Technische Universität Dortmund

Fakultät Informatik
Design Automation for Embedded Systems Group

Training eines binären Neuronalen-Netzwerkes auf den MNIST Datensatz

Fachprojekt

Jack Diep, Florian Köhler, Yannick Naumann

Betreuung:
M.Sc. Mikail Yayla

16. September 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Neuronale Netzwerke bilden eine Unterkategorie des Machine-Learning und erlauben Auswertungen von Eingaben auf Basis von zuvor angelernten, empirischen Ergebnissen. Ein neuronales Netzwerk bildet ein System aus Neuronen ab, welche Schichtweise verbunden sind einen unidirektionalen Datenfluss erzeugen. Dieses System besteht üblicherweise aus einer Eingabeschicht, einer Ausgabeschicht und dazwischen beliebig viele *versteckte* Schichten, welche die eigentliche Arbeit des Netzwerks verrichten. Die Anzahl der Neuronen in den Eingabe- und Ausgabeschichten ist intuitiv wählbar. Die Größe der Eingabeschicht wird häufig durch die Anzahl der möglichen Eingaben bestimmt, die Größe der Ausgabeschicht durch die Anzahl der möglichen Ergebnisse. Die Größe und Anzahl der dazwischen liegenden Schichten hingegen muss je nach Anforderung und Gegebenheiten individuell ermittelt werden. Je größer das Netzwerk desto höher sind die Anforderungen an die benötigte Hardware um dieses zu betreiben. Bei schwächerer Hardware oder Einschränkungen bezüglich der Energieversorgung können kleinere Netzwerke eingesetzt werden, wenn auch häufig mit geringerer Genauigkeit verglichen mit einem größeren Netzwerk.

1.1 Motivation

Die Größe eines Netzwerks kann beim Entwurf dessen direkt beeinflusst werden. In Anwendungsgebieten, bei denen der Fokus auf geringen Hardwareanforderungen liegt, stoßen schnell das Problem der schwindenden Genauigkeit. Auf IoT-Geräten oder mobilen Platformen finden klassische neuronale Netzwerke daher nur eingeschränkt Nutzen.

2016 veröffentlichten M. Courbariaux und Y. Bengio [CB16] eine wissenschaftliche Arbeit und stellen dort das *binarisierte neuronale Netzwerk* (kurz BNN) vor. Dieses könne im Vergleich zu einem klassischen neuronalen Netzwerk eine theoretische Geschwindigkeitssteigerung auf das 32-fache erreichen. Die Genauigkeit des BNN liege jedoch nur knapp unter derer klassischer Netzwerke. Das BNN ermöglicht dank dieser Eigenschaften den Einsatz von neuronalen Netzen auf vergleichsweise schwacher Hardware und verspricht zugleich nur geringe Genauigkeitseinbüße.

In dieser Ausarbeitung wird die allgemeine Funktionsweise von neuronalen Netzwerken erläutert und anschließend der Entwurf eines binarisiertes Netzwerk mit Fokus auf Handschriftenerkennung auf Basis des MNIST Datensatzes dokumentiert. Dabei werden grundlegende Überlegungen, das Vorgehen, Herausforderungen sowie dazu erarbeitete Lösungen vorgestellt.

Kapitel 2

Neuronale Netzwerke

Unter einem neuronalen Netzwerk versteht man ein System aus Neuronen. Diese sind schichtweise organisiert wobei jedes Neuron einer Schicht jeweils zu allen Neuronen der direkt anliegenden Schichten verbunden ist. Abbildung 2.1 zeigt beispielhaft ein neuronales Netzwerk aus 19 Neuronen mit insgesamt 5 Schichten.

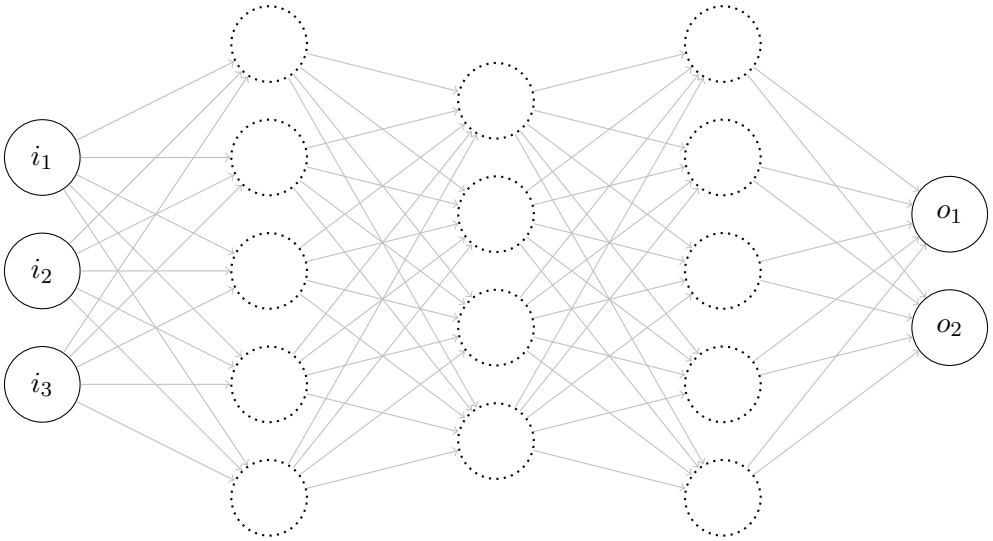


Abbildung 2.1: Struktur eines neuronalen Netzwerks

2.1 Neuronen

Unter einem Neuron versteht sich bei neuronalen Netzen lediglich ein Knoten, in dem üblicherweise ein 32-bit großer Wert hinterlegt ist. Häufig kommen hier Gleitkommazahlen zwischen -1.0 und 1.0 zum Einsatz, da sich dieser Wertebereich besonders gut zum Rechnen eignet und Eigenschaften bezüglich der Multiplikation besitzt, die eine Wertexplosion verhindern. Die Werte aller Neuronen, mit Ausnahme derer in der Eingabeschicht, setzen sich jeweils aus den Werten aller Neuronen der direkt davor liegenden Schicht zusammen. Das genaue Vorgehen bei der Wertermittlung hängt jeweils vom Netzwerk und den dort verwendeten Aktivierungsfunktionen ab.

2.2 Schichten und Kanten

In einem neuronalen Netzwerk ist jedes Neuron einer Schicht mit allen Neuronen der jeweiligen davor liegenden und danach liegenden Schicht über Kanten verbunden. Neuronale Netzwerke sind unidirektionale Graphen, demnach fließen Informationen über Schichten (und folglich Kanten) nur in eine Richtung.

Allen Kanten wird initial eine Gewichtung zugewiesen, welche erneut netzwerkabgängig generiert werden oder durch zuvor angelernte Daten bestimmt werden. Beim Trainieren des Netzwerks werden diese bei jeder Lerniteration (auch *Epoch* genannt) justiert, während sie beim Betrieb für gewöhnlich keine Änderungen mehr erfahren. Die Genauigkeit eines Netzwerks wird überwiegend durch diese Gewichte bestimmt, daher ist das Ziel beim Trainieren eines Netzes die Optimierung jener.

