Semesterprojekt
Neuronales Netz

Moritz Lechner
Konstantin Roßmann
Leon Sobotta

Mikroprozessortechnik
Computer Engineering

13. Oktober 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Für diese Seminararbeit sollte ein funktionierendes neuronales Netz von Grund auf konstruiert werden, welches letztendlich in der Lage ist, handschriftliche Zahlen zu erkennen. Das sollte ohne externe Bibliotheken in den Sprachen C oder C++ realisiert werden. Da dieser Prozess viel Rechenleistung erfordert, sollte das Programm zusätlich in zwei Varianten mit Parallelverarbeitung implementiert werden.

Damit der Algorithmus die handschriftlichen Zahlen überhaupt erkennen kann, muss er diese wie ein Mensch zuerst lernen. Dabei spricht man von maschinellen Lernen. Der Computer, also genauer der Algorithmus, lernt dabei Zusammenhänge zwischen dem Input und dem erwünschten Ergebnis zu erkennen. Ein Mensch kann in etwa ab dem 2. Lebensjahr zählen und entwickelt mit ungefähr 4 Jahren eine Zahlenverständnis. Damit der Algorithmus nicht so lange braucht, daher lohnt es sich diesen zu beschleunigen.

Diese Parallelisierung sollte die Laufzeit des Programms im Vergleich zur sequentiellen Implementierung deutlich verkürzen (*Speedup*). Theoretisch ließe sich die Laufzeit mit doppelt so vielen genutzten Kernen halbieren, dass jedoch nur wenn wirklich das gesamte Programm parallelisiert werden kann. Das wird beim neuronalen Netz voraussichtlich aber nicht umsetzbar sein, da gewisse Teile auf Ergebnisse von vorherigen Berechnungen warten müssen. Dennoch sollte sich die Laufzeit des Algorithmus deutlich verkürzen lassen.

Zur Umsetzung wurde OpenMP genutzt und die Compiler Version *SIMD* gewählt. Als Programmiersprache wurde vor allem C benutzt, da sich mit C sehr genau programmieren lässt, es werden keine unbekannten oder ungewollte Funktionen im Hintergrund ausgeführt.

2 Grundlagen von Neuronalen Netzen

Neuronale Netze sind beliebig viele verbundene Neuronen und sind damit Teil eines Nervensystems, sie sind also wichtige Objekte in der Biologie. Ihre Arbeitsweise hat man sich in der Informatik zu nutzen gemacht, wo man künstliche Neuronale Netze entwickelt, um Aufgaben zu lösen bei denen normale Programme an ihre Grenzen stoßen. Ähnlich wie das Menschliche Gehirn können künstliche Neuronale Netze in gewisser Weise lernen, somit eignen sie sich perfekt für Aufgaben bei denen man Entscheidungen treffen muss, und auf Basis der vorherigen Erfahrungen die nachfolgenden Ergebnisse anpassen kann. Das sind zum Beispiel Bild- und Stimmenerkennung, Maschinelles Übersetzen, Chatbots, autonomes Fahren oder die Robotik[1].

[1] siehe Quelle 6

Ein solches Netz besteht aus mehreren Schichten von Neuronen, wie in Abbildung 1 zu sehen ist. Dabei gibt es immer mindestens eine Eingabe- und eine Ausgabeschicht, die von mindestens einer versteckten Schicht getrennt sind. Diese versteckten Schichten sind es, die dem Netz seine Fähigkeit zum lernen geben, je mehr ein Netz besitzt, desto komplexer kann es werden. Die einzelnen Schichten bestehen aus Neuronen die von Schicht zu Schicht miteinander verbunden sind um Daten weiterzugeben. Werden die Inputs in einem Netz von der ersten zur letzten Schicht immer weitergegeben, wird dieses Netz als *feed forward* Netz bezeichnet. Mittlerweile werden die meisten neuronalen Netze auf diese Weise realisiert, so auch unseres.

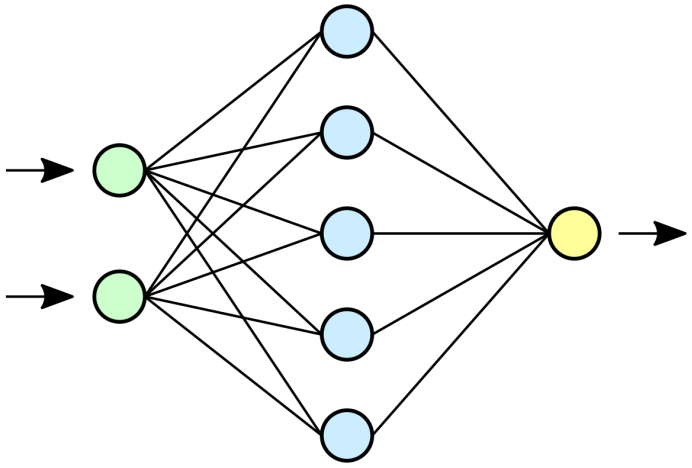

Abbildung 1: Ein vereinfachtes Neuronales Netz bestehend aus Neuronen. Quelle: Wikipedia

