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 10. April 2017

Lucas Heuser	Johannes Hill

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Studienarbeit and DHWB . . . 1
- 1.2 GitHub Link . . . 1
- 1.3 Was ist künstliche Intelligenz . . . 1
- 1.4 Aktuelle Relevanz/Themen von neuronalen Netzen . . . 1
- 1.5 Ziel der Arbeit . . . 1
- 1.6 Aufbau der Arbeit . . . 1

2 Theorie **2**
- 2.1 SetIX . . . 2
- 2.2 MNIST . . . 2
- 2.3 Perceptrons . . . 2
- 2.4 Sigmoid Neurons . . . 4
- 2.5 Architektur Neuronaler Netzwerke . . . 6
- 2.6 Netzwerk zur Klassifizierung von handgeschrieben Zahlen . . . 8
- 2.7 Stochastic Gradient Descent . . . 8
- 2.8 Backpropagation . . . 8

3 Implementierung **9**
- 3.1 Laden und Aufbereitung der MNIST Daten . . . 9
- 3.2 Implementierung des neuronalen Netzes . . . 10
- 3.3 Animation . . . 14

Abbildungsverzeichnis

2.1 Percetron mit den Eingaben x_1, x_2, x_3 und der Ausgabe $output$. 2
2.2 Unterschiedliche Möglichkeiten der Entscheidungsfindung. 3
2.3 Percetron mit zwei Eingaben -2 und einem Bias von 3. 3
2.4 `NAND` Gatter mit den Eingaben x_1 und x_2. 4
2.5 `NAND` Gatter Aufbau mit Perceptrons. 4
2.6 Vereinfachter `NAND` Gatter Aufbau mit Perceptrons. 4
2.7 Modifizieren von Weights und Biases schaffen lernendes Netzwerk. 5
2.8 Sigmoid Funktion $\sigma(z)$. 6
2.9 Sigmoid Funktion $\sigma(z)$. 6
2.10 Architektur eines neuronalen Netzwerks. 7
2.11 Aufbau des neuronalen Netzwerks hinsichtlich der einzelnen Layer. 7

Kapitel 1

Einleitung

1.1 Studienarbeit and DHWB

1.2 GitHub Link

1.3 Was ist künstliche Intelligenz

1.4 Aktuelle Relevanz/Themen von neuronalen Netzen

1.5 Ziel der Arbeit

Die menschliche Wahrnehmung ist

1.6 Aufbau der Arbeit

Kapitel 2

Theorie

2.1 SetIX

2.2 MNIST

2.3 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 $n \in \{1, \cdots, n\}$ und berechnet einen einzigen binären Ausgabewert (siehe Abb. 2.1). Für die Berechnung der Ausgabe werden

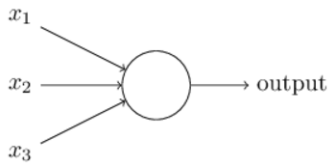

Abbildung 2.1: Percetron mit den Eingaben x_1, x_2, x_3 und der Ausgabe $output$.

sogenannte *Weights* w_n mit $n \in \{1, \cdots, n\}$ eingeführt, welche die Gewichtung der jeweiligen Eingabe festlegen. Der Ausgabewerte `output` wird mittels der gewichteten Summe $\sum_j w_j x_k$ und einem definierten Grenzwert `threshold` bestimmt.

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

Werden die *Weights* und der *Threshold* variiert, entstehen unterschiedliche Modelle zur Entscheidungsfindung. Hierbei ist zu beachten, dass eine Minimierung des *Thresholds* den binären Ausgabewert 1 mit einer höheren Wahrscheinlichkeit bedingt.

