



ReLU in N++

Implementation und Vergleich

ALEXANDER JULIAN VIETH

Matrikelnummer: 1092554

BACHELORARBEIT

im Fachbereich 2: Informatik und
Ingenieurwissenschaften

in Frankfurt am Main

im Juli 2019

Erstgutachter : Prof. Dr. Thomas Gabel
Zweitgutachter: Prof. Dr. Christian Baun

Erklärung

Ich erkläre eidesstattlich, dass ich meine Bachelorarbeit mit dem Thema "ReLU in N++ - Implementation und Vergleich" selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Frankfurt am Main, am 10. Juli 2019

Alexander Julian Vieth

Inhaltsverzeichnis

Erklärung	i
Abstract	iv
1 Einführung - Künstliche neuronale Netzwerke	1
1.1 Künstliche neuronale Netzwerke	1
1.1.1 Inspiration	1
1.1.2 Ziele	2
1.1.3 Struktur	2
1.1.4 Forward Propagation	4
1.2 Gradientenabstiegsverfahren	9
1.3 Lernalgorithmen	12
1.3.1 Backpropagation	13
1.3.2 Resilient Propagation	15
2 Aktivierungsfunktionen	17
2.1 Relevanz	18
2.2 Identität	18
2.3 TanH	20
2.4 Sigmoid	21
2.5 ReLU	23
3 Implementierung	25
3.1 Fähigkeiten	25
3.2 Erweiterung	26
4 ReLU im Vergleich	27
4.1 Trainingskonventionen – bekannte Probleme	27
4.2 Ziele	28
4.3 Versuchsaufbau	28
4.4 Erwartungen	30
4.5 Datensätze	31
4.6 Vergleich	32
4.6.1 Iris	32

4.6.2	Rotwein-Qualität	36
4.6.3	Bank	40
4.6.4	Geschwindigkeit	44
4.7	Evaluierung	45
4.7.1	Iris	45
4.7.2	Rotwein-Qualität	45
4.7.3	Bank	46
5	Mögliche Verbesserungen	47
5.1	Leaky ReLU	47
5.2	Vergleich	49
6	Fazit	52

Abstract

In einer Zeit in der maschinelles Lernen mit selbsfahrenden Autos, Bilderkennung, synthetischer Spachausgabe, künstlich generierten Gesichtern, Maschinen die GO-Weltmeister besiegen, Maschinen die Schachweltmeister besiegen, Spieler im simulierten Roboterfußball kontrollieren und vielem mehr, so populär sind wie nie zu vor, ist der Drang groß den maschinellen Lernprozess weiter zu optimieren. In den letzten Jahren hat sich die Aktivierungsfunktion ReLU als äußerst beliebt herausgestellt. ReLU soll im Vergleich zu anderen Aktivierungsfunktionen wie Sigmoid beim Lernprozess schneller zu gewünschten Zahlen konvergieren, kürzere Rechenzeit benötigen und mit weniger Lerniterationen die Fehlerrate eines Netzwerks reduzieren. Der von Martin Riedmiller geschriebene Simulator für künstliche neuronale Netzwerke "N++" wird von Forschungsgruppen an der Frankfurt University Of Applied Sciences für verschiedene Zwecke genutzt, unter anderem für das Trainieren der Agenten der simulierten Roboterfußballmannschaft "Fra-UNIted". Für diese Arbeit wurde N++ um die ReLU Aktivierungsfunktion erweitert und mit Hilfe von 3 Datensätzen auf Effizienz getestet. Neben der Geschwindigkeit und Lerneffizienz wird in dieser Arbeit auch auf Limitationen und mögliche Verbesserungen von ReLU eingegangen.

Abbildungsverzeichnis

1.1	Ausgabe eines Neurons	2
1.2	Zahlenstrahl mit beispielhaften Kategorien	3
1.3	Problematische unregelmäßige Größe der äußeren Kategorien	4
1.4	Beispiel Netzwerk mit Legende.	5
1.5	Beispielhafte lineare Aktivierungsfunktion	5
1.6	Eingabewert propagiert von der Eingabeschicht zum Hidden Layer	6
1.7	Hidden Layer Neuronen propagieren errechnete Werte an Ausgabeneuron	6
1.8	Ausgabeneuron errechnet Ausgabe des Netzwerkes	7
1.9	Berrechnung der Ausgabe eines Neurons	7
1.10	Berrechnung der Ausgabe einer Schicht	8
1.11	Berrechnung der Ausgabe einer Schicht - vereinfacht	8
1.12	Kostenfunktion: quadratischer Fehler für simples Netzwerk	9
1.13	Kostenfunktion: quadratischer Fehler für beliebige Netzwerke	10
1.14	Proportionelle Auswirkungen des Gewichts in einem simplen Netzwerk	10
1.15	Proportionelle Auswirkungen des Gewichts in beliebigen Netzwerken	11
1.16	Proportionelle Auswirkungen einer Aktivierungen der vorherigen Schicht	11
1.17	Beispielhafte Ausgaben einer Kostenfunktion	12
1.18	Ein Gradientabschnitt mit Plateau	14
1.19	Ein übersprungenes Minimum aufgrund zu hoher Lernrate	14
1.20	Etablierte Werte für η^+, η^- nach Riedmiller und Braun	15
2.1	Struktur einer Aktivierungsfunktion	17
2.2	Formel der linearen Aktivierungsfunktion: Identität	18
2.3	Graph der linearen Aktivierungsfunktion: Identität	19
2.4	Formel der Aktivierungsfunktion: TanH	20
2.5	Graph der Aktivierungsfunktion: TanH	20
2.6	Formel der Aktivierungsfunktion: Sigmoid	21
2.7	Graph der Aktivierungsfunktion: Sigmoid	21

