Projekt-Arbeit

Cynthia Odudu, Husen Muhsen , Carlo Wolter

Febuar 2023

Inhaltsverzeichnis

1 Zielstellung **5**

2 Vorbereitung **6**

3 Neuronales Netzwerk **8**
3.1 **Deep Learning vs. Machine Learning** 9
3.2 Künstliche neuronale Netze (KNN) (AI) 9
3.3 Machine Learning 10
3.4 Deep Learning 10
3.5 Gewicht 11
3.6 Schwellenwert 12
3.7 Funktionsweise auf Knotenebene 13
3.8 Arten von neuronalen Netzen 15
3.9 Einsatzbereiche für Neural Networks 16

4 MNIST-Datenbank **17**

5 Backpropagation, Verlust und Epochen **19**

6 Training **21**
6.1 Warum die Backpropagation? 22

7 Implementation **25**
7.1 Reader Construktor 25
7.2 Reader getter Funktion 27
7.3 Layer Implementation 28
7.4 Netzwerk 32
7.5 Unit-Test 38
7.6 GitLab-Runner 38

8 Laufzeit **40**

9 Herausforderung **44**
9.1 Datenaufbereitung 44
9.2 Over- / Unerfitting 44
9.3 Optimierungsproblem 45
9.4 Overparametrization 45

10 Fazit **47**

Abbildungsverzeichnis

1 Deep Learning (Quelle: https://bit.ly/3m03lmn (aufgerufen:19.02.2023,17:33)) 9
2 Architektur eines künstlichen neuronalen Netzes (Quelle: https://bit.ly/3YUNmEF (aufgerufen:19.02.2023,23:10)) 12
3 Funktionsweise auf Knotenebene (Quelle: https://bit.ly/3ZfAzfR (aufgerufen:19.02.2023,23:17)) 13
4 Single neuron Quelle: (https://bit.ly/3XZkytv (aufgerufen:19.02.2023)) 14
5 XOR-Problem 15
6 Datensatz (Quelle: https://bit.ly/3KwZ0kK (aufgerufen: 20.02.2023)) 18
7 Trainings und Testdatensatz 18
8 Überblick über der Backpropagation-Algorithmus (Quelle: https://bit.ly/3kjNQW7(aufgerufen:23.02.2023, 22:35)) 22
9 UI für das Program 37
10 learn time 42
11 think time 43

Source Code

7.1 Reader Construktor 25
7.2 Filesize Funktion 27
7.3 getLable 27
7.4 getPixel 28
7.5 Layer Constructor 29
7.6 Reset Funktion 31
7.7 Randomize Funktion 32
7.8 Think Funktion 34
7.9 Fire Funktion 34
7.10 Learn Funktion 36

Kapitel 1

Zielstellung

Unsere Zielstellung ist der Bau eines Neuronen-Netzwerkes, welche handgeschriebene Ziffern erkennt. Eine Ziffernerkennung ist ein typisches Anwendungsgebiet für neuronale Netze. Hier geht es darum, Bilder von Ziffern als Eingabe zu nutzen und sie in die korrekte Ziffer (0 bis 9) einzustufen.
Die Zielstellung eines Ziffernerkennenden neuronalen Netzes ist es, eine hohe Genauigkeit bei der Klassifikation der Ziffern zu erreichen. Hierbei ist es wichtig, die unterschiedlichen Schreibweisen der Ziffern zu berücksichtigen und in der Lage zu sein, auch unvollständige oder verzerrte Ziffern zu erkennen.
Ein weiterer wichtiger Aspekt ist das Training des neuronalen Netzwerkes. Hierbei werden große Mengen an Bilddaten verwendet, um das Netzwerk darin zu trainieren, die Ziffern richtig zu erkennen. Es ist wichtig, eine ausreichende Vielfalt an Schreibweisen der Ziffern im Trainingsdatensatz zu berücksichtigen, um das Netzwerk für unterschiedliche Schreibweisen anfällig zu machen.
Während des Trainings werden die Gewichte und Verbindungen innerhalb des Netzes angepasst, um die Fehlerfunktion zu minimieren und die Vorhersagen des Netzes immer präziser zu machen. Hierbei kann es notwendig sein, mehrere Trainingsläufe durchzuführen, um eine bessere Leistung zu erreichen.
Die Zielstellung eines Ziffernerkennenden neuronalen Netzwerkes ist es also, eine hohe Genauigkeit bei der Klassifikation von Ziffernbildern zu erreichen, indem es große Mengen an Bilddaten verwendet, um das Netzwerk zu trainieren, und die Gewichte und Verbindungen anzupassen, um die Vorhersagen zu verbessern.

Kapitel 2

Vorbereitung

Im Rahmen des Projekts wurden von Cynthia Odudu die Themen Dokumentation des Quelltextes mittels Doxygen und Git-Lab Runner bearbeitet. Von Carlo Wolter wurden die Projektziele Unit Test sowie deren Auswertung (Laufzeit) übernommen. Das Projektziel der Implementation wurde zum Großteil von Husen Muhsen bearbeitet. Dabei haben Cynthia Odudu und Carlo Wolter bei der Erstellung des neuronalen Netzwerkes, sowie Erkennung der Buchstaben und Zahlen unterstützt.

Die Belegarbeit wurde in gleichen Teilen von allen Projektteilnehmern erstellt. Um ein effektives und präzises Neuronen-Netzwerk zu bauen, ist die Vorbereitung ein wichtiger Schritt. Dabei kann man sich an diesen wichtigen Schritten orientieren. Zuerst sollte man klären, welches Problem man gelöst haben will.

In diesem Projekt geht es um das Problem des Einlesens von handgeschriebenen Ziffern. Die Anforderungen an das Netzwerk ist somit die Auswertung von Dateien mit handgeschriebenen Ziffern. Für die Problemlösung stand ein großer Datensatz zur Verfügung. Der Datensatz beinhaltete zusätzlich Trainingssets, welche die zu repräsentierende Zahl beinhaltete. Dadurch kann ein Netzwerk offline trainiert werden.
Kennt man das Problem, benötigt man noch eine Architektur. Für Neuronen-Netzwerke gibt es einige unterschiedliche Architekturen, welche alle Vorteile haben. Beispiele für solche Architekturen sind das Perceptron und Feed Forward Networks, das MultiLayer Perceptron (MLP) und das Single layer Perception, siehe Kapitel 3.1.7.

Es müssen sich auch Gedanken, um Parameter wie Anzahl der Neuronen, der Schichten und der Lehrling Rate gemacht werden. Hierbei ist es wichtig, geeignete Schätzwerte und experimentelle Methoden zu verwenden, um die besten Parameter zu ermitteln.
Damit ein Netzwerk mit Daten auch etwas anfangen kann, muss das Programm die Dateien erst in Daten umwandeln. Sie werden normalisiert, geschärft und gegebenenfalls aufgeteilt, um Overfitting zu vermeiden, siehe Herausforderungen und Kapitel 7.1.
Um zu beurteilen, wie gut das Netzwerk funktioniert, muss das Model bewertet werden. Eine Fehlerfunktion kann diese Metriken (Messgrößen oder Kennzahlen) liefern.

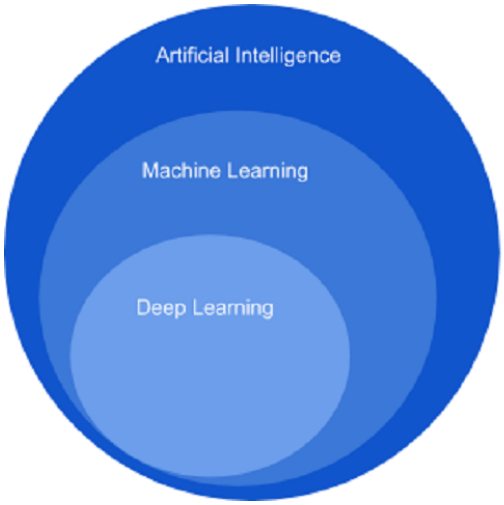
Kapitel 3

Neuronales Netzwerk

Dieses Projekt stellt eine verständliche Untersuchung dar, wie neuronale Netze implementiert werden. Das Ziel dieses Projekt ist es, dass Kernproblem des maschinellen Lernens via neuronale Netze mit mindestens zwei verschiedenen Varianten für Parallelität ohne Nutzung vorgefertigter Bibliotheken zu implementieren, die gemeinsam mit der sequenziellen Variante umfassend miteinander verglichen werden. Zu Beginn wird der Theorieteil des Neuronales Netzwerk zusammengefasst. Der Begriff "neuronales Netz bzw. künstliche neuronale Netze (KNN)", beschreibt gewisse Strukturen im menschlichen Hirn. Sie sind ein Teil der Technologien, die in der Maschine verwendet werden und bilden die Grundlage für das Deep Learning. [13]

3.1 Deep Learning vs. Machine Learning

In diesem Kapitel wird das Deep Learning dem Machine Learning gegenübergestellt. Dies ist im nachfolgenden Abbildung 1dargestellt.



(a) How deep learning is a subset of machine learning and how machine learning is a subset of AL.

