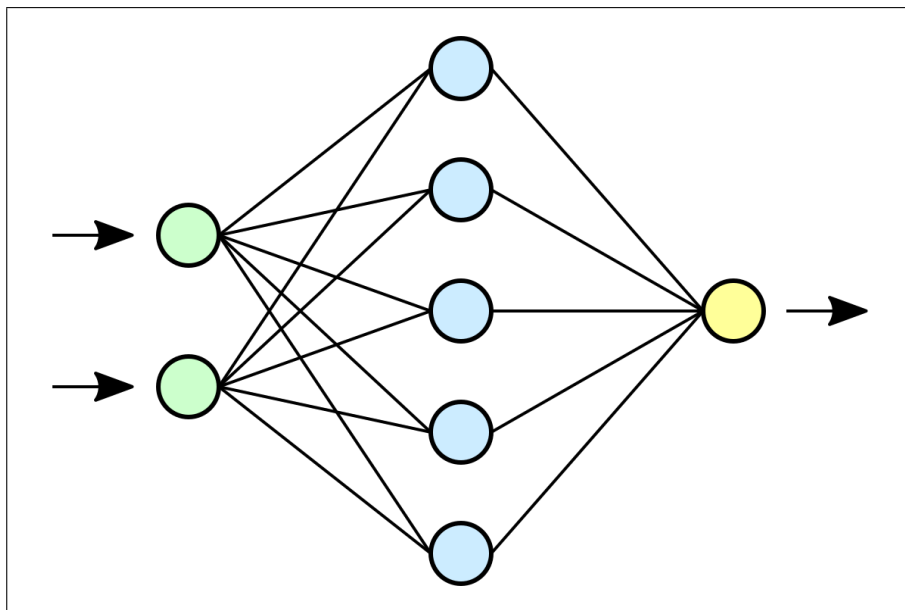


Computer Engineering

# Mikroprozessortechnik Beleg



[https://de.wikipedia.org/wiki/K%C3%BCnstliches\\_neuronales\\_Netz#/media/Datei:Neural\\_network.svg](https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz#/media/Datei:Neural_network.svg)

Neurales Netzwerk

Hai Nam La [s0578105]

Minh David Nguyen [s0579234]

Viet Anh Kopietz [s0574258]

Dozent: Prof. Dr. Sebastian Bauer

24.Juli 2022

# Inhaltsverzeichnis

<b>1</b>	<b>Vorwort</b>	<b>3</b>
<b>2</b>	<b>Problemaufstellung und Theorie</b>	<b>3</b>
2.1	Aufgabenstellung	3
2.2	Künstliches Neuron	4
2.3	Künstliches neurales Netzwerk	4
2.4	Anwendung	4
<b>3</b>	<b>Mathematische Herleitung</b>	<b>5</b>
<b>4</b>	<b>Parallelisierung und Optimierung eines Neuralen Netzwerks</b>	<b>7</b>
4.1	Parallelisierung	7
4.2	Optimierung	9
<b>5</b>	<b>Implementierung und Lösung</b>	<b>9</b>
5.1	Neurales Netzwerk mit MNIST	9
5.2	Make und Anpassbare Hiddenlayergröße	12
5.3	Parallelisierung	13
5.4	Optimierung	15
5.5	Valgrind	16
5.6	Gitlab-Runner	17
<b>6</b>	<b>Auswertung</b>	<b>18</b>
<b>7</b>	<b>Schwierigkeiten und Aufgabenverteilung</b>	<b>20</b>
	<b>Abbildungsverzeichnis</b>	<b>22</b>
	<b>Literatur</b>	<b>23</b>

# 1 Vorwort

Alle in diesem Beleg aufgeführten Testergebnisse wurden mit einer Ryzen 5 2600 CPU mit 6 Kernen und 12 Threads ausgeführt.

## 2 Problemaufstellung und Theorie

Text

### 2.1 Aufgabenstellung

Das Ziel des Projekts ist es die Inhalte der Vorlesung und Laborübungen, in Gruppen am Beispiel von neuronalen Netzen und das Lernen dieser anzuwenden, zu vertiefen und aufzubereiten. Hierfür sollte man ein künstliches neuronales Netzwerk implementieren, welches Zahlen und Buchstaben oder Objekte in Pixelbilder erkennen soll. Als Voraussetzung war die Programmiersprache C oder C++ und sollte betriebssystemunabhängig sein. Das heißt mit gcc übersetzbar und mit make, cmake oder ähnlichem über die Kommandozeile bedienbar sein. Eine weitere Aufgabe war es, das maschinelle Lernen auf 2 verschiedene Arten und Weisen zu parallelisieren. Für das neuronale Netz war eine vorgefertigte Bibliothek nicht erlaubt. Als Schnittstellen für die Parallelisierung konnte man OpenMP, welches wir am Anfang des Semesters in Laborübungen verwendet haben, OpenMPI, OpenCL oder ähnliches verwenden. SIMD-Instruktionen, um die Compiler-Optionen anzupassen und deren Auswirkungen zu untersuchen, konnte man ebenfalls nutzen. Man sollte dann die Parallelisierungen und deren Speedup des Lernverfahrens vergleichen. Die Quelltexte sollen in einem Projekt-eigenen Git-Repository auf der GitLab-Instanz der HTW entwickelt werden und die Implementation sollte auf race-conditions mithilfe von valgrind und helgrind überprüft werden.

## 2.2 Künstliches Neuron

Ein künstliches Neuron ist im Grunde eine Funktion, die eine Menge reeller Zahlen nimmt und sie auf einen reellen Output anpasst. Jeder Input hat ein bestimmtes Gewicht, wobei ein Gewicht ein Wert ist, welcher mit dem Input multipliziert wird. Die Inputs werden mithilfe der Übertragungsfunktion zusammenaddiert und optional mit einem Schwellwert verrechnet. Das Ergebnis wird dann der Aktivierungsfunktion übergeben und gibt am Ende einen Output aus. Die Aktivierungsfunktion kann eine beliebige mathematische Funktion sein, beispielsweise: linear oder logistisch.

