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 22. März 2017

Lucas Heuser	Johannes Hill

Inhaltsverzeichnis

Abbildungsverzeichnis

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

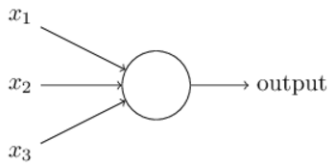

Abbildung 2.1: Percetron mit den Eingaben x_1, x_2, x_3 und der Ausgabe $output$.

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

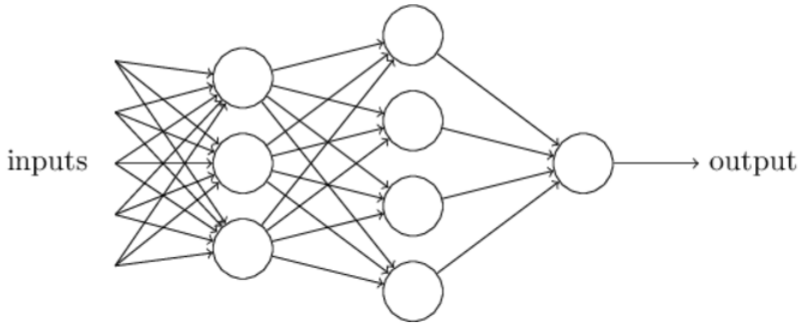

Abbildung 2.2: Unterschiedliche Möglichkeiten der Entscheidungsfindung.

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).

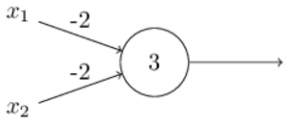

Abbildung 2.3: Percetron mit zwei Eingaben -2 und einem Bias von 3.

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.

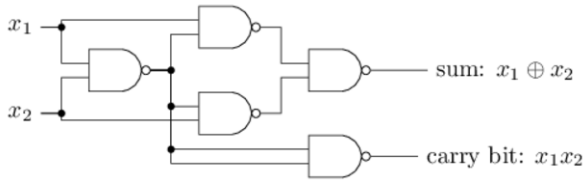

Abbildung 2.4: `NAND` Gatter mit den Eingaben x_1 und x_2.

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.

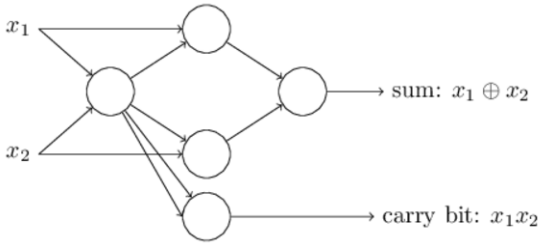

Abbildung 2.5: `NAND` Gatter Aufbau mit Perceptrons.

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.

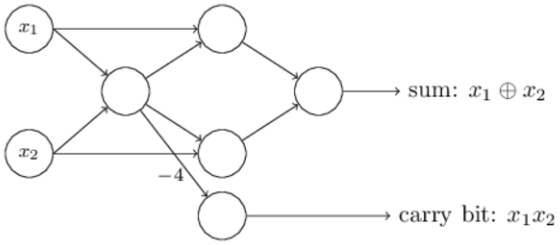

Abbildung 2.6: Vereinfachter `NAND` Gatter Aufbau mit Perceptrons.

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).

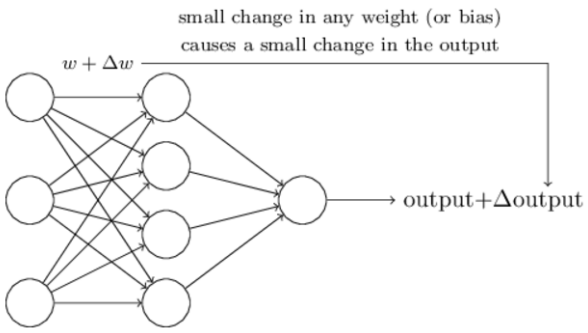

Abbildung 2.7: Modifizieren von Weights und Biases schaffen lernendes Netzwerk.