Abbildung 1: Deep Learning
(Quelle: https://bit.ly/3m03lmn (aufgerufen:19.02.2023,17:33))

3.2 Künstliche neuronale Netze (KNN) (AI)

Neuronale Netze zeichnen sich dadurch aus, dass Computer mit ihrer Hilfe eigenständig Probleme lösen und ihre Fähigkeiten verbessern können. Ein Neuron kann dabei viele Eingänge, zum Beispiel (Zahlenwerte) haben, wenn diese Zahlenwerte dann den Schwellwert überschreiten, gibt das Neuron ein Ausgabewert aus. Für jeden Eingangskanal des Neurons gibt es ein sogenanntes Gewicht und Schwellwert, der eingestellt werden muss.

3.3 Machine Learning

Machine Learning ist der Einsatz und Entwicklung von Computersystemen, die in der Lage sind, zu lernen und sich anzupassen, ohne ausdrückliche Anweisungen zu befolgen, indem sie Algorithmen und statistische Modelle verwenden, um Daten zu analysieren und aus Mustern Schlüsse zu ziehen. [13]

3.4 Deep Learning

Deep Learning ist eine Teilmenge des maschinellen Lernens auf der Grundlage künstlicher neuronaler Netze, bei der mehrere Verarbeitungsebenen verwendet werden, um schrittweise höherwertige Merkmale aus Daten zu extrahieren. Jeder Deep Learning-Algorithmus wird als maschinelles Lernen betrachtet, aber nicht jeder Algorithmus des maschinellen Lernens wird als Deep Learning betrachtet. [13]

3.5 Gewicht

Das Gewicht ist der Parameter in einem neuronalen Netz, der die Eingabedaten in den verborgenen Schichten des Netzes umwandelt. Jede Schicht hat eigene Gewichte zur Folgeschicht, je grösser das Netz, desto mehr Gewichte hat man. Hat man beispielsweise ein Netz mit 3 Schichten à 3 Knoten, sind das insgesamt 18 Gewichte. (Alle 3 Knoten der ersten Schicht verbunden mit allen 3 Knoten der zweiten Schicht = 9 Gewichte. Dann nochmals 9 Gewichte für die Knoten der zweiten zur dritten Schicht = 18 Gewichte total). Die Gewichtung dieser Verbindungen definiert hierbei, wie wichtig eine Verbindung im Netzwerk ist. Durch das Trainieren des KNN passt sich diese Gewichtung an und kann sich im Laufe des Trainings verändern.

Das Lernen eines neuronalen Netzes ist also tatsächlich die Anpassung dieser Gewichte. Ein neuronales Netz ist definiert durch die Anzahl der Eingangsknoten (Input layer), die Anzahl der verborgenen Schichten (hidden layer), die Anzahl der Knoten in jeder versteckten Schicht und die Anzahl der Knoten in der letzten Schicht (Output layer). Die Eingabeschicht (input layer) empfängt Daten bzw. nimmt die Eingabeinformation, wie beispielsweise Bilder von Hunden und Katzen, auf. Diese werden dann als Wert an das Netz bzw. die nachfolgenden verborgenen Schichten (Hidden layers) weitergegeben. Diese führt mathematische Berechnungen an den Informationen durch, lernt die Abbildungsfunktion zwischen Eingabe und Ausgabe. Die verborgenen Schichten, von denen es beliebig viele geben kann, errechnen anhand der vorhandenen Informationen die Ausgabe (output layer). Die Output layer liefert das Ergebnis (siehe Abbildung 2) . In der Abbildung repräsentiert jeder Pfeil von einem Knoten zum nächsten ein Gewicht.[10]

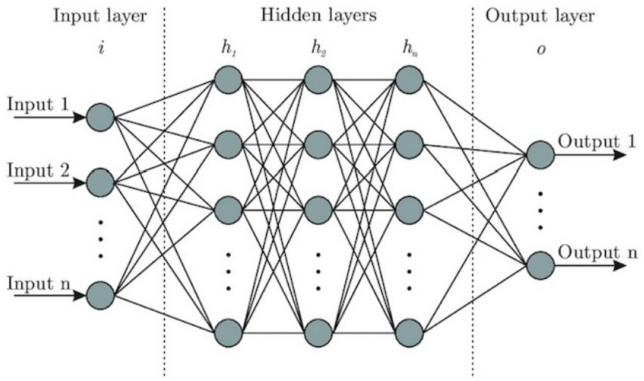


Abbildung 2: Architektur eines künstlichen neuronalen Netzes
(Quelle: https://bit.ly/3YUNmEF (aufgerufen:19.02.2023,23:10))

3.6 Schwellenwert

Der Schwellenwert ist der Grenzwert der Funktion. Wird dieser also auf 0,5 gesetzt, bedeutet alles, was darunter liegt, eine Ausgabe von 0, und alles, was darüber liegt, eine Ausgabe von 1. Das Neuron hat keine erwünschte Ausgabe, sondern nur eine erwünschte Eingabe, damit es sagt: 8, die Daten sehen aus wie Daten, die ich sehen soll.[10]

3.7 Funktionsweise auf Knotenebene

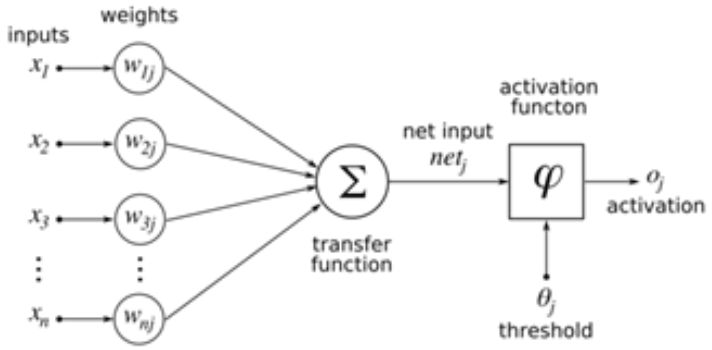


Abbildung 3: Funktionsweise auf Knotenebene
(Quelle: https://bit.ly/3ZfAzfR (aufgerufen:19.02.2023,23:17))

In der Abbildung 6.1 wird die Funktionsweise auf Knotenebene beschreiben. Jeder Knoten erhält eine Menge von Eingaben, die als x bezeichnet werden. Jede dieser Eingaben hat ein Gewicht, in dem Fall mit w betitelt. Werden Gewichtung w und Wert x multipliziert, dann erhalten wir die gewichtete Eingabe.

Das Gewicht entscheidet zusammen mit einer Übertragungsfunktion, über die Eingabe, die nun weitergeleitet wird. Die Übertragungsfunktion muss einfach dafür sorgen, dass aus diesen verschiedenen Zahlen, was in den Knoten übermittelt wird, nur noch ein Zahlenwert rauskommt. Aus diesem ergibt sich die tatsächliche Eingabe für den Knoten, indem alle gewichteten Eingaben aufsummiert werden. Ziel der Übertragungsfunktion ist es, aus den Inputs eine gemeinsame Darstellung zu generieren. Kurz gesagt, aus vielen wird eins. Diese aufsummierte Aufgabe bestimmt nun mithilfe der Aktivierungsfunktion die Ausgabe des Neurons (Y). Die Aktivierungsfunktion modifiziert diesen Wert nocheinmal. Die Aktivierungsfunktion wird in einem sogenannten Schwellwert geschrieben. Er sorgt dafür, dass bei der Aktivierungsfunktion mindestens ein gewisser Wert ausgegeben werden muss. Wenn die Bedingung erfüllt ist, funktioniert das Ganze, ansonsten kommt null raus. [14]

Es gibt Notationen, wodurch es eine Aktivierung und zusätzlich einen Schwellwert gibt. Manchmal ist die Schwellwert Funktion selber die Aktivierungsfunktion von den Neuronen. Um den Ausgabewert eines einzelnen Neurons

zu berechnen, multipliziert man jede Eingabe in dieses Neuron mit einer Gewichtung für diese Eingabe, addiert sie und fügt eine Vorspannung hinzu, die für das Neuron festgelegt wurde. Dieser *gewichtete Eingabewert* wird in eine Aktivierungsfunktion eingegeben und das Ergebnis ist der Ausgabewert dieses Neurons. [15]. In Abbildung 4 ist ein einzelnes Neuron dargestellt.

