

Belegarbeit
- Semesterprojekt - Neuronale Netze -

Mikroprozessortechnik (CE23)

Luca Alexander Schulz

Marcus Worrmann

HTW Berlin

30. September 2024



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

Inhaltsverzeichnis

1	Einleitung	5
2	Theoretische Grundlage	6
2.1	Historischer Kontext	6
2.2	Grundlagen neuronaler Netze	7
2.3	Mathematische Herleitung	10
I	Forward Propagation	11
II	Fehlerbestimmung	12
III	Back Propagation	12
3	Implementierung des neuronalen Netzwerks	13
3.1	MNIST-Datensatz	13
3.2	Implementierung in C	14
3.3	Verwendung von OpenMP	14
3.4	mpt_nn	15
I	Aktivierungsfunktionen	15
II	ForwardPass(Forwardpropagation)	15
III	Backpropagation	17
3.5	mpt_nn_utility	18
I	load_mnist	18
II	initialize_weights(), initialize_bias()	18
III	apply_dropout()	19
IV	visualize_mnist_digit()	19
V	print_options()	19
3.6	main	19
I	Verarbeitung der command line Parameter	19
II	Initialisierung	19
III	Training des Netzwerks	20
IV	Speicherfreigabe	20
3.7	mpt_nn_tests	20
I	test_sigmoid()	20
II	test_initialize_weights()	20
III	test_forward_pass(), test_backpropagation()	20
4	Ergebnisse	21
4.1	Erste Testläufe	21
4.2	Benchmarkanalyse mit Hyperfine	23
4.3	Lernfortschritt und Vorhersagbarkeit	24

5	Auswertung	26
5.1	Benchmark	26
5.2	Genauigkeit	27
5.3	Ansätze zur Verbesserung	28
6	Fazit	29

Abbildungsverzeichnis

1	Schematische Darstellung eines KNN[3]	7
2	Makefile Hyperfineskript	23
3	Benchmarkergebnisse mit Hyperfine	24
4	Mittelwerte der Genauigkeiten	25
5	Genauigkeiten der einzelnen Algorithmen	25

Tabellenverzeichnis

1	Parameter der Neuronen-Gleichung	10
2	Arten von Aktivierungsfunktionen	11
3	Genutzte Parameter des Testlaufs	21
4	Ergebnisse der Trainings-Epochen	22

Listings

1	forwardpass hidden	15
2	forwardpass output	16
3	forwardpass parallel	16
4	forwardpass simd	17
5	backpropagation outputerr	17
6	backpropagation deltaHidden	17
7	backpropagation weights	18

1 Einleitung

In der vorliegenden Seminararbeit wird der Entwicklungsprozess eines künstlichen neuronalen Netzes detailliert beschrieben. Ziel der Arbeit war die Erkennung von Ziffern, Buchstaben oder Objekten, wobei der Schwerpunkt auf der Erkennung von Ziffern mithilfe des MNIST-Datensatzes [5] liegt. Die algorithmische Implementierung erfolgt dabei in der Programmiersprache C.

Für die unterschiedlichen Varianten des maschinellen Lernens werden Ansätze des sequentiellen Lernens als induktiver Algorithmus sowie parallele und SIMD-basierte Methoden zur Erhöhung der Parallelität verwendet. Um die Leistungsfähigkeit der verschiedenen Algorithmen zu analysieren, werden die Trainingszeiten verglichen, ausgewertet und abschließend diskutiert.

Zur tiefergehenden Untersuchung der Qualität der Implementierungen werden darüber hinaus Tools wie **Valgrind** und **Helgrind** eingesetzt. Diese ermöglichen eine umfassende Analyse der Speicherverwaltung und Parallelitätsaspekte.

Im weiteren Verlauf wird zunächst die zugrunde liegende Theorie erläutert, gefolgt von einer detaillierten Beschreibung der Implementierung der Skripte. Anschließend werden die Ergebnisse visualisiert und diskutiert.

Abschließend bietet die Arbeit eine Zusammenfassung des Projekts, inklusive der gesteckten und erreichten Ziele, sowie einen Ausblick auf mögliche zukünftige Weiterentwicklungen. Ein Fazit rundet die Arbeit ab.

2 Theoretische Grundlage

In diesem Abschnitt werden die historischen Wurzeln, die theoretischen Grundlagen künstlicher neuronaler Netze umfassend erläutert. Dabei wird sowohl auf die grundlegenden Konzepte als auch auf die mathematischen Prinzipien eingegangen, die der Funktionsweise und dem Training dieser Netzwerke zugrunde liegen. Die einzelnen Aspekte werden systematisch dargestellt und eingehend analysiert.

2.1 Historischer Kontext

Die ersten wissenschaftlichen Ansätze zur Erforschung computergestützter Algorithmen, die auf der aus der Biologie bekannten Funktionsweise und Verarbeitung von Eindrücken und Informationen in neuronalen Netzen basieren, stammen aus den 1950er und 1960er Jahren. Der Psychologe Frank Rosenblatt entwickelte das Perzeptron, ein einfaches neuronales Netz, das grundlegende Muster erkennen konnte. Mit dem *Mark I Perceptron* war es möglich, Ziffern auf einem 20x20-Pixel-Sensor zu erkennen. Aufgrund seiner Unfähigkeit, nichtlinear separierbare Probleme zu lösen, verlor dieses Modell jedoch an Bedeutung.[1]

In den 1980er Jahren erlebten neuronale Netze eine Renaissance durch die Einführung des Backpropagation-Algorithmus, beschrieben im Werk "*Backpropagation of Error*", der eine Fehlerkorrektur während des Trainings ermöglichte. Dieser Fortschritt erlaubte es neuronalen Netzen, komplexe, nichtlineare Probleme zu lösen und legte den Grundstein für viele moderne Anwendungen der künstlichen Intelligenz.[1]

Durch die 2015 veröffentlichte Arbeit von Yann LeCun, Yoshua Bengio und Geoffrey Hinton wurde die Entwicklung des modernen Deep Learning maßgeblich vorangetrieben.[2] Sie kombinierten klassische Konzepte neuronaler Netze mit den heutigen Möglichkeiten leistungsfähiger Rechenkapazitäten, großer Datenmengen und verbesserter Netzwerkarchitekturen. Diese Ansätze ermöglichten den Durchbruch mehrschichtiger, tiefer neuronaler Netze, die in zahlreichen Bereichen wie der Bilderkennung und Sprachverarbeitung bahnbrechende Fortschritte erzielten. Für ihre Pionierarbeit erhielten die Forscher 2018 den renommierten Turing Award.

2.2 Grundlagen neuronaler Netze

Die im Verlauf des letzten Jahrhunderts verfolgten Ansätze zielten auf die Entwicklung und Implementierung künstlicher neuronaler Netzwerke (KNN) ab, die als leistungsfähige Datenverarbeitungsprogramme dienen und in ihrer Funktionsweise biologischen neuronalen Netzwerken ähneln [4].