2.8	Formel der Aktivierungsfunktion: ReLU	23
2.9	Graph der Aktivierungsfunktion: ReLU	23
3.1	Formel der Aktivierungsfunktion: ReLU	26
4.1	Formel der Aktivierungsfunktion: Sigmoid	29
4.2	Durchschnittliche quadratische Fehler mit Backpropagation: Iris	32
4.3	Durchschnittliche Trefferrate mit Backpropagation: Iris . . .	33
4.4	Durchschnittliche quadratische Fehler mit RPROP: Iris . .	34
4.5	Durchschnittliche Trefferrate mit RPROP: Iris	35
4.6	Durchschnittliche quadratische Fehler mit Backpropagation: Rotwein-Qualität	36
4.7	Durchschnittliche Trefferrate mit Backpropagation: Rotwein- Qualität	37
4.8	Durchschnittliche quadratische Fehler mit RPROP: Rotwein- Qualität	38
4.9	Durchschnittliche Trefferrate mit RPROP: Rotwein-Qualität .	39
4.10	Durchschnittliche quadratische Fehler mit Backpropagation: Bank	40
4.11	Durchschnittliche Trefferrate mit Backpropagation: Bank . .	41
4.12	Durchschnittliche quadratische Fehler mit RPROP: Bank . .	42
4.13	Durchschnittliche Trefferrate mit RPROP: Bank	43
4.14	Durchschnittliche Trainingsgeschwindigkeit pro Epoch mit dem Bank Datensatz	44
5.1	Formel der Aktivierungsfunktion: Leaky ReLU	47
5.2	Graph der Aktivierungsfunktion: Leaky ReLU	48
5.3	Durchschnittliche quadratische Fehler mit RPROP + ReLU und Leaky ReLU: Bank	49
5.4	Durchschnittliche quadratische Fehler mit RPROP + ReLU und Leaky ReLU: Iris	50
5.5	Durchschnittliche Trefferrate mit RPROP + ReLU und Leaky ReLU	51
5.6	Durchschnittliche Trefferrate mit BP + ReLU und Leaky ReLU	51

Tabellenverzeichnis

4.1 Lernraten für die Datensätze für Backpropagation	30
4.2 Eigenschaften der verwendeten Datensätze	31

Kapitel 1

Einführung - Künstliche neuronale Netzwerke

Künstliche neuronale Netzwerke (oder KNNs) sind zurzeit eine der beliebtesten Werkzeuge um Daten zu kategorisieren. KNNs können in einer großen Reichweite von verschiedenen Anwendungen gefunden werden. Vom erkennen/identifizieren von Objekten in Bildern [1] bis hin zum Steuern von individuellen Spielern im simulierten Roboterfußball [2]. In diesem Abschnitt gehe ich auf die Motivation, Struktur und Funktionsweise hinter KNNs ein und wie verschiedene Konfigurationen die Leistungsfähigkeit dieser Netze beeinflussen.

1.1 Künstliche neuronale Netzwerke

1.1.1 Inspiration

Vom Aufbau her sind künstliche neuronale Netzwerke an natürliche neuronale Netzwerke angelehnt, die auch in so komplexen Lebewesen wie dem Menschen gefunden werden können. Die beeindruckende Fähigkeit von natürlichen neuronalen Netzwerken, komplexe Zusammenhänge und gigantische Mengen von Informationen zu verarbeiten und sich dementsprechend anzupassen und/oder zu imitieren, ist schon seit Jahrzehnten Ziel von Entwicklern künstlicher Intelligenz und Forschern im Feld des maschinellen Lernens. Wie auch natürliche neuronale Netze, funktionieren künstliche neuronale Netze mit Hilfe von einzelnen Neuronen, die miteinander vernetzt sind und Informationen untereinander austauschen. Wie ein natürliches neuronales Netz, lernt auch ein künstliches neuronales Netz mit Hilfe eines Trainings. Ein künstliches neuronales Netz trainiert mit Hilfe von Trainingsdaten, die aus einer Reihe von Eingabewerten für das Netzwerk bestehen sowie dem gewünschten Ergebnis für die gegebene Kombination von Eingabewerten. Basierend auf der Qualität der verwendeten Trainingsdaten (oder Erfah-

rungen), Netzaufbau und Anzahl von Trainingsdaten, verfügt ein trainiertes Netz die Fähigkeit ähnliche Eingabewerte richtig den vorher definierten Kategorien zuzuweisen. Diese Vorgehensweise, ein künstliches neuronales Netzwerk zu trainieren, fällt unter die Kategorie des überwachten maschinellen Lernens, da der gewünschte Ausgabewert für jede Trainingseinheit bekannt ist, bzw. mitgegeben wird. Was Trainingsdaten qualitativ hochwertig macht und was die Qualität der Ergebnisse, die das Netzwerk produziert beeinflussen kann, wird in den folgenden Abschnitten näher betrachtet.

1.1.2 Ziele

Im Jahr 1943 beschrieben Warren McCulloch und Walter Pitts eine Verbindung von neronenartigen Konstrukten, mit deren Hilfe sich jede logische und arithmetische Funktion realisieren lassen kann [3]. Heutzutage werden künstliche neuronale Netzwerke in vielen Anwendungen dazu genutzt, Zusammenhänge zwischen beliebigen Eingabewerten zu erkennen und die Eingabewerte zu kategorisieren. Dabei können auch bei enormen Lösungsmengen Verbindungen und Muster erkannt werden. Die Qualität der Ergebnisse, die ein künstliches neuronales Netzwerk produzieren kann, hängt von vielen Faktoren ab, die in den nächsten Abschnitten näher betrachtet werden.

1.1.3 Struktur

Ein künstliches neuronales Netz besteht aus Neuronen, die in Schichten geordnet miteinander verbunden sind. Dabei gibt es in jedem Netzwerk eine Eingabeschicht, mindestens ein sogenanntes Hiddenlayer (aus dem Englischen: Hidden = versteckt und Layer = Schicht), und eine Ausgabeschicht. Jede dieser Schichten besteht aus einer endlichen Anzahl von Neuronen. Jedes Neuron verfügt über Eingaben von Neuronen aus der vorherigen Schicht und gibt selbst einen errechneten Wert an Neuronen in der nächsten Schicht weiter. Jede Eingabe eines Neurons besitzt ein Gewicht, welches den Einfluss des spezifischen Eingabewertes auf die Ausgabe des Neurons beeinflusst, sowie einen Bias, der ebenfalls in die Berechnung des Ausgabewertes miteinfließt. Die Ausgabe eines Neurons a wird durch seine Aktivierungsfunktion Φ bestimmt, deren Eingabewert aus zwei Komponenten besteht: die Eingabewerte a_0, a_1, \dots, a_n , multipliziert mit den jeweiligen Gewichten w_0, w_1, \dots, w_n und dem Bias b .

$$a = \Phi(w_0a_0 + w_1a_1 + \dots + w_na_n + b)$$

Abbildung 1.1: Ausgabe eines Neurons

Die Form dieser Aktivierungsfunktion hat maßgebliche Auswirkungen auf

das Ergebnis und Verhalten des neuronalen Netzes, wie auch auf das Lernverhalten. In dieser Arbeit werden ich 4 dieser Aktivierungsfunktionen genauer betrachten und vergleichen.