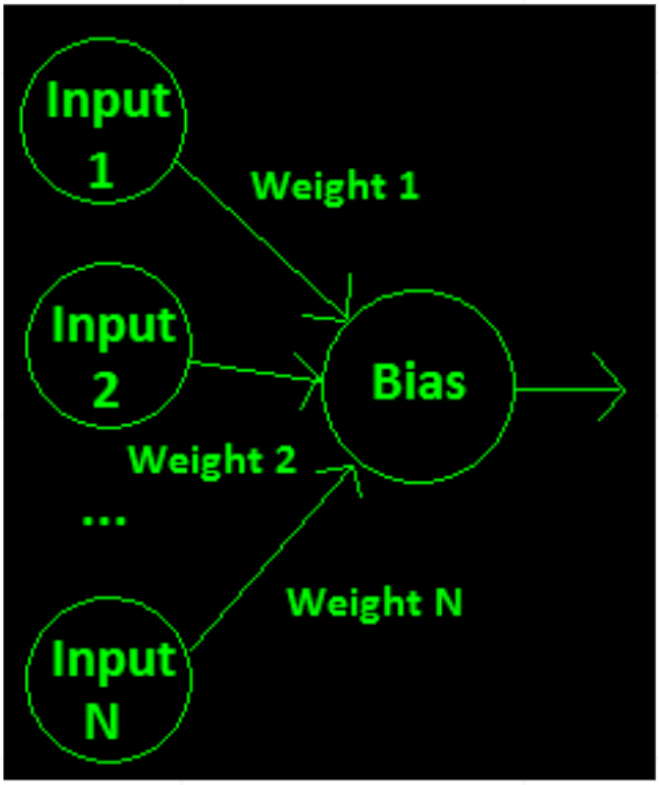


Abbildung 4: Single neuron Quelle: (https://bit.ly/3XZkytv (aufgerufen:19.02.2023))

Als Aktivierungsfunktion verwenden wir eine gängige Funktion, die Sigmoid-Aktivierungsfunktion, die manchmal auch als logistische Aktivierungsfunktion bezeichnet wird. Sie sieht wie folgt aus:

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

3.8 Arten von neuronalen Netzen

Es gibt viele Typen von neuronalen Netzwerk-Architekturen. Hier sind die wichtigsten Arten von neuronalen Netzen:

3.8.1 Perceptron und Feed Forward Networks

Sie nutzen die Eingabedaten und berechnen daraus eine Ausgabe. Die Vorgehensweise erfolgt dabei in Schichten. Das Perceptron ist sehr nützlich für die Klassifizierung von Datensätzen, die linear trennbar sind. Bei Datensätzen, die diesem Muster nicht entsprechen, stößt es an ernste Grenzen, wie das XOR-Problem zeigt. Das XOR-Problem zeigt, dass es für jede Klassifizierung von vier Punkten eine Menge gibt, die nicht linear trennbar ist.

Abbildung 5: XOR-Problem

3.8.2 Das MultiLayer Perceptron (MLP)

Das MultiLayer Perceptron durchbricht diese Einschränkung und klassifiziert Datensätze, die nicht linear trennbar sind. Dazu verwenden sie eine robustere und komplexere Architektur, um Regressions- und Klassifizierungsmodelle für schwierige Datensätze zu lernen. Bias wird hier gebraucht. Die Hauptfunktion eines Bias besteht darin, jedem Knoten einen trainierbaren konstanten Wert zu geben (zusätzlich zu den normalen Eingaben, die der Knoten erhält). Es ist unpraktisch, ein Bias-Neuron hinzuzufügen. Der Grund hierfür liegt darin, dass Sie gleichzeitig die Gewichtung und den Wert anpassen, so dass jede Änderung der Gewichtung die Änderung des Wertes, die für eine vorherige Dateninstanz nützlich war, neutralisieren kann.

3.8.3 Convolutional Neural Networks (CNN)

Dies findet häufig Verwendung in der Bild- und Objekterkennung. [5]

3.8.4 Recurrent Neural Network

Dies ist eine Art künstliches neuronales Netz, das häufig in der Spracherkennung und der Verarbeitung natürlicher Sprache eingesetzt wird. Rekurrente neuronale Netze erkennen die sequenziellen Merkmale von Daten und verwenden Muster, um das nächste wahrscheinliche Szenario vorherzusagen.

3.8.5 Single layer Perception

Das sind neuronale Netze ohne hidden Schicht.

3.9 Einsatzbereiche für Neural Networks

Anwendungsbereiche sind beispielsweise die Text-, Bild- und Spracherkennung. Künstliche Neuronale Netze können zudem Simulationen und Prognosen für komplexe Systeme und Zusammenhänge erstellen ,wie in der Wettervorhersage, der medizinischen Diagnostik oder in Wirtschaftsprozessen.[8] Anwendungsfällen der selbstlernenden neuronalen Netze gehören:

- die Sprachassistenten Alexa
- Siri
- Googles Sprachassistent

Kapitel 4

MNIST-Datenbank

Der MNIST-Datensatz ist ein klassisches Problem für den Einstieg in die neuronalen Netze. MnistData ist die Klasse, die den gesamten Code enthält, den wir zum Sammeln und Aufbereiten der Daten für unsere MNIST-Anwendung verwenden. Die MNIST-Datenbank (Modified National Institute of Standards and Technology database) ist eine große Datenbank mit handgeschriebenen Ziffern, die üblicherweise zum Training verschiedener Bildverarbeitungssysteme verwendet werden. Die Datenbank wird auch häufig zum Trainieren und Testen im Bereich des maschinellen Lernens verwendet. Sie enthält einen Trainingsdatensatz von 60.000 Beispielen und eine Testdatensatz von 10.000 Beispielen. Sie ist eine Teilmenge einer größeren Menge, die vom NIST zur Verfügung gestellt wird. Die Ziffern wurden größennormalisiert und in einem Bild fester Größe zentriert. [18]

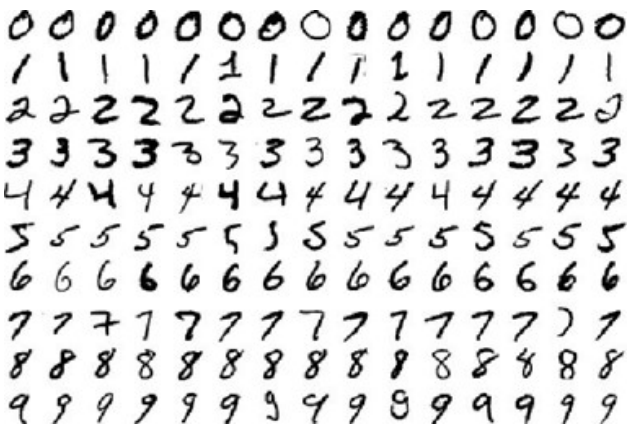


Abbildung 6: Datensatz
(Quelle: https://bit.ly/3KwZ0kK
(aufgerufen: 20.02.2023))

In der nachfolgenden Abbildung 7 sieht man 4 Dateien. Sie ist eine gute Datenbank für alle, die Techniken des Lernens und Methoden der Erkennung von Fehlern an realen Daten ausprobieren möchten, ohne viel Aufwand für die Vorverarbeitung und Formatierung zu haben. [12]

```
train-images-idx3-ubyte: training set images
train-labels-idx1-ubyte: training set labels
t10k-images-idx3-ubyte:  test set images
t10k-labels-idx1-ubyte:  test set labels
```

Abbildung 7: Trainings und Testdatensatz

Kapitel 5

Backpropagation, Verlust und Epochen

Das wichtigste Element beim Training neuronaler Netze ist die Backpropagation. Backpropagation ist eine Kurzform für "backward Propagation". Backpropagation ermittelt die richtigen Gewichtungen, die auf Knoten in einem neuronalen Netz angewendet werden sollten, indem die aktuellen Ausgaben des Netzwerks mit den gewünschten oder richtigen Ausgaben verglichen werden. Die Differenz zwischen der gewünschten Ausgabe und der aktuellen Ausgabe wird durch den Verlust berechnet. Je kleiner der Verlust für ein Netzwerk ist, desto genauer wird es.

$$f(m,b) = \frac{1}{\mathrm{N}} \sum_{i=1}^{1} (y_i - (mx_i - b))^2$$

Mathe ist der Schlüssel zu neuronalen Netzen. Der Vorgang, bei dem die Daten von der Eingabeschicht zur Ausgabeschicht und dann den ganzen Weg zurückgesendet werden, wird als Epoche bezeichnet. In jeder Epoche aktualisiert das Neuronale Netz die Gewichte der Neuronen, was auch als Lernen bezeichnet wird. Nach mehreren Epochen und Gewichtungsaktualisierungen sollte die Verlustfunktion (die Differenz zwischen der Ausgabe des neuronalen Netzes und der tatsächlichen Ausgabe) ein Minimum erreichen.
In der ersten Epoche werden die beschrifteten Daten in die Eingabeschicht eingegeben und an die Ausgabeschicht weitergeleitet, wo das Neuronale Netz eine Ausgabe berechnet. Die Differenz zwischen der tatsächlichen Ausgabe des Neuronalen Netzes und der erwarteten Ausgabe wird als Kostenfunktion

bezeichnet. Das Ziel des neuronalen Netzwerks ist es, diese Kostenfunktion so weit wie möglich zu verringern. Das neuronale Netzwerk wird also von der Ausgabeschicht bis zur Eingabeschicht zurückverfolgt und die Gewichte der Neuronen entsprechend aktualisiert, um diese Kostenfunktion zu minimieren. [11]

Kapitel 6

Training

Beim Training wollen wir mit einem neuronalen Netz mit schlechter Leistung beginnen und am Ende ein Netz mit hoher Genauigkeit haben. In Bezug auf die Verlustfunktion wollen wir, dass unsere Verlustfunktion am Ende des Trainings viel niedriger ist als zu Anfang. Eine Verbesserung des Netzes ist möglich, da wir seine Funktion durch Anpassung der Gewichte ändern können. Wir wollen eine andere Funktion finden, die besser funktioniert als die ursprüngliche. Ein neuronales Netz zu trainieren bedeutet einfach, dass wir die Werte für die Gewichtung und den Bias so anpassen, dass wir bei bestimmten Eingaben die gewünschten Ausgaben des Netzes erhalten. Herauszufinden, welche Gewichte und Verzerrungen zu verwenden sind, kann schwierig sein, insbesondere bei Netzen mit vielen Schichten und vielen Neuronen pro Schicht.