In der Abbildung 2 kann man sehen, dass jedes Neuron aus den Gewichtungen, der Übertragungsfunktion und der Aktivierungsfunktion besteht. Die Gewichtungen sind Faktoren, mit denen die Eingaben der vorherigen Neuronen multipliziert werden. Diese gewichteten Eingaben werden dann in der Übertragungsfunktion zur Netzeingabe zusammengefasst und schließlich durch die Aktivierungsfunktion zur Ausgabe dieses Neurons, welches den Wert an die nächste Schicht weitergibt, bis man an der Ausgabeschicht angekommen ist. Wenn das errechnete Ergebnis nicht mit den Erwartungen übereinstimmt, wird eine Fehlerfunktion angewendet, welche bestimmt, ob die Gewichtungen vergrößert oder verringert werden sollen, das nennt sich *Fehlerrückführung*. Wie stark die Gewichte angepasst werden, also wie groß die Lernfaktoren sind, wird mit der *Backpropagation* berechnet. Dabei wird die Kostenfunktion benötigt, welche angibt wie genau die Vorhersagen des Netzes sind. Je größer sie ist, desto ungenauer ist das Netz, dementsprechend möchte man den Punkt finden, an dem sie nahezu null ist, was mit Hilfe der Ableitung und den entsprechenden Gewichten passiert. Zuerst werden die Anpassungen für die letzte Schicht vorgenommen, dann werden die Schichten von hinten nach vorne darauf basierend angepasst. Jede Anpassung beruht auf der vorherigen, somit werden doppelte Berechnungen minimiert und der Algorithmus läuft schneller. Das macht bei einem kleinen Netz wie unserem keinen großen Unterschied, doch bei komplexeren Strukturen ist dieses Vorgehen unverzichtbar.

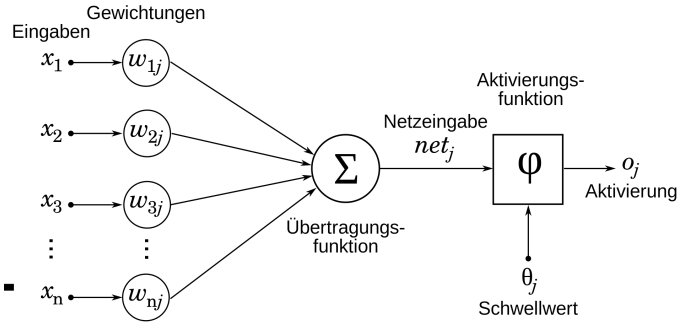

Abbildung 2: Das Schema eines künstlichen Neurons. Quelle: Wikipedia

Um ein neuronales Netz zu nutzen muss man es zuerst trainieren, dafür greift das Netzwerk auf eine enorme Datenmenge von mehreren Tausend Datensätzen zurück. Für diese berechnet das Netz dann seine Ausgabe, vergleicht diese mit dem erwarteten Ergebnis, den *Label Daten*, und passt gegebenenfalls seine Gewichtungen an. Nachdem sich das Netz durch sämtliche Trainingsdaten durchgearbeitet hat wird es an einem kleineren Testdatensatz getestet, es gibt also wieder Ausgaben, doch diesmal lernt es dabei nicht. Da selbst die vielen Beispiele aus den Trainingsdaten oft nicht reichen um eine hohe Genauigkeit zu erzielen, fängt das Netz dann erneut an die Daten durchzugehen, diesmal aber nicht mit zufälligen Gewichtungen, sondern mit den bereits verbesserten. Jeder dieser Durchläufe ist eine neue Epoche des Netzwerkes.

2.1 Grundlagen unseres Netzes

In der ersten funktionierenden Version hatte unser Netz nur eine Eingabe- und eine Ausgabeschicht. Es handelte sich also noch um ein sehr simples Netzwerk, das Perceptron (siehe Abbildung 3), und ebenso simpel war auch die Berechnung der Lernfaktoren, welche mit der *Delta Learn Regel* [2] berechnet wurden.

Diese Berechnet die Anpassung der Gewichte unter der Zuhilfenahme der Lernrate ϵ, der Differenz aus der erwarteten Ausgabe t_j und der tatsächlichen Ausgabe y_j sowie der entsprechenden Eingabe x_i. Das alles wird in folgender Formel zusammengefasst:

$$\Delta w_{ji} = \epsilon(t_j - y_j)x_i$$

Mithilfe der Lernrate kann einerseits die Geschwindigkeit des Lernprozesses, andererseits aber auch die Genauigkeit kontrollieren kann. Je größer die Lernrate, desto größer sind auch die Schritte, mit denen das Netz lernt. Das Problem ist, dass man mit großen Sprüngen das Optimum verfehlen kann. Daher nutzt man eher einen Wert irgendwo in der Mitte, bei unserem Perceptron hat sich eine Lernrate von 0,1 als am effektivsten herausgestellt. Die Lernrate ist in der Regel umgekehrt proportional zur Größe des Netzes.

[2] siehe Quelle: 4

Das Perceptron oder auch One Layer Neural Net verhält sich dabei sehr ähnlich wie die Linear Regression und kann daher nur in einem bestimmten Breich der Standardabweichung Zusammenhänge erkennen. Dadurch ist die Genauigkeit auch deutlich niedriger als in einem komplexeren Neuronalen Netz.