Abbildung

## 2.3 Künstliches neuronales Netzwerk

Ein künstliches neuronales Netzwerk besteht aus vernetzten Neuronen. Hierbei ist die Topologie der Neuronen abhängig von der Anwendung des Netzes. Darauf folgt die Trainingsphase des Netzes. Es gibt verschiedene Arten, wie das Netz lernen kann. Wie zum Beispiel durch Entwicklung oder Löschung von Verbindungen, Änderung der Gewichte, Anpassung des Bias bzw. Schwellenwerts, hinzufügen oder löschen von Neuronen und zu guter Letzt die Änderung von Aktivierungs-, Propagierungs-, oder Ausgabefunktion. Das häufigste verwendete Netz für Zahlenerkennung ist das mehrschichtige feedforward-Netz. Hierbei sind die Kanten gerichtet, das heißt, dass das Netz nur in eine Richtung geht. Das Netz hat ein Input, Output und mehrere sogenannte „hidden layer“ also verdeckte Schichten. Diese Schichten verbessern die Abstraktion und können als mehrschichtiges Perzeptron das XOR-Problem optimal lösen.

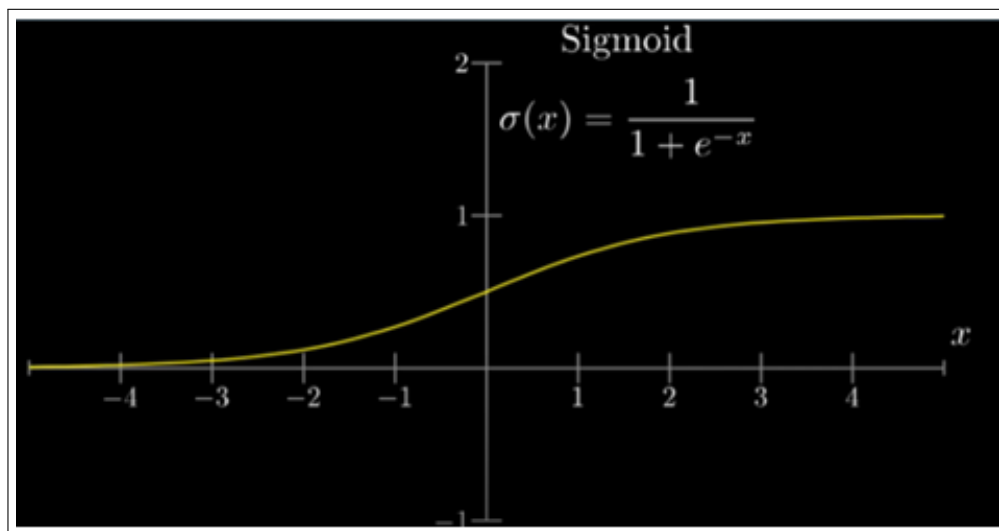
## 2.4 Anwendung

Das künstliche neuronale Netzwerk hat besondere Eigenschaften, welches es bei Anwendungen besonders interessant macht. Es braucht zum Beispiel kein oder nur geringes synthetisches Wissen über das zu lösende Problem, um diese mit Lernverfahren zu lösen. Häufige Anwendungen für künstliche neuronale Netzwerke sind: Frühwarnsysteme, Fehlererkennung,

Optimierung, Bildverarbeitung und Musterkennung, künstliche Intelligenz und mehr.

### 3 Mathematische Herleitung

Um das neuronale Netzwerk lernen zu lassen, wird das feedforward verwendet um die Daten durch das Netzwerk zu verarbeiten. Das eigentliche Lernverfahren ist die Backpropagation, worauf später im Detail eingegangen wird. Das feedforward „füttert“ das Netzwerk sagesagt mit den Daten. Bevor die Daten allerdings vom Inputlayer zum Outputlayer gelangen, werden die Daten erst in das nächste Layer gegeben. Dabei wird die Summe aller Gewichte der Neuronen des derweiligen Layers multipliziert mit den Werten des derweiligen Layers jeweils an das erste Neuron des nächsten Layers weitergegeben. Das geschieht dann für alle Neuronen des nächsten Layers und wiederholt sich dann vom Inputlayer bis zum Outputlayer. Dabei wird jede Summe von der Sigmoidfunktion verarbeitet. Die Sigmoidfunktion ist dafür da, um die Werte aus der Summe zu komprimieren. Das ist wichtig, weil gewollte Werte nur zwischen 0 und 1 liegen sollten um vom neuronalen Netzwerk verarbeitet werden zu können.



<https://www.youtube.com/watch?v=aircAruvnKk> timestamp: 10:35

Abbildung 1: Sigmoidfunktionskurve<sup>1</sup>.

Das  $x$  ist hierbei die Summe aus den Gewichten  $weight$  multipliziert mit den Werten der Input-Neuronen  $inputVals$ .  $x = \sum_{n=0}^{sizeofNeurons} (inputVals * weight)$

$$x = \sum_{n=0}^{sizeofNeurons} (inputVals * weight)$$

Je kleiner das Ergebnis der Summe  $x$  ist, desto näher läuft sie gegen 0 und je höher das Ergebnis  $x$  ist, desto näher läuft  $x$  gegen 1. Durch das Komprimieren der Werte, kann die Information an die Neuronen der nächsten Layer weitergegeben werden. Dabei gilt: je näher die Zahl an der 1 ist, desto wichtiger ist das Gewicht und das dazugehörige Neuron und je näher die Zahl an der 0 ist, desto weniger Bedeutung weist das Gewicht und das dazugehörige Neuron auf.

Da anfangs die Gewichte zufällig gewählt wurden, werden sehr wahrscheinlich falsche Ergebnisse, beziehungsweise falsche Prognosen, des neuronalen Netzwerks ausgegeben. Um die erwarteten Werte und Endergebnisse zu bekommen, wird nun die Backpropagation angewendet. Die Backpropagation funktioniert im Prinzip wie das feedforward, bloß rückwärts. Das heißt, dass die Werte nun von der Outputlayer bis hin zum Inputlayer durchverarbeitet werden. Dabei werden die Gewichte nicht nur wie beim feedforward zur Multiplikation angewendet, sondern auch neu angepasst, damit das richtige Ergebnis gefunden werden kann. Allerdings wird für die Anpassung der Gewichte das gradient descent verwendet.

