

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

17. September 2021

# Inhaltsverzeichnis

Abbildungsverzeichnis	iii
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
<b>2 Neuronale Netzwerke</b>	<b>3</b>
2.1 Neuronen . . . . .	4
2.2 Schichten und Kanten . . . . .	4
2.3 Aktivierungsfunktionen . . . . .	5
2.4 Training . . . . .	7
2.4.1 Ablauf . . . . .	7
2.4.2 Backpropagation . . . . .	8
<b>3 Das BNN</b>	<b>12</b>
3.1 Aktivierungsfunktion . . . . .	12
3.2 Binärer Linear-Layer . . . . .	13
3.3 BatchNorm . . . . .	14
3.4 Auswertung der letzten Schicht . . . . .	16
3.5 Verluste durch binäre Linear-Layer . . . . .	16
<b>4 Training des BNNs</b>	<b>18</b>
4.1 Lernrate . . . . .	18
4.2 Batchgröße . . . . .	20
<b>5 Binarisierung</b>	<b>23</b>
5.1 Binarisierung der Bilder . . . . .	23
5.1.1 Threshold . . . . .	23
5.1.2 Probability . . . . .	24
5.1.3 Evaluation . . . . .	25

## *Inhaltsverzeichnis*

<b>6</b>	<b>Export</b>	<b>27</b>
6.1	Export der Kantengewichte . . . . .	27
6.2	Ausblick: Export der Schwellwerte . . . . .	28
<b>7</b>	<b>Fazit</b>	<b>29</b>
	<b>Literatur</b>	<b>30</b>
<b>8</b>	<b>Anhang</b>	<b>i</b>
8.1	Probability-Binarized Images . . . . .	i

# Abbildungsverzeichnis

2.1	Struktur eines neuronalen Netzwerks . . . . .	3
2.2	Berechnung eines Neuronenwertes mit der <i>tanH</i> Aktivierungsfunktion . .	5
2.3	ReLU . . . . .	6
2.4	Sigmoid . . . . .	6
2.5	Softmax . . . . .	6
2.6	<i>tanH</i> . . . . .	6
2.7	Neuronenkette: Backpropagation . . . . .	9
2.8	Zu niedrige Lernrate . . . . .	11
2.9	Zu hohe Lernrate . . . . .	11
3.1	Die Vorzeichenfunktion . . . . .	12
3.2	Aufbau des binären Linear-layer . . . . .	13
3.3	Training mit und ohne BatchNorm . . . . .	15
3.4	Verlustmessung für den <i>Linear-Layer</i> . . . . .	16
4.1	Training mit verschiedenen Lernraten . . . . .	19
4.2	Training mit verschiedenen Batchgrößen . . . . .	21
4.3	Zeit pro Epoche für verschiedene Batchgrößen . . . . .	22
5.1	3, Threshold 100 . . . . .	24
5.2	3, not binarized . . . . .	24
5.3	4, Threshold 200 . . . . .	24
5.4	4, not binarized . . . . .	24
5.5	4, overlapped 50 prob-trans . . . . .	25
5.6	4, not binarized . . . . .	25
5.7	Vergleich . . . . .	26

# 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 beliebig vielen *versteckten* Schichten, welche die eigentliche Arbeit des Netzwerks verrichten und für die Erkennung von Merkmalen bei der Klassifizierung von Eingabedaten zuständig sind. Die Größe der Eingabeschicht wird durch die Anzahl der Faktoren bestimmt, welche bei der Klassifizierung berücksichtigt werden sollen. Die Größe der Ausgabeschicht hingegen wird durch die Anzahl der erwarteten Klassen bestimmt, in welche klassifiziert werden soll. Die Größe und Anzahl der dazwischen liegenden Schichten wird je nach Anforderung und Gegebenheiten individuell ermittelt. Je größer das Netzwerk desto höher sind die Anforderungen an die benötigte Hardware um dieses zu betreiben. Schwächer Hardware oder Einschränkungen bezüglich der Energieversorgung führen demnach häufig zur Verwendung kleinerer Netzwerke, auch wenn diese typischerweise im Vergleich zu einem größeren Netzwerk eine geringere Genauigkeit bieten.

### 1.1 Motivation

Die Größe eines Netzwerks kann beim Entwurf direkt beeinflusst werden. Beim Fokus auf den Einsatz von neuronalen Netzen auf klassisch langsamen Systemen liegt der Schwerpunkt auf der Balance zwischen Performanz und Genauigkeit. Auf IoT-Geräten oder

mobilen Plattformen finden klassische neuronale Netzwerke aufgrund der Hardwareanforderungen daher nur eingeschränkt Nutzen. 2016 veröffentlichten M. Courbariaux und Y. Bengio [CB16] eine wissenschaftliche Arbeit und stellen dort *binarisierte neuronale Netzwerke* (kurz BNNs) vor. Diese erlauben im Vergleich zu klassischen neuronalen Netzwerken eine theoretische 32-fachen Performanzsteigerung. Dieses Versprechen folgt aus der Tatsache, dass klassische neuronale Netze mit 32-Bit Zahlen rechnen, während das vorgestellte BNN mit einem Bit auskommt. Somit können im Optimalfall 32 1-Bit Rechenoperationen parallel und in nur einer Instruktion ausgeführt werden. Dennoch liege die Genauigkeit von BNNs nur knapp unter derer klassischer Netzwerke. Das BNN ermöglicht dank dieser Eigenschaften den Einsatz von neuronalen Netzen auf vergleichsweise schwacher Hardware mit nur geringen Genauigkeitseinbußen.