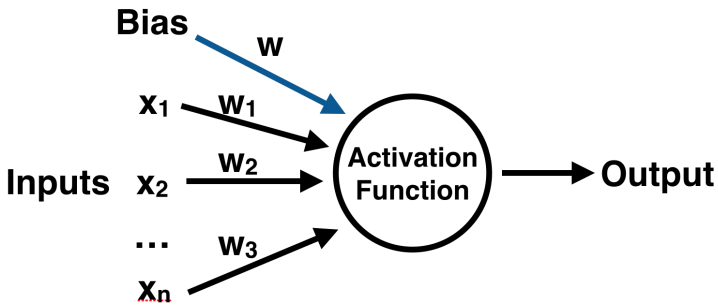

Abbildung 3: Darstellung eines Perceptrons

Mit diesem Aufbau ließsich überraschender Weise innerhalb von 50 Epochen bereits eine Genauigkeit von über 90 % erreichen. Als wir dann jedoch versucht haben eigene Beispiele von Zahlen durch das Netz erkennen zulassen hatte es eine deutlich schlechtere Genauigkeit. Wir haben das auf die geringe Varianz der Zahlen des Datensatzes zurückgeführt, mit denen es gelernt hat.

In der nächsten Entwicklungsstufe wird unser Netzwerk um eine Schicht erweitert und besteht nun aus einer Eingabeschicht, zwei versteckten Schichten und einer Ausgabeschicht. Dabei hat die Eingabeschicht 784 Neuronen, da die Bilder der Zahlen in $28 \cdot 28$ Pixeln auflösen. Die versteckten Schichten haben 16 bzw. 10 und die Ausgabeschicht hat 10 Neuronen, für jede der möglichen Antworten eins. Für den ersten Durchlauf werden die Gewichtungen zufällig generiert. Die Übertragungsfunktion unseres Netzes ist ein einfaches Kreuzprodukt aus den Gewichten und den Eingaben:

$$a \times b = \sum_{n=1}^{N} a[i] \cdot b[i]$$

Als Aktivierungsfunktion nutzen wir die Sigmoidfunktion, allerdings ohne Vorfaktor. Diese Funktion ist als Aktivierungsfunktion sehr verbreitet, da sie sich in der Übergangsregion sehr gleichmäßig verhält und daher gut differenzierbar ist. Außerdem gibt sie für alle Eingaben einen Wert zwischen 0 und 1 aus, so dass sämtliche Inputs in einen festen Wertebereich konvertiert werden. Der Graph, welcher diese Funktion beschreibt ist in Abbildung 4 zu sehen. Die Funktion selbst lautet wie folgt:

$$\frac{1}{1 + e^{-x}}$$

Durch das hinzufügen der weiteren Schichten zwischen Input und Output, können nun nicht mehr einfach die Differenz zu dem gewünschten Ergebniss mit der Gewichtung addiert werden. Die dadurch neu enstandenen Verbindungen lassen sich nicht mehr so leicht verändern und sind auch nicht mehr intuitiv mit der Wertigekeit der Eingabe und Ausgabe verbunden.

Um diese Gewichtungen nun auch richtig anpassen zu können müssen wir unsere Formel erweitern und eine Fallunterscheidung machen. Dabei helfen die Erkenntnis aus der Delta Learn Regel weiter hin. Den für die Verbindungen, die in den Output führen, ist Δw_{ji} weiterhin direkt vom Output und dem erwarteten Wert abhängig. Erst für die folgenden Schichten muss die Formel angepasst werden.

Die Berechnung der Lernfaktoren mit der Backpropagation wird mit der folgenden Formel realisiert. Dabei entspricht z_j^l dem Faktor für das Neuron *j* in der Schicht *l*, während *m* für die Anzahl an Schichten steht.

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l$$

Beim Perceptron hatte jeder Output einen jeweiligen Fehler, um nun den Fehler der jedes Hidden Neurons zu berechnen, wird der Fehler des Outputs nach hinten gereicht. Dieser Fehler dient dabei als Richtung und gibt an ob eine Flanke erhöht oder gesenkt werden muss. In der Hidden Schicht ergibt sich der Fehler aus der Summe der Fehler der vorherigen Schicht und der Wertigkeit der jeweiligen Verbindung mit der die Neuronen verbunden sind. So gibt jedes Output Neuronen seinen Fehler nach hinten, dort geben die Neuronen, dann mit Hilfe der gleichen Technik ihreren vorher berrechneten Fehler nachhinten und der Schritt wird so oft wiederholt, bis der Input erreicht wird.

Damit jede Anpassung nicht nur in die richtige Richtung sonder auch in der richtigen Stärke geschiet, muss bei jeder Anpassung auch die Relevanzen der Verbindungen betrachtet werden und als Faktoren mit eingrechnet werden. Dieser Faktoren ergeben sich einmal schon aus der größe der jeweiligen Fehler, denn umso stärker die Abweichungen, umso stärker können die Gewichtung angepasst werden. Die weiteren Faktoren sind die jeweiligen Inputs, die über die anzupassenden Gewichtung in das Neuron gegeben werden und der eigene Wert des Neurons.

Die von uns verwendeten Daten stammen aus dem weit verbreiteten MNIST-Datensatz[3], welcher aus 60.000 Trainingsdaten und 10.000 Testdaten besteht.

