

Projektarbeit
in der Angewandten Informatik

Ziffernerkennung mittels neuronalem Perzeptron-Netz

Hannes Dröse

Abgabedatum: 16.03.2021

Dr. Jürgen Löffelholz

1 Aufgabenstellung

Ziel dieses Projektes ist die Entwicklung eines Computerprogramms. Dieses Programm soll in der Lage sein, die arabischen Ziffern von 0 - 9 in einem Pixelraster zu erkennen.

Das Pixelraster ist 7 Pixel hoch und 5 Pixel breit. Ein Pixel besteht aus einer Zahl, die entweder 0 oder 1 ist. 0 bedeutet, der Pixel ist nicht gefüllt oder leer. 1 bedeutet, dieser Pixel ist gefüllt oder ausgemalt. Sollten die Grafiken kleiner sein als das Pixelraster, so werden sie nach Möglichkeit zentriert. In der folgenden Abbildung ist eine beispielhafte Pixelgrafik dargestellt:

0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
1	1	1	1	1

Abbildung 1.1: Beispiel einer Pixelgrafik mit 7x5 Pixeln

Dazu wird zunächst ein neuronales Netz entwickelt und mit Trainingsdaten mittels Gradientenabstiegsverfahren trainiert. Am Ende gibt es eine interaktive Anwendung, in welche ein beliebiges Pixelmuster eingegeben werden kann. Das Programm nutzt nun das entstandene Netz, um eine der Ziffern in diesem Muster zu erkennen.



Abbildung 1.2: Vorschau der interaktiven Anwendung am Ende des Projektes

2 Lösungsmethode

Überwachtes Lernen

Das Erkennen von Ziffern ist eine recht komplexe Aufgabe. Sie lässt sich nur sehr schwierig mittels festen Regeln umsetzen. Das liegt nicht zu letzt an einer Vielzahl von Schriftarten und Darstellungsformen für Ziffern. Anstatt ein sehr komplexes und umfangreiches Regelwerk zu entwickeln, wird dieses Projekt mittels **maschinellem Lernen** realisiert. Dazu wird anhand von Beispieldaten vom Computer ein Modell zur Lösung der Aufgabe erstellt. Dabei kommen Algorithmen zum Einsatz, welche in der Lage sind, aus den Beispieldaten die entsprechenden Regeln selbstständig abzuleiten. (vgl. Papp u. a. 2019, Kap. 6 Machine Learning)

Im Speziellen setzt dieses Projekt auf das **Überwachte Lernen** (engl. Supervised Learning). Das heißt, die Beispieldaten bestehen aus Paaren von Eingabe- und erwarteten Ausgabewerten. Der Algorithmus soll nach dem Training in der Lage sein, die Eingabewerte korrekt auf die erwarteten Ausgaben abzubilden. Dabei soll der Algorithmus gleichzeitig eine **Generalisierung** durchführen. Dadurch sollen auch Eingabewerte richtig verarbeitet werden, die nicht in den Beispieldaten enthalten gewesen sind. (vgl. Papp u. a. 2019, Kap. 5.2.1 Überwachtes Lernen)

Häufig werden die Beispieldaten in Trainings- und Testdaten unterteilt. Trainingsdaten werden für das Training des Algorithmus verwendet. Die Testdaten werden nach dem Training verwendet. Damit wird überprüft, ob auch noch nicht gesehene Daten richtig zugeordnet werden können. (vgl. Papp u. a. 2019, Kap. 5.6 Wie gut ist der Algorithmus?)

Neuronale Netze

Der Algorithmus zur Lösung der Aufgabe ist ein **künstliches neuronales Netz**. Neuronale Netze sind mathematische Konstrukte, welche beliebige Funktionen mithilfe von Beispieldaten approximieren können. Die Erkennung von Ziffern ist eine Funktion, welche Pixelwerte entgegennimmt und die erkannte Ziffer als Ausgabe zurückliefert. (vgl. Sutton und Barto 2018, Kap. 9.7 Nonlinear Function Approximation: Artificial Neural Networks)

Neuronale Netze sind besonders für die Erkennung von Mustern (vor allem von Ziffern und Buchstaben) geeignet. (vgl. Lämmel und Cleve 2020, Kap. 6.3 Typische Anwendungen)

Biologisches Vorbild

Die Inspiration für neuronale Netze stammt aus dem menschlichen Gehirn. Dieses besteht aus einem Netzwerk von Neuronen. Diese haben nun (z.B. durch äußere Reize) ein elektrisches Aktivierungslevel. Liegt dieses über einem bestimmten Schwellenwert (welcher vom Neuron abhängig ist), ist das Neuron "aktiv". Benachbarte Neuronen sind über Nervenbahnen (Synapsen) miteinander verbunden. Diese Verbindung kann stärker oder auch schwächer sein. Das Aktivierungslevel eines Neurons strahlt dadurch auch auf seine Nachbarn in unterschiedlicher Intensität aus. Durch die unterschiedlichen Stärken der Synapsen und die verschiedenen Verbindungen der Neuronen untereinander entstehen unterschiedlichste Aktivierungsmuster. (vgl. Lämmel und Cleve 2020, Kap. 5.1 Das künstliche Neuron)

Einzelnes Neuron

Ein einzelnes Neuron wird nun mathematisch mit einem sog. **Perzeptron** nachgebildet. Das Perzeptron bekommt eine feste Menge an Eingabewerten. Diese werden mit **Gewichten** versehen, welche die unterschiedlich starken Synapsen darstellen. Die Eingabewerte werden mit den Gewichten multipliziert und anschließend aufsummiert. Hinzu wird ein weiterer Wert addiert – der sog. **Bias**. Dieser stellt den Schwellenwerte zur Aktivierung eines Neurons dar. Zum Schluss wird auf diesen Wert eine **nicht-lineare Aktivierungsfunktion** angewandt (näheres im nächsten Abschnitt). (vgl. Lämmel und Cleve 2020, Kap. 5.1 Das künstliche Neuron)

Zusammengefasst ist das Aktivierungslevel bzw. der Output des Neurons wie folgt definiert:

$$y = \phi(b + \vec{w} \cdot \vec{x})$$

Wobei ϕ die Aktivierungsfunktion, \vec{x} die Eingabewerte, \vec{w} die Gewichte und b den Bias darstellt.

Aktivierungsfunktion

Als **Aktivierungsfunktionen** werden sehr unterschiedliche Funktionen eingesetzt. Je nach Anwendungsfall muss die Funktion verschiedene Anforderungen erfüllen:

Eine nicht-lineare Aktivierungsfunktion wird eingesetzt, wenn ein Neuron auch nicht lineare Ausgaben liefern können soll. Andernfalls sind die möglichen darstellbaren Funktionen für ein Neuron stark eingeschränkt. (vgl. Sutton und Barto 2018, Kap. 9.7 Nonlinear Function Approximation: Artificial Neural Networks)

Weiterhin bilden verschiedene Aktivierungsfunktionen ihre Ausgaben in unterschiedlichen Wertebereichen ab. Die Funktion beim klassischen Perzeptron ist bspw. die Schwellenwertfunktion. Diese liefert nur entweder 0 (inaktiv) oder 1 (aktiv) als Ausgabe. (vgl. Lämmel und Cleve 2020, Kap. 5.1 Das künstliche Neuron, Absch. Aktivierungsfunktionen)

Eine der populärsten Funktionen in neuronalen Netzen ist die **logistische Funktion**. Sie bildet Werte im Bereich zwischen 0 und 1 ab (inklusive aller reellen Zahlen dazwischen). Die Funktion hat einen sigmoiden (s-förmigen) Charakter. Der große Vorteil dieser Funktion ist, dass sie differenzierbar ist. Dies ist wichtig für das Training. (vgl. Lämmel und Cleve 2020, Kap. 5.1 Das künstliche Neuron, Absch. Aktivierungsfunktionen)

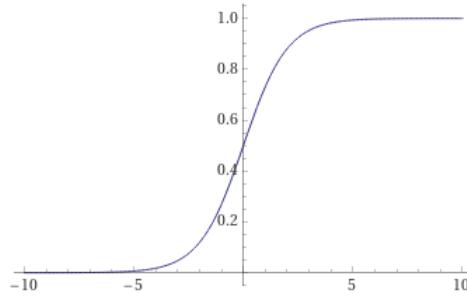


Abbildung 2.1: Die logistische Funktion zwischen -10 und 10

Für dieses Projekt wird daher ausschließlich die **logistische Funktion** und ihre Ableitung verwendet.