Der Anwendungsbereich dieser Methoden lässt sich heutzutage in zwei wesentliche Kategorien unterteilen:

- Modellierte KNN zur Simulation und Erforschung der Funktionsweise des menschlichen Gehirns.
- KNN zur Lösung spezifischer Anwendungsprobleme in den Bereichen Statistik, Wirtschaft und Technik.

In diesem Zusammenhang erweitern künstliche neuronale Netzwerke unsere mathematischen Modelle der menschlichen Denkprozesse.

Der Aufbau eines solchen Netzwerks ist in der nachfolgenden Abbildung 1 dargestellt.

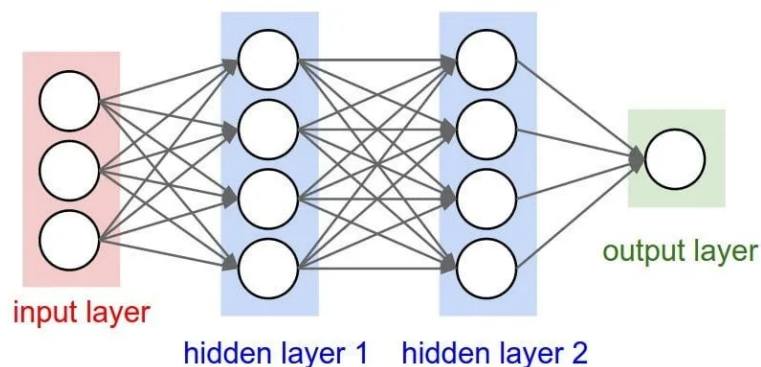


Abbildung 1: Schematische Darstellung eines KNN[3]

Wie in Abbildung 1 dargestellt, bestehen neuronale Netze aus mehreren Knoten - den sogenannten Neuronen - welche auch als Units oder Einheiten bezeichnet werden können. Die Hauptfunktion der Neuronen besteht darin, Informationen entweder aus der Umgebung oder von anderen Neuronen zu empfangen, diese zu modifizieren und an die Außenwelt oder wiederum an andere Neuronen weiterzuleiten. Dabei lassen sich die Neuronen in drei Schichten unterteilen:

- **Input-Layer:** Der Input-Layer besteht aus Units, die in der Lage sind, externe Signale unterschiedlicher Art zu empfangen. Die Anzahl dieser Units entspricht der Anzahl der Merkmale in den Eingangsdaten.
- **Hidden-Layer:** Die Hidden-Layer bilden die Schicht von Neuronen, die sich zwischen dem Input- und dem Output-Layer befinden. In diesen Schichten werden die Eingangsdaten weiterverarbeitet, wodurch die Komplexität und Leistungsfähigkeit des Netzwerks maßgeblich beeinflusst wird.
- **Output-Layer:** Der Output-Layer ist die letzte Schicht des Netzwerks und besteht aus Neuronen, die für die Ausgabe des endgültigen Signals an die Außenwelt verantwortlich sind. Die Anzahl der Neuronen in dieser Schicht repräsentiert die Anzahl der Ausgabesignale.

Die Units der einzelnen Layer sind durch Kanten miteinander verbunden, deren Verbindungsstärken durch Gewichte zwischen den Neuronen definiert werden. Diese Gewichte repräsentieren den Einfluss, den ein Signalwert auf das nachfolgende Neuron ausübt, und bestimmen somit die Verstärkung oder Abschwächung des Signals.

Das Lernverhalten eines neuronalen Netzwerks wird als Anpassung der Gewichte zwischen den Neuronen beschrieben.[4] Die Veränderung dieser Gewichte unterliegt den zugrunde liegenden Lernregeln. Diese Lernregeln beruhen häufig auf der Anwendung nichtlinearer Aktivierungsfunktionen, die auf die Summe der gewichteten Eingangssignale angewendet werden.

Zusammenfassend lassen sich die fundamentalen Bestandteile und Strukturen eines neuronalen Netzwerks wie folgt beschreiben:

- **Neuron:** Ein Neuron empfängt eine ein oder mehrere Eingangssignale, welche jeweils mit zugehörigen Gewichten multipliziert werden. Anschließend werden die gewichteten Eingangssignale summiert, und das Ergebnis wird durch eine Aktivierungsfunktion verarbeitet. Das Ergebnis dieser Aktivierungsfunktion bestimmt, ob und in welcher Intensität das Neuron ein Ausgangssignal an nachfolgende Neuronen weitergibt.
- **Gewichte:** Die Verbindungen zwischen Neuronen werden durch Gewichte charakterisiert, die den Einfluss eines Eingangssignals auf das nachfolgende Neuron bestimmen. Diese Gewichte werden während des Trainingsprozesses kontinuierlich angepasst, um die Leistung des Modells zu optimieren. Der Trainingsprozess zielt darauf ab, die Gewichte so zu modifizieren, Abweichungen zwischen den Vorhersagen des Modells und den tatsächlichen Ausgaben minimiert werden. Die Gewichte

lassen sich basierend auf ihrem Einfluss auf die Signalübertragung in drei Kategorien einteilen:

- [A] *Positives Gewicht*: Ein Neuron übt einen exzitatorischen, also stimulierenden Einfluss auf ein anderes Neuron aus.
 - [B] *Negatives Gewicht*: Ein Neuron übt einen inhibitorischen, also hemmenden Einfluss auf ein anderes Neuron aus.
 - [C] *Gewicht von Null*: Ein Neuron hat keinen Einfluss auf ein jeweils anderes Neuron.
- **Schichten**: Die verschiedenen Schichten eines neuronalen Netzwerks dienen als spezialisierte Einheiten, die unterschiedliche Aufgaben der Datenverarbeitung und Mustererkennung übernehmen.
 - **Aktivierungsfunktion**: Aktivierungsfunktionen, auch als Transferfunktionen bezeichnet, bestimmen die Beziehung zwischen dem Netzeingang und dem Aktivitätsniveau eines Neurons. Sie spielen eine zentrale Rolle in neuronalen Netzwerken, in der Umsetzung der nichtlinearität eines Modells, welche notwendig ist, um komplexe Muster zu lernen und darzustellen. Zu den wichtigsten Aktivierungsfunktionen gehören hierbei die **Sigmoid-Funktion**, der **Hyperbolische Tangens (Tanh)** sowie die **ReLU-Funktion (Rectified Linear Unit)**. Durch diese Funktionen besitzt ein neuronales Netzwerk die Fähigkeit, differenzierte Zusammenhänge im Datenraum zu modellieren.