[3]Modified National Institute of Standards and Technology database

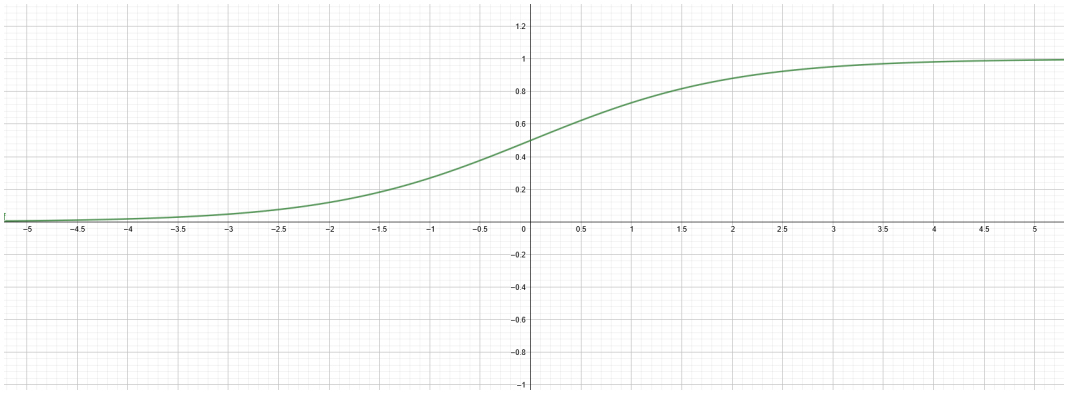

Abbildung 4: Darstellung der Sigmoidfunktion

3 Vergleich Sequentiell zu Parallelisiert

Wenn das Perceptron beispielsweise 200 Epochen sequentiell durchlaufen würde, müsste man über 2 Minuten auf ein Ergebnis warten, das ist nicht sonderlich praktisch wenn man gewisse Parameter ausprobieren und anpassen möchte. Vor allem da das Programm mit steigender komplexität des Netzes und hinzufügen der Packprogation noch langsamer wird. Daher muss man einen Weg finden um die Laufzeit des Programmes deutlich zu verkürzen, ohne dabei die Komplexität und damit die Leistungsfähigkeit des Netzes zu verringern. Da man das Programm also nur sehr begrenzt simpler machen kann muss man die Abarbeitung effizienter gestalten, indem man das Programm parallel laufen lässt.

Um diese Parallelisierung zu implementieren können verschiedene Methoden genutzt werden. Für unser Programm haben wir zum einen das Konzept *Single Instruction Multiple Data* (*SIMD*), sowie die Parallelisierung mit der C-eigenen Funktion *omp parallel* genutzt. OMP bedeutet Open-Multi-Processing, dabei werden Schleifen in Programmen auf mehrere Threads[4] aufgeteilt, so dass diese deutlich schneller abgearbeitet werden können. Wenn das Durchlaufen einer Schleife beispielsweise auf 8 Threads aufgeteilt wird, braucht dieser Prozess nur noch ein Achtel der Zeit. Bei SIMD wird das Programm weiterhin nur auf einem Kern ausgeführt, dabei werden die Elemente der zu verarbeitenden Vektoren aber an mehrere ALUs (Arithmetic Logic Unit) verteilt und können so parallel auf *Lanes* verarbeitet werden[5]. Da ein Vektor eine Reihe einzelner Werte vom gleichen Typ ist, kann die ALU sie als eine Einheit verarbeiten. Somit eignet sich dieses Verfahren gut für Neuronale Netze, da Vektoren dort bei fast allen Berechnungen vorkommen.

[4] Teil eines Prozesses, der unabhängig durchlaufen werden kann

[5] siehe Quelle: 1

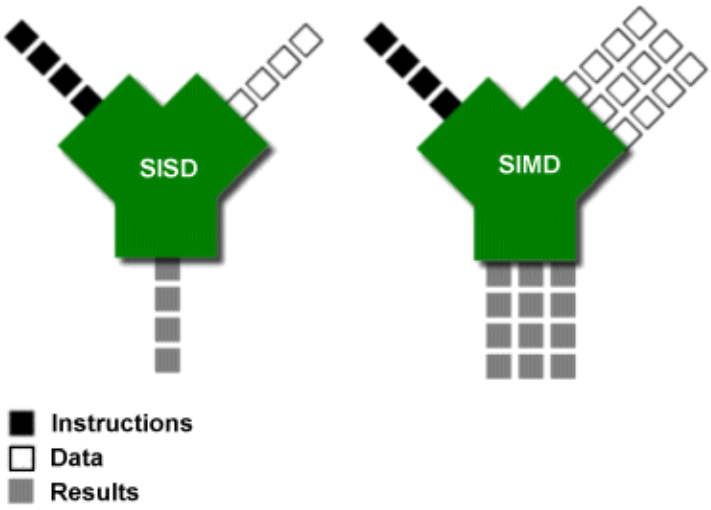

Abbildung 5: Verbildlichung von SIMD. Quelle: arstechnica.com

3.1 Darstellung der Ergebnisse

Wir wollen nun unser neuronales Netz auf seine Genauigkeit und Laufzeit auswerten. Umgesetzt haben wir das, indem wir ein Python-Script geschrieben haben, welches das neuronale Netz automatisch mit stetig steigenden Epochen trainiert und getestet hat. Die daraus resultierenden Laufzeiten und Genauigkeiten wurden von dem Script zwischengespeichert und anschließend in anschaulichen Graphen dargestellt. Dieser Prozess wurde drei mal ausgeführt, einmal für die nicht parallelisierte Version, dann für die mit OMP parallelisierte Version und anschließend für die mit SIMD parallelisierte Version.