In dieser Ausarbeitung wird eine Einführung in die Funktionsweise von neuronalen Netzwerken gegeben und anschließend der Entwurf eines binarisierten Netzwerks dokumentiert. Dieses wird für eine einfache Handschriftenerkennung der Zahlen null bis neun konzipiert und mit dem MNIST-Datensatz trainiert. 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.

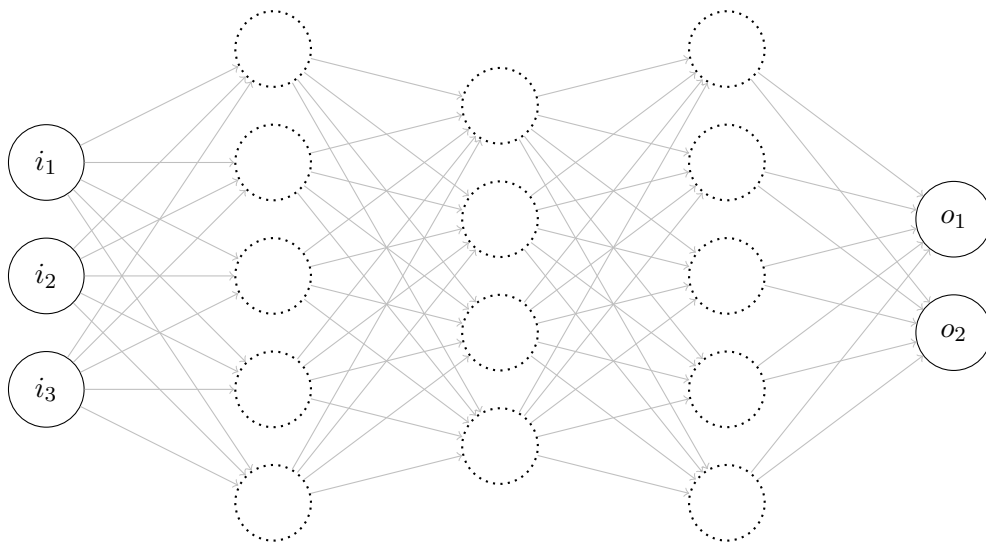


Abbildung 2.1: Struktur eines neuronalen Netzwerks

## 2.1 Neuronen

Unter einem Neuron versteht sich in neuronalen Netzen lediglich ein Knoten, in dem üblicherweise eine 32-Bit Zahl hinterlegt ist. Die Semantik dieses Wertes hängt von dem jeweiligen Typ der Schicht ab, in dem sich das Neuron befindet. Die Werte aller Neuronen, mit Ausnahme jener aus der Eingabeschicht, setzen sich aus den Werten aller Neuronen der direkt davor liegenden Schicht zusammen. Das genaue Vorgehen bei der Wertermittlung hängt vom Entwurf des Netzwerks und den dort verwendeten Aktivierungsfunktionen ab.

## 2.2 Schichten und Kanten

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

Die Kanten eines Netzwerks sind gewichtet, welche initial netzwerkabhängig generiert, manuell zugewiesen oder durch zuvor erlernte Daten bestimmt werden. Beim Trainieren eines Netzwerks werden diese in jeder Lerniteration (auch *Epoch* genannt) justiert, während sie im Betrieb keine Änderungen mehr erfahren. Der Entwurf eines Netzwerks bestimmt die Performanz dessen und die damit verbundenen Hardwareanforderungen, nicht jedoch die Genauigkeit. Diese wird überwiegend von den Kantengewichten bestimmt, daher ist das Ziel des Trainings diese zu optimieren und damit die Genauigkeit des Netzwerks bei gleicher Performanz zu erhöhen. Der Unterschied zwischen einem genauen und einem ungenauen Netzwerk ist daher oft nur das investierte Training.

Kantengewichte wirken sich maßgeblich auf die Werteberechnung 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.