2.3 Mathematische Herleitung

Grundsätzlich lassen sich neuronale Netze kompakt in Form von Matrizen darstellen. Eine solche Matrix, bezeichnet als W , besteht aus einer Menge von Elementen w_{ij} , wobei i die Output-Unit und j die Input-Unit repräsentiert. Da das Lernen in neuronalen Netzen auf der Anpassung von Gewichten basiert, werden diese Netzgewichte durch die Matrix abgebildet. Einfache neuronale Netze, die keine Hidden-Layers beinhalten, können durch eine einzelne Matrix dargestellt werden. Für neuronale Netze mit Hidden-Layern wird für jede dieser Schichten eine separate Gewichtsmatrix benötigt, um die entsprechenden Verbindungen darzustellen.

Die Struktur einer solchen Matrix für ein einfaches neuronales Netz ohne Hidden-Layer kann folgendermaßen aussehen:

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}$$

Diese mathematische Darstellung ist jedoch nicht ausreichend für neuronale Netze, die in der Lage sein sollen, komplexe und differenzierte Zusammenhänge zu modellieren, wie sie die in Unterabschnitt 2.2 genannten Aktivierungsfunktionen erfordern. Um die Transformationen solcher Funktionen zu modellieren, kann folgende mathematische Grundlage verwendet werden, bei der ein einzelnes Neuron sein Ausgangssignal y wie folgt berechnet:

$$y = f \left(\sum w_i * x_i + b \right)$$

Symbol	Beschreibung
y	Ausgangssignal des Neurons
f	Aktivierungsfunktion (z.B. Sigmoid, Tanh, ReLU)
\sum	Summe der gewichteten Eingaben
w_i	Gewicht für den i -ten Eingabewert
x_i	Eingabewert zum Neuron
b	Bias (Verschiebungsterm)

Tabelle 1: Parameter der Neuronen-Gleichung

Wie in Tabelle 1 dargestellt, können Aktivierungsfunktionen hinsichtlich ihrer Form unterschieden werden. Sie lassen sich nach dem zugrunde liegenden Netzinput und dem gewünschten Aktivitätsniveau klassifizieren. Die nachfolgende Tabelle bietet einen kurzen Überblick über verschiedene Aktivierungsfunktionen und ihre Eigenschaften:

Funktionsart	Funktion	Formel
Sigmoid	Transformiert Eingaben in einen Bereich zwischen 0 und 1. Häufiger Einsatz für binäre Entscheidungen. Ähnlich der binären Schwellenfunktion.	$f(x) = \frac{1}{1+e^{-x}}$
Hyperbolischer Tangens	Transformiert Eingaben in einen Bereich zwischen -1 und 1. Nützlich für zentrierte Ausgaben.	$f(x) = \tanh(x)$
ReLU	Setzt negative Werte auf Null und behält positive Werte bei.	$f(x) = \max(0, x)$

Tabelle 2: Arten von Aktivierungsfunktionen

Ein wesentliches Problem, das bei der Verwendung von Netzen mit Hidden-Layers auftritt, ist die mangelnde Transparenz des Fehlerpotenzials der Neuronen in diesen Schichten. Im Gegensatz zu einfachen Netzen, bei denen der Fehlerausdruck im Output direkt erkennbar ist, muss dieser Fehlerterm in komplexeren Systemen auf andere Weise ermittelt werden. Die Anpassung der Gewichte erfolgt im Laufe verschiedener Trainingsphasen, wobei die Gewichtsänderung in drei Schritten vollzogen wird.

I Forward Propagation

Die Inferenz, auch als *Forward Propagation* bekannt, beschreibt den Prozess, bei dem der Input durch die Neuronen im Netzwerk weitergeleitet wird, um bestimmte Outputs zu berechnen. Jede Schicht erhält dabei schrittweise den Ausgabewert der vorherigen Schicht als Eingabe, bis das Netzwerk komplett durchlaufen wurde und ein Output am Output-Layer berechnet werden kann. Mathematisch lässt sich dieser Prozess durch die folgende Gleichung darstellen, wobei l_i den Output der i -ten Schicht darstellt:

$$l_{i+1} = f(w_i \cdot l_i + b_i)$$

II Fehlerbestimmung

Im zweiten Schritt, der Fehlerbestimmung, wird die Differenz zwischen dem gewünschten Ausgabewert y_{true} und dem durch die Forward Propagation tatsächlich erzielten Ausgabewert y_{pred} ermittelt. Dieser Fehler E wird durch die nachfolgende quadratische Fehlerfunktion berechnet. Der Vorteil einer quadratischen Mittelwertbestimmung liegt darin, dass ausschließlich positive Ergebnisse erzielt werden, was die Aufsummierung erleichtert. Zudem fallen große Fehler stärker ins Gewicht, was für den dritten Schritt im Lernprozess ausschlaggebend ist.

$$E = \frac{1}{2}(y_{\text{true}} - y_{\text{pred}})^2$$

III Back Propagation

Überschreiten die in Unterunterabschnitt II genannten Fehler einen definierten Gütewert, so werden im finalen Schritt die Gewichte der Hidden-Layer angepasst. Hierfür werden die Fehlerterme entgegengesetzt rückwirkend zum Input-Layer angepasst. Das bedeutet, dass zunächst die Gewichte zwischen dem Output-Layer und dem letzten Hidden-Layer modifiziert werden und sich sukzessiv zum Input-Layer vorgearbeitet wird. Die Gewichte werden nach folgender Gleichung angepasst, um die Fehlerterme zu verringern. Für jedes Gewicht w wird das Update über die Lernrate η und $\frac{\partial E}{\partial w}$ als Gradient des Fehlers vom anfänglichen Gewicht abgezogen:

$$w_{\text{neu}} = w_{\text{alt}} - \eta \cdot \frac{\partial E}{\partial w}$$

Es sei erwähnt, dass es durchaus noch weitere Lernregeln und Methodiken wie die *Hebb-Regel*, die *Delta-Regel* oder das *Competitive-Learning* gibt. Da in dieser Arbeit jedoch die Methoden der Backpropagation im Fokus stehen, werden die anderen Regeln nicht weiter beleuchtet.

3 Implementierung des neuronalen Netzwerks

In diesem Kapitel wird die Herangehensweise zur Umsetzung und Implementierung des Codes eines neuronalen Netzes beschrieben. Der Schwerpunkt liegt auf dem zugrunde liegenden MNIST-Datensatz, der als Trainingsgrundlage dient. Zudem werden die einzelnen Strukturen innerhalb des Codes, die in Abschnitt 2 erläutert wurden, sowie verschiedene Ansätze zur Optimierung der Algorithmen und des implementierten Codes detailliert betrachtet.

3.1 MNIST-Datensatz

Der MNIST-Datensatz (Modified National Institute of Standards and Technology) gilt als einer der bekanntesten Datensätze im Bereich des maschinellen Lernens. Zudem wird er häufig als Test- und Trainingsgrundlage für Bildklassifikationsalgorithmen verwendet. Der Datensatz wurde über *TensorFlow* heruntergeladen und umfasst folgende zwei separate CSV-Dateien:

```
train-images.idx3-ubyte
```

```
train-labels.idx1-ubyte
```

Insgesamt enthält der Datensatz 70.000 Bilder handgeschriebener Ziffern im Bereich von 0 bis 9. Für Trainingszwecke werden 60.000 der Bilder verwendet, während die restlichen 10.000 zur Validierung und zur Berechnung der Genauigkeit herangezogen werden.