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

$$\phi'(x) = \phi(x) * (1 - \phi(x))$$

Vorwärtsgerichtete neuronale Netze

Durch die Zusammenschaltung mehrerer Neuronen entsteht nun ein Netz. Sehr geläufig sind hierbei **vorwärtsgerichtete Netze**. Dabei werden mehrere Neuronen zu einer sog. **Schicht** zusammengefasst. Innerhalb einer Schicht besteht keine Verbindung zwischen den Neuronen. Die Schichten werden in einer festen Reihenfolge nacheinander angeordnet. Jedes Neuron einer Schicht wird mit jedem Neuron der Folgeschicht verbunden. Die folgende Abbildung zeigt eine schematische Darstellung eines solchen Netzes:

Die weißen Kreise symbolisieren ein Neuron. Dabei sind die Neuronen auf der ersten (linken) Schicht lediglich Input-Neuronen. Diese bekommen also einfach die Inputwerte des Netzes übergeben. Alle Neuronen auf den Folgeschichten sind "echte" Neuronen wie sie zuvor beschrieben worden sind. Diese Werte werden nun über gewichtete Verbindungen (Kanten) an die Folgeschichten weitergeleitet. Die Biaswerte sind in dieser Grafik leider nicht dargestellt. Die nächsten zwei Schichten sind sog. verdeckte Schichten und die letzte ist die Ausgabe-Schicht. Das Netz ist also in der Lage einen Vektor an Eingabewerten zu verarbeiten und auch einen Vektor an Ausgabewerten zurückzugeben. (vgl. Sutton und Barto 2018, Kap. 9.7 Nonlinear Function Approximation: Artificial Neural Networks)

Die Berechnung des Ausgabe-Vektors einer Schicht setzt sich also aus den einzelnen Berechnungen für jedes einzelne Neuron zusammen. Da die Neuronen auf einer Schicht nicht miteinander verbunden sind, kann die Berechnung mit Vektoren und Matrizen umgesetzt werden. Die Mathematik dahinter ist dadurch recht elegant. Anstatt die Gewichte und den Bias für jedes Neuron separat zu speichern,

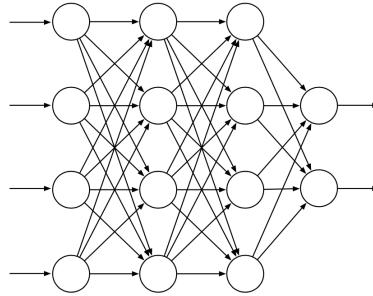


Abbildung 2.2: Ein vorwärtsgerichtetes Netz mit mehreren Schichten. (Grafik aus Sutton und Barto 2018)

kann eine Matrix W für die Gewichte und ein Vektor \vec{b} für die Biaswerte jedes Neurons verwendet werden. Die Gewichtsmatrix enthält die Gewichte zwischen den Neuronen der Schichten. Eine Spalte ist dabei dem Neuron auf der vorherigen, eine Zeile dem Neuron der Folgeschicht zugeordnet. Dadurch lässt sich die Ausgabe einer einzelnen Schicht wie folgt berechnen:

$$\vec{y} = \phi(\vec{b} + W \cdot \vec{x})$$

Zu beachten ist dabei, dass die Aktivierungsfunktion ϕ einen Vektor als Parameter übergeben bekommt. Sie verhält sich nun so, dass auf jeden Wert im Vektor die Funktion einzeln angewandt wird. Für jedes Element im Eingabevektor wird ein entsprechendes Wert im Ausgabevektor \vec{y} berechnet.

Topologie des Netzes

Da Pixelgrafiken mit 7×5 Pixeln analysiert werden sollen, muss die *Eingabeschicht* des Netzes über 35 Neuronen verfügen.

Es gibt 10 Ziffern (von 0 bis 9), die erkannt werden können. Auf der *Ausgabeschicht* wird es also 10 Neuronen geben. Jedes symbolisiert eine der Ziffern. Das Ausgabeneuron mit dem höchsten Wert ist die jeweils erkannte Ziffer.

Eine verdeckte Schicht ist vollkommen ausreichend, damit ein neuronales Netz jede beliebige Funktion darstellen kann. Zumal wenn nicht-lineare Aktivierungsfunktionen verwendet werden (vgl. Sutton und Barto 2018, Kap. 9.7 Nonlinear Function Approximation: Artificial Neural Networks). Daher wird in diesem Projekt auch nur mit *einer versteckten Schicht* gearbeitet.

Für die Anzahl der Neuronen auf der verdeckten Schicht gibt es kein Patentrezept. Je mehr Neuronen es sind, desto genauer kann die Zielfunktion gelernt werden. Allerdings werden die Berechnungen aufwendiger. Gleichzeitig sinkt die Fähigkeit des Netzes, Generalisierungen vorzunehmen. Man spricht hier vom sog. **Overfitting**. Das Netz „lernt“ eher die Beispieldaten auswendig, als den dahinterliegenden Algorithmus zu finden (vgl. Lämmel und Cleve 2020, Kap. 6.5.1 Die Größe der inneren Schicht). Für dieses Projekt wird daher folgendes Vorgehen angewandt: Es werden verschiedene Netze mit verschiedenen Neuronenzahlen auf der verdeckten Schicht generiert. Die Netze, welche direkt einen sehr kleinen Fehler über die Beispieldaten aufweisen, werden für ein paar Epochen (näheres in den folgenden Abschnitten) trainiert. Anschließend wird analysiert, wie schnell sich das jeweilige Netz trainieren lässt und wie gut es generalisiert. Das Netz, welches hier die besten Ergebnisse zeigt, wird dann für das weitere Training verwendet.

Aus der beschriebenen Topologie ergibt sich die folgende Formell zur Berechnung der Ausgabe \vec{y} des gesamten Netzes für die Eingabe \vec{x} :

$$\vec{y} = \phi(\vec{b}_o + W_o \cdot \phi(\vec{b}_h + W_h \cdot \vec{x}))$$

Die Bezeichner geben die jeweilige Schicht an, zu welcher ein Wert gehört. \vec{b}_o sind also bspw. die *Biaswerte* auf der *Output-Schicht* oder W_h bezeichnet die *Gewichtsmatrix* der verdeckten Schicht (von engl. *hidden layer*).

Gradientenabstiegsverfahren

Nun stellt sich die Frage, wie das neuronale Netz die gesuchte Funktion approximiert bzw. wie überhaupt das Training durchgeführt wird. Die Gewichte und Biaswerte der versteckten und der Output-Schicht müssen richtig eingestellt werden, damit das Netz die Ziffernerkennung durchführen kann. Eines der gängigsten Lernverfahren für diese Aufgabe ist das **Gradientenabstiegsverfahren**. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren)

Grundprinzip

Gradientenabstiegsverfahren werden verwendet, um das Minimum einer Funktion zu finden. Man beginnt an einem zufälligen Punkt auf der Funktion. Nun wird der *Anstieg* (auch Gradient genannt) in diesem Punkt berechnet. Da ein Minimum der kleinste mögliche Wert einer Funktion ist, bewegt man sich nun entgegengesetzt des Anstiegs entlang der Funktion. Je stärker der Anstieg, umso größer fällt auch der Schritt in die entgegengesetzte Richtung aus. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren)

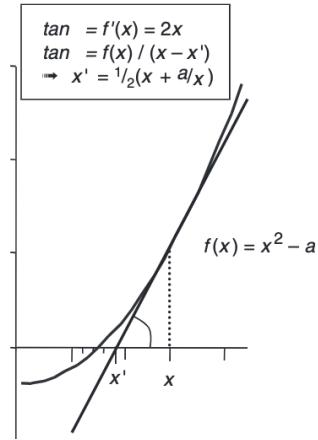


Abbildung 2.3: Das Newton-Verfahren als der Ursprung des Gradientenabstiegsverfahren. (Grafik aus Lämmel und Cleve 2020)

Mit diesem Verfahren geschieht aber nur eine Annäherung an das Minimum. Die Idee ist, dass man dieses Vorgehen mehrmals hintereinander durchführt, bis das Minimum der Funktion gefunden ist. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren)

Fehlerfunktion

Das neuronale Netz muss nun also so dargestellt werden, dass die Gewichte und Biaswerte als Parameter für eine Funktion dienen. Gleichzeitig muss diese Funktion ein Minimum haben, welches gleichzeitig bei der Lösung der Aufgabe hilft. Die Lösung: man definiert eine sog. **Fehlerfunktion**, welche die Abweichung zwischen den berechneten Ausgaben des neuronalen Netzes und den erwarteten Ausgabewerten aus den Beispieldaten bestimmt. Die verschiedenen Fehlerwerte zwischen den berechneten und den erwarteten Ausgabewerten werden in einen einzelnen Fehlerwert komprimiert. Da die Ausgabe des Netzes einzig von den Gewichten und Biaswerten beeinflusst wird, kann durch die Anpassung der selbigen die Differenz und damit der Fehler *minimiert* werden. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren; und Patterson und Gibson 2017, Kap. 2 Foundations of Neural Networks and Deep Learning, Absch. Loss Functions)