Die Ausgabe von künstlichen neuronalen Netzen kann, je nach Aufbau des Netzes, unterschiedlich interpretiert werden. Der in dieser Arbeit verwendete Ansatz, bedeutet für den Netzaufbau, dass jeder mögliche Ausgabewert sein eigenes Ausgabeneuron darstellt. Welches der Ausgabeneuronen vom Netzwerk als Ergebnis gewählt wird, entscheidet der höchste Wert der Ausgabeneuronen. Die in dieser Arbeit verwendeten Netzwerke wurden so darauf trainiert, dass auf dem richtigen Ausgabeneuron ein Wert zu finden ist, welcher größer ist als der der restlichen Ausgabeneuronen. Eine Alternative zu diesem Ansatz wäre ein einziges Ausgabeneuron zu verwenden, dessen Wert möglichst nahe an vorher definierten Schwellenwerten sein muss, um auf diese Weise die Ergebniskategorie zu bestimmen. Diese Methode hat jedoch den Nachteil, dass Schwellenwerte mit dem kleinsten und größten numerischen Wert bevorzugt werden, da alle Ausgabewerte die gegen negativ oder positiv Unendlich gehen, den äußeren Kategorien zugewiesen werden, auch wenn der Ausgabewert nicht zwangsläufig sinnvoll ist. Man nehme hierzu die folgende beispielhafte Aufteilung des Ausgabewertes eines neuronalen Netzes in 7 Kategorien.

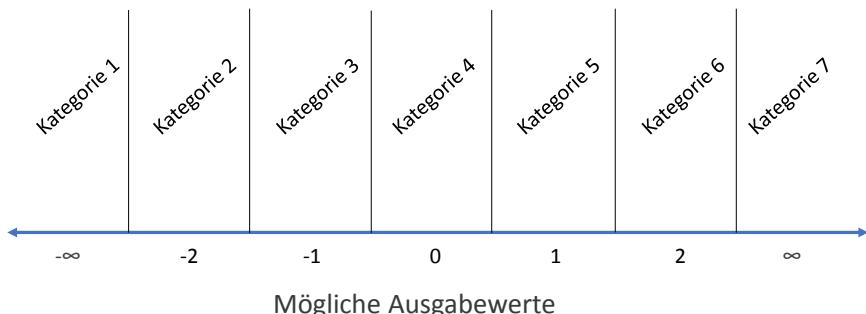


Abbildung 1.2: Zahlenstrahl mit beispielhaften Kategorien

Ausgabewerte zwischen $-1,5$ und $-0,5$ würden in diesem Fall der Kategorie 3 zugeordnet werden, etc. Für Kategorien 2 bis 6 ist die Aufteilung der Werte fair/gleichmäßig verteilt. Betrachtet man jedoch den Wertebereich von Kategorie 1 und 7 wird schnell klar, dass ein Ungleichgewicht in der Werteverteilung existiert.

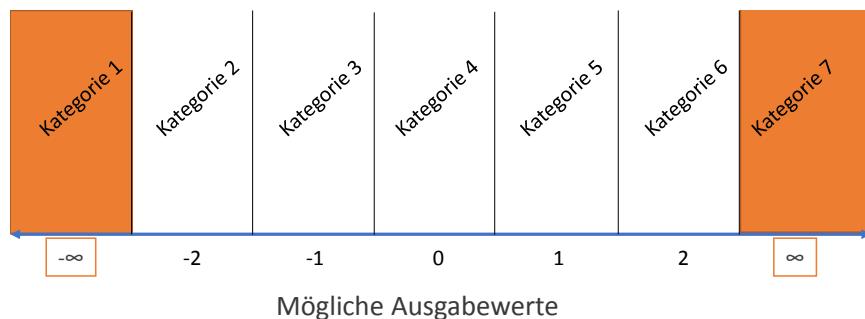


Abbildung 1.3: Problematische unregelmäßige Größe der äußeren Kategorien

Da für die Ausgabewerte in dieser Arbeit eine Fließkommazahl mit einer Genauigkeit von 6 Nachkommastellen verwendet wurde, existieren im Schwellenbereich von Kategorie 3 wesentlich weniger mögliche Werte als im Schwellenbereich von Kategorie 1, welcher bis zu dem maximalen Wert der Fließkommazahl geht. Aufgrund dessen gäbe es mit diesem Aufbau ggf. einen Bias für alle Kategorien, deren Schwellenwert an den Außenrändern des gewählten Zahlenbereichs liegen.

Um das Verhalten des neuronales Netzes bei gleichen Eingabewerten zu beeinflussen, müssen die Gewichte und der Bias für die Neuronen angepasst werden. Dieser Anpassungsprozess der Gewichte und Bias' ist der "Lernen"-Anteil des "maschinellen Lernens".

1.1.4 Forward Propagation

Der Prozess, bei dem Eingabewerte durch das neuronale Netzwerk fließen und letztenendes über die Ausgabeneuronen ein Ergebnis liefern, wird als "Forward Propagation" (aus dem Englischen: Forward = vorwärts, Propagation = Weitergabe) bezeichnet. Bei der Forward Propagation fließen die Eingabewerte sukzessive durch alle Schichten des Netzwerks, angefangen bei der Eingangsschicht. Um zu verstehen wie neuronale Netze funktionieren und wie der Lernprozess die Gewichte und den Bias anpasst, hilft es die Forward Propagation sowohl optisch an einem Beispiel wie auch als mathematischen Ausdruck zu betrachten.

Beispiel

Gegeben sei ein neuronales Netz mit 3 Schichten, einem Eingabeneuronen, 1 Hidden Layer mit 2 Neuronen und 1 Ausgabeneuron. Ein spezifisches Neuron kann mit dem Index der Schicht und dem Index des Neurons innerhalb der Schicht identifiziert werden.