3.2 Implementierung in C

In diesem Kapitel wird die Implementierung des neuronalen Netzwerks in der Programmiersprache C erläutert. Das Projekt ist modular aufgebaut und besteht aus mehreren Dateien, die jeweils unterschiedliche Aspekte des Netzwerks abdecken. Die Hauptkomponenten hierbei sind:

- `mpt_nn.c`: Enthält die Kernfunktionen für die ForwardPass und die Backpropagation des Netzwerks.
- `mpt_nn_utility.c`: Beinhaltet Hilfsfunktionen für die Datenverarbeitung und Initialisierung.
- `main.c`: Die Hauptprogrammdatei, welche das Training und die Auswertung des Netzwerks steuert.
- `mpt_nn_test.c`: Enthält Unit-Tests zur Validierung der Netzwerkfunktionen.

Das neuronale Netzwerk wurde so konzipiert, dass es sowohl sequenziell als auch parallel ausgeführt werden kann. Durch die Verwendung von OpenMP wird die Möglichkeit geschaffen, die Berechnungen auf mehrere Threads zu verteilen und somit die Leistung auf Mehrkernprozessoren zu optimieren.

3.3 Verwendung von OpenMP

OpenMP ist eine API zur Unterstützung von Multithreading in C und ermöglicht die einfache Parallelisierung von Codeabschnitten durch Compiler-Direktiven. Die in diesem Projekt verwendeten Direktiven lauten:

- `#pragma omp parallel for`: Parallelisiert die nachfolgende Schleife über die verfügbaren Threads.
- `#pragma omp simd`: Ermöglicht die Vektorisierung von Schleifen für SIMD-Instruktionen.
- `schedule(static)`: Teilt die Schleifeniterationen gleichmäßig auf die Threads auf.

Durch die Kombination dieser Direktiven werden die Berechnungen effizient auf die Hardware verteilt, was zu einer verbesserten Leistung führt.

3.4 mpt_nn

Die Datei `mpt_nn.c` enthält die Kernimplementierung des neuronalen Netzwerks, einschließlich der Funktionen für den ForwardPass und die Backpropagation.

I Aktivierungsfunktionen

Sigmoid-Funktion

Die Sigmoid-Funktion ist eine häufig verwendete Aktivierungsfunktion in neuronalen Netzwerken. Sie transformiert den Eingabewert x in einen Wert zwischen 0 und 1. Dies ermöglicht es dem Netzwerk, nichtlineare Beziehungen zu modellieren und Wahrscheinlichkeiten abzubilden.

Die aus Tabelle 2 beschriebene Funktion reflektiert hierbei folgende Eigenschaften:

- Für sehr große negative Werte von x nähert sich $f(x)$ dem Wert 0.
- Für sehr große positive Werte von x nähert sich $f(x)$ dem Wert 1.
- Bei $x = 0$ nimmt die Funktion den Wert $f(0) = 0,5$ an.

Zusätzlich zur Sigmoid-Funktion wird die Ableitung dieser benötigt, um die Gradienten während der Backpropagation zu berechnen.

II ForwardPass(Forwardpropagation)

Der Forwardpass berechnet die Ausgaben des `mpt_nn` basierend auf den Eingaben, den aktuellen Gewichten und Biases. Die Berechnung erfolgt in zwei Hauptschritten:

1. **Berechnung der hidden Layers:** Jedes Neuron der Hidden-Layer summiert die gewichteten Eingaben und den Bias und wendet die Aktivierungsfunktion an.¹

```
1 for (int i = 0; i < numHiddenNodes; i++)
2 {
3     double activation = hiddenLayerBias[i];
4     for (int j = 0; j < numInputs; j++)
5     {
6         activation += inputs[j] * hiddenWeights[j][i];
7     }
8     hiddenLayer[i] = sigmoid(activation);
9 }
10
```

Listing 1: forwardpass hidden

2. Anwendung der Dropout - Rate

3. **Berechnung der Ausgabeschicht:** Wie in den Hidden-Layers werden hier die Ausgaben dieser verarbeitet, um die endgültigen Ausgaben des Netzwerks zu erhalten.²

```
1 for (int i = 0; i < numOutputs; i++)
2 {
3     double activation = outputLayerBias[i];
4     for (int j = 0; j < numHiddenNodes; j++)
5     {
6         activation += hiddenLayer[j] * outputWeights[j][i
7     ]
8     }
9     outputLayer[i] = sigmoid(activation);
10 }
```

Listing 2: forwardpass output

Dabei wurden drei Versionen des ForwardPass implementiert und unterscheiden sich wie folgt:

sequential: Die Berechnungen in einer einzelnen Schleife ohne Parallelisierung wird durchgeführt und dient als Basis für parallel und simd.

parallel: Nutzt OpenMP, um die Schleifen, die über die Neuronen iterieren, auf mehrere Threads zu verteilen. Die Threads werden auf den einzelnen Kernen der CPU verteilt und beschleunigen so die Berechnung.³

```
1 #pragma omp parallel for schedule(static)
2 for (int i = 0; i < numHiddenNodes; i++)
3 {
4     double activation = hiddenLayerBias[i];
5     for (int j = 0; j < numInputs; j++)
6     {
7         activation += inputs[j] * hiddenWeights[j][i];
8     }
9     hiddenLayer[i] = sigmoid(activation);
10 }
```

Listing 3: forwardpass parallel

simd: Die OpenMP-Direktiven `parallel for` und `simd` werden kombiniert. Dies ermöglicht sowohl die Parallelisierung auf Thread-Ebene als auch die Nutzung von Vektorisierungsfähigkeiten moderner Prozessoren, um mehrere Datenpunkte gleichzeitig zu verarbeiten.⁴


```

1 #pragma omp parallel for simd schedule(static)
2   for (int i = 0; i < numHiddenNodes; i++)
3   {
4       double activation = hiddenLayerBias[i];
5   #pragma omp simd
6       for (int j = 0; j < numInputs; j++)
7       {
8           activation += inputs[j] * hiddenWeights[j][i];
9       }
10      hiddenLayer[i] = sigmoid(activation);
11  }

```

Listing 4: forwardpass simd

III Backpropagation

Die Backpropagation aktualisiert die Gewichte und Bias des Netzwerks basierend auf dem Fehler zwischen der vorhergesagten und der tatsächlichen Ausgabe. Dies erfolgt durch:

1. **Berechnung der Fehler in der Ausgabeschicht:** Der Fehler wird als Differenz zwischen dem Zielwert und der tatsächlichen Ausgabe berechnet.⁵

```

1 for (int i = 0; i < numOutputs; i++)
2 {
3     double error = target[i] - outputLayer[i];
4     deltaOutput[i] = error * dSigmoid(outputLayer[i]);
5 }
6

