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:	-, TINF14AI-BI
Matrikelnummer, Kurs:	-, 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 habe.

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 12. Mai 2017

Lucas Heuser	Johannes Hill

Inhaltsverzeichnis

Abbildungsverzeichnis

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“.[1] 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[2] verwendet. 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.[3]

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 [ToDo: Verweis auf Seite]

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.

[1] Die Vorlesungsunterlagen sind unter https://github.com/karlstroetmann/Artificial-Intelligence zu finden
[2] https://randoom.org/Software/SetlX
[3] http://neuralnetworksanddeeplearning.com/ [ToDo: Verweis auf Seite]

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.[4]
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.[5]
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.[6]

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.

[4] https://research.googleblog.com/2016/09/a-neural-network-for-machine.html
[5] https://deepmind.com/research/alphago/
[6] http://prisma-ai.com/ [ToDo: Alle ins Literaturverzeichnis]

Kapitel 2

Theorie

2.1 Neuronales Netzwerk

In diesem Abschnitt der Arbeit wird der Aufbau eines neuronalen Netzwerks näher betrachtet und entsprechend auf die Terminologie in diesem Bereich eingegangen.
Neuronale Netzwerke sind den biologischen Neuronen nachempfunden und setzen sich daher aus Vielzahl von Neuronen zusammen. Darüber hinaus ist ein Netzwerk in mehrere Schichten untergliedert (siehe Abb.2.1). So wird die erste Schicht auf der linken Seite auch als *Eingabeschicht* und alle korrespondierenden Eingabeknoten als Eingabeneuron bezeichnet. Die letzte Schicht, die sogenannte *Ausgabeschicht*, beinhaltet dagegen alle *Ausgabeneuronen*. Alle Schichten zwischen der Eingabe und der Ausgabe tragen die Bezeichnung des *Hidden Layer*, wobei ein Netzwerk mehrere dieser Schichten aufweisen kann. In der folgenden Grafik ist ein 4-Layer-Netzwerk abgebildet, das zwei *Hidden Layer* besitzt. Diese mehrschichtigen Netzwerke werden ebenfalls als *Multilayer Perceptrons* oder *MLPs* bezeichnet, obwohl das Netzwerk sich aus Sigmoid-Neuronen zusammensetzt.

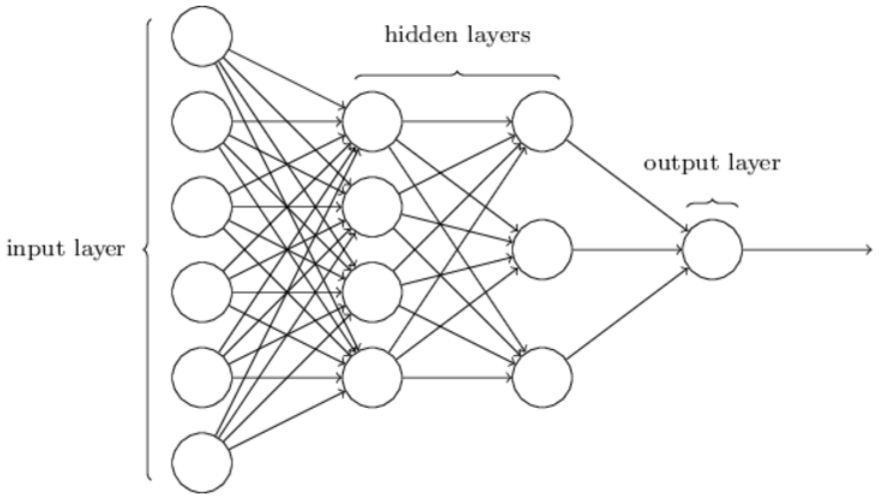

Abbildung 2.1: Aufbau des neuronalen Netzwerks hinsichtlich der einzelnen Layer.

Für die Zusammensetzung von Eingabe- und Ausgabeschichten in einem Netzwerk, fällt die Betrachtung auf die Erkennung einer handgeschrieben "9". Als Eingabewerte dienen dem Netzwerk die Intensitäten der Bildpixel. Liegt ein Graustufenbild der Größe von 64x64 Pixeln vor, leiten sich daraus 4096 Eingabeknoten mit skalierten Intensitätswerten zwischen 0 und 1 ab. Die Ausgabeschicht beinhaltet dagegen nur ein Neuron, um eine entsprechende Klassifizierung der "9" vornehmen zu können.

$\texttt{output} < 0.5 \quad \rightarrow \quad$ "Eingabebild ist keine 9"

$\texttt{output} \geq 0.5 \quad \rightarrow \quad$ "Eingabebild ist eine 9"

Im Vergleich hierzu ist der Aufbau der Hidden-Layer nicht durch irgendwelche Regeln ableitbar. Zum Einsatz kommen Heuristiken, die das Verhalten eines Netzwerkes bestimmen, welches ausgehend vom Experiment erwartet und gewünscht wird. Zum Beispiel kann untersucht werden, wie die Lernrate des Netzwerks sich im Verhältnis zu der Anzahl an Hidden-Layer verhält.