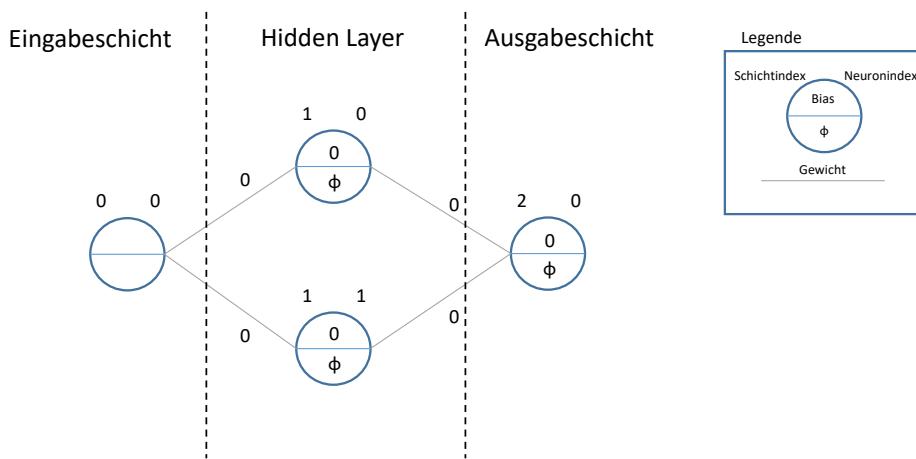


Abbildung 1.4: Beispiel Netzwerk mit Legende.

Jedes Neuron verfügt über den Index der Neuronenschicht, zu der Es gehört, sowie einen Index für die Position innerhalb der Schicht. Zusätzlich hat jedes Neuron einen Bias, pro Eingang ein Gewicht und eine Aktivierungsfunktion. Der Wert des Bias und der Gewichte wird zu Beginn für jedes Neuron zufällig gewählt. Für dieses Beispiel verwenden ich die lineare Aktivierungsfunktion

$$\Phi(x) = x.$$

Abbildung 1.5: Beispielhafte lineare Aktivierungsfunktion

Gegeben sei ein zufällig gewählter Eingabewert 0,4. Dieser Wert wird vom Eingabeneuron an beide Neuronen im Hidden Layer propagiert. (abb. 1.6) In den Neuronen im Hidden Layer wird mit dem Eingabewert, Gewicht und Bias ein Ausgabewert über die Aktivierungsfunktion Φ errechnet und an das Ausgabeneuron propagiert. (abb. 1.7) Das Ausgabeneuron verrechnet beide Ausgabewerte der Hidden Layer Neuronen in seiner Aktivierungsfunktion. Der resultierende Wert (0,556) ist die Ausgabe des Netzwerkes für den Eingabewert 0,4. (abb. 1.8)

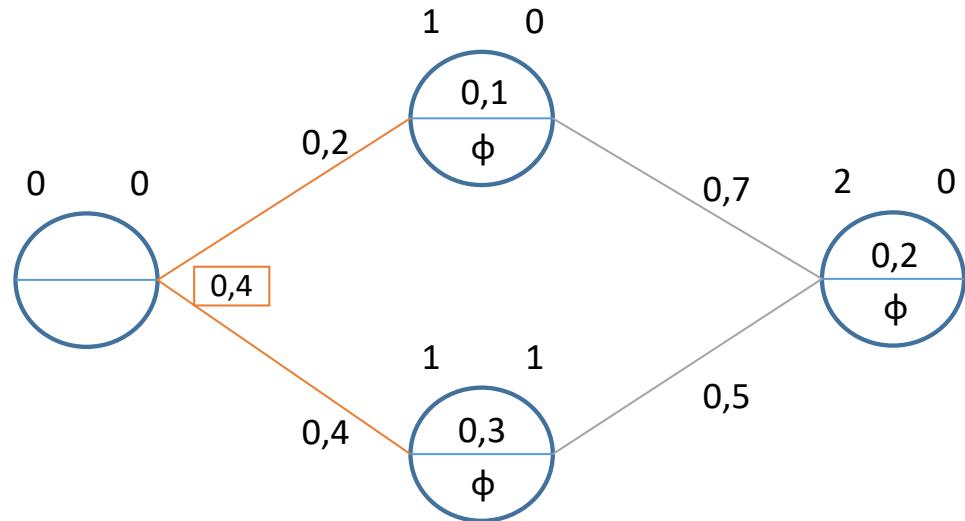


Abbildung 1.6: Eingabewert propagiert von der Eingabeschicht zum Hidden Layer

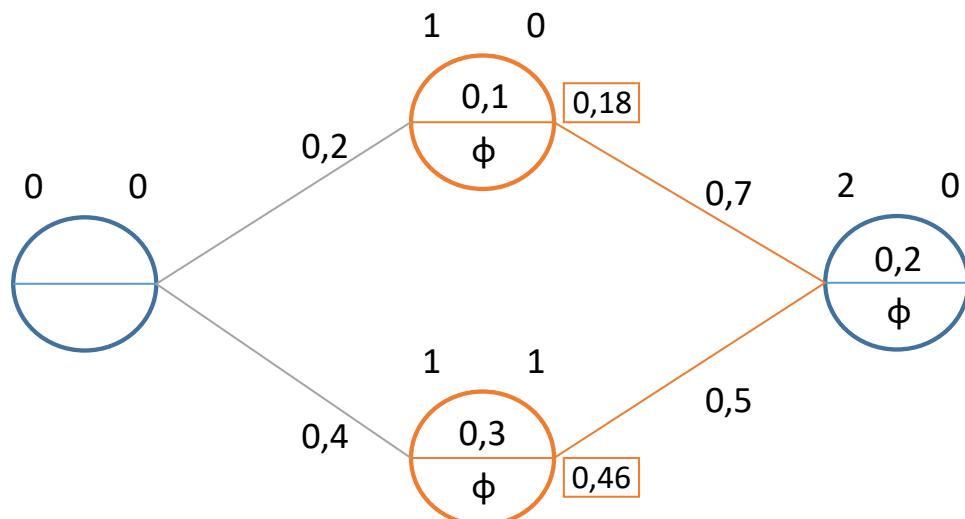
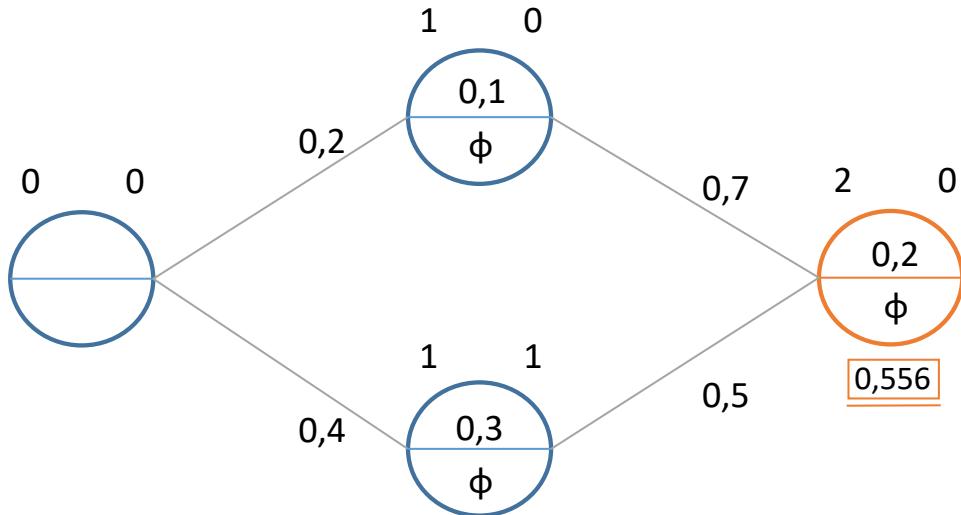