Es gibt eine Vielzahl von Algorithmen, die Funktionen optimieren. Diese Algorithmen können gradientenbasiert oder nicht gradientenbasiert sein, d. h, sie verwenden nicht nur die von der Funktion bereitgestellten Informationen, sondern auch ihren Gradienten [2]. Einer welchen wir benutzen können ist die Backprobagation. Dies ist die am häufigsten verwendete Methode für das Training neuronaler Netze. Wie oben schon beschrieben 5, Backpropagation in neuronalen Netzen ist eine Kurzform für "Rückwärtsfortpflanzung von Fehlern". Es handelt sich um eine Standardmethode für das Training künstlicher neuronaler Netze. Diese Methode hilft bei der Berechnung des Gradienten einer Verlustfunktion in Bezug auf alle Gewichte im Netz.

6.1 Warum die Backpropagation?

Beim Training neuronaler Netze wollen wir zunächst alle Werte der Neuronen berechnen und sehen, welche Ausgabeschicht dabei herauskommt (manchmal auch als forward pass bezeichnet). Wenn wir mit dem tatsächlichen Ergebnis konfrontiert werden, können wir die Verlustfunktion berechnen. Aber wir wollen das Netzwerk trainieren und daher versuchen, den Wert, den wir als Verlust erhalten haben, zu minimieren. In diesem Schritt müssen wir rückwärts gehen und prüfen, wie sich kleine Änderungen unserer Gewichte auf das Ergebnis auswirken würden, und die Gewichte entsprechend den Informationen über ihre Auswirkungen auf eine Verlustfunktion aktualisieren. Dazu müssen wir die partiellen Ableitungen der Verlustfunktion in Bezug auf jedes Gewicht und jede Verzerrung im Netz berechnen. [9]
In der nachfolgenden Abbildung 8 ist das Übersicht über den Backpropagation-Algorithmus dargestellt.

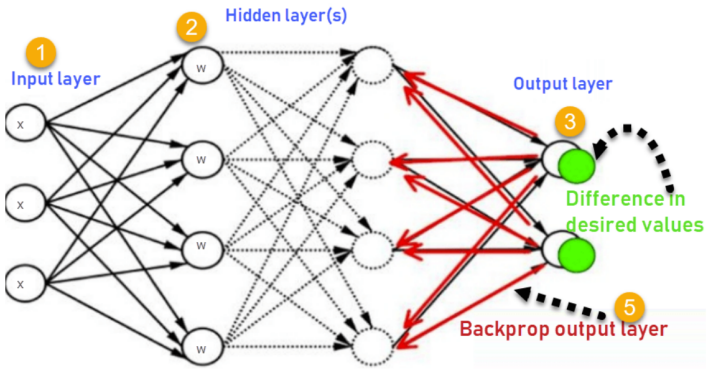


Abbildung 8: Überblick über der Backpropagation-Algorithmus (Quelle: https://bit.ly/3kjNQW7(aufgerufen:23.02.2023, 22:35))

So funktioniert der Backpropagation-Algorithmus

- ◇ Die Eingaben X kommen über den vorverknüpften Pfad an.
- ◇ Die Eingabe wird durch reale Gewichte W modelliert. Die Gewichte werden normalerweise zufällig ausgewählt.
- ◇ Die Ausgabe für jedes Neuron von der Eingabeschicht wird über die versteckten Schichten bis hin zur Ausgabeschicht wird berechnet.

- ◇ Es werden die Fehler in den Ausgaben berechnet.
- ◇ Es wird von der Ausgabeschicht zur versteckten Schicht(hidden layer) zurückgekehrt, um die Gewichte so anzupassen, dass der Fehler verringert wird.
- ◇ Der Vorgang wird wiederholt, bis das gewünschte Ergebnis erreicht ist.

Es gibt Zwei Arten von Backpropagation-Netzwerken:

- ◇ Statische Backpropagation
- ◇ Rekurrente Backpropagation

Statische Backpropagation: Dies ist eine Art von Backpropagation-Netzwerk, das eine Abbildung einer statischen Eingabe auf eine statische Ausgabe erzeugt. Es ist nützlich, um statische Klassifizierungsprobleme wie optische Zeichenerkennung zu lösen.

Rekurrente Backpropagation: Bei der rekurrenten Backpropagation im Data Mining wird so lange weitergeleitet, bis ein fester Wert erreicht ist. Danach wird der Fehler berechnet und rückwärts propagiert.

Der Hauptunterschied zwischen diesen beiden Methoden besteht darin, dass die Zuordnung bei statischer Backpropagation schnell erfolgt, während sie bei rekurrenter Backpropagation nicht statisch ist [3]. Um die Gewichte neu anzupassen, benötigen wir die Ableitung der Kostenfunktion nach W, da diese uns sagt, wie sich die Kosten der Modellvorhersage ändern, wenn wir das Gewicht W ändern. Diese Ableitung wird benötigt, um die Gradientenabstiegsmethode anzuwenden und die Gewichte entsprechend zu aktualisieren, um die Kosten zu minimieren und damit das Modell zu verbessern.

$$\frac{\partial E}{\partial W_i} = \frac{1}{2}\sum_{x_i}^{x} = \frac{\partial}{\partial W_i}(y_i - y_i')^2$$

$$f(x) = g(h(x)) \Rightarrow \frac{\partial f(x)}{\partial x} = \frac{\partial g(h(x))}{\partial x} * \frac{\partial h(x)}{\partial x}$$

$$\frac{\partial E}{\partial W_i} = \frac{1}{2} \sum_{x_i}^{X} 2 * (y - y_i') \frac{\partial}{\partial x} (y - y_i')$$

$$y_i' = \frac{1}{1 + e^{-\sum (Weight * Input) + bias}}$$

Nach W ableiten:

$$\frac{\partial E}{\partial W_i} = \sum_{x_i}^{X} (y - y_i') \frac{\partial}{\partial x} (-y_i')$$

$$\frac{\partial E}{\partial W_i} = \sum_{x_i}^{X} (y_i - \varphi(W * X_i)) * -\varphi(W * X_i) * (1 - \varphi(W * X_i)) * X_i$$

die Gewichte anpassen durch:

$$W_{new} = W_{alt} + (\frac{\eta}{batchsize}) * \frac{\partial E}{\partial W_i}$$

Kapitel 7

Implementation

In diesem Kpitel wird der Aufbau des Programmes anhand der einzelen code Implementierungen beschrieben.

7.1 Reader Construktor

Zuerst soll die MNIST-Datenbankdateien von der folgenden Webseite heruntergeladen heruntergeladen werden, um das Netzwerk mit dem Inhalt dieser Dateien zu trainieren und später zu testen. Daher soll eine Reader-Klasse erstellt werden, um den Inhalt der Dateien zu lesen.

```
1  Reader()=default;
2  Reader(const char fName[])
3  {
4
5    char c;
6    int index = 0;
7
8    ifstream infile;
9          /// ios::in allows input (read operations)
10         ///from a stream.
11   infile.open(fName, ios::binary | ios::in);
12
13         /// Calculates the number of characters in the
14         ///transferred file
15   f_Size = (int)filesize(fName);
16
```

```
17          /// resize() resizes the vector
18    fMapV.resize(f_Size);
19
20          /// The forloop continues until the last digit
21          ///in the file is read
22    for (int i = 0; i < f_Size; i++)
23    {
24        /// read()copies a block of data
25              /// read(where items ares tored, number of items
        to read)
26      infile.read(&c, 1);
27
28              /// store the read letter in the Vector
29      fMapV[i] = c;
30    }
31          /// Close the file, to create the file
32    infile.close();
33    read_Header();
34
35 }
```

Source Code 7.1: Reader Construktor

fileMapV: Ist ein Vector mit unsigned char Werten.
ifstream: Ist eine Dateibehandlungsklasse, die den Eingabedateistrom definiert und verwendet wird, um Daten aus einer Datei zu lesen.
open( ):Um die angegebene Datei zu öffnen.
Ios::binary:Um sicherzustellen, dass die Daten ohne die Zeilenumbrüche gelesen oder geschrieben werden.
Ios::in:Ermöglicht das Lesen von Eingaben aus einem Stream.
resize(n):Änderung der Größe eines Vectors. n ist die neue Größe
read(m, n):Diese Funktion kopiert einen Datenblock.
M: Speicherort der kopierten Elemente
N: Anzahl der zu lesende Elemente
close( ): Datei schließen.

7.2 Reader getter Funktion

1. getFileSize: Diese Funktion wird benötigt, um die Größe der Datei zu erfahren und um die Größe des Vectors fileMapV festzulegen.

```
1  /// uses absolute positions in the stream
2  std::ifstream::pos_type filesize(const char* filename)
3  {
4      /// ios::ate the stream's position indicator to
5      ///the end when opening
6      /// binary it is a binary file
7      std::ifstream in(filename, std::ifstream::ate |std::
       ifstream::binary);
8
9      /// tellg/() to know where the get pointer is
10   return in.tellg();
11 }
```

Source Code 7.2: Filesize Funktion

pos-type: Wird verwendet, um auf eine absolute Position im Stream zu verweisen.
Ifstream::ate: Setzt den Cursor des Streams beim Öffnen auf das Ende des Streams.
tellg(): Wird verwendet, um die Position des Lesezeigers in einer Datei zu ermitteln. Es gibt die Anzahl der Bytes zurück.