Der Aufbau des Netzwerks leitet sich aus den unterschiedlichen Modellen der Entscheidungsfindung ab und wird mit Hilfe der Perceptrons abgebildet (siehe Abb. 2.2). Eine Entscheidungsmöglichkeit wird hierbei durch das Perceptron dargestellt. Weiterhin wird eine Spalte von Perceptrons als *Layer* bezeichnet. Der erste Layer fällt Entscheidungen auf Basis der Eingabewerte, indem er diese abwägt. Jedes Perceptron des zweiten Layers hingegen, wägt für die Entscheidungsfindung die Resultate des ersten Layers ab. Ein Perceptron auf dem zweiten Layer kann somit eine Entscheidung auf einer abstrakteren

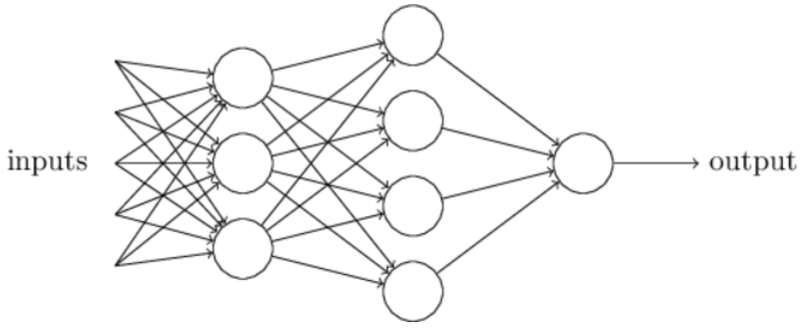

Abbildung 2.2: Unterschiedliche Möglichkeiten der Entscheidungsfindung.

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 w und x eingeführt, wodurch sich die Schreibweise $w \cdot x \equiv \sum_j w_j x_j$ ergibt. Des Weiteren wird der `threshold` auf die andere Seite der Ungleichung gezogen und erhält die Bezeichnung *Bias*, $b \equiv -$`threshold`.

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

Somit nimmt der Bias Einfluss auf die binäre Ausgabe `output`, weshalb der Bias eine Ausgabe mit einem hohen positiven oder negativen Wert mit einer höheren Wahrscheinlichkeit bedingen kann.

Im vorherigen Abschnitt wurden Perceptrons als ein Methode für die Entscheidungsfindung beschrieben. Ein weiterer Anwendungsfall besteht in der Berechnung von logischen Funktion wie z.B. `AND`, `OR` und `NAND`. Fällt die Betrachtung auf ein Perceptron mit 2 Eingaben deren Gewichtung jeweils den Wert -2 aufweisen und einen Bias von 3, so ergibt sich folgende Abbildung (siehe Abb. 2.3).

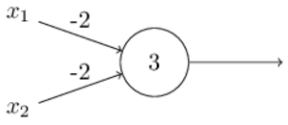

Abbildung 2.3: Percetron mit zwei Eingaben -2 und einem Bias von 3.

Weisen die Eingaben x_1, x_2 den Wert 0 auf, ergibt sich für den `output` den Wert 1. Sind die Eingabewerte für x_1, x_2 -1 ergibt sich für den `output` den Wert -1. Der Aufbau beschreibt somit einen `NAND` Gatter.

$$\begin{aligned} w \cdot x + b & = \texttt{output} \\ (-2) * 0 + (-2) * 0 + 3 & = 1 \\ (-2) * 1 + (-2) * 1 + 3 & = -1 \end{aligned}$$

`NAND` Gatter können verwendet werden, um die unterschiedlichsten Berechnungen durchzuführen. Im Folgenden fällt die Betrachtung auf die Addition von zwei Bits x_1 und x_2. 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 gleich 1.

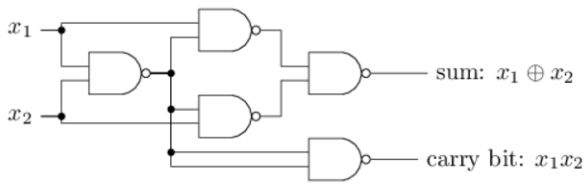

Abbildung 2.4: `NAND` Gatter mit den Eingaben x_1 und x_2.