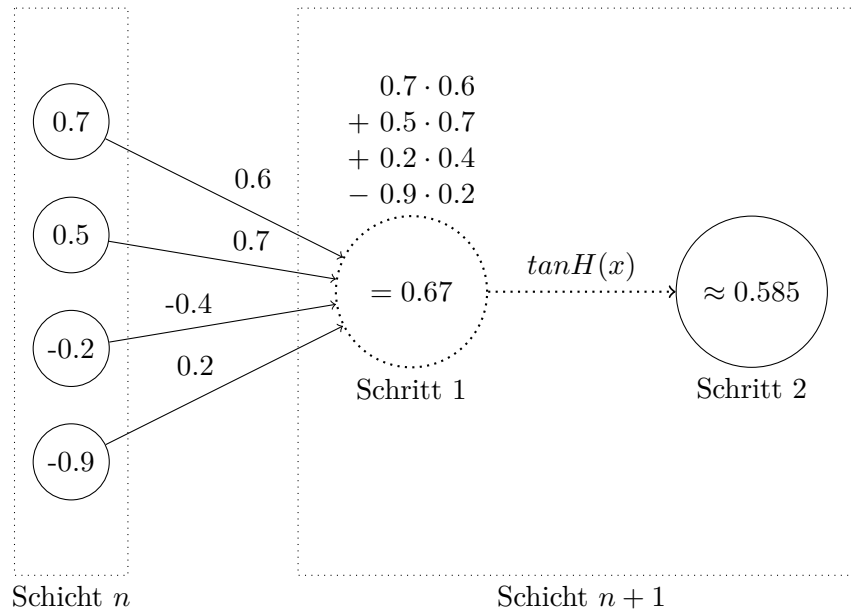


Abbildung 2.2: Berechnung eines Neuronenwertes mit der  $\tan H$  Aktivierungsfunktion

## 2.3 Aktivierungsfunktionen

Im zweiten Schritt wird die zuvor bestimmte Produktschritte durch eine Funktion modifiziert. Sogenannte *Aktivierungsfunktionen* können beliebig gewählt werden, müssen jedoch offensichtlich alle möglichen Eingaben auf einen Wert abbilden können. Aktivierungsfunktionen erwarten genau einen Eingabewert und bilden diesen auf genau einen Wert ab. Ein einfaches Beispiel ist die Identitätsfunktion  $\text{id}(x) = x$ . Die eingesetzten Aktivierungsfunktionen können schichtweise variieren. Einmal gewählt sind diese jedoch für das entworfene Netzwerk fest.

Diese beiden 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  $\tan H$ -Aktivierungsfunktion verwendet.

Die Wahl der Aktivierungsfunktion beeinflusst die möglichen Werte, die Neuronen im Netzwerk annehmen können. Um eine Wertexplosion zu vermeiden (z.B. bei Verwendung von  $\text{ReLU}(x) : \max(0, x)$  als Aktivierungsfunktion), können Normalisierungsschichten

verwendet werden, dessen Aufgabe die Einschränkung von Neuronenwerten auf einen erwünschten Zielbereich ist. Es existieren jedoch auch Aktivierungsfunktionen, die diese sogenannte *Squashing*-Eigenschaft direkt besitzen. Darunter zählt auch die im vorherigen Beispiel verwendete Funktion  $\tanh$ , bei der alle Eingaben auf den Zielbereich  $[-1,1]$  abgebildet werden.

Im Folgenden sind einige Aktivierungsfunktionen abgebildet, welche üblicherweise in neuronale Netzwerken eingesetzt werden.