2. getLabel
Es soll eine getLabel Funktion implementiert, die 'getLabel'-Funktion gibt die korrekte Antwort für das trainierte Bild zurück. Die Zahl 8 repräsentiert eine Lücke in den Daten am Anfang der Labeldatei, die bei jeder Verwendung überschritten werden muss, um den richtigen Index zu erreichen.

```
1  int getLabel(int numIndex) const
2  {
3      /// 8 for label file this is the gap after which the
4      ///first pixel of an image begins a digit
5      /// numIndex the position of a pixel
6    return fMapV[numIndex + data_Index];
7  }
```

Source Code 7.3: getLable

3. getPixel
Es soll ein Funktion implementiert, für die Lücke von 16 Bits am Anfang und am Ende der Datei. Die Zahlen sind in Hexadezimal-Notation angegeben und müssen in Float-Werte umgewandelt werden, indem sie durch 255 geteilt werden.

```
///numIndex of this picture stands for the number
///at which the next picture in the fMapV
float getPixel(int numIndex) const
{
    /// numIndex stands for the gap between
    ///one picture and the next
    /// file train-images, this gap is 16 digits
  if (numIndex + data_Index >= fMapV.size())
  {
    return 0.0f;
  }
  /// store in a 16 bit hex number
  int a = fMapV[numIndex + data_Index];
  /// divide the number by 255 to return a number between
        /// 0 and 1 if it is not 0
  return ((float)a / 255);
}
```

Source Code 7.4: getPixel

7.3 Layer Implementation

Die Layer Implementierung wird in diesem Projekt genutzt, um die einzelen Neuronen zusammenzufügen und in einzelne Schichten zu unterteilen. Dies ist wichtig, um das Netzwek in wenigen Schritten anzupassen und gegebenenfalls zu beschleunigen.

1. Layer Klasse:
Mit der Implementierung des Layer-Konstruktors werden die Größen der einzelnen Schichten mithilfe der 'resize'-Funktion definiert. Die folgende Abbildung zeigt die Komponenten einer Schicht.
Die Komponenten einer Schicht hängen davon ab, ob es sich um ein Input-, Hidden- oder Output-Layer handelt. 'dCdW', 'dCdA' und 'dCdB' stehen für die Ableitungen der Kostenfunktion nach 'W', 'Bias' und 'A'. Die Abbildung

15 zeigt den Quellcode für den Layer-Konstruktor.

```
Layer::Layer(int n, int w){

    neuron_Count = n;
    weight_Count = w;
    /**
    * @brief Assignment of vectorsSize the number of neurons
    * The size of the array is determined by the "
   neuronCount"
    *variable, which is the number of neurons in the network
    * Within the loop, a list of weights is created for each
    *neuron, also specified by the weightCount variable.
    * The size of the list of weights thus corresponds to
    *the number of inputs for the neuron.
    * */

    weight.resize(neuron_Count);
    for (int j = 0; j < neuron_Count; j++)
    {
       weight[j].resize(weight_Count);
    }
    bias.resize(neuron_Count);
    a.resize(neuron_Count);
    z.resize(neuron_Count);
    dCdA.resize(neuron_Count);
    dCdW.resize(neuron_Count);
    for (int i = 0; i < neuron_Count; i++)
    {
       dCdW[i].resize(weight_Count);
    }
       dCdB.resize(neuron_Count);
    reset();
    randomize();

}
```

Source Code 7.5: Layer Constructor

2. reset function:
Die Ableitungsvektoren sollten mit Null initialisiert werden, um Vermeidung von Fehlern zu gewährleisten. Ein solcher Fehler kann zum Beispiel durch das Hinzufügen einer Zahl zu einer nicht initialisierten Variablen entstehen. Die Ableitungsvektoren sollten mit Null initialisiert werden, da sie in vielen Algorithmen verwendet werden, um den Gradienten einer Funktion zu

berechnen. Wenn sie nicht mit Null initialisiert werden, kann es zu falschen Berechnungen und somit zu Fehlern in den Ergebnissen kommen. Außerdem kann es zu inkonsistenten oder unerwarteten Ergebnissen führen, wenn die Ableitungsvektoren von einer bereits veränderten oder verfälschten Werten ausgehen.

```
/**
 * @brief Reset values of these arrays to 0
 * to update the neural network weights and biases.
 * to ensure that the updated values of the weights
 * and biases are calculated correctly.
*/
void Layer::reset()
{
    //uses the std::fill function from the C++
    //standard library to reset the values of
    // all elements in the dCdA and dCdB arrays
    //to 0.0 in a single line of code.

    for (int j = 0; j < neuron_Count; j++)
    {
        dCdA[j] = 0.0f;
        dCdB[j] = 0.0f;
        for (int k = 0; k < weight_Count; k++)
        {
            dCdW[j][k] = 0.0f;
        }
    }
}
```

Source Code 7.6: Reset Funktion

3. randomsize:
Am Anfang werden die w_i und b_i durch Zufallszahlen definiert. Die Ergebnisse werden dann überprüft, um zu bestimmen, ob sie korrekt oder falsch sind. Mit dem Backpropagation-Algorithmus werden sie so lange angepasst, bis eine bestimmte Float-Zahl gefunden wird, die zu den richtigen Ergebnissen führt. Dies wird durch die Funktion erreicht:

$$f(x) = 2\frac{rand}{rand - max} - 1$$

```
/**
* initialize the weights and biases of the
* "Layer" object to random values
*/
void Layer::randomize()
{
    for (int j = 0; j < neuron_Count; j++)
    // We divide rand_num by rand_mak to get a number
   between
    // 0 and 1 then multiply that by
    // 2 to fix errors, and then subtract -1 to get the
   correct
    //number, which is between 0 and 1 or 0 and -1

    {
        bias[j] = 2 * ((static_cast <float> (rand())) /
        (static_cast <float> (RAND_MAX))) - 1;

        for (int k = 0; k < weight_Count; k++)
        {
            weight[j][k] = 2 * ((static_cast <float> (rand()
   ))
            / (static_cast <float> (RAND_MAX))) - 1;
        }
    }
}
```

Source Code 7.7: Randomize Funktion

Am Ende bekommt man zufällige Gleitkommazahlen zwischen 0 und 1 oder 0 und -1

7.4 Netzwerk

Als erstes sollte ein Network Konstruktor initialisiert werden, mit der gegebenen Anzahl an Layers, Inputs, Hidden-Units und Outputs. Es sollte als nächstes prüfen, ob die Übergabeparameter gültig sind (mindestens 2 Layers, mindestens 1 Input und mindestens 1 Output). Wenn die Übergabeparameter ungültig sind, wird eine Fehlermeldung ausgegeben und das Programm beendet. Ansonsten wird ein Array für die korrekten Antworten initialisiert, ein Array für die Layer definiert und mithilfe von *new* dynamisch Speicherplatz reserviert. Danach werden die einzelnen Layer definiert, beginnend mit

dem Input-Layer über die Hidden-Layers bis zum Output-Layer. Jeder Layer wird mit den passenden Anzahlen an Inputs und Gewichten initialisiert.

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

2. think Funktion:
In der think Funktion ist die Aktivierungsfunktion implementiert.

$$y_i = s\frac{1}{1 + e^{-} \sum(W_{ij} * V_i) + b_i}$$

Die Funktion "think"wird genutzt, um den Wertebereich der Aktivierung jedes Neurons im Netzwerk zu berechnen. Die "think"Funktion berechnet den Wertbereich der Aktivierung a jedes Neurons in einer bestimmten Schicht des neuronalen Netzwerks, indem es die Summe der Produkte der aktivierten Neuronen in der vorhergehenden Schicht **(previous-layer->a[k])** mit den entsprechenden Gewichten.
(lyrList[i]->W[j][k]) berechnet und dann mit dem Bias-Wert.
(lyrList[i]->bias[j]) addiert. Der resultierende Wert.
(lyrList[i]->z[j]) wird anschließend durch die Anwendung der Sigmod-Funktion.
(lyrList[i]->a[j] = sigmod(lyrList[i]->z[j])) skaliert, um den endgültigen Wertbereich der Aktivierung a des Neurons zu bestimmen.

```
/**
* @brief activate the neural network and calculate
* the outputs for all neurons
*/
void Network::think()
{
    /// The loop calls the "fire" method for each layer,
    /// passing the previous layer as a parameter.
    for (int i = 1; i < n_Layers; i++)
        {
            ///store the previous layer in a pointer p
            Layer const * p = layer_List[i - 1];
            layer_List[i]->fire(p);
        }
    }
}
```