Das Gradient descent  $g$  ergibt sich aus dem erwarteten Ergebnis  $expectedoutcome$  minus dem aktuellen Ergebnis  $outcome$  und das multipliziert mit der abgeleiteten Sigmoidfunktion des aktuellen Ergebnisses  $outcome$ .

$$g = (expectedoutcome - outcome) * \sigma'(outcome)$$

Diese Gleichung des gradient descent wird berechnet für die Neuronen im Outputlayer, denn ab dem Hiddenlayer wird dann eine etwas andere Gleichung verwendet. Sie ergibt sich aus der Summe der Gewichte  $weight$  vom momentanen Layer multipliziert mit dem gradienten vom aktuellen Ergebnis  $g_{outcome}$  und das multipliziert mit der abgeleiteten Sigmoidfunktion des aktuellen Ergebnisses  $outcome$ .

$$g = \sum_{n=0}^{sizeofNeurons} weight * g_{outcome} * \sigma'(outcome)$$

Durch das Ausrechnen der Gradienten kann jetzt die Anpassung der Gewichte stattfinden. Dafür wird eine weitere Formel verwendet. Und zwar ergibt sich das neue Gewicht  $w$  aus einer selbst ausgewählten Lernrate  $\eta$  multipliziert mit dem Ergebnis  $\phi$  multipliziert mit dem ausgerechneten Gradienten  $g$  und das addiert mit dem Momentum  $\alpha$  multipliziert mit dem alten Ergebnis  $\phi_{old}$ .

$$w = \eta * (\phi * g) + (\alpha * \phi_{old})$$

Mit dieser Formel werden nun die Gewichte der Neuronen angepasst. Das geschieht für jedes einzelne Neuron vom Outputlayer angefangen bis hin zum Inputlayer.

## 4 Parallelisierung und Optimierung eines Neuralen Networks

### 4.1 Parallelisierung

Im Grunde gibt es 2 mögliche Arten um ein Neurales Netzwerk zu parallelisieren.

1. Es wird ein neurales Netzwerk gebaut, welches in Abschnitte eingeteilt wird, wobei es so viele Abschnitte gibt, wie Threads von der CPU genutzt werden. Dabei arbeitet jedes Thread seinen Abschnitt des neuronalen Netzwerks ab und gibt sein Ergebnis an die anderen Abschnitte der anderen Threads ab.

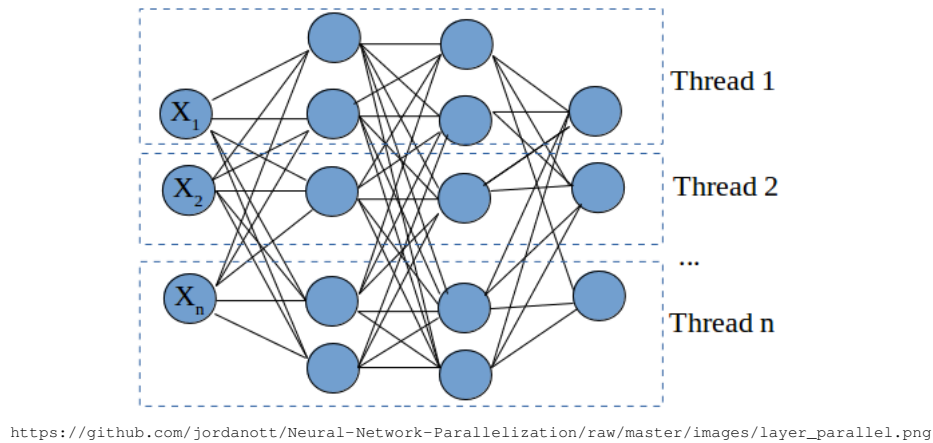


Abbildung 2: 1. Parallelitätsform<sup>2</sup>.

2. Jedes Thread erstellt sich ein eigenes neuronales Netzwerk und jedes Thread lernt mit einer bestimmten Anzahl an Trainingsdaten. Dabei werden die Trainingsdaten, je nach Anzahl an Threads, geteilt. Gibt es als Beispiel 10000 Trainingsdaten und 10 Threads, arbeitet jedes Thread mit 1000 Trainingsdaten. Am Ende werden alle Gewichte in einem globalen neuronalen Netzwerk gespeichert, wobei alle Gewichte aufsummiert werden und der Durchschnitt gebildet als Gewicht genutzt wird.

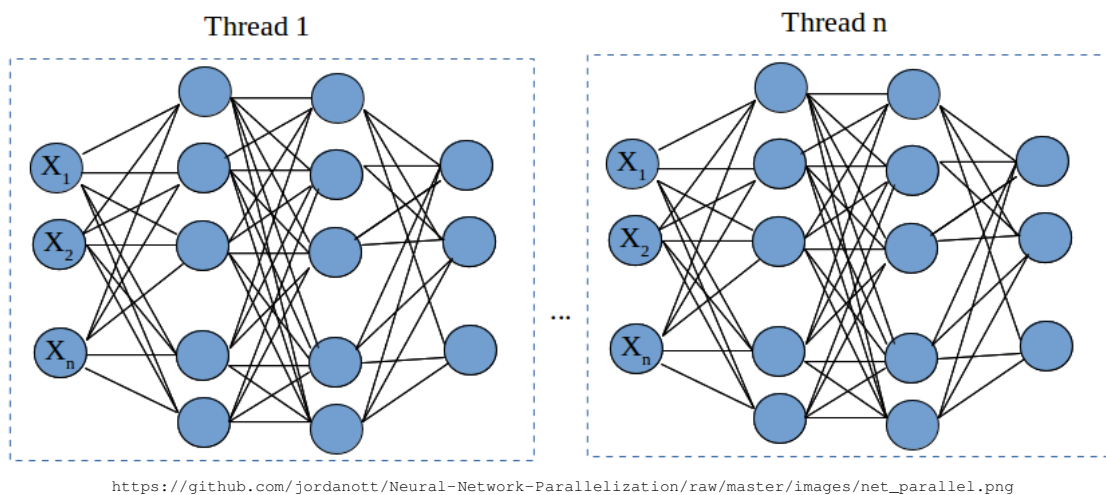


Abbildung 3: 2. Parallelitätsform<sup>3</sup>.



## 4.2 Optimierung

Es gibt viele Wege um einen Code zu optimieren. Dazu gehören sinnvolle Nutzung von Aufrufen und Funktionen, sinnvolle Datenstrukturen und vieles mehr. In unserem Projekt konzentrieren wir uns bei der Optimierung hauptsächlich auf die Nutzung von CXXFLAGS bei der Kompilierung, wobei auch ein angemessener Standard bei den anderen Fällen versucht wird zu halten. CXXFLAGS werden auch Variablen genannt, die bei der Kompilierung automatische Optimierung aktivieren können. Dabei benutzen wir die Option -OX, wobei für X eine Zahl zwischen 0 und 3 eingesetzt wird. Diese Variable aktiviert die Optimierung je nach Stufe. Die Stufe 0 gleicht dabei keiner Optimierung und 3 fast der maximalen Optimierung. Die nächst höhere Optimierungsstufe benutzt dabei alle Optimierung die von der vorherigen Optimierungsstufe verwendet wird. Es gibt auch weitere Optimierungen die aktiviert werden können sowie ftree-vectorize oder ffast-math die auch von uns genutzt werden.