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 der Bias b den Wert 3 auf.

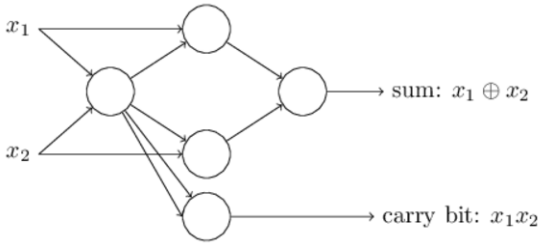

Abbildung 2.5: `NAND` Gatter Aufbau mit Perceptrons.

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 einem sogenannten *Input Layer* zusammengefasst, wobei durch die Notation eine Eingabe nicht mit einem Perceptron gleichzustellen ist.

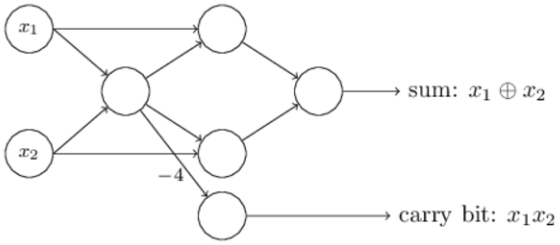

Abbildung 2.6: Vereinfachter `NAND` Gatter Aufbau mit Perceptrons.

Dieser Anwendungsfall zeigt, dass mit Perceptrons unterschiedliche Berechnungen durchgeführt werden können. Implementierte Lernalgorithmen können Gewichtungen sowie den Bias automatisch durch entsprechende Stimuli im Netzwerk anpassen und ermöglichen die Nutzung von künstliche Neuronen, die sich von herkömmlichen Logik Gattern unterscheiden. Neuronale Netze können somit über einen definierten Zeitraum lernen, wie bestimmte Probleme zu lösen sind.

2.4 Sigmoid Neurons

Für die Entwicklung lernender Algorithmen in einem Netzwerk mit Perceptrons, fällt unsere Betrachtung auf das Beispiel der Erkennung von handgeschriebenen Zahlen. Die Eingabe für das Netzwerk könnten die Raw Pixeldaten der eingescannten Bilder darstellen, welche die handgeschriebenen Zahlen abbilden. Das Ziel an dieser Stelle ist, dass das Netzwerk anhand der Veränderungen von *Weights* und

Biases lernt eine korrekte Klassifizierung der Zahlen vorzunehmen.
Das Modifizieren der *Weights* und *Biases* 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 *Weights* und *Biases* eine Korrektur durchgeführt werden. Dieses stetige Modifizieren der Werte über einen definierten Zeitraum ermöglicht ein lernendes Netzwerk (siehe Abb. 2.7).

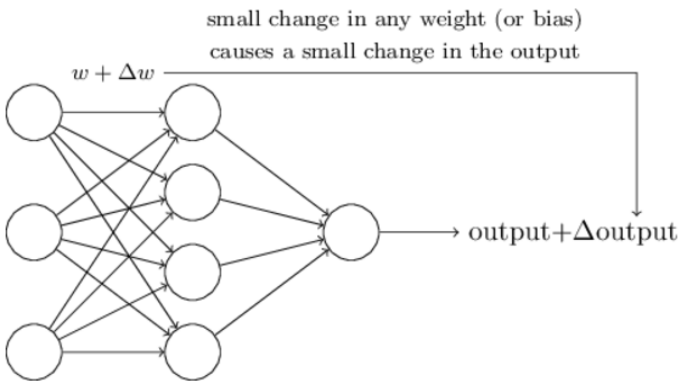

Abbildung 2.7: Modifizieren von Weights und Biases schaffen lernendes Netzwerk.