Auch für die Fehlerfunktion gibt es eine große Fülle an Möglichkeiten. Eine der universellsten ist die **Mean Squared Error** Funktion (zu deutsch: die Funktion der durchschnittlichen Fehlerquadrate). Diese Funktion bietet mehrere Vorteile: Zum Einen sind die Differenzen nicht vorzeichenbehaftet. Dadurch heben sich zwei fehlerhafte Neuronen mit entgegengesetzten Fehlerwerten nicht gegenseitig auf. Zum Anderen sind die Fehlerwerte für große Abweichungen sehr groß, kleinere Abweichung werden hingegen weniger stark gewichtet. Dadurch ist die Gefahr des Overfitting geringer und das Netz neigt zu einer besseren Generalisierung. Zu guter Letzt sei erwähnt, dass diese Funktion gut differenzierbar ist. (vgl. Patterson und Gibson 2017, Kap. 2 Foundations of Neural Networks and Deep Learning, Absch. Loss Functions)

$$E(\vec{y}, \vec{\hat{y}}) = \frac{1}{n} \cdot \sum (y_i - \hat{y}_i)^2$$

$$E'_i(\vec{y}, \vec{\hat{y}}) = \frac{2}{n} \cdot (y_i - \hat{y}_i)$$

Die Ableitung der Mean Squared Error Funktion ist hier nur für das i-te Ausgabeneuron dargestellt. Natürlich wird sie auf alle Ausgabeneuronen angewendet, sodass man einen Gradienten für jedes der Ausgabeneuronen erhält.

Anwendung im neuronalen Netz

Man beginnt mit zufälligen Werten für die Gewichte und Bias. Als nächstes wird der Gradient Δ zu jedem veränderbaren Parameter ermittelt. Anschließend wird dieser Gradient mit einer **Lernrate** λ multipliziert. Das Ergebnis wird nun vom ursprünglichen Wert abgezogen. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren)

$$W_{neu} = W_{alt} - \lambda \cdot \Delta$$

Wie bereits erwähnt, nähert man sich mit einem Trainingsdurchlauf dem Minimum der Fehlerfunktion lediglich an. Um tatsächlich ein Minimum zu erreichen, müssen viele solcher Durchläufe hintereinander durchgeführt werden. Ein solcher Trainingsdurchlauf über das gesamte Set an Trainingsdaten wird auch als **Epoche** bezeichnet. (vgl. Patterson und Gibson 2017, Kap. 6 Tuning Deep Networks, Absch. Controlling Epochs and Mini-Batch Size)

Da es verschiedene Gewichte und Biaswerte auf den verschiedenen Schichten gibt, muss die Fehlerfunktion stets nach dem jeweiligen Wert abgeleitet werden. Daher ergeben sich folgende Gleichung für die Gradienten:

$$\begin{aligned}\Delta \vec{b}_o &= \frac{\delta E}{\delta \vec{b}_o} = E'(\vec{y}, \vec{\hat{y}}) \cdot \phi'(\vec{b}_o + W_o \cdot \vec{x}_h) \\ \Delta W_o &= \frac{\delta E}{\delta W_o} = E'(\vec{y}, \vec{\hat{y}}) \cdot \phi'(\vec{b}_o + W_o \cdot \vec{x}_h) \cdot \vec{x}_h^T \\ \Delta \vec{b}_h &= \frac{\delta E}{\delta \vec{b}_h} = E'(\vec{y}, \vec{\hat{y}}) \cdot \phi'(\vec{b}_o + W_o \cdot \vec{x}_h) \cdot W_o^T \cdot \phi'(\vec{b}_h + W_h \cdot \vec{x}) \\ \Delta W_h &= \frac{\delta E}{\delta W_h} = E'(\vec{y}, \vec{\hat{y}}) \cdot \phi'(\vec{b}_o + W_o \cdot \vec{x}_h) \cdot W_o^T \cdot \phi'(\vec{b}_h + W_h \cdot \vec{x}) \cdot \vec{x}^T\end{aligned}$$

Die Netzausgabe ist eine stark in sich verschachtelte Funktion. Daher muss hier exzessiv von der Kettenregel Gebrauch gemacht werden. Bei \vec{x}_h handelt es sich um den Ausgabevektor der versteckten Schicht, welcher ja gleichzeitig der Eingabevektor für die Output-Schicht ist.

Mit den Gradienten können nun die Gewichte und Biaswerte im gesamten Netz angepasst werden. Da man hier praktisch zuerst eine Berechnung über das Netz durchführt und dann nachträglich die Parameter korrigiert, wird dieses Verfahren auch als **Backpropagation** bezeichnet. (vgl. Lämmel und Cleve 2020, Kap. 6.2.1 Das Backpropagation-Verfahren)

Lernrate

Die Annäherung an das Minimum der Fehlerfunktion erfolgt über viele Epochen. Dieses Verfahren ist aber alles andere als präzise. Würde der Gradient voll von den Gewichten und Biaswerten abgezogen werden, so würden die Korrekturen recht stark ausfallen. Im schlimmsten Fall “springt” der Fehler mit jedem Training um das Minimum herum, erreicht es aber nicht, da die Korrekturen zu groß sind. Daher wird der Gradient mit einer Lernrate zwischen 0 und 1 multipliziert. Dadurch ist die Korrektur wesentlich feiner und die Chancen stehen besser, tatsächlich genau das Minimum zu treffen. Allerdings fallen dadurch die Korrekturen insgesamt viel kleiner aus und es sind viel mehr Epochen nötig als ohne Lernrate. (vgl. Patterson und Gibson 2017, Kap. 2 Foundations of Neural Networks and Deep Learning, Absch. Learning Rate)

Es muss also ein Gleichgewicht gefunden werden. Die Lernrate muss groß genug sein, damit das Training nicht ewig dauert. Gleichzeitig darf sie nicht zu groß sein, damit eine möglichst präzise

Annäherung an das Minimum möglich ist. Dies ist kein leichtes Unterfangen, zumal die Lernrate fast der wichtigste Hyperparameter beim Training neuronaler Netz ist. (vgl. Patterson und Gibson 2017, Kap. 6 Tuning Deep Networks, Absch. Understanding Learning Rates)

Es existieren sehr vielfältige Ansätze zur Lösung dieses Problems: Einige Verfahren versuchen mit einer möglichst hohen Lernrate zu beginnen und diese im Verlaufe des Trainings geschickt zu reduzieren. Andere Optimierungen berechnen spezielle Lernraten für jedes der Neuronen einzeln. Der bekannteste und wohl erfolgreichste Algorithmus in diesem Zusammenhang ist *Adam*. (vgl. Kingma und Lei Ba 2014; und Patterson und Gibson 2017, Kap. 6 Tuning Deep Networks, Absch. Understanding Learning Rates)

Diese Verfahren sind recht komplex zu implementieren und der Erfolg ist nicht Gewiss. Außerdem ist der Umfang an Beispieldaten und die Netzgröße für dieses Projekt überschaubar klein. Daher wird das Training ausschließlich mit einer *konstanten Lernrate* durchgeführt werden. Dennoch werden zunächst verschiedene Lernraten über ein paar Epochen getestet und anschließend der optimale Wert bestimmt.

Batch-Learning

Eine letzte Optimierung, die beim Training zum Einsatz kommen wird, ist das **Batch-Learning**.

Die Beispieldaten umfassen mehrere verschiedene Grafiken mit unterschiedlichen Ziffern. Da das Netz alle diese Ziffern erkennen soll, müsste nacheinander stets eine andere Ziffer für das Training verwendet werden. Sobald man mit dem Set durch ist, beginnt man wieder von vorne. Dabei kann es passieren, dass zwei Ziffern genau gegensätzliche Gradienten erzeugen und sich die Korrekturen im Kreis drehen. Dadurch dauert das Training sehr lange.

Das Batch-Learning verbessert diesen Prozess. Dabei werden mit dem gleichen Netz die Gradienten für mehrere Datensätze gleichzeitig berechnet. Aus den verschiedenen Gradienten wird nun der Durchschnitt gebildet. Dieser wird anschließend mit der Lernrate multipliziert und von den Gewichten abgezogen. Dadurch wird mit einem Trainingsschritt ein ganzes Set an Daten angelernt. (vgl. Patterson und Gibson 2017, Kap. 2 Foundations of Neural Networks and Deep Learning, Abs. Backpropagation and Mini-Batch Stochastic Gradient Descent)

3 Umsetzung in Python

Dieses Projekt befindet sich in einem Git-Repository. Dort befinden sich alle Dateien des Projektes, inklusive Source-Code, Dokumentation usw. Das Repository ist unter diesem Link erreichbar: <https://github.com/hd-code/digit-recognition-ci/>

Installation

Das Projekt ist in Python umgesetzt. Zusätzlich ist der Package-Manager Pipenv verwendet worden, um alle benötigten Software-Komponenten zu installieren und zu verwalten.