## 5 Implementierung und Lösung

### 5.1 Neutrales Netzwerk mit MNIST

Zur Lösung aller erforderlichen Aufgaben, war es erst nötig ein Neutrales Netzwerk zu kodieren. Es gab viele Schwierigkeiten zur Kodierung, wobei wir im späteren Verlauf des Belegs die Probleme ansprechen werden. Wir haben letztendlich mithilfe eines Youtube-Tutorials (<https://www.youtube.com/watch?v=sK9AbJ4P8ao>) ein, auf Klassen basierendes, Neutrales Netzwerk gebaut, welches das XOR Problem erlernen konnte. Es fehlten einige Elemente, welche wir für unsere Aufgaben, wie der Erlernung von Ziffern und Buchstaben, neu implementieren mussten. Dazu haben wir zuallerst die Tangenshyperbolicus-Funktion ersetzt mit der Sigmoid-Funktion. Wir hätten auch mit der Tanh-Funktion arbeiten können aber wir haben uns entgegenentschieden, da auch viele Kommilitonen in Ihrer Präsentation eher

auf die Sigmoid-Funktion eingegangen sind. Darauf folgend haben wir die schon bestehende Klasse "TrainingData", welches die Trainingdaten einliest, anpassen müssen. Es gab erste Versuche mithilfe von Tensorflow keras die Daten des Mnist einzulesen und in C++ wiederzuverwenden, was nicht erfolgreich umgesetzt werden konnte. Wir sind dann auf eine Internetseite: <http://yann.lecun.com/exdb/mnist/> gestoßen in der die MNIST Daten in binary dargestellt wurden und auch direkt downloadbar waren. Mit der Datei konnten wir dann 3 Funktionen beschreiben: "getNextInputs(vector<double>)" welches die ersten Zeilen der binary Datei eingelesen und "zerstört" hat, da die ersten Zeilen nur Informationen über die wichtigen Daten besaßen, " getNextInputs(vector<double>)" welches genau ein Trainingsbild einliest und "getTargetOutputs(vector<double>)" welches die erwarteten Werte für das vorher eingelesene Bild einliest. Schlussendlich haben wir die verschiedenen Layer des neuronalen Netzwerks angepasst und ermöglichen es insgesamt 4 Layer zu besitzen wobei die 2 Hidden-layer über der Konsole mit "make" anpassbar sind, wie später auch beschrieben wird.

Für das neuronale Netzwerk haben wir dann wie folgt den Lernprozess implementiert:

```
for(int trainingEpoch = 1; trainingEpoch <= trainingsize; ++trainingEpoch)
{
    trainData.getNextInputs(inputVals);

    myNet.feedForward(inputVals);

    myNet.getResults(resultVals);
    trainData.getTargetOutputs(targetVals);

    myNet.backProp(targetVals);
}
```

Abbildung 4: main for-loop<sup>4</sup>.

Dabei wird in der For-Schleife von 1 bis trainingsize=60000 Epochen gelernt, welches von den 60000 gegebenen Lernbeispiele der Datei genommen wurde. Es wird zuerst das erste Lernbeispiel eingelesen(getNextInput()) und im nächsten Schritt durch das feedforwar(feedforward()) im neuronale Netzwerk verarbeitet. Dabei werden wie in Chapter 3 "Mathematische Herlei-

tung" die aufgezählten Prozesse abgeschlossen und ein vom neuronalen Netz errechnetes Ergebnis gespeichert und im nächsten Schritt durch die Backpropagation(backProp()) mit dem errechneten Ergebnis und dem erwarteten Ergebnis, welches ausgelesen wird, die Gewichte angepasst. Wir haben mit einer Trainingrate  $\eta$  von 0.035, einem Momentum  $\alpha$  von 0.9 und 2 Hiddenlayer mit jeweils 128 Neuronen gearbeitet und mit dem Testen, welches nur das feed-forward des neuronalen Netzwerkes nutzt, folgende Erfolgsrate erzielt:

Successrate: 93.86%

Abbildung 5: Erfolgsrate ohne Parallelisierung<sup>5</sup>.

Mit folgendem Recent-Error-Graph:

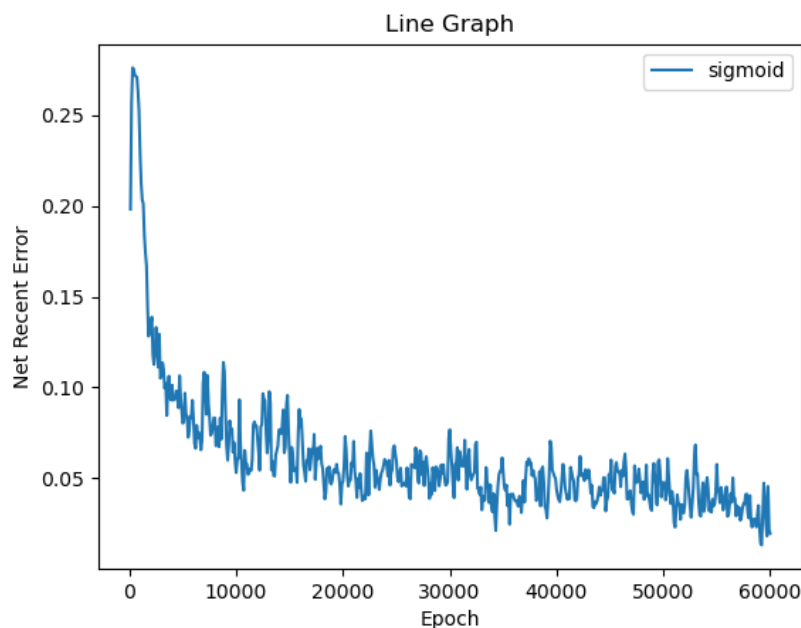


Abbildung 6: Recent Error Graph <sup>6</sup>.