```

Listing 5: backpropagation outputerr

2. **Berechnung der Fehler in den Hidden-Layers:** Der Fehler wird rückwärts durch das Netzwerk propagiert, wobei die Gewichte und die Ableitung der Aktivierungsfunktion berücksichtigt werden.⁶

```

1 for (int i = 0; i < numHiddenNodes; i++)
2 {
3     double error = 0.0;
4     for (int j = 0; j < numOutputs; j++)
5     {
6         error += deltaOutput[j] * outputWeights[i][j];
7     }
8     deltaHidden[i] = error * dSigmoid(hiddenLayer[i]);
9 }
10

```

Listing 6: backpropagation deltaHidden

3. **Aktualisierung der Gewichte und Bias:** Basierend auf den berechneten Fehlern werden die Gewichte und Bias angepasst, um den Gesamtfehler zu minimieren.⁷

```
1 for (int i = 0; i < numOutputs; i++)
2 {
3     outputLayerBias[i] += deltaOutput[i] * lr;
4     for (int j = 0; j < numHiddenNodes; j++)
5     {
6         outputWeights[j][i] += hiddenLayer[j] *
        deltaOutput[i] * lr;
7     }
8 }
9 for (int i = 0; i < numHiddenNodes; i++)
10 {
11     hiddenLayerBias[i] += deltaHidden[i] * lr;
12     for (int j = 0; j < numInputs; j++)
13     {
14         hiddenWeights[j][i] += inputs[j] * deltaHidden[i]
        * lr;
15     }
16 }
17
```

Listing 7: backpropagation weights

Wie auch bei dem forwardpass wurden hier ebenfalls sequenzielle und parallele Versionen implementiert, die analog zur Forwardpropagation funktionieren.

3.5 mpt_nn_utility

Die Datei `mpt_nn_utility.c` enthält verschiedene Hilfsfunktionen, wie z.B. das Laden der MNIST-Datensätze, die das neuronale Netzwerk unterstützen.

I load_mnist

Die Funktion öffnet die entsprechenden Dateien für Bilder und Labels, liest die Daten ein und normalisiert die Pixelwerte. Die Labels werden in One-Hot-Vektoren umgewandelt, um sie für das Training zu nutzen.

II initialize_weights(), initialize_bias()

Initialisierung der Gewichte und Bias auf kleine Zufallswerte. Dies ist wichtig, um Symmetrien im Netzwerk zu aufzubrechen und den Lernprozess zu ermöglichen.

III `apply_dropout()`

Setzt bestimmte Neuronen mit einer festgelegten Wahrscheinlichkeit auf Null. Dies dient der Regularisierung und hilft ein Overfitting zu verhindern.

IV `visualize_mnist_digit()`

Ermöglicht die ASCII-Visualisierung der MNIST-Bilder. Dies dient dazu, die Daten zu überprüfen und sicherzustellen, dass diese korrekt geladen wurden. Außerdem bietet sie eine Möglichkeit den Lehrnvorgang des `mpt_nn` visuell nachverfolgen zu können

V `print_options()`

Gibt eine Liste der verfügbaren `command line` Parameter mit dazugehöriger Info auf dem Terminal aus.

3.6 `main`

Die `main.c`-Datei implementiert die Nutzung von `command line` Parametern und steuert hierdurch den Ablauf des Trainingsprozesses.

I Verarbeitung der `command line` Parameter

Durch die Verwendung von `getopt_long()` ist es den Nutzern möglich, das `mpt_nn` beim starten nach belieben zu konfigurieren. Der Nutzer kann Parameter wie den Modus (sequenziell, parallel, SIMD), die Anzahl der Epochen, die Lernrate und diverse andere Einstellungen anpassen. Eine vollständige Auflistung aller `command line`-Parameter kann der im Gitlab hinterlegten `ReadMe` entnommen werden. Es ist wichtig zu beachten, dass das `mpt_nn` gewisse Parameter voraussetzt um es starten zu können.

`-D`, startet das Netzwerk unter festgelegten Defaultwerten

`-t, -m, -i, -h, -o, -e, -l` bildet ein Netzwerk mit gewünschter Parametrisierung.

II Initialisierung

Nach dem Einlesen der Parameter werden die Datenstrukturen für das Netzwerk initialisiert. Dies umfasst die Allokation von Speicher für die Gewichte, Bias und Datenarrays.

III Training des Netzwerks

Das Training erfolgt über eine festgelegte Anzahl von Epochen. In jeder Epoche wird der gesamte Trainingssatz durchlaufen und für jeden Trainingssatz die in Unterabschnitt 2.3 genannten Schritte (I - III) umgesetzt.

Nach jeder Epoche werden der durchschnittliche Verlust und die Genauigkeit berechnet und ausgegeben.

IV Speicherfreigabe

Am Ende des Programms wird der belegte Speicher freigegeben, um Speicherlecks zu vermeiden.

3.7 mpt_nn_tests

Die Datei `mpt_nn_test.c` enthält Unit-Tests, die sicherstellen, dass die einzelnen Komponenten des Netzwerks korrekt funktionieren.

I test_sigmoid()

Überprüft, ob die Sigmoid-Funktion und ihre Ableitung für bekannte Eingaben die erwarteten Ausgaben liefern.

II test_initialize_weights()

Testet die Initialisierung der weights, um sicherzustellen, dass die Gewichte und Biases korrekt gesetzt werden und innerhalb des erwarteten Wertebereichs liegen.

III test_forward_pass(), test_backpropagation()

Überprüft, ob die Berechnungen korrekt durchgeführt werden und die Ausgaben plausibel sind. Dies funktioniert für jeweils alle drei Implementierungen des Forwardpass und Backpropagation gleich.

4 Ergebnisse

In diesem Abschnitt werden die Ergebnisse, welche aus diskreten Parametern erzielt wurden, näher beschrieben und anhand visueller Ausgaben durch das Plottool *R* vorgestellt. Dabei wird der Seed zur Erzeugung der Daten detailliert beschrieben sowie die durch Tools wie *Hyperfine* ermittelten Benchmarks genutzt. Die Ergebnisse der unterschiedlichen Algorithmen sowie das zur Ermittlung der Daten genutzte System können der Abbildung 3 entnommen werden.

4.1 Erste Testläufe

Nach der erfolgreichen Implementierung des Codes wurden erste Testläufe anhand diskreter Daten, wie sie in der nachfolgenden Tabelle aufgestellt sind, durchgeführt. Primär ging es bei diesem Testlauf darum zu prüfen, ob der Algorithmus lernfähig ist und ob sich dessen Leistung mit zunehmender Anzahl an Epochen verbessert.