Voraussetzungen

Die Voraussetzungen um das Projekt ausführen zu können sind:

- Das Projekt inklusive aller Dateien muss sich auf dem Computer befinden.
- **Python** ab *Version 3.9 oder höher* – Informationen zur Installation gibt es hier: <https://www.python.org>
- **Pipenv** ab *Version 2020.11 oder höher* – Installation ist über folgenden Befehl im Terminal möglich: `pip install pipenv`

Hinweis: Damit der Befehl zur Installation von Pipenv funktioniert, muss Python bereits auf dem System installiert sein.

Installation der externen Software-Komponenten

Nun muss ein Terminal im Ordner des Projektes geöffnet werden. Durch die Ausführung des folgenden Befehls werden alle benötigten Software-Komponenten automatisch installiert:

```
pipenv install
```

Das ist alles. Nun kann das Projekt verwendet werden.

Externe Software-Komponenten

Wie die benötigten Software-Komponenten installiert werden, ist im vorherigen Abschnitt erklärt worden. Hier folgt eine Auflistung, was an externen Software-Komponenten benutzt worden ist und warum.

- **numpy** – eine Bibliothek für schnelles und effizientes Rechnen mit Vektoren und Matrizen.
- **pandas** – eine Bibliothek zum einfachen Speichern, Laden und Visualisieren von tabellarischen Daten.
- **matplotlib** – wird von **pandas** benötigt, um Daten in Form von Grafiken visualisieren zu können.
- **jupyter** – ist eine Entwicklungsumgebung, in der Codeschnipsel geschrieben, ausgeführt und die Ergebnisse direkt angezeigt werden können.
- **ipykernel** – wird von **jupyter** benötigt, um Python-Code ausführen zu können.
- **PySimpleGui** – eine Bibliothek, um interaktive Programme mit einer GUI zu erstellen.

Projektübersicht

Projektstruktur

Damit das Projekt übersichtlich bleibt, ist es in Ordner und Unterordner gegliedert. Die Ordner sind:

- **data**
 - **cache** – enthält die Ergebnisse einer Simulation
 - **digits** – enthält die Ziffern für die Trainings- und Testdaten als CSV-Dateien
- **docs** – Dokumentation des Projektes (also faktisch dieses Dokument).

- **src** – Source-Code, weitere Erklärung in den folgenden Abschnitten

Hauptprogramm

Im Hauptprogramm wird das neuronale Netz initialisiert, trainiert und analysiert. Die entsprechende Datei ist **src > main.ipynb**.

Zum komfortableren Arbeiten ist hierfür ein **Jupyter Notebook** verwendet worden. Ein Jupyter Notebook ist ein Dokument, welches im Webbrowser geöffnet und bearbeitet werden kann. Man hat hier sog. Code-Zellen. In diese kann Code eingefügt und direkt ausgeführt werden. Die Ergebnisse werden direkt unter der Zelle dargestellt. Ändert man den Code in der Zelle und führt ihn erneut aus, so werden die Ergebnisse entsprechend aktualisiert. Dadurch eignen sich Jupyter Notebooks optimal für das maschinelle Lernen. Es ist sehr einfach Simulationen mit verschiedenen Parametern auszuführen bis die optimalen Werte gefunden sind.

Vor der Verwendung von Jupyter muss der folgende Befehl im Terminal im Ordner des Projektes ausgeführt:

```
pipenv run jupyter notebook
```

Nun muss im Webbrowser die folgende Seite aufgerufen werden: <http://localhost:8888>. Jetzt kann man zur besagten Datei navigieren, diese öffnen und bearbeiten.

Zuletzt noch einige Hinweise zu der Datei: Wenn in den Code-Zellen eine Variable ausschließlich in Großbuchstaben geschrieben ist (z.B. **DIGITS**), dann wird diese Variable auch in anderen Code-Zellen der Datei verwendet. Die Zelle, wo die Variable definiert wird, muss also auf jeden Fall ausgeführt werden, bevor eine der folgenden Zellen ausgeführt werden kann. Die generierten Daten werden im Ordner **data > cache** abgespeichert (in den entsprechenden Code-Zellen befinden sich Kommentare). Dadurch können die Ergebnisse der letzten Simulation geladen werden und das zeitintensive Training muss nicht jedes mal erneut durchgeführt werden.

Demo-App

Mit der Demo-App kann eine interaktive Anwendung gestartet werden, um das generierte neuronale Netz zu testen. Die Implementierung befindet sich in der Datei **src > app.py**.

Zum Starten der App bitte folgenden Befehl im Terminal im Ordner des Projektes ausführen:

```
pipenv run python src/app.py
```

Es erscheint ein Fenster mit einem Feld aus Pixeln auf der linken und der Ausgabe des neuronalen Netzes auf der rechten Seite. Die Pixel können angeklickt werden. Dadurch wird ein Pixel von leer zu gefüllt geändert und umgekehrt. Nach einem Klick wird sofort die Berechnung über das Netz durchgeführt und das Ergebnis angezeigt.

Hilfsmodule und -packages

Python erlaubt die Gliederung des Source-Codes in Module (einzelne Dateien) und Packages (ein Ordner mit mehreren Dateien, die zu einer Einheit zusammengefasst werden). Dies wird benutzt, um verschiedene Teile des Programms in wiederverwendbare Komponenten auszulagern.

Digits-Module

Dieses Modul lädt die Ziffern (also die Beispieldaten), welche sich im Ordner **data > digits** als CSV-Dateien befinden. Die Implementierung befindet sich in der Datei **src > digits.py**. Es ist möglich die Ziffern zu filtern (nach der Ziffer selbst oder nach dem Set, zu welchem eine Ziffer gehört).

Die Ziffern sind so gespeichert, dass der Dateiname mit der Ziffer beginnt, welche sich hinter der Datei verbirgt. Es folgt der Name des Ziffern-Sets mit Bindestrichen getrennt (näheres im nächsten Kapitel). So lautet der Dateiname für die Ziffer 5 des evag Datensets z.B. **5-evag.csv**.

Net-Package

Dieses Package implementiert das neuronale Netz als wiederverwendbare Bibliothek. Es besteht aus mehreren Dateien, die sich alle im Ordner **src > net** befinden. Alle Teilespekte eines neuronalen Netzes finden sich hier wieder (Aktivierungs- und Fehlerfunktionen, die Schichten sowie die Möglichkeit ein Netz zu speichern und zu laden).

Die Datei `__init__.py` legt fest, welche Methoden dieses Package nach außen bereitstellt. Es gibt Methoden, um ein Netz mit verschiedenen Neuronen zu initialisieren (`init`), Berechnungen durchzuführen (`calc` und `calcBatch`), Fehlerwerte zu ermitteln (`calcError` und `calcBatchError`) und natürlich ein Netz zu trainieren (`train` und `trainBatch`). Zusätzlich kann ein Netz auf der Festplatte gespeichert (`save`) und wieder geladen werden (`load`).

Testing

Die Hilfsmodule müssen ordentlich funktionieren, damit sie bedenkenlos eingebunden und wieder verwendet werden können. Deshalb befindet sich am Ende von allen diesen Dateien ein Abschnitt, welcher den Code entsprechend testet. Um eine Datei zu testen, muss sie direkt im Terminal ausgeführt werden. Dies geht im Projektordner im Terminal über folgenden Befehl:

```
PYTHONPATH=src pipenv run python src/<path-to-file>.py
```

Sollte es bei der Ausführung zu Fehlern kommen, so werden diese im Terminal angezeigt. Andernfalls ist am Ende `SUCCESS` zu sehen.

Mit folgendem Befehl können auch alle Hilfsmodule auf einmal getestet werden:

```
files=(src/digits.py src/net/[^_]*.py)
for f in $files; do PYTHONPATH=src pipenv run python $f; done
```

4 Computersimulation

Net-Package

Das Herzstück des Projektes ist das Net-Package, welches das neuronale Netz umsetzt. Dies ist als erstes implementiert und fertiggestellt worden.

Man kann damit neuronale Netze mit verschiedenen Neuronenzahlen auf Input-, Hidden- und Output-Schicht erzeugen. Es kann allerdings nur genau eine versteckte Schicht verwendet werden. Die Gewichte und Biaswerte werden mit Zufallswerten zwischen -1 und $+1$ initialisiert. Der Zufallsgenerator der Bibliothek `numpy` ist hierfür verwendet worden. Es ist möglich den Zufallsgenerator mit einem festen Startwert zu initialisieren (`seed()`). Dadurch wird stets die gleiche Reihe an Zufallszahlen generiert.

Die Aktivierungsfunktion ist die logistische Funktion und sie ist fest in das neuronale Netz programmiert. Gleichermaßen bei der Fehlerfunktion, welche die Mean Squared Error Funktion ist.