Bei dem Recent-Error-Graphen kann man deutlich eine herkömmliche Lernkurve erkennen, in der anfangs durch die zufällig generierten Gewichte eine relativ gute aber zufällige recenterror

erzielt wird, die dann nach den ersten hundert Trainingsepochen sich deutlich verschlechtert und schlussendlich wieder exponentiell bis eher hyperexponentiell, durch das Lernverfahren, abnimmt. Außerdem wird folgende Trainingszeit vermerkt:

```
$ ./train1 128 128
187.838335 s
```

Abbildung 7: Zeit des Trainierens <sup>7</sup>.

Der Code für diesen Abschnitt kann wie folgt gefunden werden: [https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/tree/main/mpt\\_neural\\_network\\_ohne\\_omp](https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/tree/main/mpt_neural_network_ohne_omp)

## 5.2 Make und Anpassbare Hiddenlayergröße

Zum Ausführen beziehungsweise zum Kompilieren und Debuggen soll eine Betriebssystem(OS) übergreifende Methode verwendet werden. Anfangs haben wir mit der IDE Visual Studio Code gearbeitet, mit der auf allen OSs gearbeitet werden kann, doch durch die nicht generalisierte Beschreibung einer "quasi" Makefile haben wir uns entschlossen mit einer richtigen Makefile zu arbeiten. Dabei nutzen wir als Grundlage die Makefile, die aus der Vorlesung gegeben ist (<https://github.com/sbal/benchmarks/blob/master/ann/makefile>). Die Makefile wurde ein wenig an unser Beispiel angepasst und kann zum Beispiel mit "make benchmark", 7 verschiedene Optimierungsarten ausprobieren, wobei auch die Zeit immer ausgegeben wird und mit "make test", der Test gestartet werden, wenn einmal schon trainiert wurde. Die Makefile wurde dann jeweils für die normale Version sowie die parallelisierte Version unseres Neuralen Netzwerkes erstellt, welches im nächsten Abschnitt erläutert wird. Die Makefiles können jeweils unter

[https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/blob/main/mpt\\_neural\\_network\\_mit\\_omp/Makefile](https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/blob/main/mpt_neural_network_mit_omp/Makefile) (mit Parallelisierung)

und

[https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/blob/main/mpt\\_neural\\_network\\_ohne\\_omp/Makefile](https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/blob/main/mpt_neural_network_ohne_omp/Makefile) (ohne Parallelisierung) gefunden werden.

## 5.3 Parallelisierung

Die eigentliche Hauptaufgabe dieses Semesterprojekts ist es das Lernverfahren der neuronalen Netzwerke zu beschleunigen. Es wurden Parallelisierung und Optimierung gefordert. In diesem Abschnitt wird die Implementierung der Parallelisierung beschrieben. Es wird mit OpenMP, einer Programmierschnittstelle für shared-memory-Programmierung, gearbeitet. Dafür wird die library "#include <omp.h>" inkludiert und "#pragma omp parallel" genutzt. Wir haben uns dabei für die 2. Methode der Parallelisierung entschieden in der wir jedem Thread eine Kopie des neuronalen Netzwerkes geben und sie jeweils ein Teil der Trainingsdateien trainieren lassen. Dabei werden, wenn beispielsweise mit 12 Threads gearbeitet wird,

$$\frac{AnzahlAnEpochen}{ThreadAnzahl} = \frac{60000}{12}$$

also 5000 Trainingsbilder pro Thread verarbeitet. Es wurde wie folgt implementiert:

```

int main()
{
    omp_set_num_threads(12);
    #pragma omp parallel
    {
        //initialize Neural Network
        #pragma omp for
        for(int trainingEpoch = 0; trainingEpoch < trainingsize; trainingEpoch++)
        {
            //learn
        }
    }
}

```

Abbildung 8: Vereinfachte Veranschaulichung von omp<sup>8</sup>.

Dabei wird durch das "#pragma omp parallel" 12 Threads "geschaffen", die dann jeweils ein neuronales Netz erschaffen und trainieren dann durch das "#pragma omp for" jeweils die 5000 Bilder. Um die Gewichte zu speichern, zur Wiederverwendung, werden die Gewichte aller neuronalen Netzwerke die geschaffen wurden summiert und der Durchschnitt als endgültiges Gewicht des neuronalen Netzes gespeichert. Dabei wird der Codesnippet zur parallelen Speicherung nicht in die Zeitrechnung genommen, da zum einen es ein Vergleich der Zeit beider Codes im ähnlichsten Zustand ist und zum Anderen die Speicherung, so wie wir es implementiert haben, sehr Zeitaufwendig ist und die Zeit bis zu 10 Minuten verlängern kann. Es ergibt sich durch Ausführen des Tests folgende Erfolgschance, wobei eine Trainingrate  $\eta$  von 0.035, Momentum  $\alpha$  von 0.9 und 2 Hiddenlayer mit jeweils 128 Neuronen gewählt wurden:

**Successrate: 78.01%**

Abbildung 9: Erfolgsrate mit Parallelisierung<sup>9</sup>.

Vergleichsweise zu der Version ohne Parallelisierung büßen wir bei der Erfolgschance um die 20% ein. Dafür können wir eine deutliche Zeitverringerung feststellen:

```
$ ./train1 128 128
33.014517 s
```

Abbildung 10: Zeit des Trainierens mit omp<sup>10</sup>.

Wobei eine Beschleunigung von 570% besteht.

Der Code kann unter [https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/tree/main/mpt\\_neural\\_network\\_mit\\_omp](https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/tree/main/mpt_neural_network_mit_omp) gefunden werden. Außerdem wurde versucht eine 2. Parallelisierungsform zu implementieren doch mit SIMD ist es uns nicht gelungen mit unserer Klassenstruktur zu arbeiten, sowie mit CUDA.