Kantengewichte wirken sich maßgeblich auf die Wertberechnung von Neuronen aus. Diese wird in zwei Schritten ausgeführt. Im ersten Schritt wird aus den Kantengewichten aller eingehenden Kanten und den Werten der darüber verbundenen Neuronen die Produktsumme gebildet. Für den zweiten Schritt sind jeweils Aktivierungsfunktionen notwendig, welche im nachfolgenden Kapitel erläutert werden.

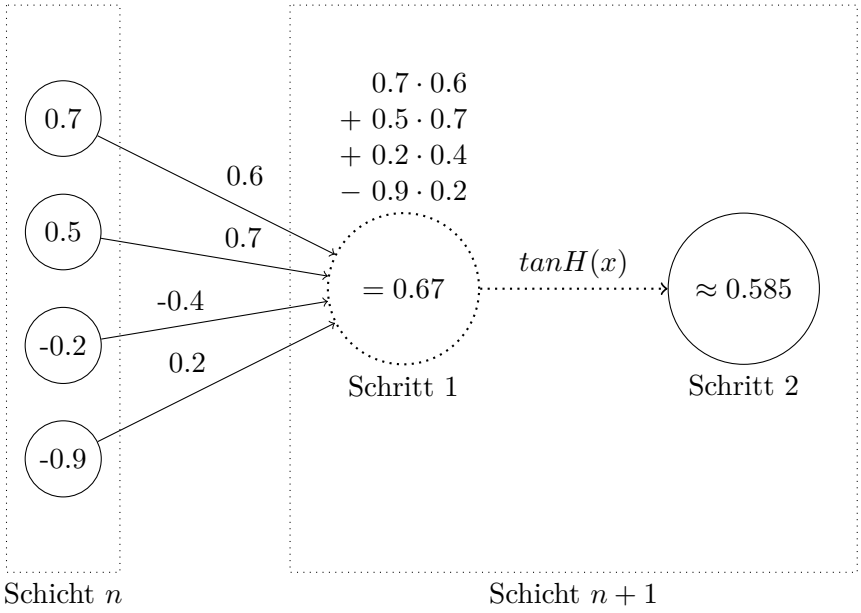


Abbildung 2.2: Berechnung eines Neuronenwertes
mit der $tanH$ Aktivierungsfunktion

2.3 Aktivierungsfunktionen

Im zweiten Schritt wird der zuvor errechnete Wert durch eine weitere Funktion modifiziert. Sogenannte *Aktivierungsfunktionen* können beliebig gewählt werden, müssen jedoch offensichtlich alle möglichen Eingaben auf einen Wert abbilden können. Die verwendete Aktivierungsfunktion kann je nach Schicht variieren. Einmal gewählt, ist diese jedoch für die jeweilige Schicht im Netzwerk für die Laufzeit fest.

Die beiden erläuterten Berechnungsschritte werden schichtweise in Richtung des Datenflusses des Netzwerks für alle Neuronen durchgeführt. Im folgenden Beispiel wird bildhaft dargestellt, wie solch eine Neuronenwertberechnung für ein Netzwerk aussieht, welche in der betrachteten Schicht die $tanH$-Aktivierungsfunktion verwendet.

Die Wahl der Aktivierungsfunktion beeinflusst die möglichen Werte, die Neuronen innerhalb einer Schicht annehmen können. Um eine Wertexplosion zu vermeiden (z.B. bei Verwendung von $ReLU(x) : \max(0, x)$ als Aktivierungsfunktion), können zusätzliche Normalisierungsschichten verwendet werden, welche die Neuronenwerte einer Schicht auf einen erwünschten Zielbereich einschränken. Es existieren jedoch auch Aktivierungs-

funktionen, die diese *Squashing*-Eigenschaft direkt besitzen. Darunter zählt auch die im vorherigen Beispiel verwendetete Funktion $tanH$, bei der alle Eingaben auf den Zielbereich [-1,1] abgebildet werden.

Im Folgenden sind sind einige, für neuronale Netzwerke übliche Aktivierungsfunktionen abgebildet.