Das Package ist in der Lage sowohl im Batch- als auch im Online-Modus zu arbeiten. In dieser Simulation ist allerdings nur das Batch-Verfahren verwendet worden.

Trainings- und Testdaten

Für das Training sind fünf verschiedene Sets an Ziffern von 0 bis 9 (also insgesamt 50 Ziffern) erstellt worden. Die folgende Grafik zeigt die verschiedenen Ziffern:

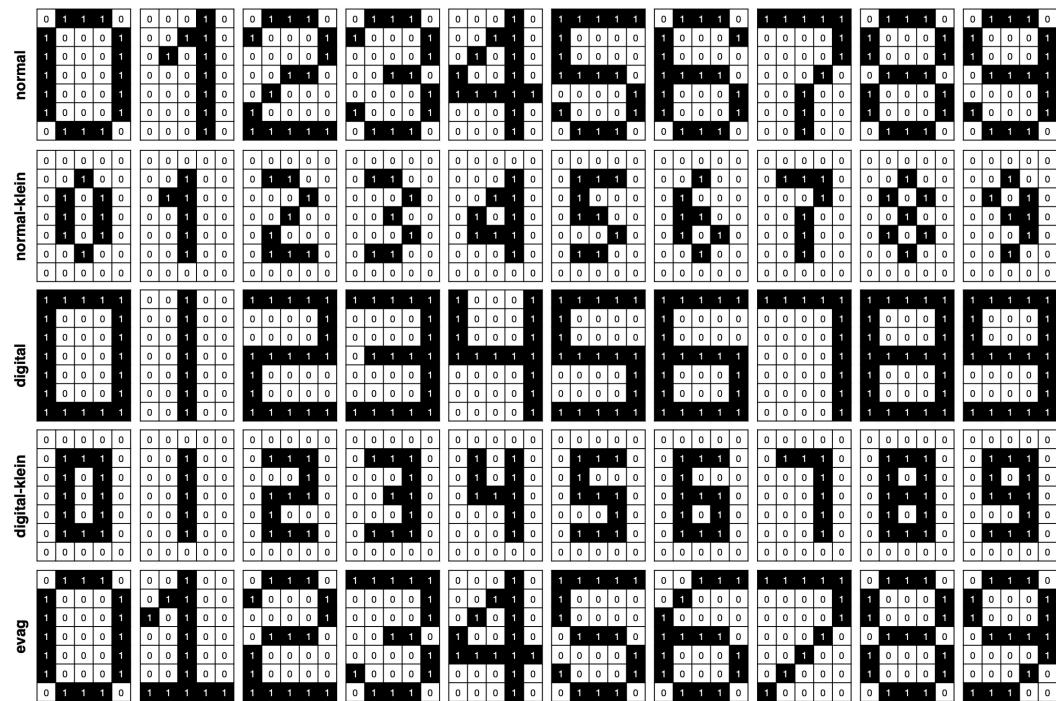


Abbildung 4.1: Die Beispieldatensätze – verschiedene Grafiken von Ziffern zwischen 0 und 9

Wie in der Aufgabe beschrieben, sind Grafiken mit 7×5 Pixeln erstellt worden. Ein Pixel kann entweder leer (0) oder gefüllt (1) sein.

Ziffern aus dem Set namens “**normal**” füllen das gesamte Raster aus. Sie entsprechen den typischen Darstellungen der arabischen Ziffern in geläufigen Schriftarten.

Mit dem Set “**normal-klein**” ist eine kleinere Variante des Sets “normal” geschaffen worden. Bei diesem Set bleibt die äußerste Reihe der Pixel auf allen vier Seiten frei. Vom Stil her ähneln sie aber dem Set “normal”.

Ein weiteres Set trägt den Namen “**digital**”. Das komplette Raster wird von einer Ziffer ausgefüllt. Die Darstellung orientiert sich an Ziffern auf digitalen Uhren. Sie zeichnen sich durch klare gerade und einfache Linien aus.

Auch hierzu gibt es eine kleinere Variante namens “**digital-klein**”. Ähnlich wie bei “normal-klein” bleibt hier ebenfalls die äußere Pixelreihe auf allen Seiten frei. Es handelt sich um die Ziffern aus “digital” in einer kleineren Form.

Diese vier Sets bilden zusammen die *Trainingsdaten*. Sie bilden ein breites Spektrum von Ziffern in verschiedenen Darstellungen und Größen ab.

Es gibt allerdings noch ein weiteres Set namens “**evag**”. Die Erfurter Verkehrsbetriebe AG nutzt das gleiche Pixelraster von 7 x 5 Pixel auf den Anzeigetafeln an den Straßenbahn-Haltestellen in Erfurt. Die Ziffern sind denen aus dem Set “normal” sehr ähnlich. Die 0, 4 und 8 sind sogar komplett identisch. Bei allen anderen Ziffern gibt es leichte Abweichungen zu den bisherigen Sets.

Das “evag” Set stellt somit die *Testdaten*. Es wird also nicht für das Training verwendet. Stattdessen wird damit überprüft, ob das neuronale Netz auch mit unbekannten, leicht abgewandelten Datensätzen zurechtkommt.

Wie in Kapitel 3 beschrieben, sind die Datensätze in CSV-Dateien gespeichert. Sie werden durch das Digits-Module geladen und zur Verfügung gestellt.

Ermittlung der Neuronenzahl auf der versteckten Schicht

Ab diesem Punkt findet sich die gesamte Implementierung im Hauptprogramm (siehe Kapitel 3).

In Kapitel 2 ist dargelegt worden, dass die Neuronenzahl auf der versteckten Schicht experimentell ermittelt werden muss. Dies ist also die erste Aufgabe, die es zu lösen gilt.

Geringster initialer Fehler

Zunächst werden Netze mit 5, 10, 15, 20, 25, 30, 35, 40, 50, 60, 70, 80 und 90 Neuronen auf der versteckten Schicht betrachtet. Zu jeder Neuronenzahl werden jeweils 100 verschiedene Netze zufällig generiert (also 100 Netze mit 5 Neuronen auf der versteckten Schicht, 100 Netze mit 10 Neuronen usw.). Von diesen Netzen wird allerdings nur das Netz mit dem geringsten Fehlerwert über alle Beispieldaten ausgewählt. Am Ende bleibt also ein Sieger-Netz je Neuronenzahl übrig.

Das Ziel hierbei ist es, für jede Neuronenzahl ein Netz zu generieren, welches schon ganz gute Ausgangs-Werte liefert. Nur so können die Netze nun weiter miteinander verglichen werden.

Fehler im Verlauf des Trainings

Als nächstes werden diese Sieger-Netze mit einer Lernrate von 0,1 über 1.000 Epochen mit den Trainingsdaten trainiert. Während des Trainings wird der Fehler über die Trainings- und Testdaten ermittelt und gespeichert. Am Ende werden die Ergebnisse miteinander verglichen.

Das Netz, welches hier die besten und schnellsten Lernerfolge zeigt und gleichzeitig gut generalisieren kann, ist der Gewinner. Dieses Netz verfügt über die geeignete Anzahl an Neuronen auf der versteckten Schicht.

Ermittlung der optimalen Lernrate

Nachdem die optimale Anzahl an Neuronen auf der versteckten Schicht gefunden worden ist, geht es nun um die Lernrate. Auch sie ist ein wichtiger Parameter, der am ehesten experimentell ermittelt werden kann.

Es wird nun wieder die initiale Version des optimalen Netzes verwendet (d.h. die bereits durchgeführten Trainingsdurchläufe werden verworfen). Diese wird nun mehrmals mit verschiedenen Lernraten trainiert. Der Trainingsverlauf wird wieder mitverfolgt und anschließend analysiert. Es gilt die Lernrate als optimal, wo der Fehler am schnellsten, gleichmäßigsten und zuverlässigsten minimiert wird. Dauert das Training zu lange, ist sie zu klein. Wenn der Fehler an einem Punkt ein Plateau erreicht oder anfängt zu “springen”, dann ist sie zu hoch.

Es werden die Lernraten 1; 0,1; 0,01 und 0,001 betrachtet. Um sichere Erkenntnisse zu gewinnen, werden 10.000 Epochen durchlaufen.

Finales Training

Da nun (hoffentlich) die optimale Anzahl an versteckten Neuronen und eine geeignete Lernrate gefunden ist, geht es an das finale Training. Das Netz wird nun solange weitertrainiert, bis der Fehler über den Beispieldaten nahezu 0 ist. Damit ist das Netz fertig modifiziert.

Abschließend wird mittels der Demo-App stichprobenartig analysiert, ob das Projekt tatsächlich erfolgreich gewesen ist und die Ziffern richtig erkannt werden.