## 5.4 Optimierung

Für die Optimierung durch Nutzung von CXXFLAGS haben wir ebenfalls die aus der Vorlesung gegebenen Optimierungen aus der Makefile genutzt (<https://github.com/sbal/benchmarks/blob/master/ann/makefile>). Dabei werden folgende CXXFLAGS genutzt und folgende Zeit vermerkt, am Beispiel unseres nicht parallelisierten neuronalen Netzwerks mit einer Trainingrate  $\eta$  von 0.035, Momentum  $\alpha$  von 0.9 und jeweils 2 Hidden-layer mit 128 Neuronen:

```
CXXFLAGS1=keine Optimierung.....
CXXFLAGS2=-O0.....
CXXFLAGS3=-Os.....
CXXFLAGS4=-O2.....
CXXFLAGS5=-O3.....
CXXFLAGS6=-O2-ftree-vectorize.....
CXXFLAGS7=-O2-ftree-vectorize-ffast-math
```

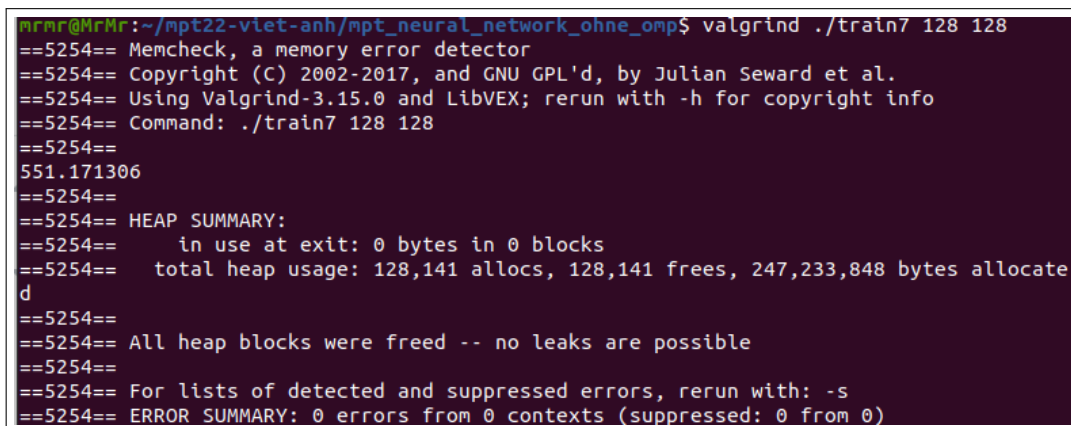
```
186.913837 s
189.206775 s
30.756480 s
28.112472 s
22.819214 s
27.863012 s
25.497112 s
```

Abbildung 11: Zeit mit Optimierung<sup>11</sup>.

Es wird erwartet, dass CXXFLAGS1 sowie CXXFLAGS2 die gleiche Zeit braucht, doch es gibt einen kleinen Unterschied von 2 Sekunden. Dabei wird beim -O0 ein Optimierungslevel von 0 benutzt, welches einer Kompilierung ohne Optimierung gleichen sollte. Mit CXXFLAGS5 wird hierbei die schnellste Kompilierungszeit erreicht, wobei -O3 auch das von uns höchstgenutzte Optimierungslevel ist. Bei der Betrachtung zwischen CXXFLAGS4 mit CXXFLAGS6 und CXXFLAGS7, welche das gleiche Optimierungslevel von -O2 benutzen, wird auch ein Geschwindigkeitszuwachs festgestellt bei der erweiterten Nutzung von -ffree-vectorize und -ffree-vectorize mit -ffast-math.

## 5.5 Valgrind

Folgend wird die Nutzung von Valgrind beschrieben. Dabei wird mit Valgrind, ein analyse tool, das memory management überprüft. Es gibt dabei keine memory leaks bei der Version ohne Parallelisierung:



```
mrnr@MrMr:~/mpt22-viet-anh/mpt_neural_network_ohne_omp$ valgrind ./train7 128 128
==5254== Memcheck, a memory error detector
==5254== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5254== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5254== Command: ./train7 128 128
==5254==
551.171306
==5254==
==5254== HEAP SUMMARY:
==5254==    in use at exit: 0 bytes in 0 blocks
==5254==   total heap usage: 128,141 allocs, 128,141 frees, 247,233,848 bytes allocated
==5254==
==5254== All heap blocks were freed -- no leaks are possible
==5254==
==5254== For lists of detected and suppressed errors, rerun with: -s
==5254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Abbildung 12: Valgrind ohne Parallelisierung<sup>12</sup>.

Jedoch gibt es memory leaks bei der parallelisierten Version:



```

mrmr@mrmr:~/mpt22-viet-anh/mpt_neural_network_mt_omp$ valgrind ./train7 128 128
==6500== Memcheck, a memory error detector
==6500== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6500== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6500== Command: ./train7 128 128
==6500==
518.002734
==6500==
==6500== HEAP SUMMARY:
==6500==     in use at exit: 7,792 bytes in 16 blocks
==6500==   total heap usage: 337,934 allocs, 337,918 frees, 530,102,792 bytes allocated
==6500==
==6500== LEAK SUMMARY:
==6500==    definitely lost: 0 bytes in 0 blocks
==6500==   indirectly lost: 0 bytes in 0 blocks
==6500==    possibly lost: 3,344 bytes in 11 blocks
==6500==   still reachable: 4,448 bytes in 5 blocks
==6500==     suppressed: 0 bytes in 0 blocks
==6500== Rerun with --leak-check=full to see details of leaked memory
==6500==
==6500== For lists of detected and suppressed errors, rerun with: -s
==6500== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Abbildung 13: Valgrind mit Parallelisierung<sup>13</sup>.

Durch kurzer Recherche sind wir darauf gestoßen, dass die library libgomp, des OpenMPs, eigene "synchronisation primitives" erstellt, welche nicht von Valgrind erkannt werden und daraus folgend memory leaks und data races, welche nicht bestehen, aufgezählt werden.

## 5.6 Gitlab-Runner

Es wurde auch für unser neurales Netz ein Gitlab-Runner errichtet, der kontinuierlich den Code kompiliert wenn es durch git gepushed wird. Doch wir haben den Gitlab-Runner erst spät angefangen zu bauen und daraus folgend auch kaum benutzt. Dennoch konnten wir es zum Laufen bringen in dem wir einen Linux-Rechner als Runner registriert haben. Der Gitlab-Runner öffnet dabei beide Versionen unseres Codes, einmal die Version ohne Parallelisierung und einmal die Version mit Parallelisierung. In beiden Fällen wird make all ausgeführt und die schnellste Optimierungsstufe ausgeführt. In der CI/CD Pipeline kann dann geprüft werden ob der Code erfolgreich ausgeführt wurde.