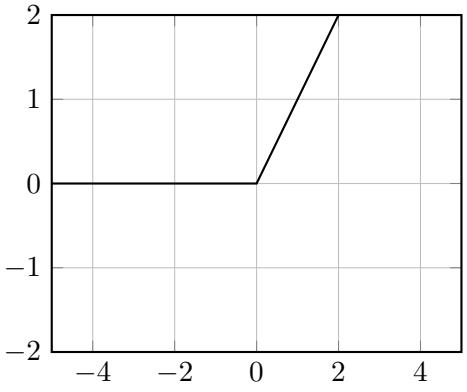


Abbildung 2.3: ReLU

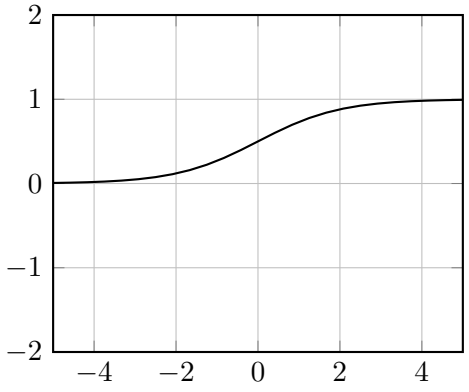


Abbildung 2.4: Sigmoid

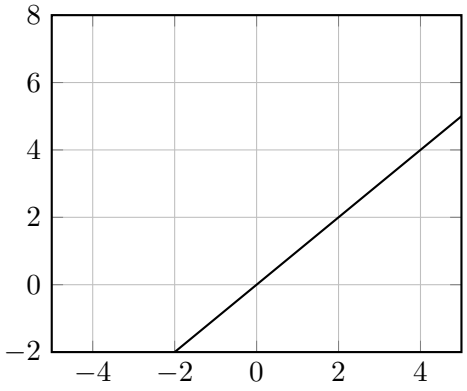


Abbildung 2.5: Softmax

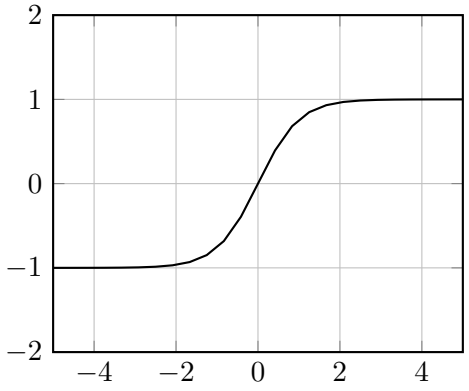


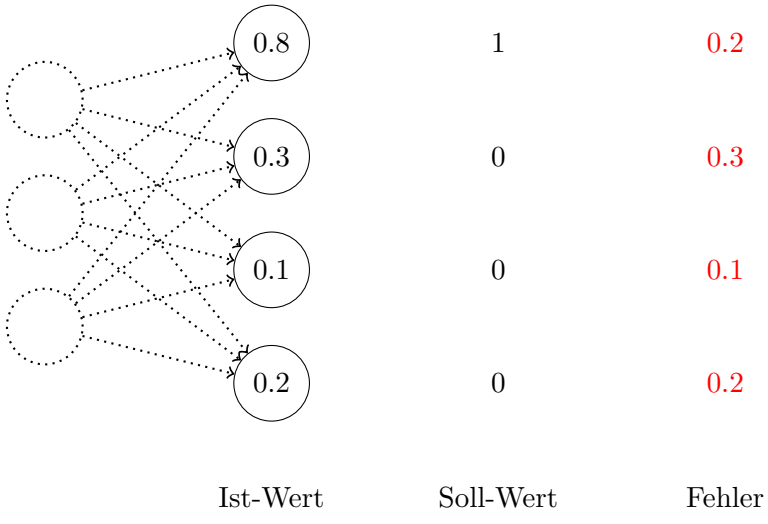
Abbildung 2.6: tanH

2.4 Training

Die Stärken und Schwächen eines neuronalen Netzwerks liegen neben der Struktur der Neuronen, der Schichten und den gewählten Aktivierungsfunktionen vor allem in den verwendeten Kantengewichten. Diese zu optimieren kann ein langwieriger und rechenintensiver Prozess sein. Kleine Änderungen im Aufbau des Netzwerks können große Teile von zuvor erlernten Gewichten unbrauchbar machen, daher werden Netzwerke häufig an sich selbst trainiert. Das bedeutet, dass beim Trainieren eines neuen Netzwerks die initialen Kantengewichte keine emprischen Daten aus anderen (ähnlichen) Netzwerk verwenden, sondern diese zu Beginn pseudozufällig gewählt werden.

2.4.1 Ablauf

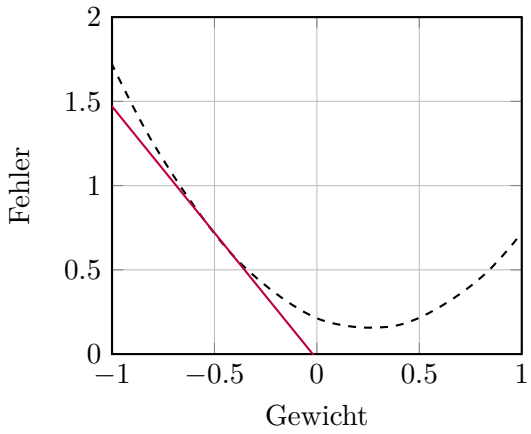
Beim Training werden wiederholt verschiedene Eingaben in das Netzwerk getätigt, zu denen das korrekte Ergebnis bekannt ist. In der letzten Schicht des Netzwerks, der Ausgabeschicht, werden die vom Netzwerk bestimmten Werte der jeweiligen Neuronen mit den erwarteten Werten verglichen. Die absolute Differenz dieser wird als Fehler bezeichnet. Das Ziel des Trainings ist es, den Fehler durch Anpassung der Kantengewichte zu verringern oder im Optimalfall ganz zu eliminieren (Fehler = 0). In folgender Abbildung ist ein Fehlerbeispiel dargestellt.



2.4.2 Backpropagation

Es stellt sich nun die Frage, ob und wie errechnet werden kann, welches Kantengewicht wie angepasst werden muss, um den Fehler zu verringern. Der Begriff der *Backpropagation* beschreibt das Durchlaufen des Netzwerks, jedoch entgegen der eigentlichen Laufrichtung und das Rückschließen der notwendigen Gewichtsanpassung, um die aktuelle Ausgabe in die gewünschte Richtung zu verändern.

Um eine Anpassung an einem Kantengewicht vorzunehmen, muss demnach zuerst festgestellt werden, wie die errechneten Fehler sich bei Änderungen am Gewicht verhalten. Dazu wird die sogenannte *Slope* ermittelt, also die Kurve auf der das Kantengewicht im Verhältnis zum Fehler steht. Anschließend wird die Steigung dieser Kurve an der aktuellen Stelle (Kantengewicht) errechnet. Aus dem Ergebnis lässt sich ableiten, in welche Richtung das Kantengewicht justiert werden muss, um den Fehler zu verringern. Ein Beispiel dazu befindet sich in der folgenden Abbildung, dort ist die Tangente der Fehlerkurve an der Stelle -0.5 in rot dargestellt. Daraus folgt, dass eine Erhöhung des zugehörigen Kantengewichts den Fehler weiter verringert.



Diese Anpassungen werden in jedem Trainingsdurchlauf für alle Kantengewichte durchgeführt, wobei die Stärke der Anpassung häufig konfigurierbar ist. Die Ermittlung der Fehlerkurve kann mathematisch durch das Verwenden der sogenannten *Chainingrule* [RHW86] erleichtert werden. Zur Verbildlichung dieser Regel zeigt die folgende Abbildung einen Ausschnitt aus einem einfachen neuronalen Netzwerk mit drei Schichten.

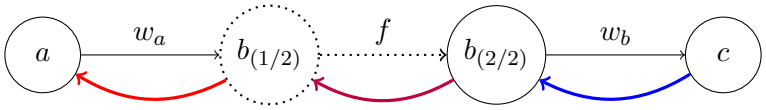


Abbildung 2.7: Neuronenkette: Backpropagation

Die notwendige Anpassung des Kantengewichts w_a zur Verringerung des Fehlers in c kann wie folgt ermittelt werden:

$$\frac{\delta c}{\delta w_a} = \frac{\delta b_{(1/2)}}{\delta w_a} \cdot \frac{\delta b_{(2/2)}}{\delta b_{(1/2)}} \cdot \frac{\delta c}{\delta b_{(2/2)}}$$

Diese Kette lässt sich weiter vereinfachen:

$$\begin{aligned}
\frac{\delta c}{\delta w_a} &= a \cdot \frac{\delta b_{(2/2)}}{\delta b_{(1/2)}} \cdot \frac{\delta c}{\delta b_{(2/2)}} && \left(\frac{\delta b_{(1/2)}}{\delta w_a} = a\right) \\
&= a \cdot f'(b_{(1/2)}) \cdot \frac{\delta c}{\delta b_{(2/2)}} && \left(\frac{\delta b_{(2/2)}}{\delta b_{(1/2)}} = f'(b_{(1/2)})\right) \\
&= a \cdot f'(a \cdot w_a) \cdot \frac{\delta c}{\delta b_{(2/2)}} && \left(b_{(1/2)} = a \cdot w_a\right) \\
&= a \cdot f'(b_{(1/2)}) \cdot w_b && \left(\frac{\delta c}{\delta b_{(2/2)}} = w_b\right)
\end{aligned}$$

Offensichtlich lässt sich dieses Vorgehen auf jede beliebig lange Neuronenkette anwenden und erlaubt somit die Ermittlung der Richtungen aller notwendigen Gewichtsänderungen. Im zuvor erwähnten Optimalfall wird jedes Kantengewicht solange angepasst, bis ein Fixpunkt erreicht wurde und damit jedes Gewicht genau im Tal der jeweiligen Fehlerkurve liegt. Ein häufig auftretendes Problem ist das sowohl das zu starke als auch das zu schwache Anpassen der Gewichte. Ersteres führt zum hin- und herspringen des verwendeten Gewichts auf der Fehlerkurve sodass sich dem Tal nicht mehr angenähert wird. Letzteres hingegen führt dazu, dass die Korrektur außerhalb des Optimums zum Stillstand kommt. Siehe dazu auch Abbildungen 2.8 und 2.9.