5 Ergebnisse

Ermittlung der Neuronenzahl auf der versteckten Schicht

Geringster initialer Fehler

Wie bereits beschrieben, sind zunächst jeweils 100 Netze pro Neuronenzahl auf der versteckten Schicht zufällig generiert worden. Anschließend wurde der Fehler über alle Beispieldaten berechnet. Die folgende Grafik zeigt die Ergebnisse:

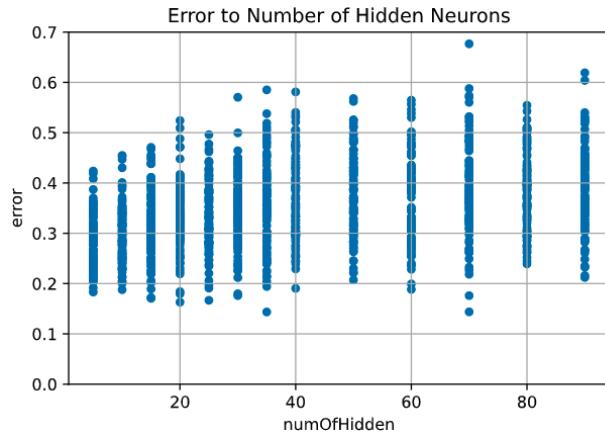


Abbildung 5.1: Fehler der verschiedenen generierten Netze nach Neuronenzahl auf der versteckten Schicht

Es ist deutlich zu sehen, dass die Fehler aller Netze ziemlich gleichmäßig verteilt sind. Ab 40 Neuronen auf der versteckten Schicht werden die Fehler aber tendenziell eher höher. Das ist ein Zeichen, dass die optimale Neuronenzahl eher unter 40 liegt. Allerdings gibt es hier eine Ausnahme: Einige Netze mit 70 Neuronen haben ebenfalls sehr geringe Fehlerwerte.

Für die Lösung der Aufgabe sind nur Netze mit geringen Fehlerwerten von Belang. Daher zeigt die folgende Tabelle zu jeder Neuronenzahl das Netz mit dem geringsten Fehler. Die Werte sind außerdem vom geringsten zum größten Fehler sortiert worden.

Neuronenzahl	Fehlerwert des besten Netzes
35	0,1435
70	0,1437
20	0,1629
25	0,1667
15	0,1707
30	0,1760
5	0,1830
10	0,1878
60	0,1885
40	0,1904
50	0,2071
90	0,2121
80	0,2391

Die Tabelle bestätigt die erste Beobachtung: Netze mit 40 Neuronen oder mehr haben einen sehr hohen Fehler. Zusätzlich ist zu sehen, dass die Netze mit 5 und 10 Neuronen ebenfalls einen etwas höheren Fehler aufweisen. Die optimale Neuronenzahl scheint also eher zwischen 15 und 35 Neuronen zu liegen. Einzige Ausnahme ist der Ausreißer mit 70 Neuronen, welcher den 2. Platz belegt.

Fehlerwerte im Verlauf des Trainings

Nun soll überprüft werden, wie schnell die jeweiligen Netze lernen können. Auch ihre Fähigkeit zur Generalisierung wird nun auf die Probe gestellt.

Wie bereits beschrieben, sind die Netze mit den kleinsten Fehlerwerten für 1.000 Epochen mit einer Lernrate von 0,1 trainiert worden. Allerdings sind direkt die Erkenntnisse aus der ersten Betrachtung hier eingeflossen: Es sind nur die Netze mit Neuronenzahlen von 15 bis 35 und zusätzlich das 70er Netz trainiert worden. Die folgende Tabelle zeigt den Lernerfolg der verschiedenen Netze:

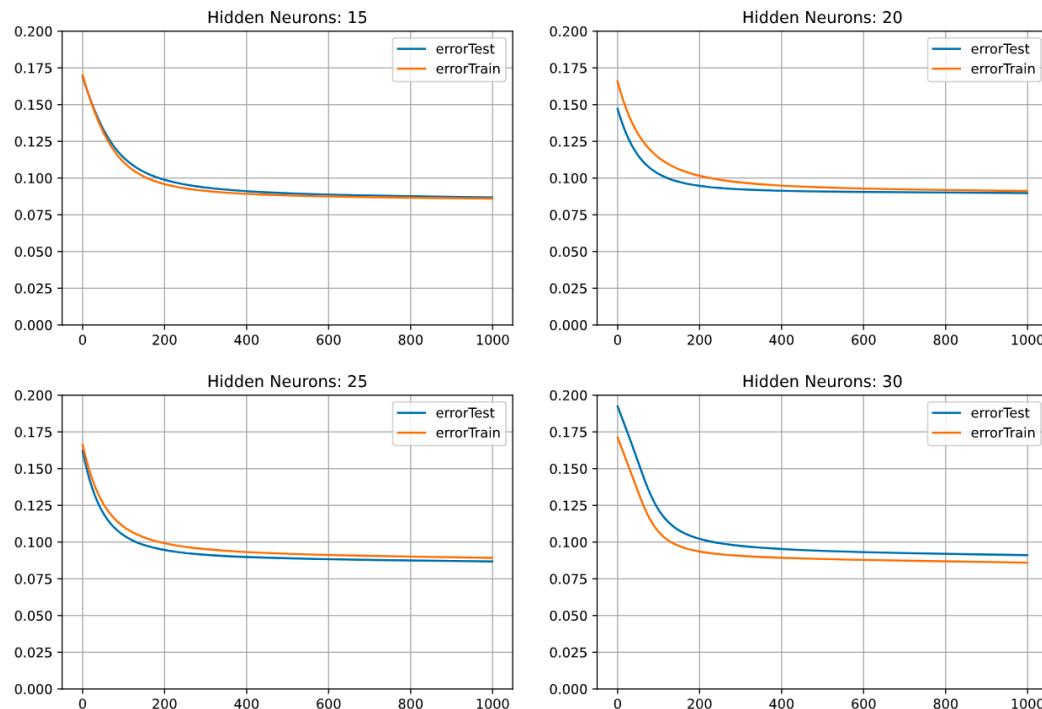
Neuronenzahl	Fehlerwert zu Beginn	Fehlerwert nach dem Training	Differenz
30	0,1760	0,0860	0,0900
15	0,1707	0,0860	0,0846
25	0,1667	0,0892	0,0775
20	0,1629	0,0912	0,0717
35	0,1435	0,0840	0,0595
70	0,1437	0,0885	0,0551

Die Werte sind absteigend nach der Spalte “Differenz” sortiert. Diese Spalte gibt an, wie stark sich der Fehlerwert durch das Training verbessert hat.

Interessanterweise haben die beiden Netze, welche zu Beginn den geringsten Fehlerwert hatten (35 und 70), einen sehr geringen Trainingserfolg zu verbuchen. Sie haben einfach zu viele Neuronen auf der verdeckten Schicht und lassen sich dadurch schlechter trainieren. Damit scheiden sie als Kandidaten ebenfalls aus.

Fähigkeit zur Generalisierung

Es verbleiben 4 Netze mit 15, 20, 25 und 30 Neuronen auf der verdeckten Schicht. Um hier eine Entscheidung zu treffen, werden die Verläufe der Fehlerwerte von den Trainings- und den Testdaten nun analysiert:



Die blaue Linie zeigt den Verlauf der Fehlerwerte der Trainingsdaten. Die orangene zeigt den Verlauf für die Testdaten.

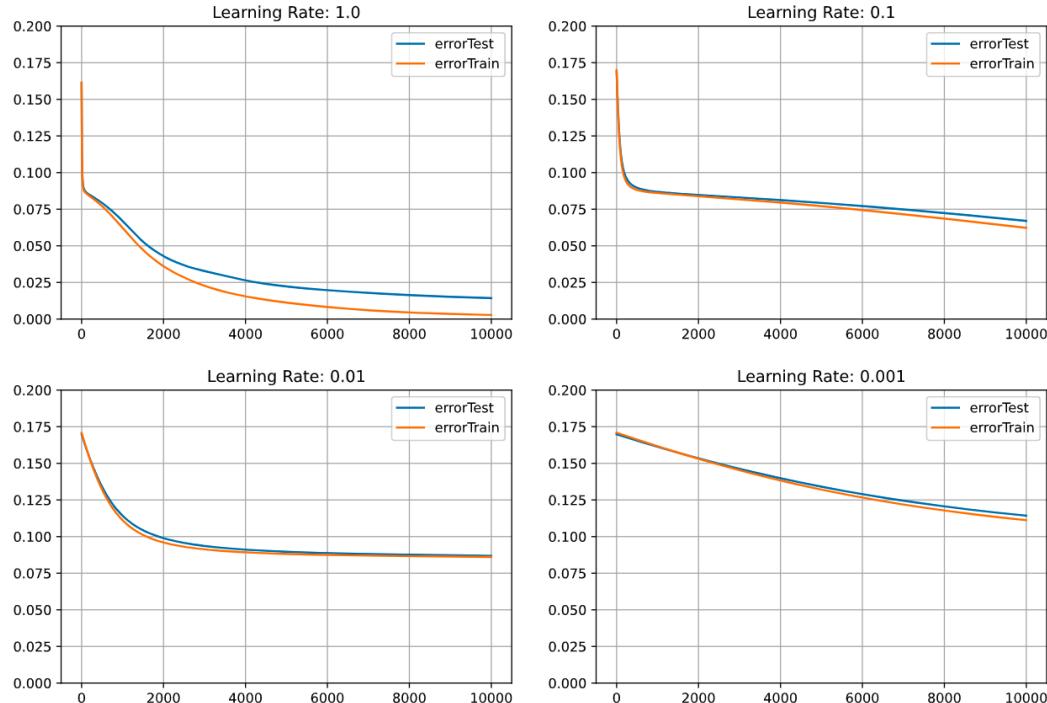
Die Fehlerwerte der Netze entwickeln sich ziemlich gleichmäßig. Allerdings fällt auf, dass die Fehlerwerte der Trainings- und Testdaten umso weiter auseinanderliegen, je höher die Neuronenzahl auf der verdeckten Schicht ist. Dies lässt sich mit der theoretischen Grundlage gut erklären: Je