Dieses wünschenswerte Verhalten eines lernenden Netzwerks kann durch Perceptrons nicht gewährleistet werden, da eine Veränderung der Weights und Biases den Ausgabewerte `output` eines Neurons umkehren kann. Dadurch kann das komplette Verhalten zur Klassifizierung von Zahlen beeinflusst werden, wobei zuvor falsch erkannte Zahlen nun richtig klassifiziert werden und umgekehrt. Mit Hilfe der Einführung des Sigmoid Neurons soll dieser Fehler behoben werden. Eine Änderung der Weights und Biases 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.

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

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

Unter Berücksichtigung der Eingabewerte x_n mit $n \in \{1..n\}$ und der Weights w_n mit $n \in \{1..n\}$ ergibt sich die folgende Formel:

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

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

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

Dieses Verhalten wird weiterhin verdeutlicht, wenn die Betrachtung auf den folgenden Funktionsgraph

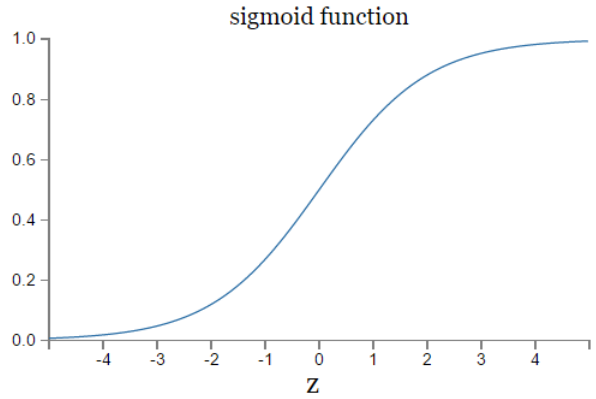

Abbildung 2.8: Sigmoid Funktion $\sigma(z)$.

fällt (siehe Abb. 2.8). Für große z nimmt die Funktion den Wert 1 an und für kleine z nimmt die Funktion den Wert 0 an.
Im Vergleich hierzu die Stufenfunktion, die das Verhalten eines Perceptrons abbildet (siehe Abb. 2.9). Die Vorteile der Sigmoid Funktion liegen in den marginalen Änderungen δw_j bei den Gewichtungen

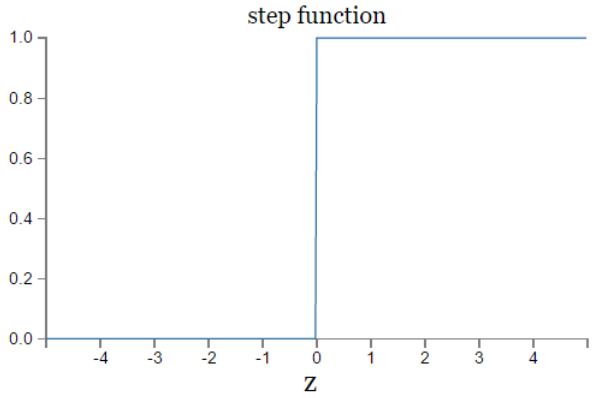

Abbildung 2.9: Sigmoid Funktion $\sigma(z)$.

und δb im Bias, welche eine marginale Änderung am $\delta \texttt{output}$ vornehmen. Damit stellt $\delta \texttt{output}$ die lineare Funktion bezüglich der Änderungen δw_j und δb in den Weights und Bias dar.

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

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

2.5 Architektur Neuronaler Netzwerke

In diesem Abschnitt der Arbeit wird der Aufbau eines neuronalen Netzwerks näher betrachtet und entsprechend auf die Terminologie in diesem Bereich eingegangen (siehe Abb. 2.10).