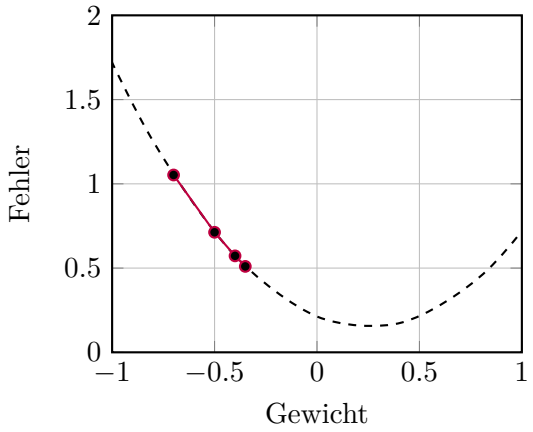


Abbildung 2.8: Zu niedrige Lernrate

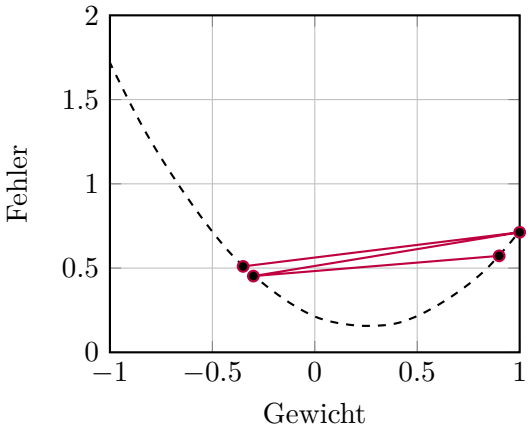


Abbildung 2.9: Zu hohe Lernrate

Nachdem das Training eines neuronalen Netzwerks abgeschlossen ist, werden diese erlernten Kantengewichte gespeichert und häufig zur Verwendung exportiert.

Kapitel 3

Das BNN

Bei unserem Neuronalen Netz handelt es sich um ein binäres Netzwerk. Im Folgenden werden die Anpassungen, beziehungsweise die verwendeten Komponenten beschrieben.

3.1 Aktivierungsfunktion

In neuronalen Netzwerken kann, wie in Kapitel 2.3 beschrieben, eine ganze Reihe von Aktivierungsfunktionen verwendet werden. Jede dieser Funktionen hat, je nach Anwendungskontext, verschiedene Vor- und Nacheile.
Für BNNs ist hier die $sign(x)$ Funktion eine populäre Wahl. Diese überführt die Aktivie-

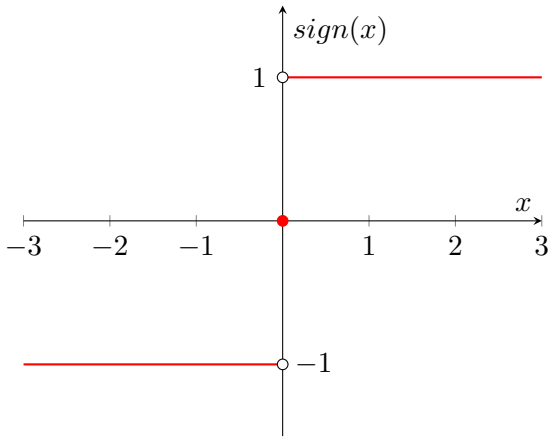


Abbildung 3.1: Die Vorzeichenfunktion

rungen direkt in binäre Werte. Da für die *backpropagation* allerdings die Ableitung der

Aktivierungsfunktion benötigt wird und die Vorzeichenfunktion nicht stetig differenzierbar ist, wird diese approximiert. In unserem Netzwerk wird sie, wie häufig verwendet, durch die *hard tanH* Funktion approximiert [Kim+20]. Diese ist, in den Ableitungen, günstiger zu berechnen als die Funktion *sign* und ist bei Werten $x \leq -1 \vee x \geq 1$ gleich zur Vorzeichenfunktion.

3.2 Binärer Linear-Layer

Für die Binarisierung der Linear-Layer wurde eine *Wrapper-Klasse* für die Pytorch-Klasse *nn.linear* erstellt.

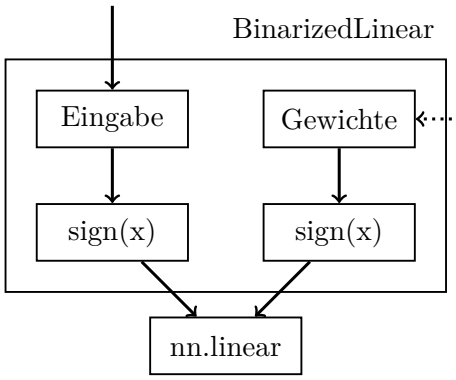


Abbildung 3.2: Aufbau des binären Linear-layer

Wie in Abbildung 3.2 zu erkennen, erhält der binäre Linear-Layer, wie normaler Weise *nn.linear*, die Ausgaben der vorherigen Schicht. Diese werden im Folgenden über die Vorzeichenfunktion binarisiert.
Die Gewichte, welche über das Training aktualisiert werden, werden bei erster Verwendung ebenfalls binarisiert. Im folgenden übernimmt die Pytorch-Klasse *nn.linear* die Berechnung der Aktivierung der Neuronen. Da die Berechnung der Aktivierungen also immer auf binären Gewichten und Eingaben geschieht, ist das Training binär.

3.3 BatchNorm

Die Rolle des *BatchNorm-Layers* in nicht-binären Neuronalen-Netzwerken ist eine leicht andere als in unserem BNN. Durch *BatchNorm* werden die Eingaben einer Schicht, innerhalb eines *Batches*, normalisiert. Hier wird dafür gesorgt, dass *Batches* einen Mittelwert von Null und eine Standartabweichung von Eins haben[IS15] .
Durch diese reduzierte Streuung der Werte, können in Neuronalen Netzen höhere Lernraten verwendet werden, da stark ausschlagende Daten nicht zu einer starken Überanpassung führen.
Bei binären Netzwerken hat dies, aufgrund der diskreten Kantengewichte,einen weniger großen Effekt.In BNNs erfüllt die *BatchNorm* Schicht eine zentrale Rolle für das Lernen des Netzwerkes. Primär dient die Normalisierung der Schicht dazu, das *expolding-gradient* Problem zu verhindern. Hier wird beim Training des Netzwerkes, welches die Error-Werte reduzieren soll, ein sehr hoher Error-Wert akkumuliert. Dies führt zu einer zu starken Anpassung der Gewichte, welche folgend zu einer niedrigeren Genauigkeit führt[SBN19].
Da durch Auftreten des *exploding gradients* die Genauigkeit stark abgesenkt wird, kann dieses Netzwerk sich, ab einer gewissen Schwelle, nicht mehr verbessern. Der stetige Lernerfolg wird hier durch regelmäßige Überkorrekturen nichtig gemacht.