weniger Neuronen sich auf der versteckten Schicht befinden, desto eher muss das Netz generalisieren, um die korrekten Ausgaben zu erzeugen (siehe Kapitel 2).

Damit ist die optimale Anzahl an Neuronen für die versteckte Schicht gefunden: Mit **15 Neuronen** können die Ergebnisse zuverlässig und schnell gelernt werden. Gleichzeitig führt das Netz ein hohes Maß an Generalisierung durch und kommt so mit einer Vielzahl von Eingabewerte zurecht.

Ermittlung der optimalen Lernrate

Als nächstes gilt es, die optimale Lernrate zu finden. Dazu sind alle durchgeführten Trainingsepochen wieder verworfen worden. Das Netz besitzt also wieder die initial generierten Werte. Es ist nun mit 4 unterschiedlichen Lernraten für 10.000 Epochen trainiert worden. Die Lernraten sind: 1; 0,1; 0,01 und 0,001. Die folgenden Grafiken zeigen die Ergebnisse:



Alle vier Grafiken zeigen einen identischen Verlauf der Fehlerwerte. Einziger Unterschied: je niedriger die Lernrate, desto länger zieht sich das Training.

Zunächst sinkt der Fehler sehr schnell. Dann gibt es eine recht starke "Kurve" und die Geschwindigkeit des Trainings wird stark gebremst. Fortan verläuft das Training recht langsam. Im weiteren Verlauf nimmt es wieder etwas Fahrt auf. Je weiter sich der Fehler nun der 0 annähert, desto langsamer verläuft das Training wiederrum. Außerdem bewegen sich die Trainings- und Testdaten auf diesem Abschnitt weiter auseinander als zuvor. Über den gesamten Verlauf sind aber sowohl die Fehlerwerte für die Trainings- als auch für die Testdaten am fallen. Dies gilt für alle betrachteten Lernraten.

Unterm Strich kann ohne Probleme die Lernrate von 1 (was bedeutet, dass gar keine Lernrate nötig ist) verwendet werden. Der Fehler für diese Lernrate ist am Ende dieses Testlaufes bereits nahe der 0, was der optimale Fehlerwert ist. Es scheint kein Plateau zu entstehen. Zu keiner Zeit fangen die Fehlerwerte an, sich wieder nach oben zu bewegen oder zu springen. Dies lässt sich durch mehrere Faktoren erklären:

Es sind nur sehr wenige Beispieldaten zu lernen. Es sind so wenige Datensätze, dass alle Trainingsdaten in einen einzigen Batch passen. Das heißt, in jeder Epoche wird nur der durchschnittliche Gradient für einen einzigen Batch berechnet. Mehr Batches gibt es gar nicht. Dadurch werden die Gradienten stets in eine ähnliche Richtung zeigen wie im vorherigen Trainingsschritt. Eine Reduzierung der Schrittweite durch die Lernrate würde die Bewegung also nur bremsen, aber nicht optimieren.

Ein weiterer Grund liegt in der Mean Squared Error Funktion. Wie bereits beschrieben, liefert diese Funktion recht hohe Gradienten, wenn die Differenz zwischen berechneten und erwarteten Werten groß ist. Bei geringen Differenzen entstehen aber extra kleine Gradienten (das liegt an

der Quadrierung). Dadurch sind die Gradienten am Anfang des Trainings sehr groß und werden zunehmend immer kleiner. Das ist genau das Vorgehen, welches auch für die Lernrate empfohlen wird. Die Fehlerfunktion unterstützt das Training also bereits optimal. Daher sind die Anpassungen trotz hoher Lernrate fein genug.

Die kleineren Lernraten würden das Training nur unnötig in die Länge ziehen. Daher ist die **1,0** die optimale Lernrate für dieses Projekt.

Finales Training

Mit dem vorherigen Schritt ist der Fehler des Netzes bereits auf 0,0028 gesunken. Das ist schon ein sehr geringer Fehler. Daher wird das Training nur noch ein kleines bisschen weiter fortgesetzt. Konkret wird das Netz nun solange weiter trainiert, bis der Fehler *0,001* erreicht hat. Die folgende Grafik zeigt den kompletten Verlauf der Fehlerwerte von Beginn des Trainings, bis die 0,001 erreicht ist:

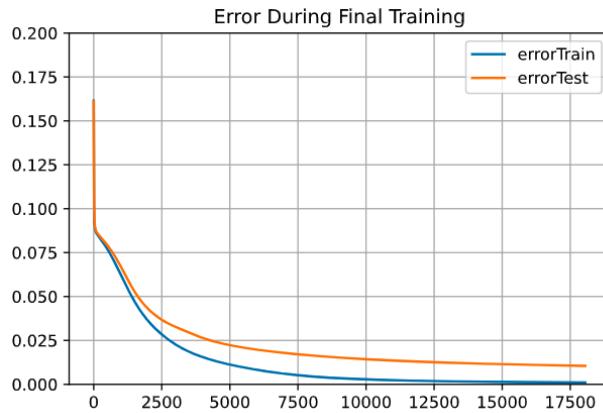


Abbildung 5.2: Verlauf der Fehlerwerte des finalen neuronalen Netzes

Nach insgesamt 18.052 Epochen ist der Fehler endlich unter den Zielwert gefallen. Der Einfluss der Mean Squared Error Funktion ist deutlich zu erkennen. Je geringer der Fehler, desto länger zieht sich das Training. Die letzten 10.000 Epochen haben sich die Werte nur sehr schleppend bewegt.

Der Fehlerwert über das Testdatenset weicht nun deutlich von den Trainingsdaten ab. Dennoch erreichen die Testdaten einen finalen Fehlerwert von 0.01. Das ist zwar 10 mal schlechter als die Trainingsdaten, aber dennoch sehr präzise. Zu keiner Zeit haben sich die Fehlerwerte wieder aufwärtsbewegt. Es ist stets eine Abwärtsbewegung zu erkennen.

Damit ist das Training abgeschlossen und das finale neuronale Netz gefunden.

Abschließende Analyse

Performance per Ziffern-Set

Die folgende Grafik zeigt die durchschnittlichen Fehlerwerte über die jeweiligen Ziffern-Sets:

Wenig überraschend ist, dass der Fehler für das "evag" Set am größten ist. Unter den Trainingssets weisen alle einen recht ähnlichen niedrigen Fehler auf. Lediglich das Set "digital-klein" schneidet etwas schlechter ab als die anderen. Dies lässt sich aber auch gut erklären: Digitale Ziffern nutzen lediglich horizontale und vertikale Linien. Dadurch sind sich die Ziffern sehr ähnlich und schwieriger auseinander zu halten. Durch die kleine Größe unterscheiden sich manche Ziffern nur noch um einen Pixel (z.B die 8 von der 6 oder der 9). Entsprechend ist hier der Fehler größer.

Performance über die Trainingsdaten per Ziffer

Die folgende Grafik zeigt die durchschnittlichen Fehlerwerte über die jeweiligen Ziffern der Trainingsdaten:

Hier zeigt sich ein recht ausgeglichenes Bild. Lediglich die Unterscheidung zwischen der 8 und der 9 scheint etwas schwieriger zu sein. Dies lässt sich auf die große Ähnlichkeit der Ziffern zurückführen.