Source Code 7.8: Think Funktion

3. fire Funktion: Die Funktion 'fire' berechnet die Aktivierungen der Neuronen in der aktuellen Schicht (Layer) anhand der als Parameter übergebenen Aktivierungen aus der vorherigen Schicht (prevLayer).
Für jedes Neuron 'j' in der aktuellen Schicht berechnet es die gewichtete Summe der Aktivierungen aus der vorherigen Schicht 'z[j]' durch Iterieren über die Gewichte weight[j][k], die die vorherige Schicht mit der aktuellen Schicht verbinden, und fügt einen 'Bias-Term' hinzu Voreingenommenheit[j]. Schließlich wird die Aktivierungsfunktion, hier die **Sigmoidfunktion 1,0f / (1 + exp(-z[j]))**, auf die gewichtete Summe **z[j]** angewendet, um die Aktivierung **a[j]** des Neurons 'j' zu erzeugen.

```
// The function takes in a pointer to a previous layer (
   prevLayer)
void Layer::fire(Layer const *prevLayer)
{
    for (int j = 0; j < neuron_Count; j++)
    {
        z[j] = 0.0f;
        for (int k = 0; k < weight_Count; k++)
        {
        //output from the previous layer and the weights
        //connecting the previous layer
        //to the courrent layer

        z[j] += prevLayer->getActivation(k) * weight[j][k];
        }
```

```
15          //followed by the addition of a bias term (bias[j])
16          //and the application of an
17          // "activation function"
18          z[j] += bias[j];
19          //a[j] = sigmoid_func(z[j]);
20          a[j] = 1.0f / (1 + exp(-z[j]));
21      }
22  }
```

Source Code 7.9: Fire Funktion

4. learn Funktion: Die Funktion *Learn* implementiert den Backpropagation-Algorithmus zum Trainieren eines neuronalen Netzes.
Das Netzwerk hat ein Array von Schichten layer-List, und n-Layers gibt die Anzahl der Schichten an. n-Outputs und n-Hiddens geben die Anzahl der Ausgaben bzw. versteckten Neuronen an. Das Array correct-Answer speichert die erwarteten Ausgabewerte.

Die Funktion beginnt mit der Ausgangsschicht und berechnet den Gradienten der Kosten in Bezug auf die Aktivierungen ***dCdA*** und Bias ***dCdB*** für jedes Ausgangsneuron ***j***. Es aktualisiert auch die Gewichte ***dCdW***, die die Ausgangsschicht mit der vorherigen verborgenen Schicht verbinden. Die Gradientenberechnung verwendet die ***inputdCdA-Funktion***, um den Gradienten zu akkumulieren, die ***movedCdW*** Funktion, um die Gewichtungen zu aktualisieren, und die „movedCdB"-Funktion, um die Abweichungen zu aktualisieren.

Die Funktion durchläuft dann die verborgenen Schichten vom Ende bis zum Anfang und berechnet den Gradienten *dCdA* für jedes verborgene Neuron ***j*** in jeder Schicht. Der Gradient wird mit der Funktion ***movedCdA*** akkumuliert und basiert auf dem Gradienten der nächsten Schicht, **layer-List[i + 1]**, und der Ableitung der Aktivierungsfunktion ***transform-Prime***. Die Funktion aktualisiert auch die Gewichtungen und Vorspannungen, die die verborgene Schicht mit der vorherigen Schicht verbinden, indem sie die Funktionen ***movedCdW*** und ***movedCdB*** verwendet.

Die Lernrate *learn-Rate* und die Stapelgröße *batch-Size* werden verwendet, um die Aktualisierungen auf die Gewichtungen, Abweichungen und Aktivierungen zu skalieren, wo bei die Stapelgröße auf die Anzhal der Traningsbeispiele sich bezieht [16].

```
void Network::learn(float learn_Rate, int batch_Size) //
    stochastic gradient descent + back propegation
  {

    //Calculate Output Layer

    for (int j = 0; j < n_Outputs; j++)
    {
      layer_List[n_Layers-1]->inputdCdA((layer_List[n_Layers
    -1]->getActivation(j) - correct_Answer[j]) *
    transform_Prime(layer_List[n_Layers-1]->getZ(j)), j);

      for (int k = 0; k < n_Hiddens; k++)
      {
        layer_List[n_Layers-1]->movedCdW(-(learn_Rate / (
    float)batch_Size) * layer_List[n_Layers-1]->getdCdA(j) *
    layer_List[n_Layers-2]->getActivation(k), j, k);
      }

      layer_List[n_Layers-1]->movedCdB(-(learn_Rate / (float
    )batch_Size) *  layer_List[n_Layers-1]->getdCdA(j), j);
    }

    //Calculate Hidden Layers

    for (int i = (n_Layers - 2); i > 0; i--)
    {
      for (int j = 0; j < layer_List[i]->getNumNeurons(); j
    ++)
      {
        layer_List[i]->inputdCdA(0.0f, j);

        for (int k = 0; k < layer_List[i + 1]->getNumNeurons
    (); k++)
        {
            layer_List[i]->movedCdA(layer_List[i + 1]->
    getWeight(k,j) * layer_List[i + 1]->getdCdA(k) *
    transform_Prime(layer_List[i]->getZ(j)), j);
        }

        for (int k = 0; k < layer_List[i - 1]->getNumNeurons
    (); k++)
        {
          layer_List[i]->movedCdW(-(learn_Rate / (float)
    batch_Size) * layer_List[i]->getdCdA(j) * layer_List[i -
```

```
    1]->getActivation(k), j, k);
34          }
35
36          layer_List[i]->movedCdB(-(learn_Rate / (float)
    batch_Size) * layer_List[i-1]->getdCdA(j), j);
37        }
38      }
39    }
40 }
```

Source Code 7.10: Learn Funktion

In der *learn* Funktion werden die neue Gewichte durch die Implementation folgende Gleichungen berechnet:

$$W_{new} = W_{alt} + (\frac{\eta}{batchsize}) * \frac{\partial E}{\partial W_i}$$

$$B_{new} = B_{alt} + (\frac{\eta}{batchsize}) * \frac{\partial E}{\partial A_i}$$

Im nächsten Schritt wurde eine Benutzeroberfläche erstellt, um die MNIST-Datenbank zu testen. Dies ist in Abbildung 9 zu sehen.

```
-------------Welcome--------------.
The goal of our program is to recognize the handwritten digits and continue to m
easure the execution time of the various algorithms.
*
*
Please enter a number:
1- Measure the execution time of the algorithms.

2- Costume setting.

3- Default setting.

4- Exit

Your input number is:
```

Abbildung 9: UI für das Program

Dem Benutzer stehen drei Optionen zur Verfügung:

1. Die Laufzeit von *Think- und Learn-Phasen* kann gemessen werden, indem vier verschiedene Architekturen getestet werden: *SISD, SIMD, Parallel und Parallel und SIMD.*

2. Der Benutzer kann die *MNIST-Daten* testen und die entsprechenden Ergebnisse ausgeben lassen. Dabei können benutzerdefinierte Daten eingegeben werden, wie z.B. welcher Architekturen ausgewählt werden sollen, welche Anzahl an Hidden Layers oder welche Lernrate verwendet werden soll.

3. Bei der dritten Option wird alles automatisch mit Standardeingaben erledigt. Die Anzahl der *Hidden Layers* und die Lernrate sind bereits vorgegeben.

7.5 Unit-Test

Um die erstellten Funktionen zu testen haben wir ein Unit-Test erstellt. Der Unit-Test setzt sich in diesem Projekt aus der Unit-Test Datei und dem XOR-problem Verzeichnis, siehe [4] zusammen. Da in diesem Projekt viele Mathematischen Formeln benutzt wurden, konnten nicht alle Funktionen des Netzes getestet werden. Die Unit-Tests wurden genutzt, um damit Funktionalität zu beweisen und auf Fehlersuche zu gehen. In der Unit-Test Datei wurde die Parameterübergabe der Layer Klasse getestet und im XOR-problem Verzeichnis wurden die Lern und Think Funktionen getestet.

7.6 GitLab-Runner

GitLab-Runner ist ein Open-Source-Tool, das von GitLab bereitgestellt wird und als Continuous Integration (CI)- und Continuous Deployment (CD)-Agent für *GitLab* fungiert. Mit *GitLab* Runner kann man *Builds, Tests und Deployments* automatisch ausführen, um sicherzustellen, dass die Anwendungen schnell und zuverlässig sind.
GitLab-Runner kann in verschiedenen Modi ausgeführt werden, wie z.B. als *Shell-Executor, Docker-Executor oder Kubernetes-Executor*, um Builds in isolierten Umgebungen auszuführen. Diese Modi ermöglichen Builds in einer

konsistenten Umgebung auszuführen und die Abhängigkeiten der Anwendungen zu verwalten[7].
Bevor wir unser Projekt abgeben möchten, werden die Server von *GitLab-Runner* abgeschaltet (Shared-Runner), weshalb wir unseren eigenen -Runner zum System hinzufügen müssen. Wir haben uns für *Docker Alpine* entschieden, da es schnell ausgeführt werden kann und im Vergleich zu anderen Images weniger Ressourcen benötigt. Dadurch können Builds und Tests schneller ausgeführt werden und es wird weniger Speicherplatz auf dem Host-System benötigt.
Insgesamt ist *GitLab-Runner* ein nützliches Tool für Entwickler, um sicherzustellen, dass die Anwendungen schnell und zuverlässig sind, indem sie *Builds, Tests und Deployments* automatisieren[6].