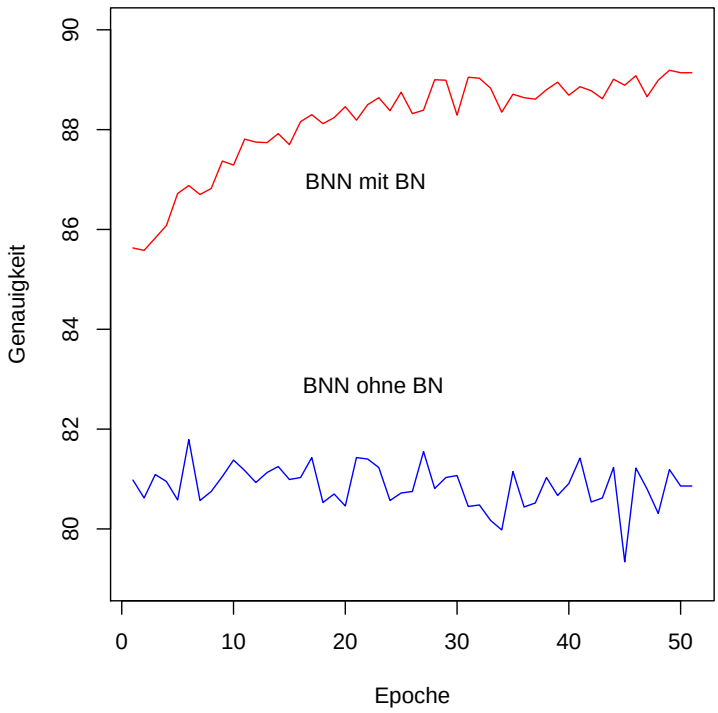


Abbildung 3.3: Training mit und ohne BatchNorm

Der Versuch in Abbildung 3.3 zeigt einen Vergleich des Netzwerkes mit und ohne *BatchNorm*. Hierfür wurde ein Netzwerk für 50 Epochen trainiert. Nach jeder Epoche wurde die Genauigkeit getestet. Anschließend wird das Training fortgesetzt. Um eine bessere Vergleichbarkeit zu erzielen, wurden die Bilder über die Schwellwert-Methode binarisiert.

Wie in Abbildung 3.3 zu sehen ist, performt das Netzwerk mit *BatchNorm* Schicht zu jeder Trainingsdauer besser als ohne *BatchNorm* Schicht. Während in den ersten 20 Epochen mit *BatchNorm* ein klarer Aufwärtstrend zu erkennen ist, stagniert die Genauigkeit ohne *BatchNorm* zwischen 79% und 82%. Hier sind besonders gut die starken Einbrüche nach Unten zu erkennen, die mit *BN* deutlich abgeschwächt sind. Diese zeigen besonders deutlich den starken Genauigkeitsverlust, der durch *exploding gradients* verursacht wird. Die Konsequenz ist, dass eine längere Trainingsdauer keine Verbesserung des Netzwerkes mehr bedeutet, was bedeutet, dass die Konvergenz-Schwelle bei < 82 erreicht ist. Ab

dieser Schwelle verbessert sich das Netzwerk ohne *BN* nicht mehr.

3.4 Auswertung der letzten Schicht

Um am Ende das Ergebnis des Netzwerkes auszuwerten, müssen die Ergebnisse des letzten *Linear-Layers* normalisiert werden. Diese Normalisierung entscheidet, ob ein Neuron feuert oder nicht. Für die Normalisierung der Daten wird hier die *logsoftmax(x)* Funktion verwendet. Durch *softmax(x)* werden die Aktivierungen der letzten Schicht so normalisiert, dass ihre Summe Eins ergibt. Dies ist mit Wahrscheinlichkeiten zu vergleichen, mit der das Bild die jeweilig zugeordnete Zahl widerspiegelt. Um bessere Ergebnisse in Kombination mit der Verlustfunktion, *negative log likelihood loss*, zu erzielen, wird die *logsoftmax*-Funktion verwendet.
Da bei der Anwendung des MNIST-Datensatzes immer nur das Neuron ausgewählt werden sollte, da immer nur eine Zahl auf einem Bild abgebildet ist, wird bei der Auswertung über die *argmax*-Funktion das aktivste Neuron ausgewählt. Dieses entspricht dann der Zahl, die am wahrscheinlichsten abgebildet ist.

3.5 Verluste durch binäre Linear-Layer

Durch die binarisierung der *Linear-Layer* ist zu vermuten, dass diese, im Vergleich zu normalen *Linear-Layer*, etwas schlechter performen. Dies ist der Fall, da die Anzahl der möglichen Kantengewichte stark, auf Null und Eins, eingeschränkt ist.

Durchgang	binarär	normal
1	88.29	97.43
2	87.32	96.98
3	87.19	97.2

Abbildung 3.4: Verlustmessung für den *Linear-Layer*

Trainiert wurde hier das gleiche Netzwerk, ein mal mit binären *Linear-Layern*, das andere mal mit normalen *Linear-Layer*. Jedes Netzwerk wurde für 50 Epochen trainiert, bevor die Genauigkeit ausgewertet wurde. Um sicher zu gehen, ob die Genauigkeit gegen

diesen Wert konvergiert, wurde jede Messung drei mal wiederholt.
Wie in Abbildung 3.4 zu sehen, leidet die Genauigkeit des Netzwerkes beachtlich unter der Binarisierung der *Linear-Layer*. Der Mittelwert für das Training mit normalem *Linear-Layer* ist hierbei $97,2\%$, während bei binären Schichten ein Durchschnitt von $87,6\%$ erreicht wird. Es ist klar zu sehen, dass die Einschränkung der Gewichte auf Null und Eins und der damit einhergehende Granularitätsverlust, sich stark auf die Genauigkeit des Netzwerkes, bei gleicher Größe, auswirken.

Kapitel 4

Training des BNNs

Wie bereits in Kapitel 2.4 beschrieben, werden Netzwerke über eine rechenintensive Annäherung der Kantengewichte an das optimale Ergebnis trainiert. Diese Konvergenz in Richtung eines besseren Ergebnisses kann hierbei, selbst bei schlechtem Training, meist erreicht werden, da selbst marginale Verbesserungen eine Auswirkung haben. Da *BNNs* jedoch die stärkste Form von quantisierten Netzwerken sind, können hier keine stetigen Änderungen an den Kantengewichten vorgenommen werden. Somit ist Abstimmung der Trainingsparameter essentiell.