Als erstes Betrachten wir Laufzeit der verschiedenen Parallelisierungen in Abbildung 6. In dem Graph ist sehr gut zu erkennen, dass die Laufzeit des Netzes ohne Parallelisierung bei einer niedrigen Anzahl an Epochen noch relativ nah bei der Laufzeit des parallelisierten Codes liegt. Dies Ändert sich jedoch im weiteren Verlauf des Graphen. Bereits bei 25 Epochen beträgt die Differenz zwischen parallelisiert und nicht parallelisiert 25 Sekunden. Auch Auffällig in dem Graph ist, dass die Laufzeit der mit OMP parallelisierten Version, der mit SIMD parallelisierten Version ähnelt. Das lässt sich dadurch erklären, dass wir SIMD nicht richtig angewendet haben. Wir sind an dieser Stelle an unsere Grenzen gekommen und wussten nicht wie wir SIMD korrekt implementieren sollen. Daher lässt sich kein wirklich erkennbarer Unterschied zwischen diesen beiden Laufzeiten feststellen. Trotzdem haben wir, wie in der folgenden Abbildung 7 erkennbar, durch die Parallelisierung des Codes einen Speedup von im Schnitt 1.5 erreichen können.

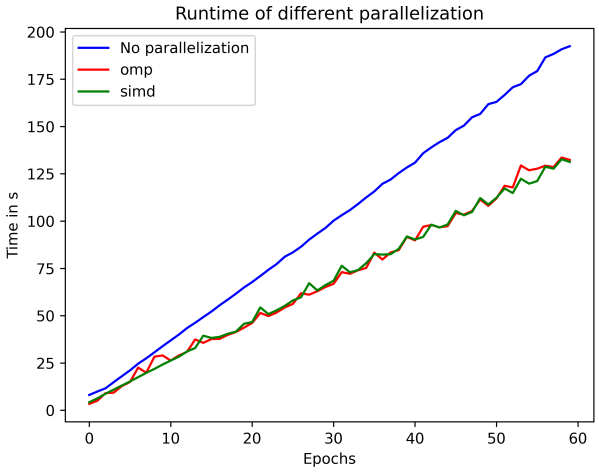

Abbildung 6: Vergleich der Laufzeit zwischen sequentiell, OMP und SIMD

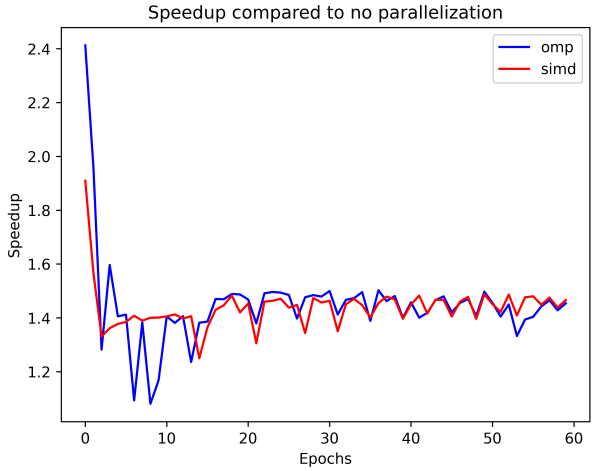

Abbildung 7: Vergleich des Speedups zwischen OMP und SIMD

Als letztes möchten wir noch auf die Genauigkeit des Netzes eingehen. Im folgenden Graph in Abbildung 8 lässt sich diese gut veranschaulichen. Wie man sehen kann, steigt die Genauigkeiten in den ersten 10 Epochen sehr stark an und fängt dann langsam an bei ca. 97 Prozent zu stagnieren. Dies ist bei allen drei Implementierung identisch. Leider muss man sagen, dass 97 Prozent keine gute Genauigkeit für ein Neuronales Netz ist, wir es jedoch nicht geschafft haben eine höhere Genauigkeit zu erreichen. An der Stelle hätte mit man mit umfangreichen Tests zu sämtlichen Parametern, wie zum Beispiel Lernfaktor, Batch Size oder Größe und Anzahl der Hidden Layer, ein genaueres Ergebnis erzielen können.

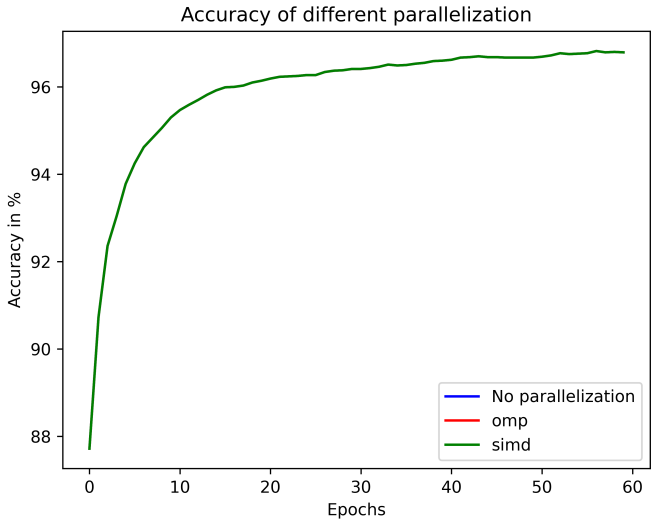

Abbildung 8: Vergleich der Genauigkeit zwischen sequentiell, OMP und SIMD

4 Umsetzung

Die theoretische Lösung sowie der Aufbau des geplanten Neuronalen Netzt muss nun ausgeführt werden. Da das niemand händisch machen will, wird dafür eine Software Lösung benötigt.