Bisher fiel die Betrachtung in dieser Arbeit auf neuronale Netze, bei denen die Ausgabe einer Schicht die Eingabe in der nächsten Schicht darstellte. Solche Netzwerke werden auch *Feedforward Neural Networks* bezeichnet. Hierunter ist das nicht Auftreten von Schleifen zu verstehen - sprich, Informationen werden im Netzwerk immer von Layer n zu Layer $n+1$ übergeben. Somit kann verhindert werden, dass das Netzwerk in gewissen Fällen bei der Eingabe der Sigmoid-Funktion σ von dessen Ausgabe abhängig ist.
Ebenfalls gibt es Netzwerke bei denen sogenannte *Feedback Loops* möglich sind. In diesem Fall handelt es sich um *Recurrent Neural Networks*. Die Idee hinter diesem Modell ist, dass bestimmte Neuronen über einen definierten Zeitraum aktiv sind bevor sie inaktiv werden. Dies kann andere Neuronen anregen, ebenfalls über einen gewissen Zeitraum aktiv zu sein und eine entsprechende Kettenreaktion auslösen (Kaskade). Schleifen stellen in diesem Modell kein Problem dar, da die Ausgabe eines Neurons erst zu einem späteren Zeitpunkt die Eingabe beeinflusst.
Stellt man diese Arten von neuronalen Netzwerken gegenüber, so lässt sie zum heutigen Zeitpunkt die Aussage treffen, dass Feedback Neural Networks weniger Verbreitung finden. Dies ist begründet in der Leistungsfähigkeit der Lernalgorithmen. Jedoch sollte an dieser Stelle berücksichtigt werden, dass mittels Feedback Neural Networks bestimmte Probleme mit einem geringeren architektonischen Aufwand gelöst werden können. Darüber hinaus bildet der komplexere logische Aufbau eines solchen Netzwerks das menschliche Verhalten besser ab.

2.2 Perceptrons

Ein Perceptron ist ein mathematisches Modell zur Abbildung eines künstliches Neurons in einem Netzwerk. Es wird für die Entscheidungsfindung herangezogen, indem verschiedene Aussagen abgewägt werden. Hierbei nimmt das Perceptron eine Menge von Eingaben x_n mit $n \in \{1, \cdots, m\}$ und berechnet einen einzigen binären Ausgabewert (siehe Abb. 2.2).

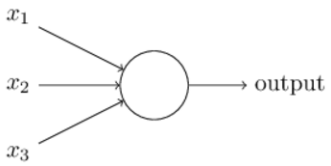

Abbildung 2.2: Percetron mit den Eingaben x_1, x_2, x_3 und der Ausgabe output.

Für jeden Eingabewert x_n wird eine Gewichtung w_n mit $n \in \{1, \cdots, m\}$ zugeordnet. Der Ausgabewerte output wird mittels der gewichteten Summe $\sum_j w_j x_k$ und einem definierten Grenzwert threshold bestimmt.

$$\texttt{output} := \begin{cases} 0 & \text{falls} \quad \sum_j w_j x_j \leq \texttt{threshold} \\ 1 & \text{falls} \quad \sum_j w_j x_j > \texttt{threshold} \end{cases} \tag{2.1}$$

Der Aufbau des Netzwerks leitet sich aus den unterschiedlichen Modellen der Entscheidungsfindung ab und wird mit Hilfe der Perceptrons abgebildet (siehe Abb. 2.3). Eine Entscheidungsmöglichkeit wird hierbei durch das Perceptron dargestellt. Weiterhin wird eine Spalte von Perceptrons als Schicht

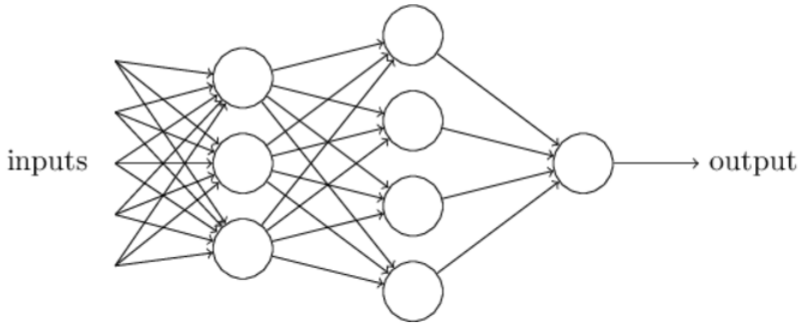

Abbildung 2.3: Unterschiedliche Möglichkeiten der Entscheidungsfindung.

bezeichnet. Die erste Schicht fällt Entscheidungen auf Gewichtung der Eingabewerte, indem er diese abwägt. Jedes Perceptron der zweiten Schicht hingegen, wägt für die Entscheidungsfindung die Resultate der ersten Schicht ab. Ein Perceptron auf der zweiten Schicht kann somit eine Entscheidung auf einer abstrakteren und komplexeren Ebene durchführen. Auf diese Weise kann sich ein vielschichtiges Netzwerk von Perceptrons in ein anspruchsvolles Modell zur Entscheidungsfindung entwickeln.