Abbildung 1.7: Hidden Layer Neuronen propagieren errechnete Werte an Ausgabeneuron

**Abbildung 1.8:** Ausgabeneuron errechnet Ausgabe des Netzwerkes

Mathematische Darstellung

Für die mathematische Darstellung wird ein Neuron mit a_n^i beschrieben, wobei $i = \text{Schichtindex}$ und $n = \text{Neuronindex innerhalb der jeweiligen Schicht}$. Jedes Neuron verfügt über einen Vektor an Aktivierungen von der vorherigen Schicht a^{i-1} sowie einen Vektor an Gewichten w und einen Bias b . Demnach ist die Ausgabe eines spezifischen Neurons a_n^i :

$$a_n^i = \Phi \left(\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \begin{bmatrix} a_0^{i-1} \\ a_1^{i-1} \\ \vdots \\ a_n^{i-1} \end{bmatrix} + b \right)$$

Abbildung 1.9: Berechnung der Ausgabe eines Neurons

Diese Formel kann erweitert werden, um die Aktivierungen der gesamten Schicht zu berechnen. Für diese Rechnung werden alle Gewichte für jedes Neuron innerhalb der zu berechnenden Schicht, in eine Matrix W^i aufgenommen. Außerdem bilden wir mit allen Bias' der Schicht einen Biasvektor b^i . Wobei i weiterhin den Schichtindex darstellt. Über eine Vektormultiplikation wird nun die gewichtete Summe der Aktivierungen errechnet und anschließend mit einer Vektoraddition der Bias hinzugefügt. Auf das Ergebnis

nis jeder "Reihe" in dieser Berechnung muss nun die Aktivierungsfunktion angewendet werden, was in diesem Fall dargestellt wird, indem der gesamten Ausdruck der Φ Aktivierungsfunktion übergeben wird. Die Aktivierung der Schicht a^i kann nun folgendermaßen beschrieben werden:

$$a^i = \Phi \left(\begin{bmatrix} w_0^0 & w_1^0 & \dots & w_n^0 \\ w_0^1 & w_1^1 & \dots & w_n^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_0^i & w_1^i & \dots & w_n^i \end{bmatrix} \begin{bmatrix} a_0^{i-1} \\ a_1^{i-1} \\ \vdots \\ a_n^{i-1} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right)$$

Abbildung 1.10: Berechnung der Ausgabe einer Schicht

Oder weiter vereinfacht geschrieben, mit W^i als Gewichtematrix, a^{i-1} für die Aktivierungen der vorherigen Schicht und b^i für den Biasvektor.

$$a^i = \Phi(W^i a^{i-1} + b^i)$$

Abbildung 1.11: Berechnung der Ausgabe einer Schicht - vereinfacht

Diese Formel kann nun für alle Schichten in Folge angewendet werden, um das Ergebnis für einen gegebenen Eingabewert zu berechnen.

Die Berechnung als Vektorprodukt und Vektorsumme darzustellen ist hilfreich, da viele Bibliotheken der meisten Programmiersprachen diese Operationen stark optimiert haben und dadurch potentiell Zeit eingespart werden kann.

1.2 Gradientenabstiegsverfahren

Ähnlich wie auch Ihre biologische Vorlage, lernen künstliche neuronale Netzwerke indem Sie mit "Erfahrungen" konfrontiert werden und anschließend beigebracht bekommen was auf die gegebene "Erfahrung" die richtige Antwort ist. Erfahrungen im Kontext des maschinellen Lernens sind in diesem Fall die Eingabewerte für ein neuronales Netzwerk und die richtige Antwort stellt die gewünschte Kategorie da, die zu den Eingabewerten gehört. Diese Art zu lernen setzt voraus, dass es eine Sammlung von Trainingsdaten gibt, die Einagbewerte und die dazugehörige Lösung beinhalten. Damit ein neuronales Netzwerk mit Hilfe von Trainingsdaten besser darin werden kann, die gegebenen Eingabewerte zu kategorisieren, müssen die Trainingsdaten ein nicht zwangsläufig offensichtliches Muster aufweisen. Wie ich in den nächsten Kapiteln feststellen werde, kann ein neuronales Netzwerk bei unzuverlässigen und/oder inkonsistenten Trainingsdaten keine guten Ergebnisse erzielen. Um zu verstehen wie neuronale Netzwerke lernen und welche Rolle die Aktivierungsfunktion in diesem Vorgang spielt, hilft es, das Gradientenabstiegsverfahren im Hinblick auf neuronale Netzwerke näher zu betrachten.