Ein Netzwerk setzt sich aus mehreren Layern zusammen. So ist wird der erste Layer auf der Linken Seite auch als *Input Layer* und alle korrespondierenden Neuronen als *Input Neuronen* bezeichnet. Der letzte

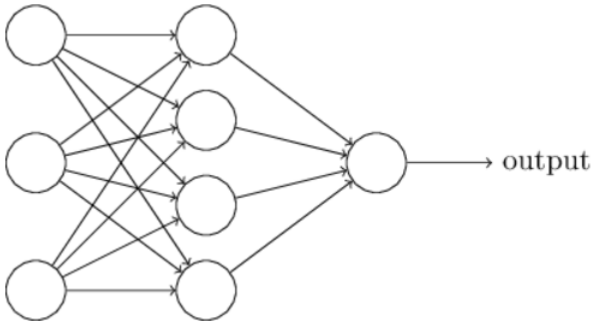

Abbildung 2.10: Architektur eines neuronalen Netzwerks.

Layer oder auch *Output Layer* beinhaltet dagegen alle *Output Neuronen*. Die mittleren Layer tragen die Bezeichnung des *Hidden Layers*, da die Neuronen weder zu dem Layer der Inputs, noch zu denen der Outputs gehören. Hierbei kann ein Netzwerk mehrere so genannte *Hidden Layer* aufweisen (siehe Abb. 2.11). 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.

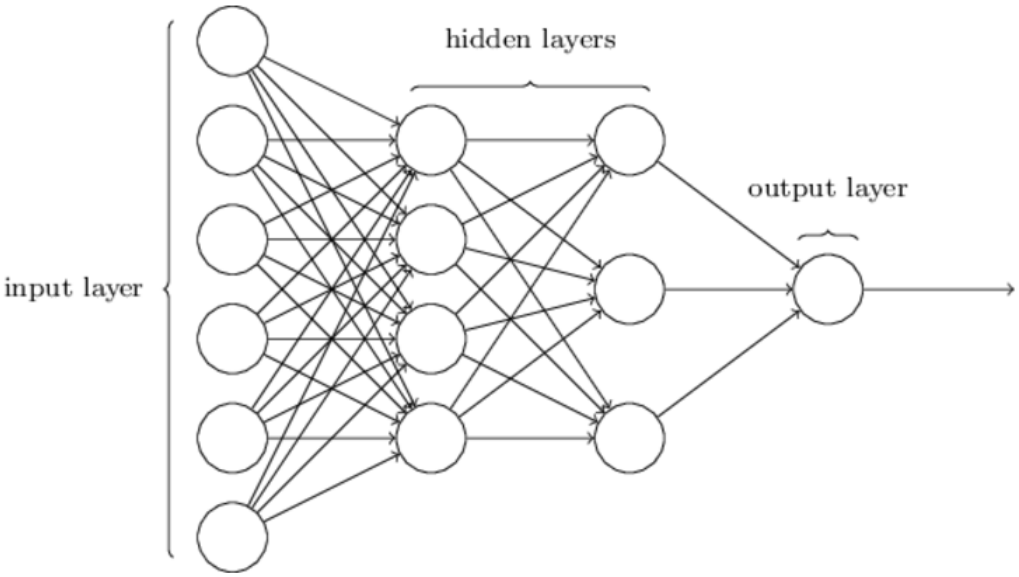

Abbildung 2.11: Aufbau des neuronalen Netzwerks hinsichtlich der einzelnen Layer.

Die Zusammensetzung der Input- und Output-Layer in einem Netzwerk sind vergleichsweise einfach. Fällt unsere Betrachtung auf die Erkennung einer handgeschriebenen "9", so können die Intensitäten der Bildpixel als Eingabewerte für den Input-Layer dienen. Liegt ein Graustufenbild der Größe von 64x64 Pixeln vor, leiten sich daraus 4096 Input-Neuronen mit skalierten Intensitätswerten zwischen 0 und 1 ab. Der Output-Layer hingegen weißt nur ein Neuron auf, um eine neun entsprechend klassifizieren zu können.

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

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

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 viel 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 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 sogenannte *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.