Wie eingangs erwähnt nutzen wir für den Code des Algorithmus die Programmiersprache C. Die Vorteile von C in diesem Projekt sind neben einer hohen Portabilität auch die Möglichkeit schnelle und ressourcensparende Programme zu schreiben. Durch diese Vorteile eignet sich C am besten für dieses Projekt. Als Compiler verwenden wir GCC, ein Open-Source Compiler-Suit der im Rahmen des GNU-Projektes entwickelt wird. Dieser ist auf den meisten Systemen anwendbar und steht kostenlos zur Verfügung.

Zu erst wird der simplere Teil des Netzes implementiert, das Berechnen des Outputs. Dabei gehen wir vorwärts über die Schichten bis hinzu dem Output. Dafür starten wir in dem ersten Layer, dem Input, und berechnen wir für jeden Knoten der darauffolgenden Schicht das Kreuzprodukt aus dem Input und der jeweiligen Gewichtungen zwischen den Knoten. Das Ergebnis wird daraufhin an die Aktivierungsfunktion übergegeben, bei uns die Sigmoidfunktion, und der Bias addiert. So gelangen wir zum Output, die Funktion wir auf Grund des "vorwärts durchgehen" in den meisten Literaturen FeedForward genannt.

```
1 // code snippet from the inner for loop
2 nodes[row][layer + 1][node] =
    sigmoid(dotProduct(nodes[row][layer],net->weights[layer][node],
    net->n_nodes[layer]) + net->biases[layer][node]);
```

Am Anfang wird ein Neuronales Netz mit einem zufälligen Werten für die Gewichtungen initialisiert, damit ein Startpunkt kreiert werden kann. Doch dadurch ist das Ergebnis auch zufällig und der Algorithmus rät eigentlich nur. Damit er nun auch tatsächlich seinen Input erkennt und weißwie der Output sein sollte, muss das Programm "lernen".

Für eine bessere Verständlichkeit wird hier das Beispiel der Zahlenerkennung benutzt. Das Programm bekommt eine Zahl als Input und soll diese erkennen. Das Ergebnis wird dann mit eigentlichen Wert verglichen.

Für ein gut trainiertes Netz wird ein guter Lern-Datensatz benötigt, denn ein Netz wird nur das erkennen, was es so ähnlich schon mal gesehen hat. Hier wird der MNIST Datensatz benutzt, dieser enthält 60.000 Trainingsbilder und 10.000 Testbilder und steht zur freien Verfügung.

4.1 Perceptron

Um in die Welt der neuronalen Netze einzutauchen und deren Lernweise zu verstehen, eignet sich das Perceptron. Da es auf zwei Layer reduziert ist, besteht es nur aus einer Input Schicht, die direkt mit einer Output Schicht verbunden ist (siehe Abbildung 3). Dadurch besteht ein direkter, linearer, Zusammenhang zwischen dem Input und den Output und es gibt nur einen Wert den wir anpassen können, die Gewichtungen der Verbindungen (siehe Deltalearn auf Seite 5).

```
// inner loop from delta Learn
weights[i_output][i_input] += derivedSigmoid(noutputs[train_row]
    [i_output]) * train_image_data[train_row][i_input] * earn_rate * -2 *
    (outputs[train_row][i_output] - train_label_data[train_row][i_output]);
```

Die Differnz gibt also die Richtung an in die die Gewichtungen angepasst werden, die weiteren Werte geben die Stärke an in der die Gewichtung geändert werden soll.

4.2 Implementation der Hidden Schicht

Die Anzahl der Gewichtungen war bis jetzt noch von der Anzahl der Inputs und Outputs abhängig und auch auf diese begrenzt. Das Perceptron wird nur besser, wenn es ein größeres Bild als Input erhält. Damit das Neuronale Netz besser wird ohne den Input zu verändern, muss die Lernfunktion nicht linear werden (siehe Backpropagation auf Seite 4).

Durch die Implementierung der Hidden Schicht erlangt das Netz eine deutlich höhere Komplexität, da so wie der Lernalgorithmus aufgebaut ist, nur bei einer direkten Verbindung zwischen Input und Output die Gewichtungen richtig angepasst werden können.

Der bestehende Algorithmus kann mit wenig Änderung weiterhin verwendet werden um die Gewichtungen anzupassen die in den Output führen. Die Gewichtungen die noch zwischen den Schichten davor sind, sollen auch angepasst werden, da so erst ein geringerer Fehler erreicht werden kann.

Das nach hinten "weiterreichen" der Differenz zwischen Output des Netzes und des Soll-Wertes, die Backporpagation, lässt sich auch ähnlich in Code umsetzten. Der Knoten vor dem Output bekommt seinen eigenen Fehlerwert, wie der Output auch einen hatte. Das wird, wie (Verweis einfügen) schon beschrieben, dadurch erreicht, dass alle Gewichte der ausgehenden Verbindungen des Knoten mit dem Fehler des Knoten auf mit dem sie verbunden sind multipliziert werden. Das geht dann solange weiter bis der Input erreicht wird.

