoch 21: 9591 / 10000
Epoch 22: 9597 / 10000
Epoch 23: 9605 / 10000
Epoch 24: 9612 / 10000
Epoch 25: 9588 / 10000
Epoch 26: 9600 / 10000
Epoch 27: 9598 / 10000
Epoch 28: 9605 / 10000
Epoch 29: 9582 / 10000
Epoch 30: 9629 / 10000
Time needed:    1255425ms
```

4.2 Performance der SetlX Implementierung

Wie im vorherigen Kapitel zu sehen ist, beträgt die Durchlaufzeit des Programmes mit allen Datensätzen und 30 Epochen (Messzeit beginnt ab Aufruf der SGD-Funktion des Netzwerkes) 1255425 Millisekunden. Im Schnitt wurden ca. 1350000 Millisekunden benötigt, was 22,5 Minuten entspricht. Alle Messungen wurden auf einem Computer mit folgenden Hardwarekomponenten durchgeführt (nur relevante Komponenten sind aufgezählt):

- CPU: Intel Core i7-4720HQ @ 2.60GHz
- Arbeitsspeicher: 16GB RAM
- Grafikkarte: NVIDIA GeForce GTX 960M

Auffällig bei der Analyse der Laufzeit ist, dass die Durchlaufzeit der SetlX-Implementierung weit über der der Python-Implementierung liegt. SetlX benötigte für den Durchlauf der 30 Epochen auf dem selben Computer im Schnitt 5,4 mal länger als die Python-Version des Programms. Basierend auf einer Reihe Performance-Tests, die auf Grund der langen Durchlaufzeiten durchgeführt wurden, stellte sich heraus, dass die Matrizen-Multiplikation in SetlX wesentlich zeitintensiver ist als in der Numpy-Bibliothek von Python. Der Faktor hierbei beträgt circa 6,5.
Eine Vermutung, wieso die SetlX Matrizen-Multiplikation wesentlich langsamer ist, ist dass in Python die Berechnung auf die Grafikkarte des Computers ausgelagert wird. Da die Programmiersprache Java (welche die Grundlage von SetlX bildet) plattformunabhängigkeit verspricht, wäre es möglich, dass hier keine Auslagerung auf die Grafikkarte stattfindet. Allerdings sind das nur Vermutungen und müssten auf Ebene der JAMA-Bibliothek (wird von SetlX zur Matrizen-Multiplikation verwendet) in Java, sowie der Numpy-Bibliothek in Python überprüft werden.

Die gesamte Auswertung sowie alle hierfür verwendeten Programme sind im Verzeichnis

/setlx/testing/

im GitHub-Repository zu finden. Die Datei `setlx_performance_evaluation.pdf` bietet eine Übersicht und Beschreibungen zu den durchgeführten Tests.

Literaturverzeichnis

[1] Buduma, Nikhil: *Fundamentals of deep learning*. The McGraw-Hill Companies, 2017, ISBN 978-1491925614.

[2] Nielsen, Michael: *Neural Networks and Deep Learning*. http://neuralnetworksanddeeplearning.com/index.html.

[3] Stroetmann, Karl: *Artifical Intelligence*. https://github.com/karlstroetmann/Artificial-Intelligence.

[4] Stroetmann, Karl / Herrmann, Tom: *SetlX — A Tutorial*. http://download.randoom.org/setlX/tutorial.pdf.

[5] Le, Quoc V. / Schuster, Mike: *A Neural Network for Machine Translation, at Production Scale*. https://research.googleblog.com/2016/09/a-neural-network-for-machine.html.

[6] DeepMind Technologies Limited: *The story of AlphaGo so far*. https://deepmind.com/research/alphago/.

[7] Prisma Labs, Inc.: *AI Powered Art Styles*. https://prisma-ai.com/.