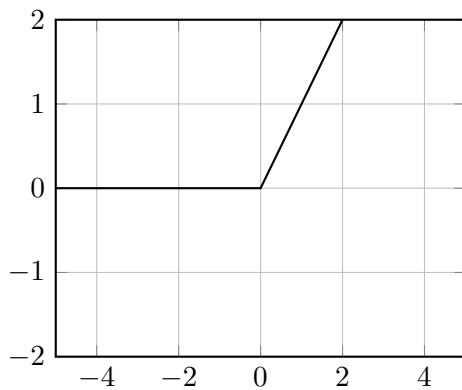


Abbildung 2.3: ReLU

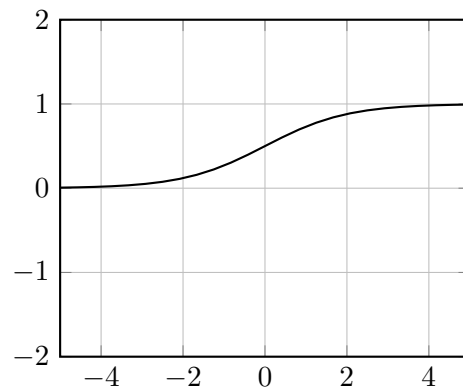


Abbildung 2.4: Sigmoid

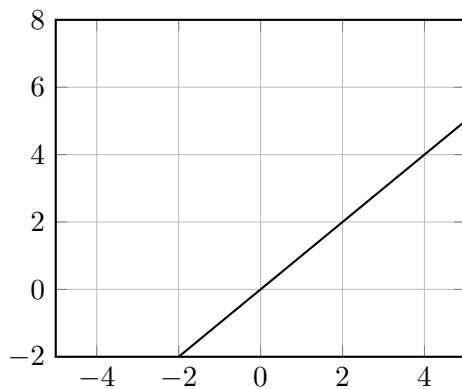


Abbildung 2.5: Softmax

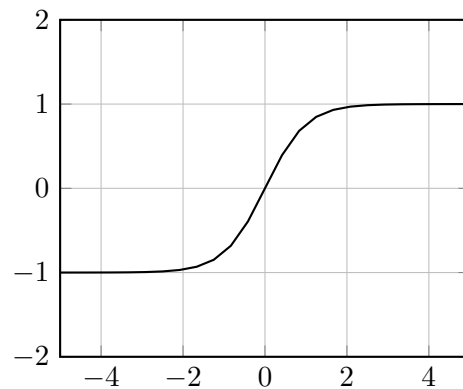


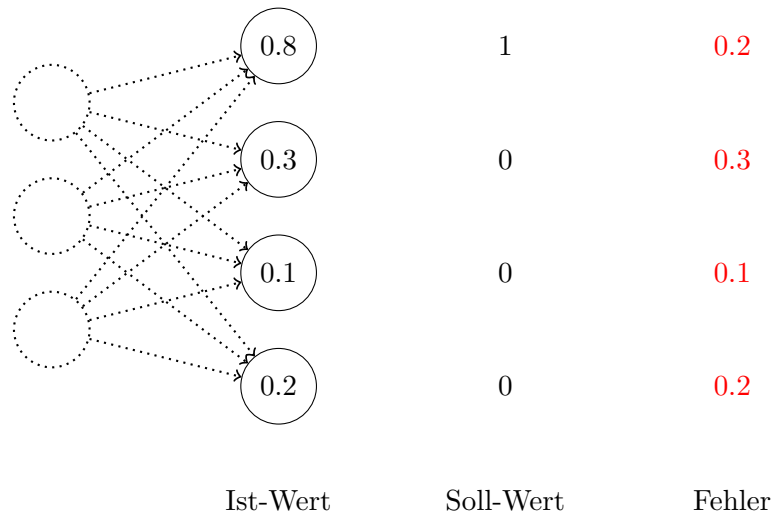
Abbildung 2.6: tanH

## **2.4 Training**

Wie zuvor erwähnt, liegen die Stärken und Schwächen eines neuronalen Netzwerks neben dem Aufbau und der Anzahl der verwendeten Schichten sowie der gewählten Aktivierungsfunktionen vor allem in den Kantengewichten. Diese zu optimieren kann ein langwieriger und rechenintensiver Prozess sein. Kleine Änderungen in der Struktur des Netzwerks können große Teile von zuvor erlernten Gewichten unbrauchbar machen, daher werden Netzwerke häufig an sich selbst trainiert. Das bedeutet, dass beim Trainieren des Netzwerks die initialen Kantengewichte keine empirischen Daten aus anderen (ähnlichen) Netzwerk verwenden, selbst wenn beide Netzwerke auf dem identischen Datensatz trainiert werden.

### **2.4.1 Ablauf**

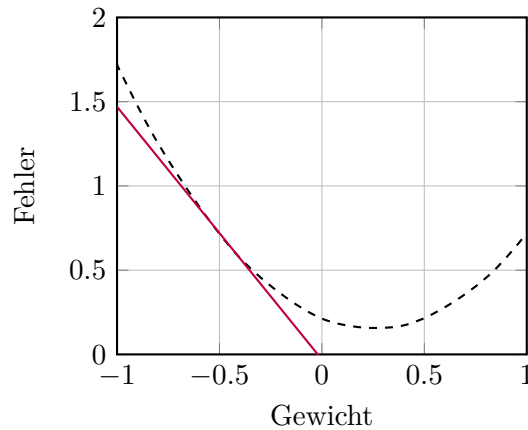
Beim Training werden wiederholt unterschiedliche Eingaben in das Netzwerk gegeben, zu denen die korrekten Klassifizierungen bekannt ist. In der letzten Schicht des Netzwerks, der Ausgabeschicht, wird die vom Netzwerk bestimmte Klassifizierung mit der tatsächlichen Klassifizierung verglichen. Konkret werden die vom Netzwerk bestimmten Neuronenwerte in der Ausgabeschicht mit den erwarteten Werten verglichen. Die absolute Differenz dieser Werte wird als Fehler bezeichnet. Beim Training wird diese Information in jedem Durchlauf verwendet, um die Kantengewichte so anzupassen, dass der Fehler verringert, oder im Optimalfall ganz zu eliminiert wird (Fehler = 0). In der folgenden Abbildung ist ein Beispiel einer Fehlerbestimmung 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 Gewichts Anpassung, um die Ausgabewerte der jeweiligen Lerneinheit in die gewünschten Richtungen zu verändern.

Um eine Anpassung an einem Kantengewicht vorzunehmen, muss zuerst festgestellt werden, wie die errechneten Fehler sich bei Änderungen an einem bestimmten Gewicht verhalten. Dazu wird die sogenannte *Slope* ermittelt, eine Kurve auf der das Kantengewicht im Verhältnis zum Fehler steht. Anschließend wird die Steigung dieser Kurve an der aktuellen Stelle, also dem aktuellen Kantengewicht errechnet. Offensichtlich lässt sich aus diesem Ergebnis, 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. Eine Erhöhung des Kantengewichts führt demnach zu einer Verringerung des gemessenen Fehlers.