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`.

Kapitel 3

Implementierung

3.1 Laden und Aufbereitung der MNIST Daten

3.2 Implementierung des neuronalen Netzes

Dieser Abschnitt beschreibt die eigentliche Implementierung des neuronalen Netzwerkes zur Erkennung von handgeschriebenen Ziffern. 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 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 Biases des Netzwerkes (genauer Aufbau wird im Folgenden erläutert)
4. `mWeights`: Alle Weights 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 Weights und Biases 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 Weights- und Bias-Initialisierung verwendet werden, zu sehen.

```
1       init := procedure() {
2               computeRndBiases();
3               computeRndWeights();
4       };
5
6       computeRndBiases := procedure() {
```

```
7               this.mBiases := [
8                       computeRndMatrix([1, mSizes[i]]) : i in [2..mNumLayers]
9               ];
10      };
11
12      computeRndWeights := procedure() {
13              this.mWeights := [
14                      computeRndMatrix([mSizes[i], mSizes[i+1]])
15                      : i in [1..mNumLayers-1]
16              ];
17      };
18
19      computeRndMatrix := procedure(s) {
20              [i,j] := s;
21              return la_matrix([
22                      [ ((random()-0.5)*2)/28 : p in [1..i] ] : q in [1..j]
23              ]);
24      };
```

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 Biases entspricht folgender Form:
   [<< << `b_layer1_neu1` >> << `b_layer1_neu2` >> ... >>, << << `b_layer2_neu1` >> ... >>, ...]
   Das heißt es kann auf die Biases mit folgendem Schema in SetlX zugegriffen werden:

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

   Hierbei ist zu beachten, dass der Wert `bias` immer 1 ist, da jedes Neuron nur einen einzigen Bias besitzt. Da es sich beim Input-Layer des Netzwerkes nicht um Sigmoid-Neuronen handelt, sondern lediglich um Eingabewerte, werden hierfür keine Biases benötigt. Deshalb wird bei der Erstellung der zufälligen Biases nur [2..`mNumLayers`] (also alle Layer außer dem Ersten) betrachtet.

3. `computeRndWeights()`: Diese Funktion ist equivalent zu der Bias-Funktion, lediglich wird folgende Struktur der Weights 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 Layer mittels [`layer` $-$ 1] resultiert aus den fehlenden Weights des Input-Layers.

4. `computeRndMatrix()`: Diese Hilfsfunktion dient zur Erstellung der Struktur der Weights und Biases 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. Die übergebende Struktur hat die Form $[\mathtt{x}, \mathtt{y}]$, 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 Weights und b die Matrix aller Biases und a bezeichnet den Aktivierungsvektor des vorherigen Layers, also deren Output (zu Beginn also die Input-Pixel der Eingabe). Nach Gleichung xyz zur Berechnung eines Sigmoid-Outputs lässt sich nun folgende Formel aufstellen:

$$a' = \sigma(wa + b) \tag{3.1}$$

Hierbei bezeichnet a' den Output-Vektor des aktuellen Layers, welcher dann dem nächsten Layer 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
9       sigmoid := procedure(z) {
10              return la_vector([ 1.0/(1.0 + exp(- z[i] )) : i in [1..#z] ]);
11      };
12
13      sigmoid_prime := procedure(z) {
14              s := sigmoid(z);
15              return la_matrix([ [ s[i] * (1 - s[i]) ] : i in [1..#s] ]);
16      };
```

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 Layer des Netzwerkes angewandt. Zurückgegeben wird der resultierende Output jedes Neurons des letztes Layers in vektorisierter Form.
2. `sigmoid(z)`: Diese Funktionen nimmt einen Vektor `z` und berechnet mit Hilfe der Sigmoid-Formel (siehe Formel xyz) den Output 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.

ToDo: Rest des Codes

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