4.1 Lernrate

Die Lernrate ist ein zentrales Parameter beim Training von Neuronalen Netzwerken. Sie gibt die Rate an, mit der Kantengewichte in einem Durchlauf angepasst werden. Je höher die Lernrate also ist, desto schneller passt sich das Model an gerade trainierte Daten an. Bei der Wahl der Rate sind die Probleme des *overfitting* und *underfitting* zu beachten.
Als *undefitting* bezeichnet man eine zu geringe Anpassungsfähigkeit des Netzwerkes, ausgelöst über zu kleine Netze oder zu niedrige Lernrate. Ist die Lernrate zu klein, kann der Error nicht reduziert werden, die Genauigkeit nimmt nur, wenn überhaupt, sehr langsam zu. Das Netz kann aus Daten keine Lernerfolge ziehen.
Overfitting ist eine zu schnelle Anpassung des Models. Ist zum Beispiel die Lernrate zu hoch, kann die optimale Genauigkeit des Netzwerkes nicht erreicht werden, da die Anpassungen pro Error-Berechnung zu groß sind. Hier wird das Optimum immer, durch eine Überkorrektur, übersprungen und kann nie erreicht werden[Smi18]. Diese Probleme

sind hier analog zu *vanishing-* beziehungsweise *exploding gradient*.

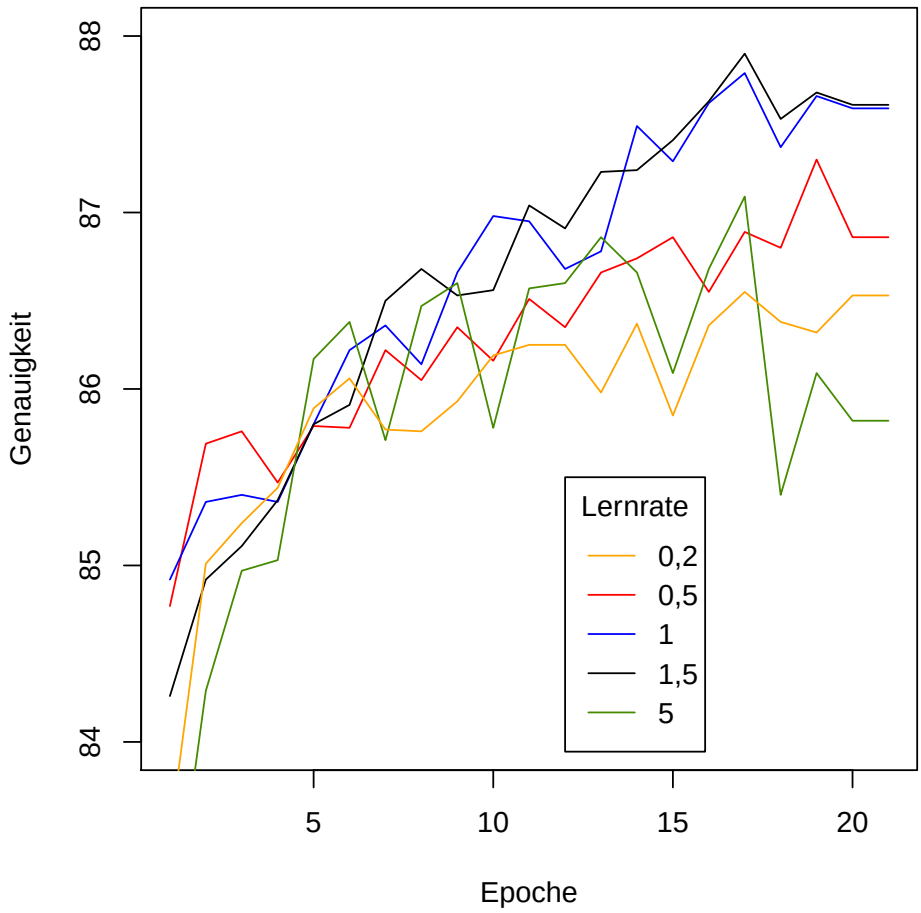


Abbildung 4.1: Training mit verschiedenen Lernraten

Zur Ermittlung der optimalen Lernrate haben wir unser Netzwerk für 20 Epochen mit verschiedenen Lernraten trainieren lassen. In Abbildung 4.1 sind die Ergebnisse mit einer Messung der Genauigkeit nach jeder Epoche dargestellt. Bei einer Lernrate von 5 ist deutlich eine Überkorrektur ab Generation 17 zu erkennen. Hier werden zu starke Anpassungen vorgenommen, weshalb sich die Genauigkeit vom Optimum entfernt. Lernraten unter Eins hingegen nähern sich dem Optimum erst gar nicht genug an. Hier ist ein potentieller *vanishing gradient* zu erkennen, da der Error nicht signifikant sinkt. Die

Lernraten 1 und $1,5$ liefern hier die besten Ergebnisse. Da das Training mit einer Rate von $1,5$ weniger große Ausschläge zeigt, wurde sich für diese Lernrate entschieden.

4.2 Batchgröße

Die Batchgröße hat Einfluss auf mehrere Punkte des Netzwerkes. Zum Einen entscheidet sie, wie viele Daten durch das Netzwerk laufen, bevor der Error gemessen wird und die Gewichte aktualisiert werden. Somit ist sie Quasi eine Rate der Error-Berechnung. Zum Anderen wirkt sich dies auf die BatchNorm Schicht aus. Je größer hier die Batchgröße ist, desto stärker ist die Normalisierung durch die BatchNorm.