Diese Anpassungen werden in jeder Trainingseinheit für alle Kantengewichte durchgeführt. Die Bestimmung der Stärke der Anpassung wird als Lernrate bezeichnet. Die Ermittlung der Fehlerkurve kann mathematisch durch das Verwenden der sogenannten *Chainingrule* [RHW86] effizient ermittelt werden. Zur Veranschaulichung dieser Regel zeigt die folgende Abbildung einen Ausschnitt aus einem einfachen neuronalen Netzwerk mit drei Schichten.

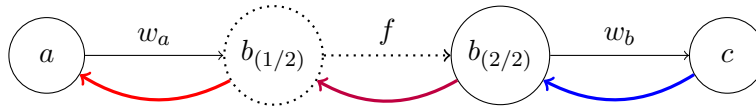


Abbildung 2.7: Neuronenkette: Backpropagation

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:

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

Mit  $a$ ,  $w_a$  bekannt und  $f'$  leicht ermittelbar wird auch deutlich, warum die üblich verwendeten Aktivierungsfunktionen häufig stetig sind und einfach abzuleiten sind. Offensichtlich lässt sich dieses Vorgehen auf jede beliebig lange Neuronenkette ausweiten und erlaubt somit die Ermittlung der Richtungen aller notwendigen Gewichtsadjustierungen zur Verringerung des Fehlers. 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. Das Erreichen dieses Tals kann erschwert werden, wenn die verwendete Lernrate beim Training entweder zu hoch oder zu niedrig ist. Daraus können zu starke oder zu schwache Anpassungen der Gewichte folgen. Ersteres führt zum hin- und herspringen des ermittelten Gewichts auf der Fehlerkurve sodass sich dem Tal nicht mehr angenähert wird. Letzteres hingegen führt dazu, dass der Umfang der Korrektur stagniert und außerhalb des optimalen Gewichts zum Stillstand kommt. Siehe dazu auch Abbildungen 2.8 und 2.9.

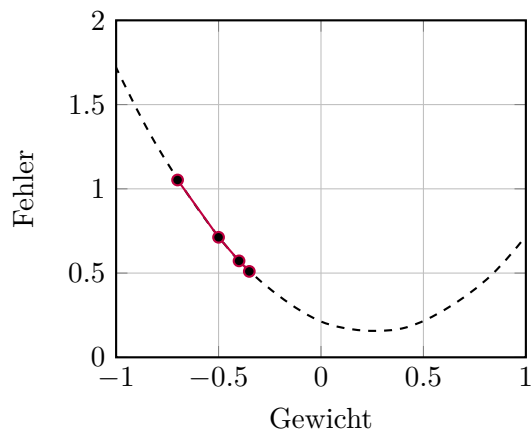


Abbildung 2.8: Zu niedrige Lernrate

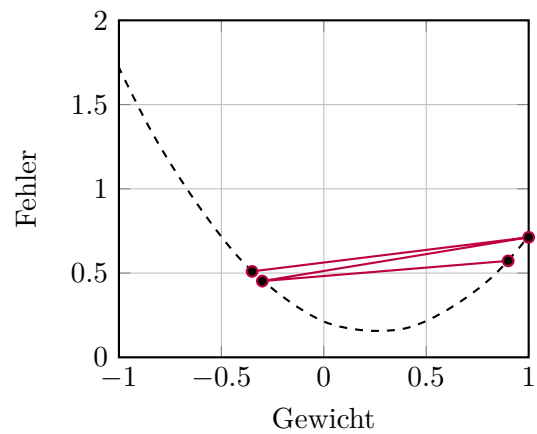


Abbildung 2.9: Zu hohe Lernrate

Das Training eines neuronalen Netzes wird aufgrund der Rechenintensität und den notwendigen Zeitaufwand für gewöhnlich nicht bis zum Optimum durchgeführt. Stattdessen wird auf eine bestimmte Genauigkeit hingearbeitet und bei Erfüllung dieser die erlernten Kantengewichte gespeichert und zur Wiederverwendung 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 Nachteile.

Für BNNs ist hier die  $\text{sign}(x)$  Funktion eine populäre Wahl. Diese überführt die Aktivie-

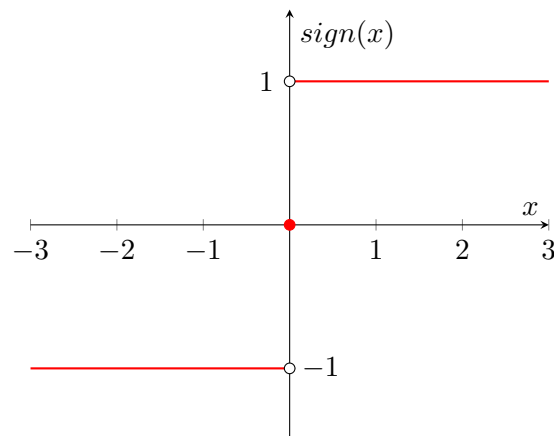


Abbildung 3.1: Die Vorzeichenfunktion

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.