Im folgenden wird die mathematische Beschreibung von Perceptrons vereinfacht, indem Änderungen an der Notation für $\sum_j w_j x_j >$ `threshold` vorgenommen werden. Für die Beschreibung der Summe $\sum_j w_j x_j$ werden die Vektoren $\mathbf{w}$ und $\mathbf{x}$ eingeführt, wodurch sich die Schreibweise $\mathbf{w} \cdot \mathbf{x} \equiv \sum_j w_j x_j$ ergibt. Des Weiteren wird der `threshold` auf die andere Seite der Ungleichung gezogen und erhält die Bezeichnung *Vorbelastung*, $b \equiv -$`threshold`.

$$\texttt{output} := \begin{cases} 0 & \text{falls} \quad \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{falls} \quad \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{2.2}$$

Dieses mathematische Verhalten wird durch die folgende Stufenfunktion verdeutlicht (siehe Abb. 2.4).

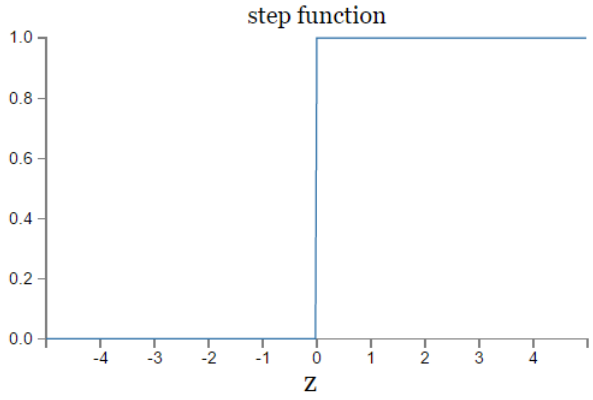

Abbildung 2.4: Sigmoid Funktion $\sigma(z)$.

2.3 Arbeitsweise von Perceptrons

Ein einzelnes Perceptron kann für die Darstellung unterschiedlicher boolescher Funktionen genutzt werden. Angenommen die Definition der booleschen Werte 1 (`true`) und 0 (`false`) liegt zugrunde, so ist es möglich alle logische Operationen z.B. `AND`, `OR` und `NAND` über ein Perceptron abzubilden.

Im weiteren Verlauf soll die Umsetzung eines `NAND` Gatters mit Perceptrons betrachtet werden. `NAND` Gatter spielen in der Digitaltechnik eine bedeutende Rolle, da sie alle logischen Verknüpfungen und somit auch komplexere Schaltungen (z.B. Addierer, Multiplexer) zusammenstellen lassen. Fällt die Betrachtung auf ein Perceptron mit 2 Eingaben deren Gewichtung jeweils den Wert -2 aufweisen und eine Vorbelastung von 3, so ergibt sich folgende Abbildung (siehe Abb. 2.5).

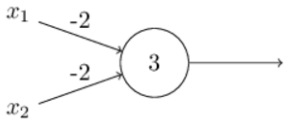

Abbildung 2.5: Percetron mit zwei Eingaben -2 und einem Bias von 3.

Um das Verhalten des Perceptrons weiter zu untersuchen werden zum einen die Eingaben x_1, x_2 mit dem Wert 0 und zum anderen mit dem Wert -1 belegt.

$$(-2) * 0 + (-2) * 0 + 3 \quad = \quad 3 \quad \geq \quad 0 \quad \rightarrow \quad \texttt{output} := 1$$

$$(-2) * 1 + (-2) * 1 + 3 \quad = \quad -1 \quad < \quad 0 \quad \rightarrow \quad \texttt{output} := 0$$

Das vorliegende Perceptron ist in der Lage das Verhalten eines `NAND` Gatters nachzubilden. Somit kann auf dessen Basis auch ein Halbaddierer, der die Addition von zwei Bits x_1 und x_2 durchführt, implementiert werden. Für die Berechnung wird die bitweise Summe $x_1 \oplus x_2$ gebildet, wobei ein `carrybit` den Wert 1 annimmt, sobald x_1 und x_2 den Wert 1 aufweisen.

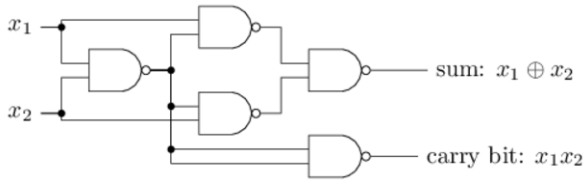

Abbildung 2.6: Halbaddierer mit den Eingaben x_1 und x_2.

Um ein gleichwertiges Netzwerk abzuleiten, werden die `NAND` Gatter durch Perceptrons mit jeweils 2 Eingaben ersetzt. Hierbei weisen die Gewichtungen w_1, w_2 den Wert -2 und die Vorbelastung b den Wert 3 auf.