Das Gradientenabstiegsverfahren - ein Optimierungsverfahren aus der Numerik - wird in neuronalen Netzwerken dazu verwendet die Gewichte und Biases für die gegebenen Eingabewerte anzupassen und damit die Fehlerquote des Netzwerks für ähnliche Eingabewerte zu reduzieren. Präziser formuliert, soll mit Hilfe des Gradientenabstiegsverfahrens das Ergebnis der Kostenfunktion des Netzwerks minimiert werden, in dem ein lokales Minimum der Funktion gefunden wird. Eine mögliche Kostenfunktion stellt den quadratischen Fehler des Netzwerks dar, alternativ kann auch der mittlere quadratische Fehler verwendet werden. Für diese Arbeit wurde der quadratische Fehler als Kostenfunktion verwendet. Für ein neuronales Netzwerk mit jeweils nur einem Neuron je Schicht, würde diese Kostenfunktion die folgende Struktur haben:

$$C = (a^i - y)^2$$

Abbildung 1.12: Kostenfunktion: quadratischer Fehler für simples Netzwerk

Wobei a^i das Ergebnis der Forward Propagation 1.1.4 darstellt und y das erwartete Ergebnis für den verwendeten Eingabewert.

Da neuronale Netzwerke meistens umfangreicher ausfallen als 1 Neuron pro Schicht, muss diese Funktion angepasst werden um mehrere Ausgabeneuronen akzeptieren zu können.

$$C^i = \sum_{k=0}^N (a_k^i - y_k)^2$$

Abbildung 1.13: Kostenfunktion: quadratischer Fehler für beliebige Netzwerke

Wobei N die Anzahl von Neuronen in der Ausgabeschicht darstellt. Hier stellt die Summe der quadratischen Fehler der jeweiligen Ausgabeneuronen die Kosten des Netzwerkes da.

Im nächsten Schritt gilt es, die Gewichte und Bias der Neuronen in der Ausgabeschicht proportional zu dem Fehlerwert anzupassen, sodass gleiche Eingabewerte eine Ausgabe produzieren, die näher an der gewünschten Lösung liegt. Um die nächsten Formeln übersichtlicher zu gestalten, substituiere ich die gewichteten Aktivierungen, die von der Aktivierungsfunktion angenommen werden, mit dem Ausdruck z_n^i , mit n = Neuronindex und i = Schichtindex. Für ein einfaches Netz mit je einem Neuron pro Schicht und ohne Hidden Layer, lässt sich die Formel für die Änderung des Gewichts des Ausgabeneurons mit partionellen Ableitungen entsprechend formulieren:

$$\Delta w^i = \frac{\delta C}{\delta w^i} = \frac{\delta z^i}{\delta w^i} \frac{\delta a^i}{\delta z^i} \frac{\delta C}{\delta a^i}$$

Abbildung 1.14: Proportionelle Auswirkungen des Gewichts in einem simplen Netzwerk

Mit dem Anwenden der Kettenregel, können die Veränderung des Gewichts proportional zum Fehler errechnet werden. Dazu beschreibt jede Ableitung einen der Werte, die in die Berechnung des Fehlers mit einfließen. Vereinfacht gesagt, beschreibt diese Formel wie stark sich Veränderungen des Gewichts w^i auf den Fehler C^i auswirken. Für komplexere Netzwerke ändert sich diese Formel nur minimal. Hierfür wird eine weitere Laufvariable für die Gewichte und Aktivierungen benötigt, um die Verbindung von zwei Neuronen ansprechen zu können.

$$\Delta w_{nk}^i = \frac{\delta C}{\delta w_{nk}^i} = \frac{\delta z_n^i}{\delta w_{nk}^i} \frac{\delta a_n^i}{\delta z_n^i} \frac{\delta C}{\delta a_n^i}$$

Abbildung 1.15: Proportionelle Auswirkungen des Gewichts in beliebigen Netzwerken

Wobei n der Neuronindex in der aktuellen Schicht darstellt und k den Neuronindex aus der vorherigen Schicht. Ein Gewicht w_{nk}^i ist demnach das Gewicht für die Verbindung von dem Neuron mit dem Index k , aus der $i-1$ ten Schicht, zu dem Neuron mit dem Index n in der Schicht i .

Um die Veränderung des Bias zu errechnen, wird der Term für das Gewicht durch den des Bias ersetzt - die fundamentale Logik ist dieselbe: Wie muss der Bias verändert werden, sodass das Ergebnis der Kostenfunktion minimiert wird, in Relation zum Fehler, Eingabewerten und Aktivierungen.

Um die restlichen Gewichte und Bias in dem Netzwerk anzupassen, wird die Auswirkung der Aktivierung der vorherigen Schicht, proportional zum Fehler, benötigt. Dieser Wert kann anschließend dafür verwendet werden, rückwärts durch das Netzwerk den selben Algorithmus anzuwenden und so alle Gewichte und Bias im Bezug auf die gegebenen Eingabewerte und Fehler anzupassen. Da die Aktivierung eines Neuron von mehreren Aktivierungen der vorherigen Schicht abhängen kann, müssen die partiellen Ableitungen aufaddiert werden, um dieses Szenario zu berücksichtigen.

$$\Delta w^i = \frac{\delta C}{\delta a_k^{i-1}} = \sum_{n=0}^N \frac{\delta z_n^i}{\delta a_k^{i-1}} \frac{\delta a_n^i}{\delta z_n^i} \frac{\delta C}{\delta a_n^i}$$

Abbildung 1.16: Proportionelle Auswirkungen einer Aktivierungen der vorherigen Schicht

1.3 Lernalgorithmen

Wie etabliert, geht es beim Lernen in neuronalen Netzwerken darum, das Ergebnis einer Kostenfunktion zu reduzieren, indem die Gewichte und Bias angepasst werden. Das Ergebnis der Kostenfunktion hängt potentiell von einer großen Menge von Eingabewerten, Gewichten, Bias' und die daraus resultierenden Aktivierungen ab. Aufgrund der Komplexität der resultierenden möglichen Ergebnisse ist es nur möglich einen lokalen Tiefpunkt für die Kostenfunktion zu finden. Um diesen Vorgang zu verdeutlichen, betrachten wir einen Ausschnitt des Verlaufs der Kostenfunktion in einem Beispielhaften neuronalen Netzwerk.