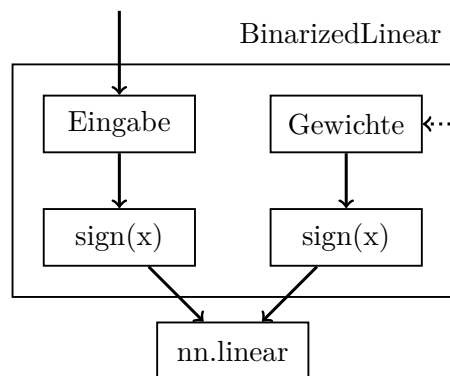


Abbildung 3.2: Aufbau des binären Linear-layer

Wie in Abbildung 3.2 zu erkennen, erhält die 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 Standardabweichung 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 *expoding-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.

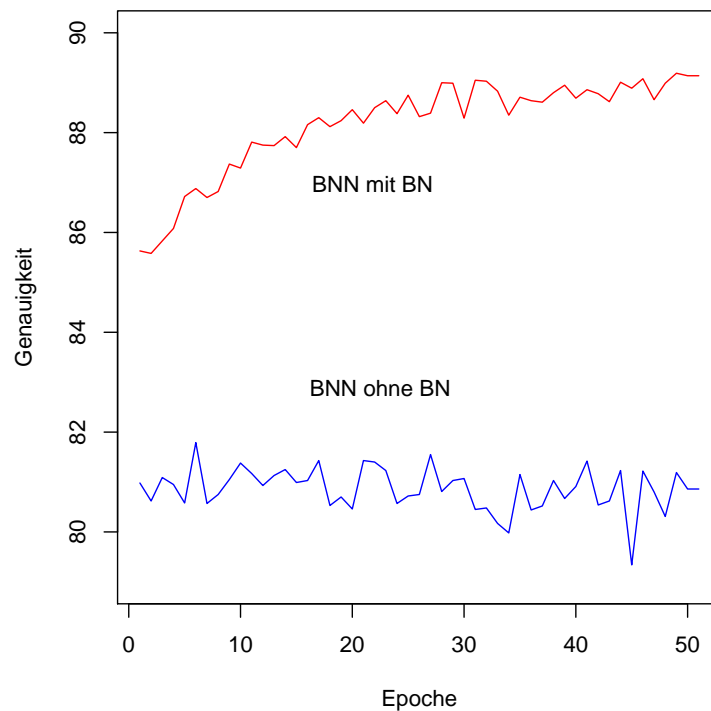


Abbildung 3.3: Training mit und ohne BatchNorm

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  $\text{logsoftmax}(x)$  Funktion verwendet. Durch  $\text{softmax}(x)$  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  $\text{logsoftmax}$ -Funktion verwendet.

Da bei der Anwendung des MNIST-Datensatzes immer nur ein Neuron ausgewählt werden sollte, da immer nur eine Zahl auf einem Bild abgebildet ist, wird bei der Auswertung über die  $\text{argmax}$ -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	binär	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 zentraler Parameter beim Training von Neuronale Netzwerke. Sie gibt die Rate an, mit der Kantengewichte in einem Durchlauf angepasst werden. Je höher die Lernrate also ist, desto schneller passt sich das Modell an gerade trainierte Daten an. Bei der Wahl der Rate sind die Probleme des *overfitting* und *underfitting* zu beachten. Als *underfitting* 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 Modells. 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*.