Nach dem nun alle Fehler für die einzeln Knoten berechnet wurde, werden diese wieder auf alle eingehenden Gewichtungen der jeweiligen Knoten addiert.

```
//learn_factor init with 0
void deltaBackProp(){
  for (int layer = n_layer - 2; layer > 0; layer--){
    for (int node = 0; node < n_nodes[layer]; node++){
      for (int past_node = 0; past_node < n_nodes[layer + 1]; past_node++)
        {learn_factors[layer - 1][node] += learn_factors[layer][past_node]
        * weights[layer][past_node][node];
      }
      learn_factors[layer - 1][node] *=
        derivedFunktion(nodes[row][layer][node]);
    }
  }
}
```

4.3 Batch Learning

Diese Routine wird jetzt auf jede Datensatz angewendet, das sind bei MNIST-Datensatz 60.000 Wiederholungen. In jeder Wiederholung muss erst FeedForward ausgerechnet werden, um dann den errechneten Output mit seinem Fehler zurückzupropagieren.

Die momentane Implementation ist mit Memory Spielen zu vergleichen, wir drehen eine Karte um und wenn wir falsch liegen müssen wir uns merken welche Karte wo lag. Nur das dieses Spiel aus 60.000 Pärchen besteht, wir werden also nie mit nur einem Versuch ein richtiges Pärchen finden, aber unsere Trefferquote würde sich verbessern, wenn wir mit einem Zug immer mehrere Felder auf Einmal umdrehen dürften.

Bei dem kleinen Gedankenspiel würde die Menge an Daten, die sich gemerkt werden müssen, zwar nicht weniger werden, aber die Anzahl an Runden würde sich verringern. Dieser Gedanke lässt sich auch auf den Lernalgorithmus anwenden.

Der Datensatz wird in Stapel, sogenannte Batches, unterteilt. Bevor ein Stapel angeschaut wird, wird FeedForward angewendet, danach werden die Ergebnisse wieder mit den Sollwerten verglichen und die Gewichte abgepasst. Wie im Beispiel schauen wir uns nicht weniger Daten an und müssen auch nicht weniger Gewichtungen lernen, aber durch das Batchlearning erreichen wir schneller eine geringe Fehlerrate und minimieren auch den Zufallseffekt, der einzelnen Datensätze.

Der kleinste Stapel wäre also ein einzelnes Datenpaar und der größte wäre der gesamte Datensatz. Je größer der Batch, umso genauer ist jede Anpassung pro Epoche und die Fehlerminimierung gleicht fast dem Gradientenabstieg. Es auch werden zufällig Sprünge, die das Netz auch verschlechtern können, vermieden. Allerdings ist dadurch die Anpassung an den Gewichtungen auch nur minimal, es werden also mehr Epochen benötigt. Die Wahl der Batch Size ist ein Zusammenspiel von der gewünschten Genauigkeit und der Geschwindigkeit, mit der diese erreicht werden soll.

4.4 Graphical User Interface

Zusätzlich zu einer terminalbasieten Ausführung des Programms hatten wir uns zudem noch als Ziel gesetzt, eine grafische Benutzeroberfläche zu implementieren, da dies einige Vorteile mit sich bringt. Zum Einen ist es für den Benutzer deutlich intuitiver eine graphische Anwendung zu bedienen, als das Programm über das Terminal aufzurufen. Außerdem wollten wir dem Benutzer die Möglichkeit geben, das neuronale Netz mit einer eigens angefertigten Skizze zu testen, sodass das Ergebnis des Netzes direkt in der GUI angezeigt werden kann. Wir einigten uns darauf, dies in Python mit Tkinter umzusetzen, da wir darin die meiste Erfahrung hatten. Aufgebaut ist unsere GUI wie folgt:

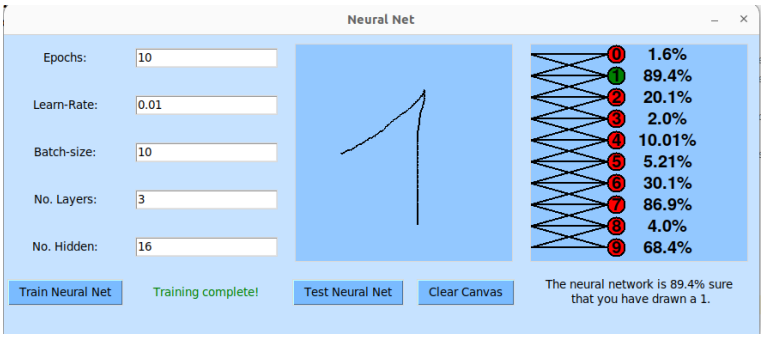

Abbildung 9: Screenshot der Grafischen Oberfläche

Wie in dem Screenshot zu sehen, ist unsere GUI grob in drei Segmente aufgeteilt. Auf der linken Seite befinden sich die wesentlichen Parameter des neuronalen Netzes. Diese kann der Benutzer selber in die Eingabefelder eintragen und mittels eines Klicks auf den darunterliegenden Button das neuronale Netz mit diesen trainieren.

Als Feedback, ob das trainieren erfolgreich war, wird ein kleiner Text rechts von dem Button eingeblendet.

Anschließend befindet sich in der Mitte der GUI eine leere Leinwand, auf der der Benutzer mit der Maus eine Ziffer malen kann. Diese Skizze wird zunächst in einem $252 \cdot 252$ Numpy Array gespeichert. Da das neuronale Netz aber nur $28 \cdot 28$ Arrays als Input akzeptiert mussten wir noch digitale Bildbearbeitung anwenden. Dazu bot sich die OpenCV Bibliothek an. Diese beinhaltet verschiedenste Algorithmen für die Bildbearbeitung und Computer Vision. Wir benötigten für unseren Anwendungsfall jedoch nur die resize()-Funktion, bei der ein Bild-Array auf ein beliebig großes Array angepasst werden kann. Wichtig dafür ist die Art der Interpolation, wir entschieden uns für "INTER_ AERA", da diese das Array unter Verwendung des Pixelflächenverhältnisses resampled.