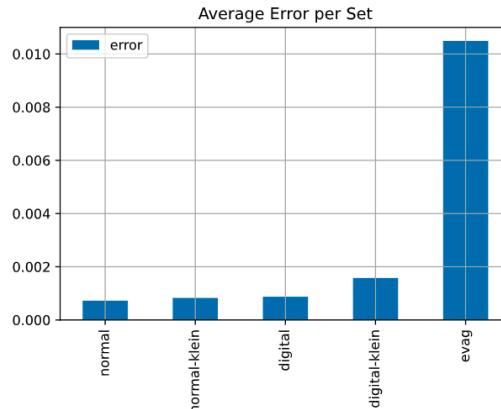


Abbildung 5.3: Fehlerwerte nach Ziffern-Set



Abbildung 5.4: Fehlerwerte der Trainingsdaten nach Ziffer

Performance über die Testdaten per Ziffer

Die folgende Grafik zeigt die durchschnittlichen Fehlerwerte über die jeweiligen Ziffern der Testdaten:

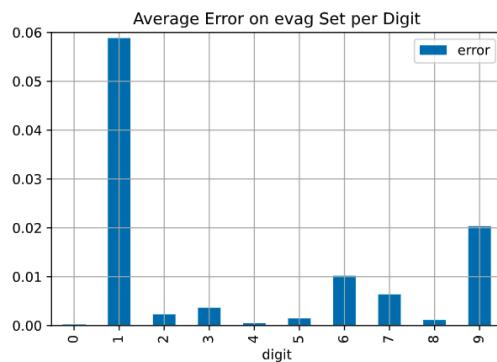
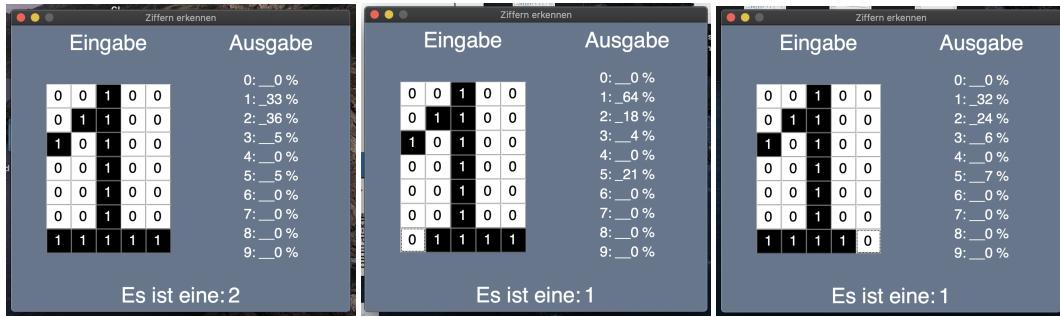


Abbildung 5.5: Fehlerwerte der Testdaten nach Ziffer

Der Fehler für die 0, die 4 und die 8 ist am niedrigsten. Das liegt daran, dass diese Ziffern auch in den Trainingsdaten im Set ‘normal’ vorkommen.

Die größten Probleme hat das Netz mit der Erkennung der Ziffer 1. Ein Blick in die Demo-App zeigt, was passiert:



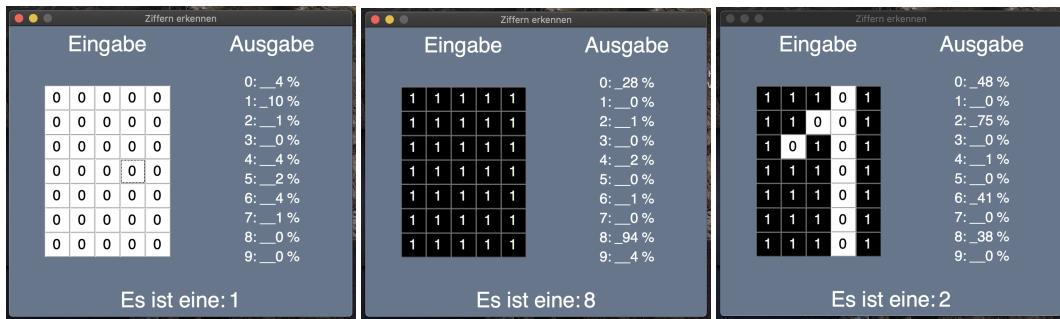
Links ist die originale 1 aus dem “evag” Datenset. Das Netz erkennt fast zu gleichen Teilen eine 1 bzw. eine 2. In den beiden rechten Abbildungen ist nur ein Pixel (jeweils ganz unten an der Seite) geändert worden und schon ist das Ergebnis eindeutiger. Dieser Unterstich im “evag” Set kommt in keinem anderen Set vor. Dadurch ist die 1 so problematisch. Allerdings ist das die einzige Ziffer aus allen Beispieldaten, die nicht korrekt erkannt wird. Und schon durch die Änderung eines Pixels, wird die Ziffer wieder korrekt erkannt.

Weitere Muster

Mit der Demo-App können nun noch eine Vielzahl weiterer Muster ausprobiert werden. Alle Ziffern aus den Beispieldaten werden zuverlässig erkannt (mit Ausnahme der besagten 1 aus dem “evag” Set). Selbst Varianten, die das Netz noch nie gesehen hat, werden hinreichend zuverlässig erkannt. Hier sind einmal drei solcher Beispiele:



Spannend sind auch die folgenden 3 Grafiken:



In einem leeren Bild wird eine 1 erkannt. Das macht Sinn, da die Ziffer 1 die wenigsten Pixel benötigt. Ein voll ausgemaltes Bild wird der 8 zugeordnet. Auch das leuchtet ein, da die 8 in der Regel die meisten Pixel belegt.

Die Grafik ganz rechts ist der Versuch, wie das Netz auf eine inverse Grafik (bei der alle Pixel genau den gegenteiligen Wert haben) reagiert. Leider ist das Modell nicht robust genug und erkennt hier etwas völlig anderes. Damit werden auch die Grenzen dieser Ziffernerkennung deutlich.

Fazit

Im Verlaufe dieses Projektes ist es gelungen, ein Programm zu entwickeln, welches die arabischen Ziffern in einem 7 x 5 Pixelraster erkennt. Die Umsetzung ist alleine mithilfe von Beispieldaten erfolgt. Außerdem ist das Modell robust genug, auch stärker abweichende Varianten der Ziffern zu erkennen. Es ist also ein erfolgreicher Abschluss geglückt.

Selbstverständlich könnte dieses Projekt in Zukunft noch in verschiedene Richtungen erweitert werden. Es könnten größere Pixelraster verwendet werden. Genauso wären Graustufen in den Pixeln denkbar. Auch das neuronale Netz kann auf verschiedenste Arten erweitert werden. Man könnte mit verschiedenen Aktivierungs- und Fehlerfunktionen experimentieren. Ebenso ist die Verwendung weiterer versteckter Schichten denkbar. Bezüglich der Lernrate sind verschiedene Optimierungen angeschnitten worden. Diese waren zwar für diesen kleinen Projektumfang nicht nötig, bieten aber spannende Forschungsfragen für die Zukunft.

Der Autor bedankt sich in jedem Fall für das spannende und lehrreiche Projekt.

Quellen

- Kingma, Diederik P., und Jimmy Lei Ba. 2014. „Adam: A method for stochastic optimization“. *arXiv preprint arXiv:1412.6980*.
- Lämmel, Uwe, und Jürgen Cleve. 2020. *Künstliche Intelligenz: Wissensverarbeitung – Neuronale Netze*. Carl Hanser Verlag GmbH & Co. KG.
- Papp, S., W. Weidinger, M. Meir-Huber, B. Ortner, G. Langs, und R. Wazir. 2019. *Handbuch Data Science: Mit Datenanalyse und Machine Learning Wert aus Daten generieren*. Carl Hanser Verlag GmbH & Co. KG.
- Patterson, Josh, und Adam Gibson. 2017. *Deep Learning: A Practitioner’s Approach*. O’Reilly Media. <https://books.google.de/books?id=qrcuDwAAQBAJ>.
- Sutton, Richard S., und Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press.

Anhang

Quellcode

Das gesamte Projekt inklusive Quellcode kann hier abgerufen werden:
<https://github.com/hd-code/digit-recognition-ci/>.

Es liegt zusätzlich nochmal diesem Dokument per CD-ROM bei.

Eine Erklärung der Projektstruktur ist in Kapitel 3 zu finden.

Hinweis zum Programm-Code: Für den Fall, dass das Jupyter Notebook (`src > main.ipynb`) nicht geöffnet werden kann. Im gleichen Ordner befindet sich eine Datei namens `main.pdf`. Dies ist ein PDF-Export des Jupyter Notebooks. Hier steht also der Code inklusive aller Kommentare und generierten Grafiken nochmal zur Sicherheit drin.

Bei Fragen und Problemen kann der Autor gerne per E-Mail kontaktiert werden.

Selbstständigkeitserklärung

Ich, Hannes Dröse, versichere hiermit, dass ich die vorliegende Projektarbeit mit dem Thema
Ziffernerkennung mittels neuronalem Perzeptron-Netz
selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Erfurt, 16.03.2021

Hannes Dröse