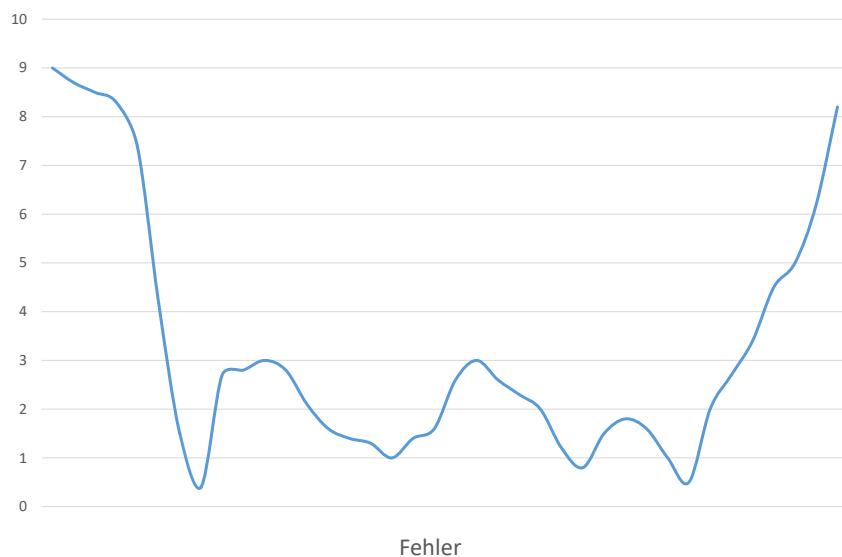


Abbildung 1.17: Beispielhafte Ausgaben einer Kostenfunktion

Ein gegebenes untrainiertes Netzwerk könnte nach einer ersten Trainingseinheit mit seinem Fehlerwert an einem zufälligen Punkt auf diesem Gradienten starten. Über verschiedene Lernalgorithmen kann nun gesteuert werden, wie sich das Ergebnis der Kostenfunktion auf dem Gradienten bewegt. Ziel ist es, anhand der Steigung an dem aktuellen Fehlerwert, sich Schritt für Schritt einem lokalen Minimum zu nähern. In dieser Arbeit wurden zwei verschiedene Lernalgorithmen verwendet um die Leistungsfähigkeit der implementierten Aktivierungsfunktion zu testen und zu vergleichen. In den nächsten Kapiteln werden ich die verwendeten Lernalgorithmen betrachten, um später eventuelle Auswirkungen auf die Qualität der trainierten Netzwerke nachvollziehen zu können.

1.3.1 Backpropagation

Obwohl Backpropagation (aus dem Englischen: Fehlerrückführung), als Konzept bereits in den frühen 1960er Jahren durch die Arbeit von Henry J. Kelley bekannt war [4], wurde erst im Jahr 1974 von Paul Werbos die Möglichkeit erwägt, die Backpropagation mit bestimmten Lernparametern auch für künstliche neuronale Netzwerke anzuwenden [5]. 1986 wurde die Backpropagation erstmal erfolgreich von David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams experimentell für tiefe neuronale Netze angewandt [6]. Bis heute wird die klassische Backpropagation in verschiedenen Simulatoren und Frameworks, wie Keras [7] oder PyTorch [8] verwendet/angeboten.

Die Backpropagation erzielt eine Annäherung an ein lokales Minimum der Kostenfunktion, indem, abhängig von dem Steigungsgrad der aktuellen Position auf dem Gradienten, in variabler Schrittgröße sich Richtung kleinerer Fehlerwerte genähert wird. Es existieren verschiedene Erweiterungen für den Backpropagation-Algorithmus, wie Momentum und Weight-Decay (aus dem Englischen: Weight = Gewicht, Decay = Zerfall), mit denen verhindert werden soll, lokale Minima zu überspringen, oder aus einem Tal im Gradienten auszubrechen. Da diese Parameter für diese Arbeit nicht verwendet wurden, gehen ich hier nur auf den Lernrate-Parameter ein. Die Lernrate stellt einen Faktor für die Schrittgröße dar. Je höher die Lernrate, desto größer ist der Schritt in die entsprechende Richtung. Wie für alle Lernalgorithmen und deren Konfigurationsmöglichkeiten gilt, dass es keine generelle, optimale Kombination an Werten für die verwendeten Parameter gibt, die für alle Netzkonfigurationen und Eingaben gute Ergebnisse produziert. Je nach Aufbau des Netzwerks und Form der Eingabewerte, muss über Experimente festgestellt werden, welche Konfiguration die augenscheinlich besten Ergebnisse liefert. Auch das ist bedingt durch die Komplexität des Gradienten in Abhängigkeit der Eingabewerte, Gewichte und Bias. Sollte sich während des Lernprozesses der Fehlerwert nicht relevant verringern, spricht man von einem Plateau im Gradienten. Eine höhere Lernrate kann helfen ein solches Plateau schneller bzw. in weniger Iterationen zu durchschreiten. Dabei besteht jedoch die Gefahr, schmale Täler, die eventuell in dem Gradienten zu finden sind, zu überspringen, was potenziell zu einer Oszillation des Fehlerwertes, schlechteren Ergebnissen und/oder längeren Trainingsprozessen führen kann.