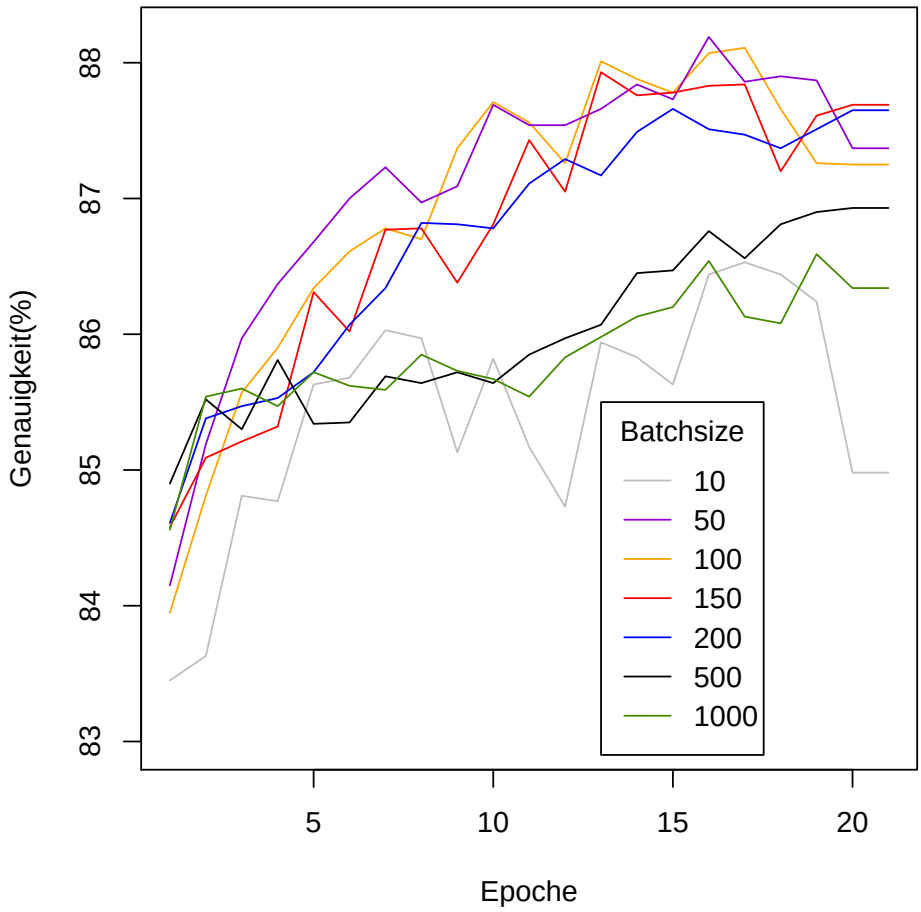


Abbildung 4.2: Training mit verschiedenen Batchgrößen

In Abbildung 4.2 zu sehen ist das immer gleiche Netzwerk, lediglich mit angepasster Batchgröße. Dieses Netzwerk wurde für 20 Epochen beobachtet.
Auffällig ist die stark ausschlagende Kurve bei einer Batchgröße von 10. Hier ist der Wert zu gering, weshalb der Error oft berechnet wird. Da außerdem wenig über die BatchNorm normalisiert wird, kommt es zu hohen Error-Werten und starken und häufigen Anpassungen des Netzwerkes.
Bei einer Batchgröße ≥ 500 sind weniger starke Ausschläge zu beobachten. Allerdings ist hier zu beobachten, dass das Netzwerk nicht mehr so effizient trainiert wie bei kleineren

Batchgrößen. Hier könnte die Normalisierung und damit die Generalisierung bereits zu stark sein um aus einzelnen Daten überhaupt zu lernen. Außerdem wird hier der Error wesentlich seltener berechnet und das Netzwerk ist somit wesentlich unagiler. Es kommt zum *underfitting*.
Bei Batchgrößen 50-200 ist keine besondere Abweichung zu erkennen, wobei 50 und 100 am besten performen.

Batchgröße	10	50	100	150	200	500	1000
Zeit (s)	30,68	11,33	8,76	7,95	7,63	6,66	6,39

Abbildung 4.3: Zeit pro Epoche für verschiedene Batchgrößen

Ein weiterer, praxisrelevanter, Faktor ist, dass das Training mit größeren Batchgrößen schneller geht. Da mehr Daten parallel bearbeitet werden und eine GPU parallele Rechnungen gut performen kann, ist die gleiche Anzahl an Daten schneller abgearbeitet. In Abbildung 4.3 sind die Zeiten für eine einzige Epoche, abhängig von der jeweiligen Batchgröße, gelistet. Es ist zu erkennen, dass besonders kleine Batchgrößen bei einer Erhöhung Zeiteinsparungspotential haben. Da die Werte 50-200 bei der Genauigkeit am besten abgeschnitten haben und $t(100) - t(50) < t(200) - t(100)$, ist hier die Laufzeitverbesserung von 50 auf 100 eine Sinnvolle Einsparung. Bei Batchgrößen ≥ 150 lohnt sich die marginale Zeiteinsparung im Vergleich zur schlechteren Genauigkeit nicht mehr.
Deshalb haben wir uns für ein Batchgröße von 100 zur optimalen Werteberechnung entschieden.

Kapitel 5

Binarisierung

5.1 Binarisierung der Bilder

Das Binarisieren der importierten Bilder stellt die erste Hürde im Binarisierungsprozess des *NNs* auf. Wir arbeiten in unserem *BNN* mit *Grey-Scaled*-Bildern, d.h. wir verarbeiten nur einen α*-Kanal* anstatt drei *RGB-Kanälen*. Dieser α*-Kanal* bewegt sich jedoch in einem Wertebereich von $\{0 - 255\}$ bzw. $(0, 1)$ mit einer Granularität von $\frac{1}{255}$.

Unsere Aufgabe bei der Binarisierung ist es also mit möglichst wenig Informationsverlust eine Funktion $\mathbb{Q}_{0-1} \to \{0, 1\}$ zu entwickeln, die unsere importierten Bilder binarisiert.

5.1.1 Threshold

Im Folgenden betrachten wir den Wertebereich der Pixel als zwischen $\{0 - 255\}$ liegend. Der erste Ansatz für eine Binarisierungsfunktion betrachten wir Thresholds. Diese Funktion flippt für ein fest gewähltes $n \in \{0..255\}$ alle Bits, die den Schwellenwert n überschreiten, auf 1 und alle anderen auf 0.

Der Vorteil dieser Methode ist, dass man eine sehr kostengünstige Binarisierung erhält (150ms pro 100 Bilder). Man kann durch das fest gewählte n die *Inbuild*-Tensor-Funktionen verwenden und muss nicht manuell über die einzelnen Pixel iterieren. Dies

ist vorallem wichtig, da man mit 60.000 Bildern je 28×28 Pixeln sehr viele Binarisierungsschritte je Epoche erhält.

Der große Nachteil an der Threshold-Methode ist, dass man kompromisse machen muss. Es gibt nicht einen guten Threshhold, da für jeden gewählten Threshold bei einer Teilmenge der Bilder zuviel, bzw. zu wenig gefiltert wird. (Siehe Abbildung 5.1 bzw. 5.2) In

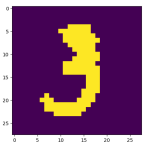


Abbildung 5.1: 3, Threshold 100

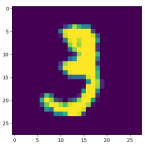


Abbildung 5.2: 3, not binarized

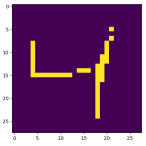


Abbildung 5.3: 4, Threshold 200

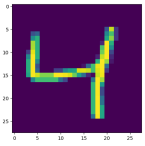


Abbildung 5.4: 4, not binarized

diesem Vergleich sieht man sehr gut, dass ein Threshold, der für ein Bild zu tief gesetzt ist, d.h. es werden zu wenig Pixel gefiltert nicht erhöht werden kann, da er sonst für ein anderes Bild zu hoch gesetzt ist(es werden zu viele Pixel gefiltert).

5.1.2 Probability

Eine Lösung für dieses Problem bietet die *Probability-Transformation*. Bei dieser Methode betrachten wir den Wert eines einzelnen Pixels als die Wahrscheinlichkeit, dass dieser Pixel in der Binarisierung mit 1 belegt wird. So werden Pixel mit den Wert 255 immer mit 1 belegt und Pixel mit dem Wert 130 ungefähr 50% der Zeit.