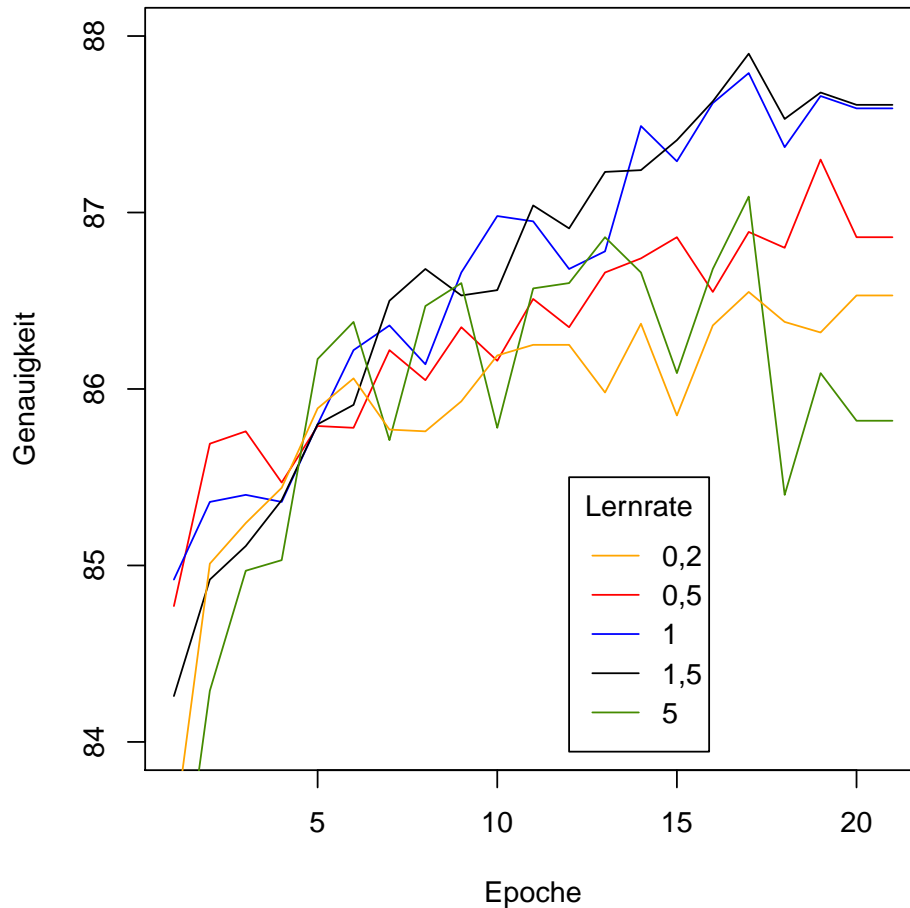


Abbildung 4.1: Training mit verschiedenen Lernraten

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.



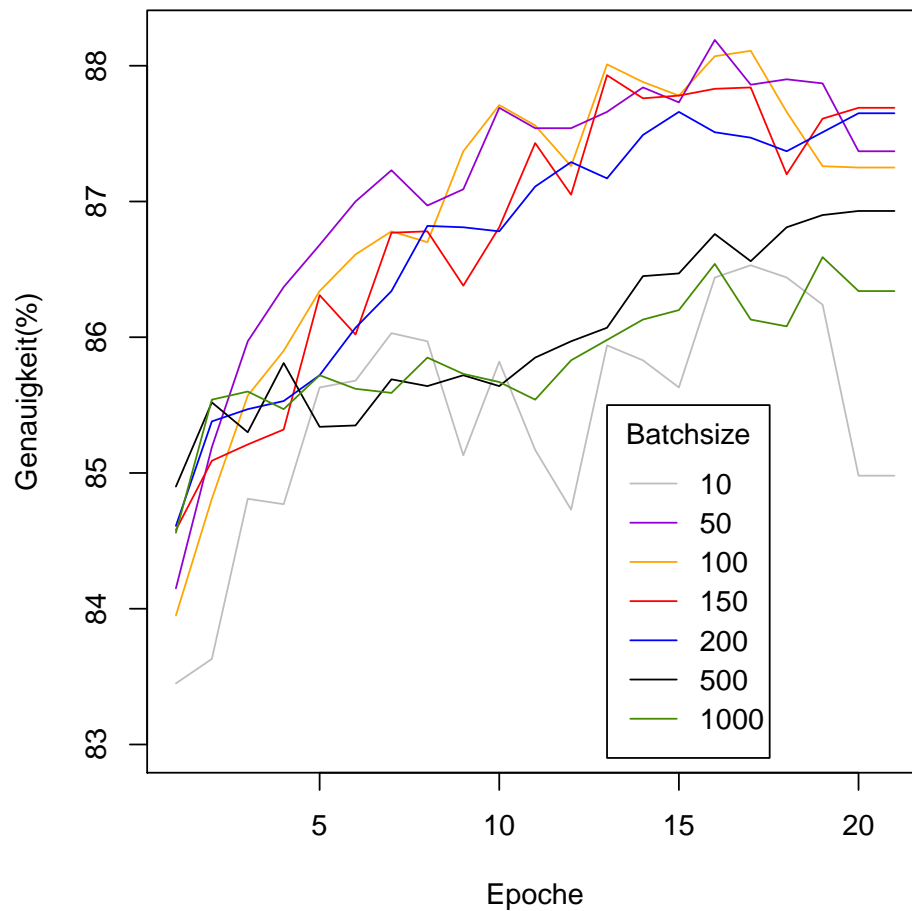


Abbildung 4.2: Training mit verschiedenen Batchgrößen

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  $\geq 500$  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  $\geq 150$  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  $\alpha$ -Kanal anstatt drei *RGB-Kanälen*. Dieser  $\alpha$ -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  $\mathbb{Q}_{0-1} \rightarrow \{0, 1\}$  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 Batch). 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  $28 \times 28$  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 Threshold, da für jeden gewählten Threshold bei einer Teilmenge der Bilder zuviel, bzw. zu wenig gefiltert wird. (Siehe Abbildung 5.1 bzw. 5.2) In

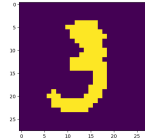


Abbildung 5.1: 3, Threshold 100

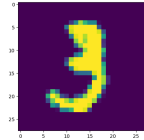


Abbildung 5.2: 3, not binarized

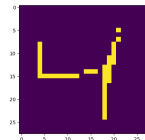


Abbildung 5.3: 4, Threshold 200

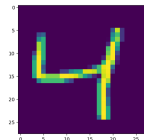


Abbildung 5.4: 4, not binarized

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*. Zudem 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 *Trainingssets* durchläuft.