Komponente	Details
Mode	parallel
Input Layer	784 Neuronen
Output Layer	10 Neuronen (Ziffern von 0 bis 9)
Anzahl Hidden Layers	2
Hidden Layer	128 Neuronen
Epochs	10
Learning Rate	0.1
Dropout Rate	0.1
Training Sets	60000

Tabelle 3: Genutzte Parameter des Testlaufs

Für den Testlauf wurde der Algorithmus in der parallelen Verarbeitungsweise ausgeführt. Diese ermöglicht die gleichzeitige Verarbeitung mehrerer Threads, was zu einer beschleunigten Berechnung führt. Im **Input-Layer** wurden Daten eines 28x28-Pixel-Bildes verwendet, was 784 Eingabeneuronen entspricht. Die Daten basieren auf dem in Unterabschnitt 3.1 beschriebenen MNIST-Datensatz.

Der *Output-Layer* besteht aus 10 Neuronen, die jeweils für eine Ziffer von 0 bis 9 stehen. Der Algorithmus bewertet, ob die erkannte Zahl korrekt klassifiziert wurde, indem das Ergebnis mit den bekannten Werten des Datensatzes verglichen wird.

In der Netzwerkstruktur sind zwei *Hidden Layer* implementiert. Diese *Feedforward-Struktur* ermöglicht dem Netzwerk die komplexe Umsetzung der Lernmethoden. Diese auf der Basis von 2^n potenzierte, absteigende Anzahl an Neuronen ermöglicht eine einfachere Abstraktion des Netzwerks und unterstützt die hardwaregekoppelte Umsetzung der Algorithmen. Zudem soll durch die schrittweise Reduktion der Neuronenanzahl eine eventuelle Überanpassung des Netzwerks vermieden werden.

Das Training wurde über 10 Epochen durchgeführt. Jede Epoche umfasst einen vollständigen Durchlauf des Netzwerks über den Trainingsdatensatz. Die *Lernrate* wurde auf 0,1 gesetzt, um sicherzustellen, dass das Netzwerk moderate Anpassungen an den Gewichten vornimmt. Zusätzlich wurde eine *Dropout-Rate* von 0,1 verwendet, um eine Überanpassung (*Overfitting*) zu vermeiden, indem einige Neuronen während des Trainings zufällig deaktiviert wurden.

Nach 10 Epochen wurde der Testlauf abgeschlossen, und die Ergebnisse sind in Tabelle 4 aufgeführt.

Epoch [x/10]	Loss	Accuracy	Hits[n/60000]
1	0.197099	88.51%	53106
2	0.115641	93.46%	56074
3	0.095635	94.54%	56726
4	0.083982	95.28%	57167
5	0.077033	95.73%	57435
6	0.071206	96.15%	57690
7	0.066532	96.40%	57838
8	0.063757	96.62%	57975
9	0.060518	96.74%	58043
10	0.058485	96.92%	58151

Tabelle 4: Ergebnisse der Trainings-Epochen

Der Testlauf wurde nach der Implementierung des Codes mehrfach wiederholt, um die Auswirkungen skriptseitiger Anpassungen nachzuvollziehen. Die hier präsentierten Ergebnisse spiegeln den finalen Stand der Optimierungen wider. Eine Genauigkeit von 96,92% weist auf eine ausgewogene Parametrisierung hin, jedoch auch auf eine sehr effiziente Lernmethodik der Backpropagation nach 10 Epochen.

4.2 Benchmarkanalyse mit Hyperfine

Für eine detaillierte Analyse der Benchmarks wurde das Programm *Hyperfine* verwendet. Hyperfine ist ein Benchmarking-Tool, das speziell dafür entwickelt wurde, die Ausführungszeiten von Befehlen präzise zu messen. Es ermöglicht dabei die Durchführung mehrerer Durchläufe und der Option von Vorlaufzyklen, um verlässliche Ergebnisse zu gewährleisten.

Für die gewählten Algorithmen - sequentiell, parallel und SIMD - konnten im Makefile einige Voreinstellungen auf Grundlage der Hyperfine-Vorgaben gesetzt werden. Wie in Abbildung 2 zu sehen ist, wurden die gleichen Parameter verwendet wie bereits in Tabelle 3. Der Unterschied besteht darin, dass Hyperfine nach einem Warmup von 3 Messzyklen mit dem eigentlichen Test beginnt.

```
$ hyperfine --warmup 3 --show-output \  
  './out/mpt_nn -m1 -t60000 -i784 -h128 -o10 -e10 -10.01 -d0.1' \  
  './out/mpt_nn -m2 -t60000 -i784 -h128 -o10 -e10 -10.01 -d0.1' \  
  './out/mpt_nn -m3 -t60000 -i784 -h128 -o10 -e10 -10.01 -d0.1' \  
  --export-markdown benchmarks/benchmark_results.md
```

Abbildung 2: Makefile Hyperfineskript

Die Ergebnisse wurden anschließend unter nachfolgendem Namen im Pfad

`benchmarks/benchmark_results.md`

im Markdown-Format abgespeichert und konnten mit einem eigens geschriebenen Skript in R visualisiert werden. Die Ergebnisse der abgespeicherten Testdaten aus Hyperfine sind in der Grafik Abbildung 3 dargestellt.

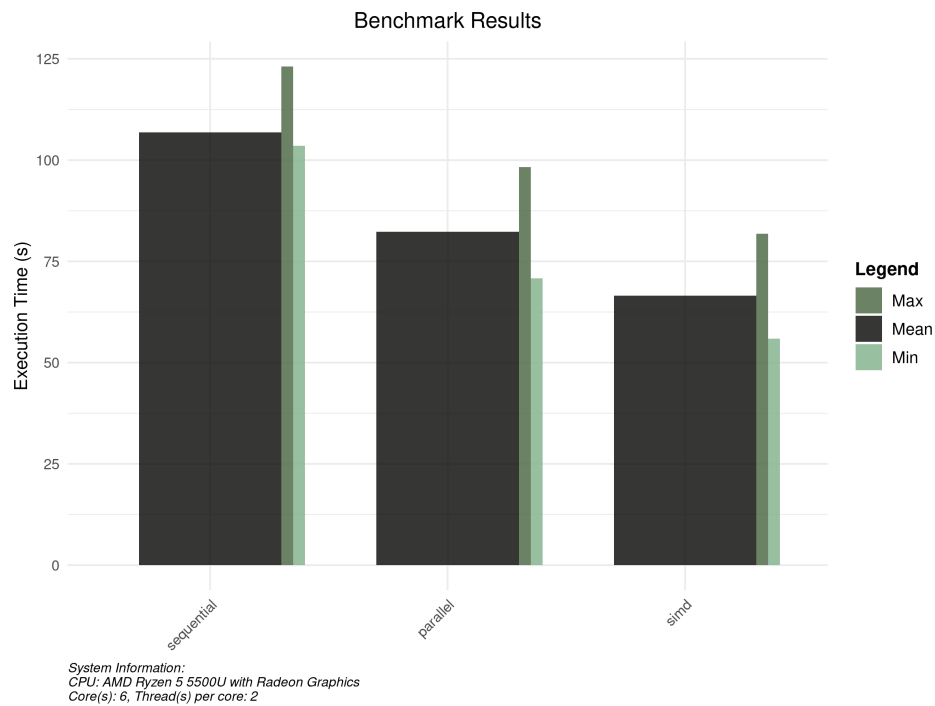


Abbildung 3: Benchmarkergebnisse mit Hyperfine

4.3 Lernfortschritt und Vorhersagbarkeit

Neben der Auswertung der Benchmarks mit *Hyperfine* wurde auch die Genauigkeit der einzelnen Algorithmen über die Epochen hinweg analysiert und dokumentiert. Anschließend konnten die erfassten Daten mithilfe eines separaten R-Skripts visualisiert werden. Die grafische Darstellung dieser Ergebnisse ist in Abbildung 4 zu sehen. Diese Visualisierung ermöglicht es, den Lernfortschritt der Algorithmen sowie deren Anpassung der Gewichte im Laufe des Trainingsprozesses detailliert zu untersuchen.