Kapitel 8

Laufzeit

Die Laufzeit eines Neuronen-Netzwerkes hängt von verschieden Faktoren ab. Beispiele sind: Geschwindigkeit des Prozessors, größe des Netzwerkes, Anzahl der Neuronen und Schichten, Anzahl der Trainingsdateien sowie der Optimierungsalgorithmus. Die Anzahl der Neuronen und Schichten spielen hierbei eine große Rolle. Zu viele Neuronen/Schichten erhöhen die Laufzeit proportional.

Die Wahl des Optimierungsalgorithmus ist wichtig, abhängig von der Schnelligkeit oder Genauigkeit. Ein schnellerer Optimierungsalgorithmus hat unter umständen eine geringe Genauigkeit als eine langsamer Optimierungsalgorithmus.

Die Laufzeit kann verbessert werden, in dem die Threads des Prozessors Parallel laufen. In diesem Projekt wurde OpenMP zum Parallelisieren verwendet, siehe Quellen (5). OpenMP ermöglicht es einfach Anwendungen parallel laufen zu lassen. Abschnitte eines Codes können für die Parallelisierung gekennzeichnet werden, wodurch OpenMP diesen Abschnitt auf mehreren Prozessorkernen ausführt und damit die Laufzeit erheblich verkürzen kann. Parallelisierung wird häufig verwendet, um die Berechnung von Neuronen und Schichten parallel auszuführen.
Parallele Ausführung kostet allerdings auch mehr Speicher und erhöht die Fehleranfälligkeit des Netzwerkes. Daher ist auf eine geschickte Anwendung zu achten.

Für das Projket wurden vier Architekturen (*sisd, simd, parallel, simd und*

parallel) verwende:
SISD (Single Instruction Single Data), ist die klassische Cpmputerarchitektur. Hier führt eim einziger CPU Kern eine einzige Anweisung auf ein Datenelement aus.

SIMD (Single Instruction Multiple Data) ermöglicht eine beschleunigte Datenverarbeitung. Hier wird eine gleiche Anweisung gleichzeitig auf mehrere Datenelemente angewendet.

Parallel bedeutet, dass mehrere CPU-Kerne gleichzeitig mehrere Anweisungen ausführen, um die Geschwindigkeit der Verarbeitung zu erhöhen.

simd-parallel ist eine spezielle Form der parallelen Verarbeitung, bei der SIMD-Technologie genutzt wird, um die Verarbeitungsgeschwindigkeit noch weiter zu beschleunigen.

Bei der Verarbeitung der MNIST-Datenbank für neuronale Netze kann die Verwendung von Parallel- oder SIMD-Architekturen die Geschwindigkeit der Verarbeitung verbessern, im Vergleich zu einer SISD-Architektur. Bei diesem Projekt man kann anhand die Laufzeitzhalen, siehe [4] feststellen, das *simd-paralle* schneller als simd und parallel ist.

Um das Projekt zu evaluieren, wird eine Laufzeitmessung für verschiedene Architekturen durchgeführt, einschließlich *SISD, SIMD, parallel, und SIMD parallel*. Die Messungen werden auf einem Linux-System (Ubuntu 20.04.5 LTS) durchgeführt. Der Prozessor stammt von der Firma *AMD* (Ryzen 5 2600 Six-Core Processor x 12).. Diese Messung dient dazu, die Leistung der verschiedenen Architekturen miteinander zu vergleichen. Hierbei werden die Ausführungszeiten für die Funktionen *Think und Learn* gemessen. Es ist wichtig, sicherzustellen, dass das Programm für jede Architektur optimiert ist, um die bestmögliche Leistung zu erzielen.
Die gemessene *Think und Learn* sind in der nachfolgenden Abbildung 10 und Abbildung 11 dargestellt.

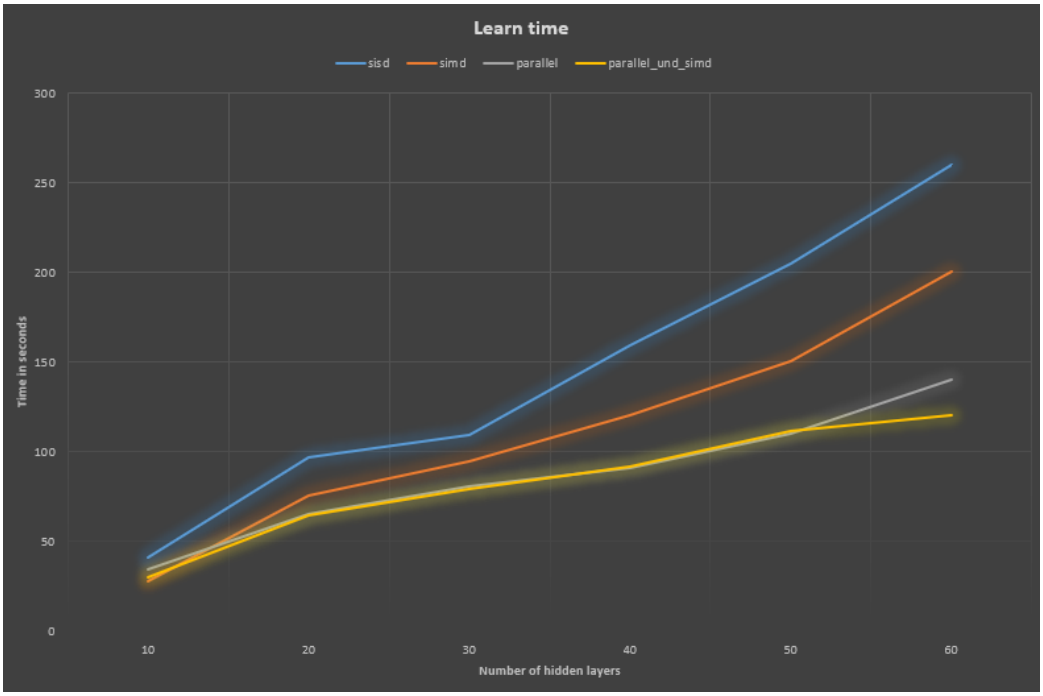


Abbildung 10: learn time

Die Ergebnisse werden auf einem *Linux-Betriebssystem* durchgeführt, wobei die Lern- und Denkzeit gemessen wird. Die Messungen sind abhängig von der Anzahl der *hidden layers* (von 10 bis 60). Anhand von Abbildung 19 und 20 lässt sich feststellen, dass die Zeit bei *sisd* am längsten und bei *parallel und parallel-simd* am kürzesten ist. Zum Beispiel beträgt die Denkzeit bei 60 *hidden layers* bei sisd fast 30, während sie bei *parallel und parallel-simd* fast 15 beträgt. Parallel-simd ist schneller, da diese Instruktionen nicht sequentiell (veraurbeitung in einer bestimmten Reinfolge) arbeitet, sondern die gleichen Operation auf mehrere Daten gleichzeitig angewenden. Das ist gerade bei großen Datenmengen sehr Sinnvoll. Somit können mehrere Pixel einer Handgeschriebenen Zahl auf einmal bearbeitet werden, um damit das Netzwerk schneller zu machen.

Die MNIST-Datenbank habe wir mit verschiedenen Einstellungen getestet, und das Gesamtergebnis betrug eine korrekte Antwortrate von 93,33%. Bei diesem Test wurden 60 Hidden Layers und eine Lernrate von 1.0.F mit dem parallelen verwendet. Die gesamte MNIST-Datenmenge wurde dabei geprüft,

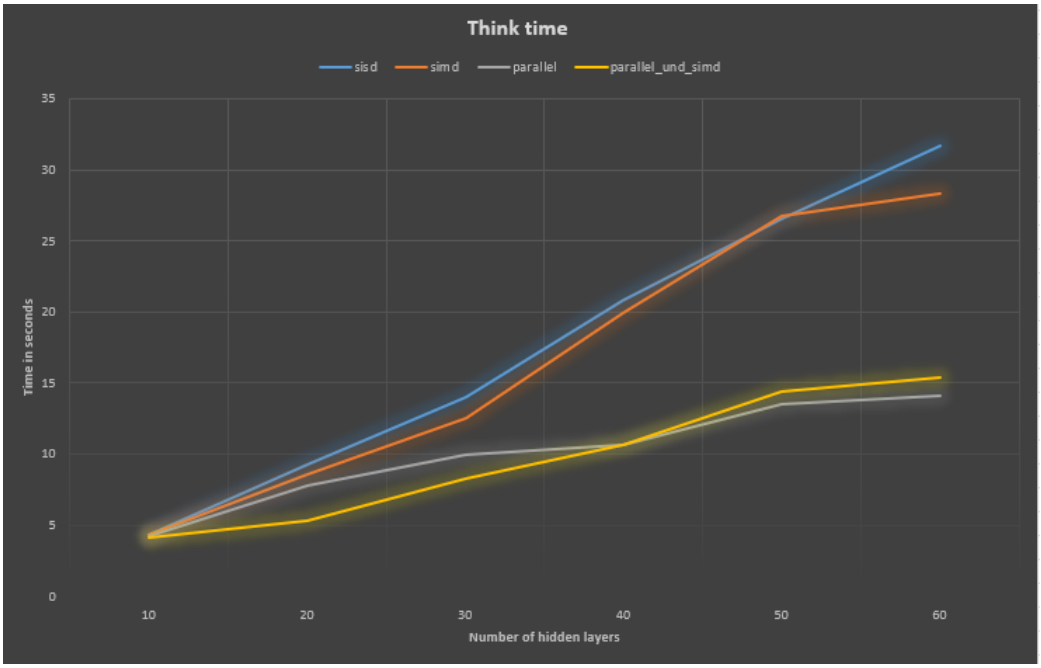


Abbildung 11: think time

einschließlich 60.000 Trainingsbildern und 10.000 Testbildern.