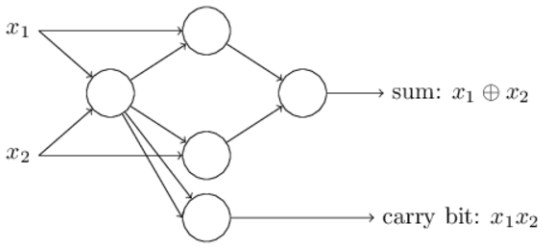

Abbildung 2.7: Halbaddierer-Aufbau mit Perceptrons.

In einem weiteren Schritt wird die Abbildung eines `NAND` Gatter mit Perceptrons vereinfacht. Dazu werden mehrere Eingänge eines Perceptrons zu einem zusammengefasst, weshalb aus den zwei Eingaben -2 der Wert -4 resultiert. Ebenfalls werden die Eingaben in der Eingabeschicht des neuronalen Netzwerkes zusammengefasst, wobei jedoch durch diese Notation eine Eingabe nicht mit einem Perceptron gleichzustellen ist. Das vorliegende Netzwerk mit Perceptrons entspricht somit dem Aufbau eines Halbaddierers (siehe Abb. 2.8).

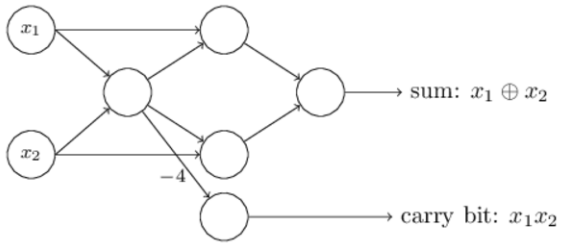

Abbildung 2.8: Vereinfachter `NAND` Gatter Aufbau mit Perceptrons.

Wie im obigen Beispiel aufgezeigt, lassen sich mittels Perceptrons unterschiedliche Berechnungen durchführen. Auf diese Weise können implementierte Lernalgorithmen Gewichtungen sowie die Vorbelastung automatisch durch entsprechende Stimuli im Netzwerk anpassen und ermöglichen die Nutzung von künstlichen Neuronen.

2.4 Sigmoid Neurons

Für die Entwicklung lernender Algorithmen in einem Netzwerk, fällt die Betrachtung in dieser Arbeit auf die Erkennung von handgeschrieben Zahlen. Die Eingabe für das Netzwerk könnten die Raw Pixeldaten der eingescannten Bilder darstellen, welche die handgeschrieben Zahlen abbilden. Das Ziel an dieser Stelle ist, dass das Netzwerk anhand der Veränderungen von *Gewichtungen* und der *Vorbelastung* lernt eine korrekte Klassifizierung der Zahlen vorzunehmen.
Das Modifizieren der *Gewichtungen* und der *Vorbelastung* von Neuronen kann das Verhalten des Netzwerkes und deren Entscheidungsfindung zu Problemen beeinflussen. Angenommen die Erkennung und Klassifizierung einer Zahl wurde durch das Netzwerk falsch vorgenommen, so können durch kleine Veränderungen an den *Gewichtungen* und der *Vorbelastung* eine Korrektur durchgeführt werden. Dieses stetige Modifizieren der Werte über einen definierten Zeitraum ermöglicht ein lernendes Netzwerk (siehe Abb. 2.9).

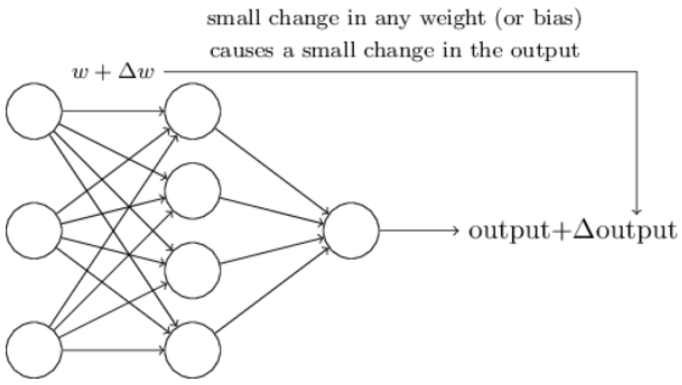

Abbildung 2.9: Modifizieren von Weights und Biases schaffen lernendes Netzwerk.

Mit Hilfe der Einführung des Sigmoid Neurons soll der Fehler zur Klassifizierung von Zahlenwerten minimiert werden. Eine Änderung der Gewichtungen und der Vorbelastung bei diesem künstlichen Neuron soll nur marginale Änderungen an dem Ausgabewert `output` vornehmen. Diese Erweiterung des Neurons begünstigt ein Netzwerk selbständig die Klassifizierung von Zahlen zu optimieren.

$$\mathbf{w} + \Delta\mathbf{w} \quad \rightarrow \quad \texttt{output} + \Delta\texttt{output}$$