2.6 Netzwerk zur Klassifizierung von handgeschrieben Zahlen

2.7 Stochastic Gradient Descent

2.8 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.
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:

`label1, pixel11, pixel12, pixel13, ...`
`label2, pixel21, pixel22, pixel23, ...`
...

Hierbei bezeichnet Label den Wert der jeweils gezeichneten Ziffer (wird benötigt um zu überprüfen, ob das Netzwerk die korrekte Ziffer identifiziert hat und um das Netzwerk zu trainieren).

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:
[
[`pixels1, label1`],
[`pixels2, label2`],
...]
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 SetIX. 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 SetIX 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 SetIX angelegt und enthält die folgenden Membervariablen:

1. `mNumLayers`: Anzahl der Layer des aufzubauenden Netzwerkes
2. `mSizes`: Aufbau des Layers in Listenform. Bsp.: $[784, 30, 10]$ beschreibt ein Netzwerk mit 784 Inputfeldern, 30 Neuronen im zweiten (hidden) Layer und 10 Output-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(s) {
16      [i,j] := s;
17      return la_matrix([
18          [ ((random()-0.5)*2)/28 : p in [1..i] ] : q in [1..j]
19      ]);
20  };
```

1. `init()`: in der init-Funktion werden die Funktionen computeRndBiases() und `computeRndWeights()` aufgerufen

2. `computeRndBiases()`: Die Funktion befüllt die Variable mBiases mit zufälligen Werten. Der für das Netzwerk benötigte Aufbau der Vorbelastungen entspricht folgender Form:
   [
   << << `b_layer1_neu1` >> << `b_layer1_neu2` >> ... >>,
   << << `b_layer2_neu1` >> ... >>,
   ...]
   Das heißt es kann auf die Vorbelastungen mit folgendem Schema in SetlX zugegriffen werden:

   $$\texttt{mBiases[layer][neuron][bias]}$$

   Hierbei ist zu beachten, dass der Wert für `bias` immer 1 ist, da jedes Neuron nur eine einzige Vorbelastung besitzt. 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 dem Ersten) betrachtet.

3. `computeRndWeights()`: Diese Funktion ist equivalent zu der Vorbelastungs-Funktion, lediglich wird folgende Struktur der Gewichte angelegt:
   [
   << << `w1_layer2_neu1 w1_layer2_neu2` ... >> << `w2_layer2_neu1` ... >> ... >>,
   << << `w1_layer3_neu1 w1_layer3_neu2` ... >> << `w2_layer3_neu1` ... >> ... >>,
   ...]
   Dies entspricht folgenden Zugriffsmöglichkeiten:

   $$\texttt{mWeights[layer} - 1\texttt{][neuron][weight for input neuron]}$$

   Der Zugriff auf die Schichten mittels $[\texttt{layer} - 1]$ resultiert aus den fehlenden Gewichten der Eingabe-Schicht.

4. `computeRndMatrix()`: Diese Hilfsfunktion dient zur Erstellung der Struktur der Gewichte und Vorbelastungen in den zuvor vorgestellten Funktionen. Die Funktion enthält als Parameter eine Matrix-Struktur in Listenform und liefert die zugehörige Matrix mit zufälligen Werten zwischen $-1/28$ und $1/28$ zurück. 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 Reihen 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 $\vec{a}$ bezeichnet den Aktivierungsvektor der vorherigen Schicht, also deren Ausgabe (zu Beginn also die Pixel der Eingabe). Nach Gleichung xyz zur Berechnung einer Sigmoid-Ausgabe lässt sich nun folgende Formel aufstellen:

$$\vec{a}' = \sigma(W * \vec{a} + B) \tag{3.1}$$

Hierbei bezeichnet $\vec{a}'$ 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      a := la_vector(a);
3      for( i in {1..#mBiases} ) {
4          a := sigmoid( (mWeights[i]*a) + mBiases[i] );
5      }
6      return a;
7  };
8  sigmoid := procedure(z) {
9      return la_vector([ 1.0/(1.0 + exp(- z[i] )) : i in [1..#z] ]);
10 };
11 sigmoid_prime := procedure(z) {
12     s := sigmoid(z);
13     return la_matrix([ [ s[i] * (1 - s[i]) ] : i in [1..#s] ]);
14 };
```