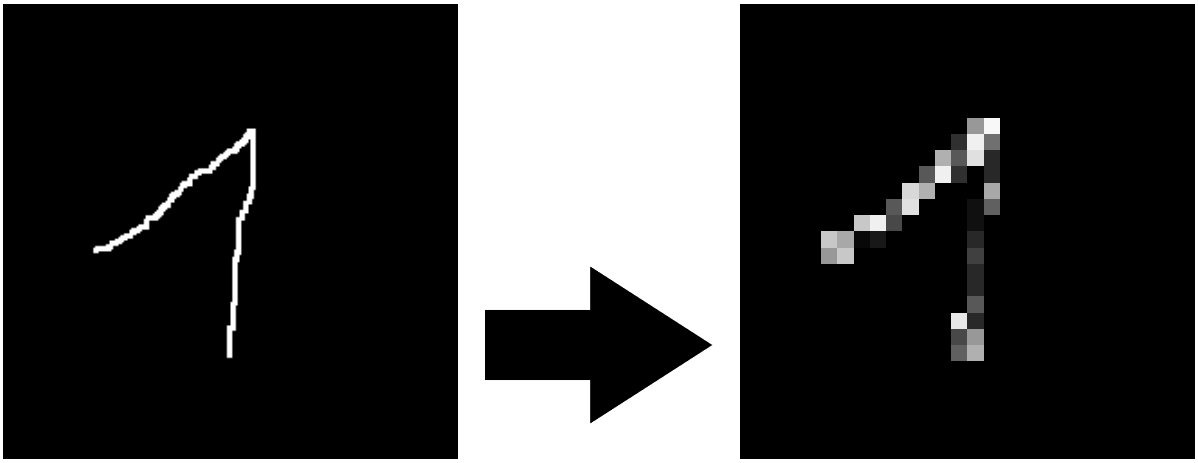

Abbildung 10: Downscaling des $252 \cdot 252$ Bildes auf $28 \cdot 28$ Pixel

Anschließend wird das neue Array als csv-Datei gespeichert und von dem neuronalen Netz eingelesen und getestet. Liegen nun die Ergebnisse vor, werden diese in der Grafik auf der rechten Seite der GUI angezeigt. Mit unserem Aufbau der GUI und des Netzes lässt sich das Netz beliebig oft testen, ohne jedes Mal das Netz neu trainieren zu lassen.

5 Fazit

Letztendlich ist es uns gelungen ein neuronales Netz ohne Bibliotheken Dritter zu implementieren. Es erkennt, wie erwartet, den Großteil der Zahlen aus den MNIST Datensätzen. Der Einstieg in ein so komplexes Thema war schwierig, vor allem da keiner von uns Erfahrungen mit neuronalen Netzen hatte. Auch nach einiger Recherche und den ersten Vorlesungen zu diesem Thema war uns unklar wie wir anfangen sollten. Doch mit Hilfe des simplen Perceptrons haben wir erfolgreich die ersten Schritte gemeistert.

Von diesem Punkt an ging die Entwicklung stetig voran, doch die Parallelisierung mit SIMD bereitete uns bis zum Schluss Probleme, da sie im Gegensatz zu OMP nicht einfach hinzugefügt werden konnte. Stattdessen musste man es von Anfang an auf komplexe Weise in den Code und die Berechnungen integrieren, was uns nicht gelungen ist. Die Implementierung und Verwendung von OMP war jedoch erfolgreich, so dass das Netz deutlich kürzere Laufzeiten hatte.

Obwohl nicht alle Anforderungen erfüllt wurden, haben wir dennoch eine Menge w"hrend diesem Semesterprojekt gelernt. Zum einen über das aktuelle Thema machine Learning, welches heutzutage in fast sämtlichen Bereichen Anwendung findet. Zum Anderen über die Parallelisierung von Programmen gelernt, welche es erlaubt rechenintensiven Code schneller ausführen zu lassen und so die Wartezeiten zu verkürzen.

Literatur

[1] https://gitlab.htw-berlin.de/bauers/ce20-mpt-sose-22/-/blob/main/slides/03-simd.pdf

[2] https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz

[3] https://de.wikipedia.org/wiki/K%C3%BCnstliches_Neuron

[4] https://en.wikipedia.org/wiki/Delta_rule

[5] https://www.freecodecamp.org/news/building-a-neural-\network-from-scratch

[6] https://datasolut.com/anwendungsgebiete-von-kuenstlicher\intelligenz/#Robotik

[7] https://www.it-treff.de/it-lexikon/c-programmiersprache

[8] https://www.kaggle.com/code/residentmario/full-batch-mini-batch-and-online-learning/notebook

[9] https://www.allaboutcircuits.com/technical-articles/sigmoid-activation-function-activation-in-a-multilayer\-perceptron-neural-network/

[10] https://deepai.org/machine-learning-glossary-and-terms/backpropagation

[11] https://www.youtube.com/watch?v=Wo5dMEP_BbI&list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3

[12] https://www.youtube.com/watch?v=4E4GhVREXBg&list=PL58qjcU5nk8sUq97ze6MZB8aHflF0uNgK&index=10