Dadurch erhalten wir ein Mittel für jedes Bild, welches gegen die Originalversion konvergiert. (Siehe Abbildung 5.5 bzw 5.6). Die Einzelbilder aus denen sich das überlappte Bild zusammenstellt finden sich im Anhang.

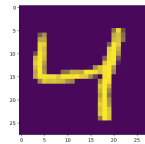


Abbildung 5.5: 4, overlapped 50 prob-trans

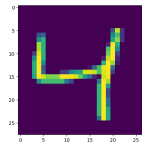


Abbildung 5.6: 4, not binarized

### 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.

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

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	<b>91.99%</b>	<b>89.24%</b>	<b>91.98%</b>

Abbildung 5.7: Vergleich

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 schneller 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.



# Kapitel 7

## Fazit

Durch die verwendeten Komponenten, sowie durch die Anpassung der Trainingsparameter konnte ein binäres Netzwerk erstellt werden, welches auf den MNIST-Datensatz trainiert wurde. Mithilfe von verbesserter Binarisierung der Eingaben konnte das Netz eine Genauigkeit von mehr als 92% erreichen. Dies entspricht insbesondere dem Ziel einer Genauigkeit von mehr als 90%.

In Zukunft könnte durch weitere Optimierung von bisher unbeachteten Parametern, wie zum Beispiel *step-size*, die Genauigkeit weiter gesteigert werden. Ebenfalls könnte das Netzwerk mit anderen Datensätzen trainiert und getestet werden.

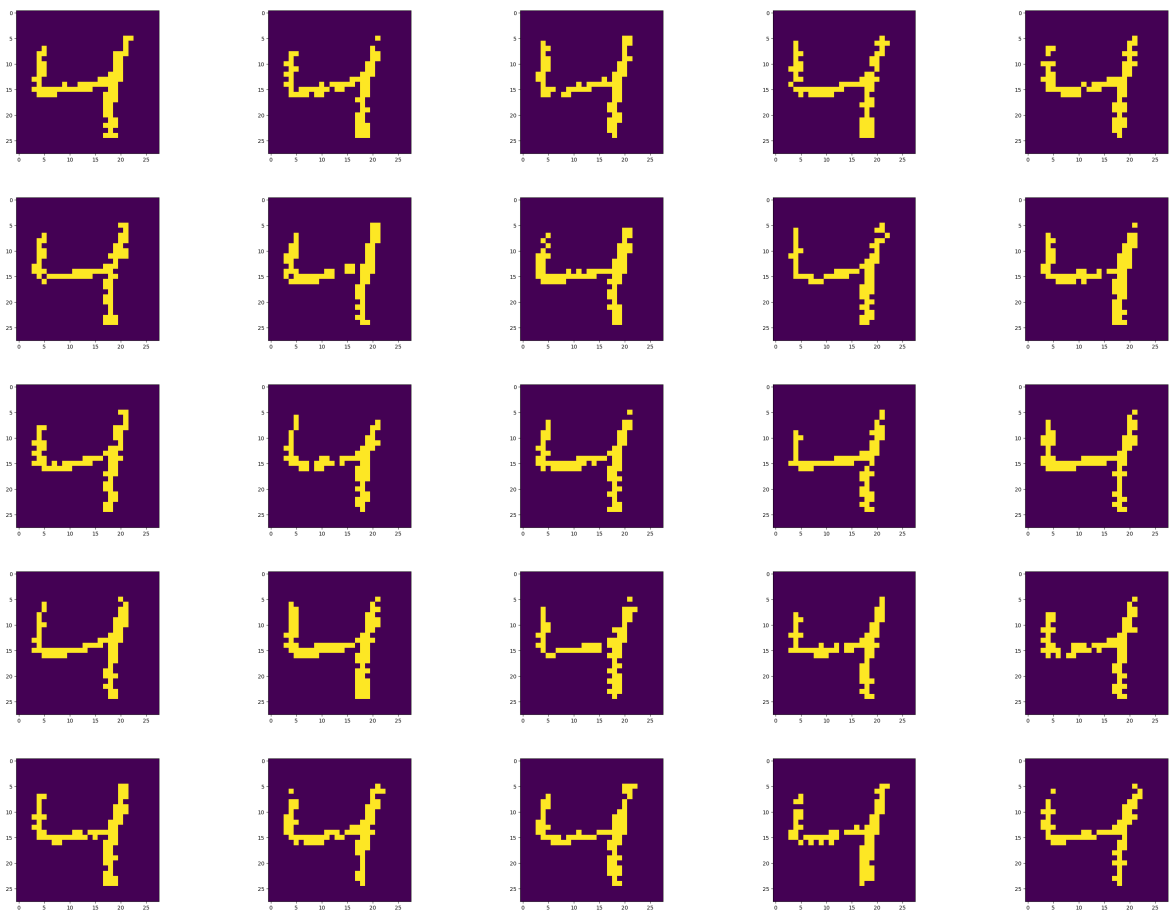
# Literatur

- [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 8

## Anhang

### 8.1 Probability-Binarized Images



## Kapitel 8 Anhang