Nachteil dieser Methode ist die Effizienz. Da wir nun nicht mehr mit einem statischen Threshold arbeiten, müssen wir manuell über die Pixel iterieren. Dabei verschlechtern wir uns im Zeitlichen um einen Faktor 20. Selbst nach großen Optimierungen in Form von

Tensor-Umwandlung in Numpy-Arrays und Numba-Integration dauert die *Probability-Transformation* doppelt solange wie die *Threshold-Transformation*. Zu dem konvergiert die Probability Methode erst nach Rund 600 Epochen wohingegen die Threshold-Methode nur rund 100 Epochen benötigt. Somit ist das Trainieren mit der *Probability-Transformation* deutlich teurer.

Der große Vorteil dieser Methode ist jedoch die Informationserhaltung. Wir arbeiten bei der *Probability-Transformation* mit sogenannten *Repetitions*. Diese durchlaufen die gleiche Generation mit dem gleichen *Trainingset*, welche jedoch neu binarisiert wurden. Das heißt, dass man bei 50 Repetitions jede Generation 50 mal, mit 50 unabhängig binarisierten *Trainginsets* durchläuft.
Dadurch erhalten wir ein Mittel für jedes Bild, welches gegen die Originalversion konvergiert. (Siehe Abbildung 5.5 bzw 5.6)

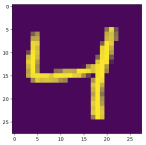


Abbildung 5.5: 4, overlapped 50 prob-trans

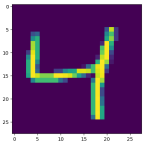


Abbildung 5.6: 4, not binarized

5.1.3 Evaluation

Um die verschiedenen Methoden zu Vergleichen, haben wir beide Transformationen und eine Referenzinstanz für 600 Epochen, bzw. 20 Epochen und 30 Repetitions, trainieren lassen.

Run	Non-Binarized	threshold	prob
1	91.99%	89.24%	92.52%
2	91.99%	89.24%	91.82%
3	91.99%	89.24%	92.56%
4	91.99%	89.24%	91.02%
avg	**91.99%**	**89.24%**	**91.98%**

Abbildung 5.7: Vergleich

Es ist gut zu erkennen, dass bei der Binarisierung mit der *Threshold*-Methode über 2% Genauigkeit verloren wird. Zudem sind unsere Referenzinstanz und die *Threshold*-Methode bereits konvergiert, heißt sie erzielen keine Verbesserung mehr.
Die *Probability-Transformation* schwankt jedoch noch in der Genauigkeit. Dies liegt überwiegend an der Varianz in der Binarisierung bedingt durch die Wahrscheinlichkeit. Trotz dieser Varianz ist zu erkennen, dass das Mittel der *Probability-Transformation* gegen unsere Referenzinstanz konvergiert. Somit haben wir eine Transformation, bei der wir durch zeitliche Einbuße keine Genauigkeitsverluste verzeichnen müssen.

Nun sind die zeitlichen Einbuße, durch den Faktor 12 beschrieben, so groß, dass es sich nicht lohnt bei dem Testen von Parametern oder Ausprobieren von Konzepten mit der *Probability-Transformation* zu arbeiten. In diesen Fällen ist es effizienter die 2% Genauigkeitsverlust in Kauf zu nehmen und deutlich schnelle arbeiten zu können. Es hilft uns im Finalen Netz aber durchaus unsere Hürde von 90% zu überschreiten.

Kapitel 6

Export

Nachdem das Netzwerk nun trainiert wurde, müssen die Ergebnisse, die Kantengewichte und Schwellwerte der Neuronen, nun exportiert werden. Im Folgenden sollen diese dann in den BNN-Beschleuniger Baustein importiert und verwendet werden. Da der Import in VHDL stattfindet, eignen sich hier simple Formate, sprich eine einfache Textdatei. Diese kann dann, Zeichen nach Zeichen, von dem Import-Buffer eingelesen und in einer Matrix gespeichert werden.

6.1 Export der Kantengewichte

Da es sich bei unserem Netzwerk um ein *FullyConnected Neural Network* handelt, ist insbesondere jedes Neuron mit jedem Neuron der Folgenden Schicht verbunden. Bei unserem BNN ergibt sich also folgende Kantenanzahl

$$784 \cdot 500 + 500 \cdot 1024 + 1024 \cdot 1024 = 1.952.576$$

Diese Gewichte müssen alle mit möglichst wenig Mehrkosten in die Datei geschrieben werden. Da es sich bei den Gewichten lediglich um binäre Werte, somit als Einsen und Nullen darstellbar, handelt, ist kein Trennzeichen zwischen den Gewichten notwendig. Die Gewichte sind außerdem, trivialer Weise, präfixfrei und können fortlaufend in die Datei geschrieben werden.
Um die Gewichte zu extrahieren, wird zuerst über jeden *Layer* iteriert. In jedem *Layer* wird nun jedes Neuron abgelaufen. Jedes dieser Neuronen hat nun jeweils eine Kante zu jedem Neuron in der nachfolgenden Schicht. Hier wird ebenfalls über alle Kanten iteriert und das jeweilige Gewicht wird hinten an eine Variabel an gehangen. Ist nun ein *Layer* fertig, wird der Inhalt der Variable, welche als Zwischenspeicher dient, in die Datei *weights.txt* geschrieben. So wird für jede Schicht ein IO-Zugriff gemacht.

6.2 Ausblick: Export der Schwellwerte

Für eine vollständige Ausführung des Netzwerkes, müssen außerdem die Schwellwerte der Neuronen exportiert werden. Zum Zeitpunkt der Fertigstellung dieser Ausarbeitung ist die Implementierung jedoch noch ausstehend.

Literatur

[CB16] Matthieu Courbariaux und Yoshua Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830.

[IS15] Sergey Ioffe und Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Hrsg. von Francis Bach und David Blei. Bd. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, S. 448–456. URL: https://proceedings.mlr.press/v37/ioffe15.html.

[Kim+20] Hyungjun Kim u. a. “Improving Accuracy of Binary Neural Networks using Unbalanced Activation Distribution”. In: *CoRR* abs/2012.00938 (2020). arXiv: 2012.00938. URL: https://arxiv.org/abs/2012.00938.

[RHW86] D. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986).

[SBN19] Eyyüb Sari, Mouloud Belbahri und Vahid Partovi Nia. “A Study on Binary Neural Networks Initialization”. In: *CoRR* abs/1909.09139 (2019). arXiv: 1909.09139. URL: http://arxiv.org/abs/1909.09139.

[Smi18] Leslie N. Smith. “A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay”. In: *CoRR* abs/1803.09820 (2018). arXiv: 1803.09820. URL: http://arxiv.org/abs/1803.09820.

Kapitel 7

Anhang

7.1 Probability-Binarized Images