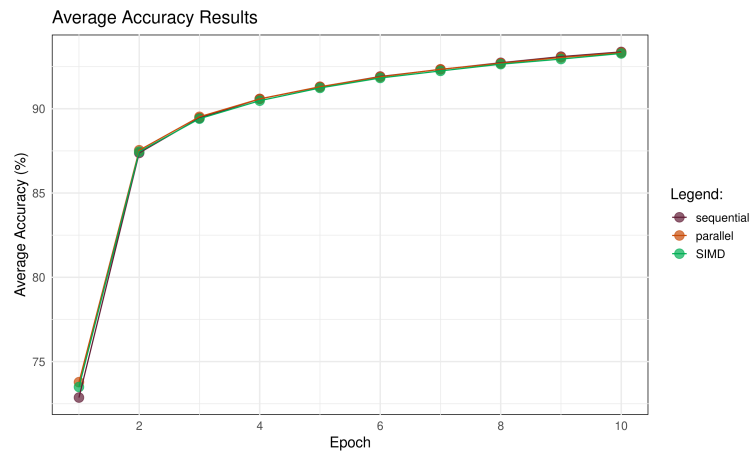


Abbildung 4: Mittelwerte der Genauigkeiten

Zu sehen ist die Genauigkeit der einzelnen Algorithmen über den zeitlichen Verlauf der Epochen ihrer Trainingsphasen. Während der Trainingsphase wurden für jeden Algorithmus zehn unabhängige Testzyklen durchlaufen. Anschließend wurden die Ergebnisse gemittelt, um potenzielle Ausreißer zu minimieren. Abbildung 5 hingegen zeigt die absoluten Messergebnisse, ohne mathematische Glättung.

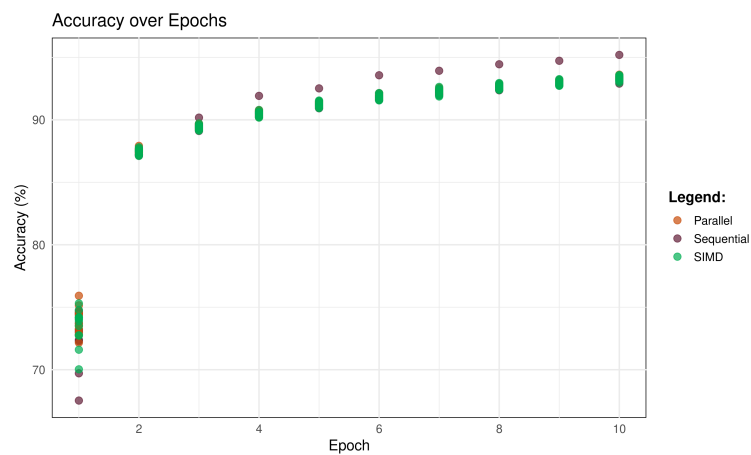


Abbildung 5: Genauigkeiten der einzelnen Algorithmen

5 Auswertung

Der vorletzte Abschnitt dieser Arbeit widmet sich der Analyse und Interpretation der erzielten Resultate. Die in Abschnitt 4 gewonnenen Ergebnisse werden im Hinblick auf ihren Inhalt und ihre Aussagekraft untersucht und reflektiert. Im Rahmen dieser Auswertung werden sowohl die durchgeführten Benchmark-Tests als auch die Genauigkeitsanalysen herangezogen, um die Leistungsfähigkeit der verschiedenen Algorithmen zu bewerten.

Anhand der Resultate lassen sich die Stärken und Schwächen der implementierten Methoden herausarbeiten. Darauf aufbauend werden mögliche Ansätze zur Optimierung und Verbesserung der Algorithmen vorgestellt und diskutiert.

5.1 Benchmark

In Abbildung 3 sind die Ausführungszeiten der verschiedenen Algorithmen dargestellt. Die Tests wurden auf einem Lenovo Laptop mit einem *AMD Ryzen 5 5500U* Prozessor durchgeführt. Während der Testläufe wurde der Computer nicht durch zusätzliche Programme belastet.

Aus der Grafik wird deutlich, dass sowohl die parallele als auch die SIMD-Implementierung zu einer signifikanten Reduktion der Trainingszeiten führen. Die Grafik zeigt sowohl die maximale und minimale Trainingszeit als auch deren Mittelwert. Die konkreten Ausführungszeiten, absteigend nach ihren maximalen Werten sortiert, lauten wie folgt:

(I)	Sequentiell:	109,24 s
(II)	Parallel:	82,35 s
(III)	SIMD:	66,71 s

Insbesondere bei der sequentiellen Ausführung ist eine signifikante Abweichung zwischen der maximalen und der minimalen Trainingszeit zu beobachten. Diese Abweichungen treten bei den parallelen und SIMD-Implementierungen deutlich geringer auf. Da alle Methoden jedoch durch **Hyperfine** nacheinander und unter der gleichen Bedingung ausgeführt wurden, deutet dies darauf hin, dass systembedingte Schwankungen, wie etwa thermische Drosselung (*Throttling*) aufgrund von Überhitzung der CPU, als Ursache ausgeschlossen werden können. Cache-Misses oder Task-Scheduling was ebenfalls zu Verzögerungen durch ausgelagerte Prozesse führen kann, können für die sequentielle Ausführung ebenfalls ausgeschlossen werden, da solch ein Verhalten eher dem parallelen- oder SIMD-Algorithmus zugeschrieben werden könnte.

Auch Fehler im Datensatz können ausgeschlossen werden, da der MNIST-Datensatz verwendet wurde, welcher zu den bekanntesten Trainingsdatensätzen gehört und in allen Algorithmen gleichmäßig sowie mit der gleichen Testhäufigkeit verarbeitet wurde.