Der Aufbau des Sigmoid Neurons ähnelt dem Perceptron, wobei das Neuron eine Anzahl von Eingabewerten x_n mit $n \in \{1, \cdots, m\}$ entgegennimmt und ausgehend von diesen Informationen den output ermittelt. Der wesentliche Unterschied zwischen diesen zwei Typen von Neuronen liegt in der differenzierbaren Funktion des Sigmoid Neurons zur Bestimmung des Ausgabewertes. Bei dem Sigmoid Neuron kann dieser alle Werte im Intervall $[0, 1]$ annehmen. Weiterhin weist auch diese Art von Neuronen für jeden Eingabewert eine Gewichtung w_n mit $n \in \{1, \cdots, m\}$ sowie eine Vorbelastung auf.
Für die Berechnung des output wird in diesem Kontext die *Sigmoid Funktion* $\sigma(z)$ angewendet.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \quad \text{mit} \quad z = \mathbf{w} \cdot \mathbf{x} + b \tag{2.3}$$

Unter Berücksichtigung des Gewichtungsvektors

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

und des Eingabevektors

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

ergibt sich mit der Indexnotation

$$\sigma(z) \equiv \frac{1}{1 + \exp(-\sum_j w_j x_j - b)}. \tag{2.4}$$

Dabei weist das Sigmoid Neuron das gleiche Verhalten wie ein Perceptron auf, wenn eine Grenzwertbetrachtung durchführt wird.

$$\begin{aligned} \lim_{z \to \infty} \sigma(z) &\approx 1 \qquad bzw. \\ \lim_{z \to -\infty} \sigma(z) &\approx 0 \end{aligned}$$

Dieses Verhalten wird weiterhin verdeutlicht, wenn die Betrachtung auf den folgenden Funktionsgraph fällt (siehe Abb. 2.10). Für große z nimmt die Funktion den Wert 1 an und für kleine z nimmt die Funktion den Wert 0 an.

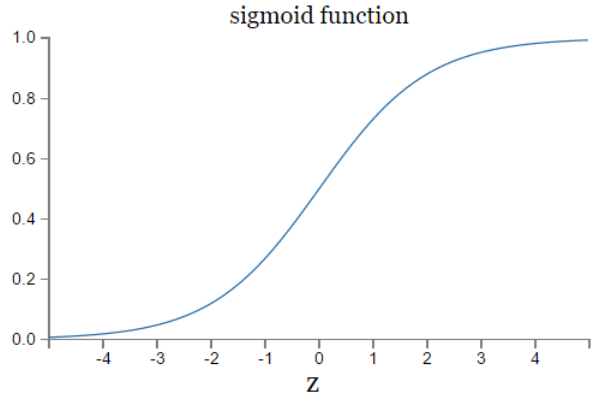

Abbildung 2.10: Sigmoid Funktion $\sigma(z)$.

Die Vorteile der Sigmoid Funktion liegen in den marginalen Änderungen Δw_j bei den Gewichtungen und Δb bei der Vorbelastung, welche eine marginale Änderung am Δoutput vornehmen. Damit stellt Δoutput die lineare Funktion bezüglich der Änderungen Δw_j und Δb in den Gewichtungen und der Vorbelastung dar.

$$\Delta \texttt{output} \equiv \sum_j \frac{\partial \texttt{output}}{\partial w_j} \Delta w_j + \frac{\partial \texttt{output}}{\partial b} \Delta b \tag{2.5}$$

Diese Linearität begünstigt die Wahl von kleinen Änderungen in den Gewichtungen und der Vorbelastung, um das Verhalten für ein lernendes Netzwerk abzuleiten.

Ein *Feedforward Neural Network* besteht aus L Schichten, wobei die Topologie des Netzwerkes durch $L \in \mathbb{N}$ und einer Liste $[m(1), \cdots, m(L)]$ geben ist. L bezeichnet die Anzahl der Schichten, während $m(l)$ mit $l \in \{2, \cdots, L\}$ die Anzahl der Neuronen in der l-ten Schicht angibt. Somit ist erste Schicht des Netzwerks, die Eingabeschicht, über die Eingabedimension $m(1)$ und die Ausgabedimension über $m(L)$ definiert.
Jedes Neuron der l-ten Schicht hat über die Gewichtung eine Verbindung zu einem Neuron in der $(l+1)$-ten Schicht. Die Gewichtung $w_{j,k}^{(l)}$ stellt die Verbindung des k-ten Neurons in der $l-1$ Schicht zu dem j-ten Neuron in der l-ten Schicht. Die Gewichtungen in Schicht L werden über die Gewichtungsmatrix $W^{(l)}$ der Schicht l zusammengefasst. Die Matrix ist wie folgt definiert

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

und ist eine $m(l) \times m(l-1)$ Matrix, wobei

$$W^{(l)} \in \mathbb{R}^{m(l) \times m(l-1)}.$$

Weiterhin hat das j-te Neuron in Schicht l eine Vorbelastung $b_j^{(l)}$. Die Vorbelastungen der Neuronen in Schicht l werden über den Vorbelastungsvektor

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

zusammengefasst. Hieraus ergibt sich für das j-te Neuron der l-ten Schicht die Sigmoid Funktion $\sigma_j^{(l)}$, die wie folgt rekursiv definiert wird:

1. Für die Eingabeschicht ergibt sich

$$\sigma_j^{(1)}(z) := x_j. \tag{2.6}$$

2. Für alle anderen Schichten ergibt sich

$$\sigma_j^{(l)}(z) := \sum_{k=1}^{m(l-1)} \left(w_{j,k}^{(l)} \cdot \sigma^{(l-1)}(z) + b_j^{(l)}\right) \quad \text{for all } l \in \{2, \cdots, L\}. \tag{2.7}$$

Die Ausgabe des neuronalen Netzwerks für eine gegebene Eingabe $\mathbf{x}$ ist durch die Neuronen der Ausgabeschicht festgelegt. Diese werden über den Ausgabevektor $\mathbf{o} \in \mathbb{R}^{m(L)}$ mit

$$\mathbf{o}(x) := \langle \sigma_1^{(L)}(x), \cdots, \sigma^{(L)}(z) \rangle = \sigma^{(L)}(x)$$

definiert. Unter Berücksichtigung der Gleichungen 2.6 und 2.7 kann nun nachvollzogen werden, wie Informationen das Netzwerk durchlaufen.

1. Der gegebene Eingabevektor $\mathbf{x}$ wird in der ersten und sogenannten Eingabeschicht des neuronalen Netzwerks gespeichert:

$$\sigma^{(1)}(z) := \mathbf{x}.$$

2. Auf der zweiten Schicht liegt die erste Neuronenebene vor bei denen die Sigmoid Funktion zum Einsatz kommt:

$$\sigma^{(2)}(z) := W^{(2)} \cdot \sigma^{(1)}(z) + \mathbf{b}^{(2)} = W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}.$$

3. Auf der dritten Schicht liegt die zweite Neuronenebene vor bei denen die Sigmoid Funktion zum Einsatz kommt:

$$\sigma^{(3)}(z) := W^{(3)} \cdot \sigma^{(2)}(z) + \mathbf{b}^{(3)} = W^{(3)} \cdot \left(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}.$$

4. Dieser Vorgang wird solange durchlaufen bis die Ausgabeschicht erreicht wird und die Ausgabe

$$\mathbf{o}(\mathbf{x}) := \sigma^{(L)}(z)$$

2.5 Stochastic Gradient Descent

Für eine zuverlässige Klassifizierung der Zahlen wird ein Algorithmus benötigt, der die Bestimmung von Gewichtungen und der Vorbelastungen ...

2.6 Backpropagation

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 $[x, y]$, 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 [ToDo: Verlinkung zu Theorieteil] zur Berechnung einer Sigmoid-Ausgabe lässt sich nun folgende Formel aufstellen:

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

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 Formel [ToDo: Verlinkung zu Theorieteil]) die Ausgabe der Neuronen in vektorisierter Form.

3. `sigmoid_prime(z)`: Für einen gegebenen Vektor z wird die Ableitung der Sigmoid-Funktion (nach Formel [ToDo: Verlinkung zu Theorieteil]) 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 verwendet:

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

   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. [ToDo: Verweis auf vorherige Formeldefinition, wenn vorhanden]

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:

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

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

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

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

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 die Implementierung der grafischen Ausgabe des neuronalen Netzwerks zur Erkennung handgeschrieben Zahlen in SetlX beschrieben. Ebenfalls wurde wie beim Hauptprogramm die Kommentarzeilen aus der Originaldatei in dieser Arbeit entfernt, um dem Code kompakt zu präsentieren. Die Animation wird als Klasse in SetlX angelegt und enthält die folgenden Membervariablen:

1. `mScreenSizeX`: Größe des Ausgabefenster in Richtung der x-Koordinate
2. `mScreenSizeY`: Größe des Ausgabefenster in Richtung der y-Koordinate
3. `mAllPixel`: Anzahl der Pixel einer einzulesenden Zahl
4. `mSqrtAllPixel`: Wurzelergebnis aus der Anzahl der Pixel einer einzulesenden Zahl
5. `mBiases`: Alle Vorbelastungen des Netzwerks
6. `mWeights`: Alle Gewichte des Netzwerks
7. `mTestData`: Testdaten des neuronalen Netzwerks
8. `mTestDataSize`: Größe der Testdaten des neuronalen Netzwerks
9. `mNetwork`: Aufbau der Layer in Listenform ohne Berücksichtigung der Eingabeschicht (`strutcure`[2..]: $[784, 30, 10] \rightarrow [30, 10]$)

Die Initialisierung der Animation erfolgt durch folgende Befehle:

```
1 netAnimation := animation(weights, biases, structure, test_data);
2 netAnimation.start_animation();
```

Als Übergabeparameter werden bei der Erstellung des Animation-Objekts die Gewichtungen, Vorbelastungen, die Struktur des Netzwerks in Form einer Liste und die entsprechenden Testdaten übergeben. Die `start_animation()`-Funktion der `network_animation`-Klasse setzt die Parameter für die Fensterausgabe und steuert die entsprechenden Animationen über die Werte des Eingabefelds an.