1. `feedforward(a)`: Zunächst werden die als Liste übergebenen Eingabewerte `a` (784 Pixel in Listenform) mit Hilfe von `la_vektor()` in einen Vektor umgewandelt und anschließend wird die 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 xyz) die Ausgabe der Neuronen in vektorisierter Form.

3. `sigmoid_prime(z)`: Für einen gegebenen Vektor `z` wird die Ableitung der Sigmoid-Funktion (nach Formel xyz) berechnet und in vektorisierter Form 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
7      for(j in {1..epochs}) {
8          training_data := shuffle(training_data);
9          mini_batches := [
10             training_data[k..k+mini_batch_size-1] : k in [1,mini_batch_size..n]
11         ];
12
13         for(mini_batch in mini_batches) {
14             update_mini_batch(mini_batch, eta);
15         }
16
```

```
17          if(test_data != null) {
18              ev := evaluate(test_data);
19              print("Epoch $j$: $ev$ / $n_test$");
20          }
21          else {
22              print("Epoch $j$ complete");
23          }
24      }
25  };
```

1. Zeile 1: Übergabeparameter der Funktion sind die Trainingsdatensätze (Liste von Tupeln $[\mathrm{x}, \mathrm{y}]$ 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 7: Der nachfolgende Programmcode wird entsprechend der übergebenen Epochenanzahl mehrfach ausgeführt.
3. Zeile 8-12: 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 14-16: Anschließend wird für jeden Mini-Batch aus `mini_batches` ein Schritt des SGD angewendet. Dies geschieht mit Hilfe der Funktion `update_mini_batches`, welche anschließend genauer 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 19-25: 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.

Nun 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 := [0 : i in [1..#test_data]];
3
4       i := 1;
5       for( [x,y] in test_data ) {
6           out := feedforward(x);
7           max := out[1];
8           index := 1;
9           for(i in {2..#out}) {
10              if( out[i] > max ) {
11                  max := out[i];
12                  index := i;
13              }
14          }
```

```
15          test_results[i] := [index-1,y];
16          i += 1;
17      }
18
19      return #[1 : [x,y] in test_results | x == y];
20  };
```

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. Die Variable wird zunächst mit 0-en initialisiert. Es könnte ebenso eine leere Liste erstellt werden, welche im späteren Verlauf um weitere Elemente erweitert wird, allerdings würde das Anhängen an die Liste zu erhöhtem Rechenaufwand führen was die Leistung des Netzwerkes negativ beeinflussen würde.
3. Zeile 5: Der nachfolgende Programmcode wird nun auf jedes Tupel $[x, y]$ des übergebenen Testdatensatzes angewandt.
4. Zeile 6-14: Mit Hilfe der bereits besprochenen Feedforward-Funktion wird zunächst die vektorisierte Ausgabe des Netzwerkes für den jeweiligen Datensatz berechnet und in `out` gespeichert. anschließend wird über den Ausgabe-Vektor iteriert und das Maximum sowie der dazugehörige Index im Vektor ermittelt [1].
5. Zeile 15: 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 nun der errechnete Wert sowie der tatsächliche Wert (y) gespeichert.
6. Zeile 19: Die Funktion gibt im Anschluss die Anzahl aller übereinstimmenden Ergebnisse in `test_results` zurück.

ToDo: update-mini-batch, backprop

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

[1] Anmerkung: Das Maximum könnte auch ohne eine Schleife mit Hilfe der `max`-Funktion von SetlX ermittelt werden. Allerdings ist es so nicht möglich den zugehörigen Index zu berechnen