Der Code für den Gitlab-Runner kann folgend gefunden werden: <https://gitlab.rz.htw-berlin.de/s0574258/mpt22-viet-anh/-/blob/main/.gitlab-ci.yml>

## 6 Auswertung

Wie schon im vorherigen Chapter zu sehen kann durch Parallelisierung ein deutlicher Geschwindigkeitszuwachs festgestellt werden. Dabei verhält sich der Zuwachs ähnlich einer exponentiellen Abnahme, wie in Abbildung 14 zu sehen. Das Gleiche wird auch für verschiedene

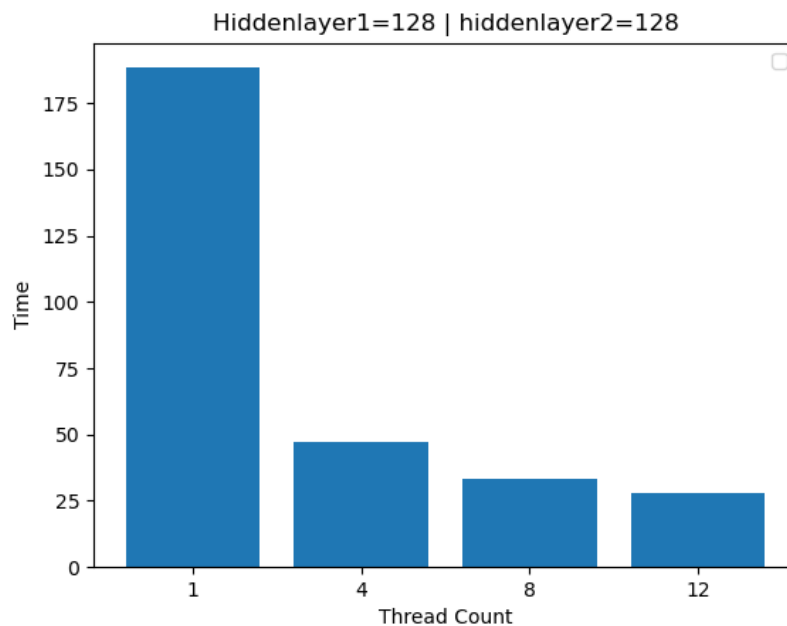


Abbildung 14: Barchart für Zeitverbrauch pro Thread<sup>14</sup>.

Hiddenlayergrößen festgestellt. Dabei sollte die Zeit aber bei kleinen Hiddenlayer, wie wir sie gewählt haben in Abbildung 15 (blau), nicht so klein werden. Grund dafür sollte der Overhead sein, der durch Nutzung von Threads, eine bestimmte Menge an Instruktionen hervorruft, damit die Threads genutzt werden können, um zu Kompilieren. Anscheinend ist der Overhead beim Aufruf unseres Codes nicht groß genug, so dass mit einer Hiddenlayergröße von 1 und 1 trotzdem eine Zeit von unter einer Sekunde erreicht werden kann.

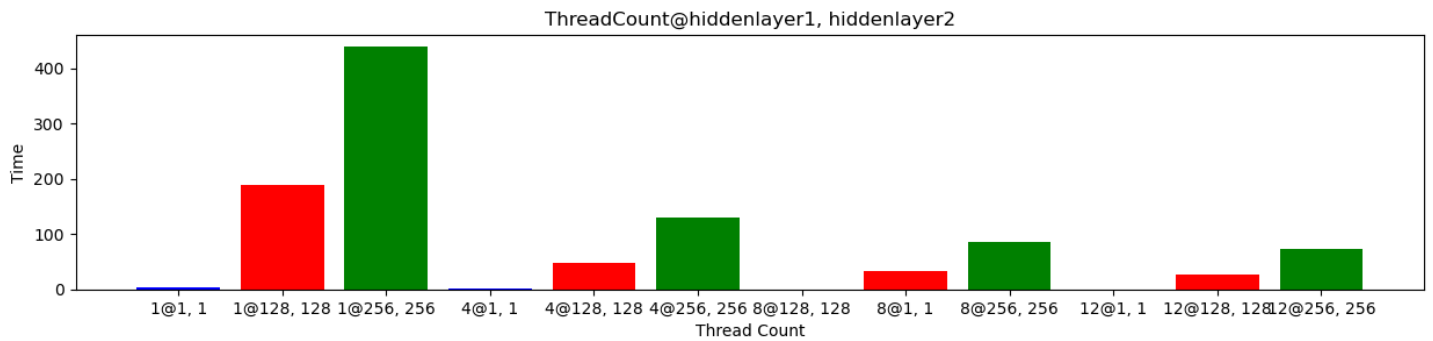


Abbildung 15: Barchart für Zeitverbrauch pro Thread für verschiedene Layergrößen<sup>15</sup>.

Auch mit Optimierung kann viel Zeit gespart werden. Dabei bietet Optimierung mit Nutzung von CXXFLAGS eine unkomplizierte und schnelle Zeiteinsparung. Es kann auch auf parallel geschriebene Codes verwendet werden, welches die Geschwindigkeit weiter verringert. Es werden folgende CXXFLAGS genutzt für Abbildung 16:

CXXFLAGS1 = -O0

CXXFLAGS2 = -Os

CXXFLAGS3 = -O2

CXXFLAGS4 = -O3

CXXFLAGS5 = -O2 -fthread-local

CXXFLAGS6 = -O2 -fthread-local -ffast-math

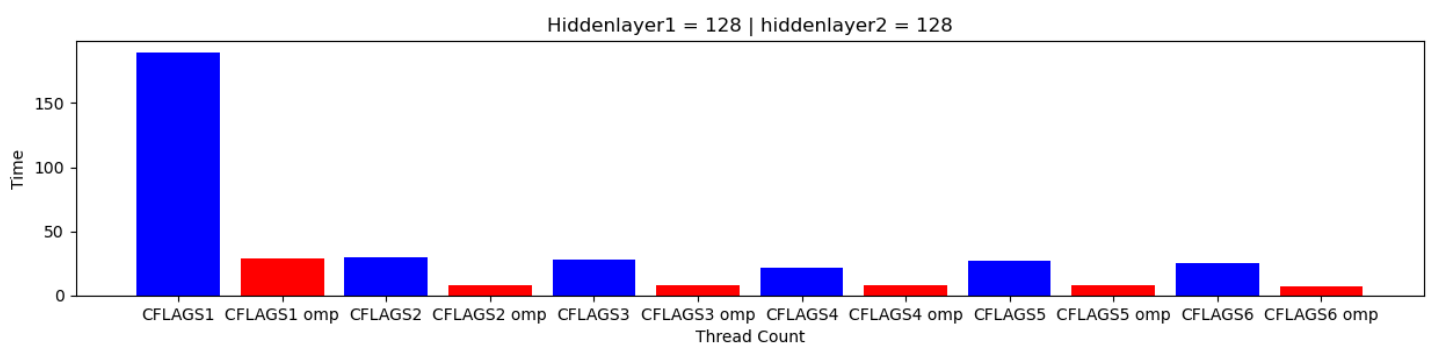


Abbildung 16: Barchart für Zeitverbrauch je nach Optimizer<sup>16</sup>.