Kapitel 9

Herausforderung

Der Bau eines Neuronen-Netzwerkes stellt viele Herausforderungen dar. Um ein effizientes und präzises Netzwerk zu erhalten gibt es einiges zu beachten:

9.1 Datenaufbereitung

Datenaufbereitung ist ein wichtiger Teil des Projektes. Damit ein Neuronen-Netzwerk Daten verarbeiten kann, müssen die Daten erst aufbereitet und gereinigt werden, [17].

9.2 Over- / Unerfitting

Over- und Underfitting sind zwei Herausforderungen, welche Schwierigkeiten darstellen. Wenn sich das Netzwerk zu sehr auf die Trainingsdaten anpasst, spricht man von Overfitting. Das Netzwerk kann in diesem Fall die Trainingsdaten sehr gut auswerten, die Testdaten hingegen haben eine hohe Fehlerquote. Overfitting kann auftreten, wenn das Netzwerk zu viele Freiheiten hat und zu Komplex ist. Underfitting tritt auf, wenn das Netzwerk nicht in der Lage ist die komplexen Beziehungen der Trainingsdaten zu erfassen. Somit ist die Leistung des Netzwerkes sowohl bei den Trainingsdaten als auch bei den Testdaten gering. Um Over und Underfitting zu vermeiden kommt es darauf an den richtigen grad an Komplexität zu finden. Dafür gibt es Methoden, welche helfen den richtigen grad an Komplexität zu erzielen.

Beispiele sind: Dropout, L1- und L2-Regularisierung und Early Stopping. Bei Dropout werden zufällig ausgewählte Neuronen während des Trainings ausgeschaltet, um eine Überanpassung des Modells an die Trainingsdaten zu vermeiden. Die L1- und L2-Regularisierung Fürgen der Kostenfunktion eine zusätzliche Konstante zu, welche die große der Neuronen beschränkt und somit eine Überanpassung vermeidet. Das Early Stopping bezieht sich auf Überwachungskriterien. Sind diese Überwachungskriterien erfüllt wird das Training frühzeitig beendet. In diesem Projekt wurde die MINST-Database mit sehr großen Trainingsdatensätzen verwendet. Dadurch bestand die Gefahr von Overfitting, [17].

9.3 Optimierungsproblem

Unter dem Optimierungsproblem versteht man, dass finden des besten Satzes von Gewichten. Damit wird die Vorhersagefähigkeit verbessert, wodurch zusätzlich die Genauigkeit des Neuronen-Netzes verbessert wird. Die Kostenfunktion, welche die Soll- und Ist-Vorhersage des Modells misst, wird angepasst. Danach werden die Gewichte angepasst, um den Fehler zu minimieren. Es existieren verschiedene Algorithmen für dieses Problem, wie den Gradientenabstieg.
Die Ableitungen der Kostenfunktion (Gradienten) werden berechnet, um die Richtung der Anpassung der Gewichte vorzugeben, in Richtung des Gardientenabstiegs. Damit dieses Verfahren einen Nutzen hat muss es mehrmals angewendet werden, bis die Fehlerfunktion minimal wird. Dieses Verfahren ist einfach zu Implementieren und trägt zu einer besseren Generalisierung bei, [1].

9.4 Overparametrization

Existieren in einem Neuronen-Netzwerk zu viele Neuronen und Schichten welche nicht den Anforderungen und Aufgaben des Netzwerkes entsprechen, dann spricht man von Overparametrization. Wenn es zu viele Neuronen und Schichten gibt hat das Netzwerk zu viel ungenutzte Freiheiten sich an das Muster der Trainingsdaten anzupassen. Wenn es zu wenige Neuronen und

Schichten hat, kann das Netz das Muster der Trainingsdaten nicht erfassen. Um Overparametrization zu vermeiden muss die Anzahl der Parameter sorgfältig gewählt werden. Um Overparametrisation zu vermeiden, ist es wichtig, ein Modell mit einer angemessenen Anzahl von Parametern zu wählen, die auf die verfügbaren Daten und das Problem, das gelöst werden soll, abgestimmt ist. Dies kann durch eine sorgfältige Auswahl der Architektur, Regulierung und Reduktion der Anzahl der Neuronen oder Schichten erreicht werden, [17].

Kapitel 10

Fazit

Zusammenfassend lässt sich sagen, dass das Projekt, bei dem ein neuronales Netzwerk Verwendung findet, um handgeschriebene Ziffern aus dem MNIST-Datensatz zu identifizieren, eine Herausforderung für jeden, der sich für den Bereich des maschinellen Lernens interessiert, aber auch eine große Chance bietet, die Fähigkeiten in diesem Bereich zu verbessern.
Durch die Verwendung des MNIST-Datensatzes kann das neuronalen Netzwerk trainiert werden, um handgeschriebene Ziffern mit hoher Genauigkeit zu erkennen.

Jedoch ist es wichtig zu beachten, dass eine gründliche Vorbereitung der Daten ein wesentlicher Faktor für den Erfolg des Projekts ist. Dies beinhaltet die Überprüfung und Bereinigung der Daten, um sicherzustellen, dass sie geeignet für das Training des neuronalen Netzwerks sind. Ebenso ist es wichtig, die Hyperparameter sorgfältig einzustellen, um eine optimale Leistung des neuronalen Netzwerks zu erzielen. Außerdem kann es hilfreich sein, verschiedene Architekturen zu evaluieren, um das bestmögliche Ergebnis zu erzielen.

Insgesamt bietet das Projekt eine einzigartige Möglichkeit, die Fähigkeiten im Bereich des maschinellen Lernens zu verbessern und ein tieferes Verständnis für die Funktionsweise neuronaler Netzwerke zu erlangen.

Literatur

[1] Amey Band. *How to find the optimal value of K in KNN?* URL: https://towardsdatascience.com/how-to-find-the-optimal-value-of-k-in-knn-35d936e554eb.

[2] Vitaly Bushaev. *How do we 'train' neural networks ?* URL: https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73.

[3] Global tech council. *Back-Propagation Algorithm:Everything You Need to Know - Global Tech Council.* URL: https://www.globaltechcouncil.org/machine-learning/propagation-algorithm/.

[4] Carlo Wolter Cynthia Odudu Husen Muhsen. *GitLab Projekt.* URL: https://gitlab.rz.htw-berlin.de/s0579286/ziffererkennung_husen_cynthia_2022_wise.

[5] datasolut. *Künstliche Neuronale Netzwerke: Definition, Einführung, Arten und Funktion.* URL: https://datasolut.com/neuronale-netzwerke-einfuehrung/#:~:text=Welche%20Arten%20von%20K%C3%BCnstlichen%20Neuronalen%20Netzen%20bestehen%3F%201,k%C3%B6nnen.%20...%204%20Recurrent%20Neural%20Networks%20%28RNN%29%20.

[6] GitLAb. *GitLab.* URL: https://github.com/gitlabhq/gitlab-runner.

[7] GitLab. *GiLab Runner.* URL: https://docs.gitlab.com/runner/.

[8] Digital Guide IONOS. *Neural Networks: Was können künstliche neuronale Netze.* URL: https://www.ionos.de/digitalguide/online-marketing/suchmaschinenmarketing/was-ist-ein-neural-network/.

[9] Daniel Johnson. *Back Propagation in Neural Network: Machine Learning Algorithm*. 2022. URL: https://www.guru99.com/backpropogation-neural-network.html.

[10] Zero knowledge. *Introduction to Deep Learning*. URL: https://zkmablog.com/2017/12/25/neuronale-netze-tutorial-uebersicht-der-konzepte/.

[11] Frank La La. URL: https://learn.microsoft.com/de-de/archive/msdn-magazine/2019/april/artificially-intelligent-how-do-neural-networks-learn.

[12] Corinna Cortes und Christopher J. C. Burges LeCun Yann. *The MNIST database of handwritten digits*. 2010. URL: http://yann.lecun.com/exdb/mnist/ (besucht am 19. 02. 2023).

[13] Judy Nduati. *Introduction to Deep Learning*. URL: https://www.section.io/engineering-education/introduction-to-deep-learning/.

[14] Berkeley Scientific. *How artificial neural networks work, from the math up*. URL: https://bsj.berkeley.edu/how-artificial-neural-networks-work-from-the-math-up/.

[15] The blog at the bottom of the sea. *A Geometric Interpretation of Neural Networks*. URL: https://blog.demofox.org/2017/02/07/a-geometric-interpretation-of-neural-networks/.

[16] Maximilian Solech. *How to learn functions on sets with neural networks*. URL: https://argmax.ai/blog/setinvariance/.

[17] Josh Tobin. *Troubleshooting Deep Neural Networks*. URL: https://fullstackdeeplearning.com/spring2021/lecture-7.

[18] wikipedia. *The MNIST database*. URL: https://de.wikipedia.org/wiki/MNIST-Datenbank.