Die wahrscheinlichste Ursache für die signifikanten Abweichungen in der sequentiellen Ausführung liegt daher vermutlich im Algorithmus selbst. Wie auch bei den anderen Methoden besteht ein hoher Grad an Variabilität aufgrund unterschiedlicher Initialisierungszustände. Allerdings zeigt die sequentielle Verarbeitung eine deutlich langsamere Konvergenz zwischen den einzelnen Schichten des neuronalen Netzes im Vergleich zu den parallelen und SIMD-Methoden. Diese Variabilität wird jedoch im Verlauf des Trainings effizient reduziert, was zu einer schnelleren Bearbeitungszeit führt. Diese bleibt zwar im Vergleich zur SIMD-Verarbeitung fast doppelt so hoch, ist jedoch deutlich schneller als im anfänglichen, zufällig gewichteten Zustand.

Abschließend zeigt die Analyse, dass die Nutzung von Parallelisierungs- und SIMD-Methoden zur Optimierung der Gewichte des neuronalen Netzes eine äußerst effiziente Technik darstellt.

5.2 Genauigkeit

Abbildung 4 und Abbildung 5 bieten einen differenzierten Einblick in die Lernmethodik der Algorithmen. Im Gegensatz zur Trainingszeit und deren Optimierung wird hier die Genauigkeit der neuronalen Identifizierung der Bilddaten über einen Lernzeitraum von 10 Epochen dargestellt.

Wie aus Abbildung 4 ersichtlich, erreichen alle drei Methoden in der letzten Epoche eine ähnliche Genauigkeit von etwa 96 %. Obwohl die sequentielle Methode zu Beginn marginal hinter den parallelen und SIMD-Methoden liegt, übertrifft sie diese am Ende der Trainingsphase um wenige Bruchteile.

Betrachtet man zusätzlich Abbildung 5, so lassen sich signifikante Schwankungen bei der SIMD-Methode in der ersten Epoche beobachten. Dieser Trend setzt sich bis zur zehnten Epoche fort, wobei auch die parallele Methode ein vergleichbares Verhalten zeigt.

Über den gesamten Verlauf hinweg übertrifft einzig die sequentielle Methode beide anderen Ansätze. Da jedoch, wie in Abbildung 4 dargestellt, der Trendverlauf für alle Methoden ähnlich ist, scheinen die Schwankungen bei der sequentiellen Methode über den gesamten Zeitraum betrachtet dennoch größer zu sein als bei den verbleibenden Algorithmen.

Der Umstand, dass alle drei Methoden eine ähnliche Genauigkeit erreichen, lässt sich dadurch erklären, dass sie denselben Datensatz verwenden. Obwohl die Methoden hinsichtlich der Berechnungszeit Unterschiede aufweisen, liegt der Fokus des neuronalen Netzes auf der Optimierung der Endgenauigkeit.

Dadurch kann ein sehr ähnliches Endresultat erzielt werden.

Die größeren Schwankungen bei den Methoden Parallel und SIMD sowie die daraus resultierende geringfügig niedrigere Genauigkeit sind auf die jeweilige Art der Verarbeitung zurückzuführen. Bei der parallelen Berechnung werden mehrere Threads genutzt, um die Aufgaben gleichzeitig auszuführen. Dies kann bei der anschließenden Zusammenführung der Ergebnisse zu Synchronisierungsproblemen führen.

Die SIMD-Methode verarbeitet mehrere Daten simultan auf Vektorebene. Dadurch kann es insbesondere in den ersten Epochen zu einer erhöhten Variabilität in der Genauigkeit kommen.

5.3 Ansätze zur Verbesserung

Die vorliegenden Ergebnisse könnten durch verschiedene Ansätze verbessert werden. Eine Möglichkeit liegt in der Wahl der verwendeten Aktivierungsfunktionen. In allen drei Methoden wurde die Sigmoid-Funktion eingesetzt. Wie bereits in Tabelle 2 erwähnt, existieren jedoch weitere, effizientere Aktivierungsfunktionen. Funktionen wie ReLU oder Softmax können schnellere und deutlich effektivere Konvergenzen während des Trainingsprozesses ermöglichen.

Der Einsatz von ReLU in den Hidden Layers könnte ebenfalls zu einer Steigerung der Genauigkeit führen. Dies ließe sich durch eine verbesserte Lernkapazität und -geschwindigkeit erreichen. Die Softmax-Funktion wird häufig im Output-Layer von Netzwerken verwendet, insbesondere in Szenarien, bei denen mehrere Klassen vorhergesagt werden.

Zusätzlich wurde durch den Einsatz von *helgrind* festgestellt, dass insbesondere zu Beginn des Trainingsprozesses ein hohes Maß an *Race-Conditions* auftrat. Durch Anpassungen im Code konnte dieses Problem teilweise reguliert werden, jedoch stellt die Funktion `forward_pass_parallel` weiterhin eine Herausforderung dar.

Weiterhin könnte die Forward- und Backpropagation durch eine stärkere Parallelisierung der Summierungen in den Hidden- und Output-Layers optimiert werden.

6 Fazit

Abschließend lässt sich für das Projekt im Fach Mikroprozessortechnik festhalten, dass alle Ziele der Aufgabenstellung erreicht wurden. Darüber hinaus wurden eigene Ziele und Verbesserungsmöglichkeiten implementiert und ein lernfähiges neuronales Netzwerk erfolgreich entwickelt.

Zu Beginn stellte vor allem die theoretische Erarbeitung der mathematischen Herleitungen eine größere Herausforderung dar. Mit fortschreitender Bearbeitung rückte dieses Problem jedoch zunehmend in den Hintergrund und die mathematischen Zusammenhänge wurden gegen Ende des Projekts klarer und verständlicher.

Die Implementierung der Ziele in der Programmiersprache C erwies sich ebenfalls zunächst als anspruchsvoll. Zwar existierten einige hilfreiche Foren und Artikel im Internet, viele Funktionen mussten jedoch an unsere spezifischen Anforderungen angepasst werden.

Der Einsatz des MNIST-Datensatzes als Lernquelle für unser neuronales Netzwerk stellte sich durch die Unterstützung und Hinweise unserer betreuenden Professoren zur Bildverarbeitung und Datenextraktion als sehr valide Grundlage heraus.

Im Verlauf des Projekts konnten wir unsere Programmierfähigkeiten in C deutlich verbessern, ein umfassendes Verständnis für die Thematik neuronaler Netze entwickeln sowie unsere Kenntnisse in der Dokumentation mittels L^AT_EX vertiefen.

Literatur

- [1] David Kriesel. Ein kleiner überblick über neuronale netze. *Download unter <http://www.dkriesel.com/index.php>*, 2007.
- [2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- [3] Jeslur Rahman. Simply understanding artificial neural networks (ann) & deep learning, 2023. Zugriff am 04.09.2024.
- [4] Günter Daniel Rey and Karl F Wender. *Neuronale netze: Eine einföhrung in die grundlagen, anwendungen und datenauswertung*. Hogrefe AG, 2019.
- [5] TensorFlow. MNIST dataset in TensorFlow. <https://www.tensorflow.org/datasets/catalog/mnist>, 2015.