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

Eine Animation setzt sich aus den Elementen des Netzwerkes zusammen, weshalb die unterschiedlichen Schichten, Neuronen und die Verbindungen zwischen den Neuronen benachbarter Schichten gezeichnet werden müssen. Für jedes Element gibt es eine entsprechende Funktion in der `network_animation`-Klasse. Dieser Sachverhalt wird bei Betrachtung der `drawNetwork()`-Funktion verdeutlicht. Die Funktion ist durch folgenden Code gegeben:

```
1  drawNetwork := procedure() {
2      r := mScreenSizeY/(3*max(mNetwork)-1);
3      drawLayer(r, -1);
4      drawConnection(r);
5      drawNeuron([], mWeights, 0, 0, 0, 0, false, false, 0,
6                 (mScreenSizeY-(mSqrtAllPixel*12))/2, 0, 0, 1, 0, 12);
7  };
```

1. Zeile 2: Berechnet den Radius r für alle Neuronen n mit $n \in \{1..\texttt{max(mNetwork)}\}$. Mit $2r * n$ für die Größe des Kreiskörpers und $r * (n-1)$ für den Abstand zwischen den Neuronen ergibt sich $2r * n + r * (n-1) = \texttt{mScreenSizeY}$. Die Umstellung der Formel nach r ergibt:

   $$r = \texttt{mScreenSizeY}/(3 * \texttt{max(mNetwork)} - 1).$$

2. Zeile 3-5: Aufruf der Funktionen `drawLayer()`, `drawConnection()` und `drawNeuron()` mit entsprechenden Aufrufparametern.

An dieser Stelle soll angemerkt werden, dass die Funktion `drawLayer()` die Darstellung des Netzwerk inklusive der Neuronen in Kreisform abbildet z.B. das Hidden Layer und Output Layer, während die Funktion `drawNeuron()` das jeweilige Verhalten innerhalb des Neurons mittels $28 \times 28 = 784$ Quadraten abbildet z.B. das Input Layer (siehe Abb.3.1).

Der Funktion `drawLayer()` werden die Parameter für die Größe des Radius `r` eines Neurons und des momentan ausgewählten Neurons `acitveNeuron` übergeben. Der Code für das Zeichnen der einzelnen Schichten ist dann gegeben durch:

```
1  drawLayer := procedure(r, activeNeuron) {
2      gfx_setPenColor("BLACK");
3      for(i in {1 .. #mNetwork}) {
4          for(j in {0 .. mNetwork[i]-1}) {
5              if(activeNeuron == j && i ==1) {
6                  gfx_setPenColorRGB( 1.0, 0.0, 0.0 );
7                  gfx_filledCircle((mSqrtAllPixel*12+300*i), ((mScreenSizeY-r*(3*mNetwork[i]-1))/2
8              } else {
```