(Im Barchart wurden CFLAGS anstatt CXXFLAGS benutzt, da der Name zu lang war)

CXXFLAGS1 stellt hierbei keine Optimierung dar(-O0 = Optimierungslevel 0).

## 7 Schwierigkeiten und Aufgabenverteilung

Anfangs war es sehr schwer für uns einen Einstieg in das Thema zu verschaffen. Zwar haben wir die Mathematik hinter einem neuronalen Netzwerk verstanden aber eins zu schreiben aus den mathematischen Formeln, war für uns nicht leicht zu implementieren. Zwar war es möglich mithilfe eines Tutorials ein Netzwerk zu bauen, doch dann hatten wir uns sehr auf die projects.pdf festgesetzt und versucht mit Python ein Tensor-MNIST Datenset in C++ zu benutzen. Es war uns nicht klar wie wir in C++ mit Python arbeiten oder das Datenset über Python in C++ laden konnten. Wir hatten dann nach einer Weile einen ersten Ansatz das Datenset aus Python in eine .txt Datei zu laden, womit wir auch anfangs gearbeitet haben aber durch das Lesen dieser Textdatei mussten wir uns letztendlich mit einer anderen Methode versuchen. Dann sind wir im Internet auf das Yann.Lecun MNIST Datenset gestoßen welches das Gleiche ist wie das von Tensorflow, bloß dass die Datei in binary zum Download verfügbar sind, womit wir auch nach dem Verstehen, wie man binary Dateien ausliest in C++, endlich mit MNIST Dateien arbeiten konnten. Nach der Präsentation, welche in der Vorlesung gehalten wurde, haben wir einige Hinweise bekommen weitere Aufgaben des Projekts zu erfüllen. Eins davon war es eine richtige makefile zu erstellen statt der IDE Visual Studio zu benutzen. Dabei hatten wir vorher das Problem wie OpenMP benutzt wird an unserem Code und auch beim Kompilieren.

Zur Aufgabenverteilung war es, dass jeder von uns anfangs nach einem Ansatzweg gesucht hat, wie wir ein neuronales Netzwerk bauen könnten. Jeder hatte dabei einen anderen Code, der mehr oder weniger verständlich war und funktioniert hat. Wir einigten uns auf den Code von Hai Nam La, da der Code in einem Klassendesign geschrieben wurde, der die Leslichkeit deutlich verbessert hat. Um dann die MNIST Datei einzulesen, war es Minh David Nguyens

Idee, erst in Python die Datei in eine txt-File einzuschreiben, wobei Viet Anh Kopietz dann die Yann.Lecun MNIST Datei gefunden hat. Da Viet Anh auch die OpenMP Aufgaben aus der Vorlesung gemacht hatte, war er für die Parallelisierung zuständig. Minh David hatte dabei die Aufgabe die Makefile zu schreiben und mit Hai Nam an dem Code zu arbeiten, wobei versucht wurde unnötige Codelines rauszustreichen und andere Codelines "sauberer" zu beschreiben. Viet Anh hat außerdem mit seiner CPU, einer Ryzen 7 5800x den immer wieder veränderten parallelisierten Code kompiliert und Minh David den nicht parallelisierten Code mit seiner Ryzen 9 5900X, auch wenn letztendlich mit dem Ryzen 5 2600 von Hai Nam gearbeitet wurde. Hai Nam hat die Valgrind Untersuchung ausgeführt sowie den Gitlab-Runner gebaut und letztendlich auch die Tests durchgeführt die hier im Beleg benutzt wurden. Als Schlusswort kann gesagt werden, dass auch wenn die Aufgaben so aufgeteilt wurden, wie beschrieben, letztlich war jeder an jeder Aufgabe beteiligt, da öfters einer keine Idee mehr zu seiner Aufgabe hatte und Andere übernommen haben mit einer frischen Idee und neuer Perspektive.

# Abbildungsverzeichnis

1	Sigmoidfunktionskurve . . . . .	5
2	1. Parallelitätsform . . . . .	8
3	2. Parallelitätsform . . . . .	8
4	Main for-loop . . . . .	10
5	Erfolgsrate ohne Parallelisierung . . . . .	11
6	Recent Error Graph . . . . .	11
7	Zeit des Trainierens . . . . .	12
8	Vereinfachte Veranschaulichung von omp . . . . .	14
9	Erfolgsrate mit Parallelisierung . . . . .	14
10	Zeit des Trainierens mit omp . . . . .	15
11	Zeit mit Optimierung . . . . .	15
12	Zeit des Trainierens mit omp . . . . .	16
13	Valgrind mit Parallelisierung . . . . .	17
14	Barchart für Zeitverbrauch pro Thread . . . . .	18
15	Barchart für Zeitverbrauch pro Thread für verschiedene Layergrößen . . . . .	19
16	Barchart für Zeitverbrauch je nach Optimizer . . . . .	19

# Literatur

- [1] **3Blue1Brown** [www.youtube.com/watch?v=aircAruvnKk](https://www.youtube.com/watch?v=aircAruvnKk), 2022
  
- [2] **Prof. Dr. Bauer**  
<https://gitlab.rz.htw-berlin.de/bauers/ce20-mpt-rose-22>,  
2022
  
- [3] **Dave Miller** [www.youtube.com/watch?v=sK9AbJ4P8ao](https://www.youtube.com/watch?v=sK9AbJ4P8ao), 2022
  
- [4] **Michael A. Nielsen** "*Neural Networks and Deep Learning*", *Determination Press* 2015,  
["neuralnetworksanddeeplearning.com/chap1.html](https://neuralnetworksanddeeplearning.com/chap1.html), 2022
  
- [5] **Jordanott** <https://github.com/jordanott/Neural-Network-Parallelization>, 2022
  
- [6] **Yann Lecun, Corinna Cortes, Christopher J.C. Burges**  
<http://yann.lecun.com/exdb/mnist/>, 2022
  
- [7] **GNU**  
<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>,  
2022
  
- [8] **Auraham Camacho** <https://medium.com/@auraham/pseudo-memory-leaks-when-using-openmp-11a383cc4cf9>, 2022