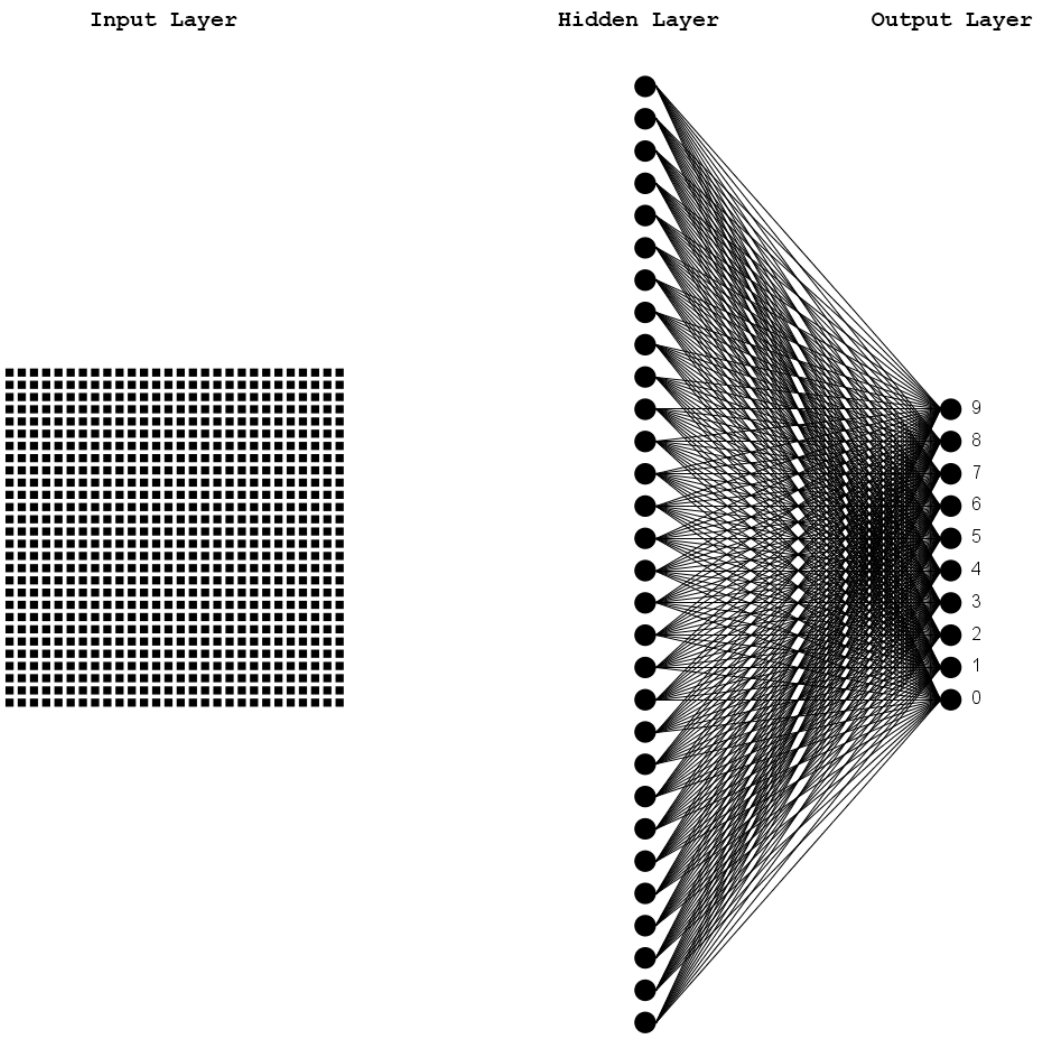

Abbildung 3.1: Default Animation, welche über die Eingabe mit dem Wert 0 aufgerufen wird.

```
9                   gfx_setPenColorRGB( 0.0, 0.0, 0.0 );
10                  gfx_filledCircle((mSqrtAllPixel*12+300*i), ((mScreenSizeY-r*(3*mNetwork[i]-1))/2)
11
12                  if(i == #mNetwork) {
13                      gfx_textRight((mSqrtAllPixel*12+300*i)+30, ((mScreenSizeY-r*(3*mNetwork[i]-1
14                  }
15              }
16          }
17      }
18  };
```

1. Zeile 3-4: Für das Zeichnen der einzelnen Schichten werden die Schichten mit der korrespondierten Anzahl von Neuronen.
2. Zeile 5-7: Das aktive Neuron auf dessen Betrachtung in der Animation fällt, wird durch die Farbe "rot" hervorgehoben.
3. Zeile 8-13: Alle nicht aktiven Neuronen werden schwarz abgebildet. Für die Ausgabeschicht wird

zusätzlich der erwartete Ausgabewert abgebildet, wenn `i` $==$ #`mNetwork` (siehe Zeile 12-13).

Die Berechnung der x-Koordinate für das abzubildende Neuron wird über

`mSqrtAllPixel` $* 12 + 300 *$ `i`

ermittelt. Dazu wird der der benötigte Bereich für die Eingabeschicht mittels `mSqrtAllPixel` $* 12$ berechnet, wobei für jede Iteration durch die Layer-Liste des Netzwerks 300px hinzukommen. Für die y-Koordinate ergibt sich die Formel

$(\texttt{mScreenSizeY} - \texttt{r} * (3 * \texttt{mNetwork}[\texttt{i}] - 1))/2 + 3 * \texttt{r} * \texttt{j}.$

Dabei wird der mittlere Abstand der grafischen Ausgabe einer Schicht zum Fensterrand in Abhängigkeit des Radius r über $(\texttt{mScreenSizeY} - r * (3 * \texttt{mNetwork}[\texttt{i}] - 1))/2$ berechnet. Für jedes Neuron j in der Schicht i wird der Durchmesser $2 * r$ des Neurons und der Abstand r zum nächsten Neuron über $3 * r * j$ addiert.

Im nächsten Abschnitt wird die Implementierung der `drawConnection`()-Funktion vorgestellt. Die Funktion verbindet jedes Neuron der Schicht n mit jedem Neuron der Schicht $n + 1$. Hierbei sind die Berechnung der Koordinaten x_1, y_1 sowie x_2, y_2 äquivalent zu den Berechnung der vorangegangenen `drawLayer`()-Funktion.

```
1  drawConnection := procedure(r) {
2      gfx_setPenColor("BLACK");
3      for(i in {1 .. #mNetwork-1}) {
4          for(j in {0 .. mNetwork[i]-1}) {
5              for(k in {0 .. mNetwork[i+1]-1}) {
6                  if(i<#mNetwork) {
7                      gfx_line((mSqrtAllPixel*12+300*i+r), ((mScreenSizeY-r*(3*mNetwork[i]-1))/2)+
8                  }
9              }
10         }
11     }
12 };
```

Als letztes wird die Funktion $drawNeuron$ diskutiert, welche ausgehend von den Gewichtungen und den Testdaten Bereiche mit höherer Relevanz farbliches hervorhebt. Die farbliche Unterteilung hat dabei die folgende Bedeutung.

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

Für die unterschiedlichen Animationsmöglichkeiten muss an dieser Stelle unterschieden werden, ob die anzuzeigenden Daten nur die Gewichtungen, die Testdaten oder eine Kombination beider abbilden sollen.

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.