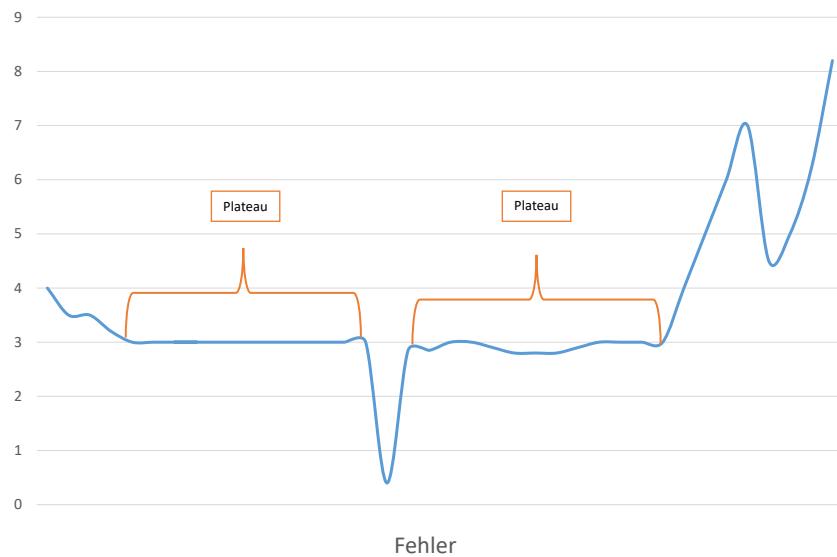


Abbildung 1.18: Ein Gradientabschnitt mit Plateau

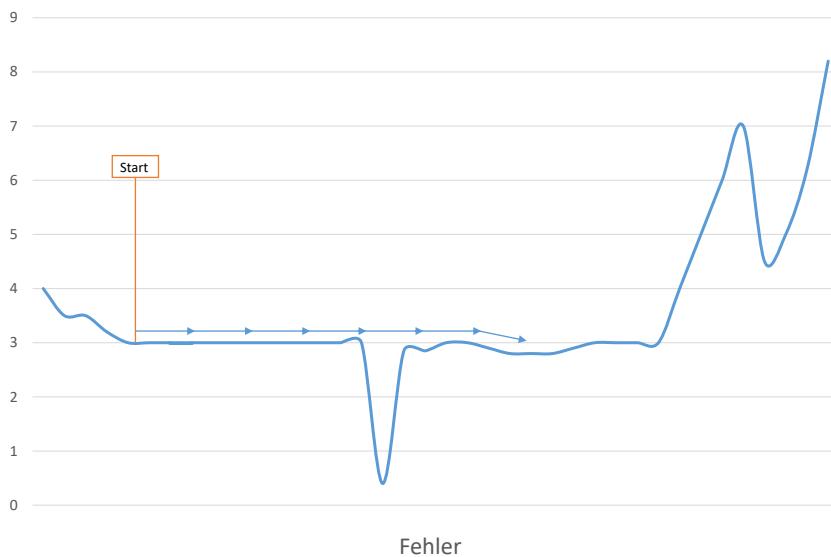


Abbildung 1.19: Ein übersprungenes Minimum aufgrund zu hoher Lernrate

In Abb. 1.19 kann man erkennen, wie aufgrund einer zu hohen Lernrate - hier gezeigt durch die Länge der Schritte/Pfeile - ein Minimum der Kostenfunktion übersprungen wurde. Selbst wenn es die Funktion in das Tal schaffen sollte, würde sie im nächsten Schritt aufgrund der zu großen Schrittgröße

wieder ausbrechen. Um dieses Problem zu vermeiden, muss in diesem Fall die Lernrate bei gleichbleibender Startkonfiguration (gleicher Ausgangspunkt auf dem Gradienten) reduziert werden. Sollte der Fehler des Netzwerks stark fluktuieren, werden höchstwahrscheinlich ebenfalls lokale Minima, die nah auf dem Gradienten liegen, übersprungen. Obwohl es Alternativen zu Backpropagation gibt und diese, je nach Anwendungsfall, bessere Ergebnisse als die klassische Backpropagation erzielen können, kann keine allgemeine Aussage über die Qualität der Lernalgorithmen für alle Anwendungsfälle getroffen werden. Je nach Netzwerkaufbau und Eingabewerten erzielen verschiedene Lernalgorithmen bessere oder schlechtere Ergebnisse, relativ zueinander.

1.3.2 Resilient Propagation

Resilient Propagation - oder RPROP, ist eine Variation der Backpropagation, die erstmals 1992 von Martin Riedmiller und Heinrich Braun beschrieben wurde [9]. RPROP, in seiner modernen Form, benötigt 4 Parameter:

1. Minimale Schrittgröße γ_{min}
2. Maximale Schrittgröße γ_{max}
3. Faktor bei steigendem Gradienten η^+
4. Faktor bei fallendem Gradienten η^-

Wie auch die klassische Backpropagation mit variabler Lernrate, passt auch RPROP die Schrittgröße je nach Steigung des Gradienten an. Dabei fungieren die Minimal- und Maximalwerte $\gamma_{min}, \gamma_{max}$ als Begrenzung für die mögliche Schrittgröße und η^-, η^+ als Faktoren für die Erhöhung oder Vergrößerung der Schrittgröße, abhängig davon ob die aktuelle Steigung des Gradienten positiv oder negativ ist. Der große Vorteil von RPROP ist, dass das Überschreiten von Tälern im Gradienten und damit eine Oszillation des Fehlerwertes stark reduziert wird, da nicht die Steigung an der aktuellen Position entscheidend ist, sondern nur das Vorzeichen der Steigung. Ist die Steigung negativ, also bei einem fallendem Gradienten, wird die Schrittgröße mit η^- multipliziert. Ist die Steigung positiv wird der Faktor η^+ verwendet. Durch die Experimente von Riedmiller und Braun haben sich Werte für η^-, η^+ ergeben, die relativ unabhängig von Eingabewerten und Netzaufbau gute Ergebnisse produzieren. Dieser Werte wurden auch in dieser Arbeit verwendet.

- $\eta^+ = 1,2$
- $\eta^- = 0,5$

Abbildung 1.20: Etablierte Werte für η^+, η^- nach Riedmiller und Braun

1993 wurde von Riedmiller und Braun eine Verbesserung des RPROP Algorithmus veröffentlicht [10], in der die minimale und maximale Schrittgröße eingeführt wurden. Die minimale Schrittgröße kann nach Riedmiller und Braun sehr gering gehalten werden, da diese nur erreicht werden kann, wenn sich bereits für viele Iterationen sich einem Minim genähert wurde. In diesem Fall ist es gewolltes Verhalten sich stetig dem Minimum zu nähern und nicht wieder aus dem Tal auszubrechen. Eine kleine minimale Schrittgröße ist hierbei ungefährlich und sicher. Des Weiteren empfehlen Riedmiller und Braun in Ihrer Arbeit eine maximale Schrittgröße von 50. Dieser Wert hat sich seitdem als anerkannte gute Wahl für die meisten Fälle erwiesen, der auch in dieser Arbeit verwendet wurde.

Kapitel 2

Aktivierungsfunktionen

Die Auswahl der Aktivierungsfunktionen, die beim Aufbau eines neuronalen Netzwerks verwenden können, ist groß und vielseitig. So kann praktisch jede Funktion ϕ verwendet werden, insofern sie differenzierbar ist und die folgende Struktur aufweist:

$$\phi(x) = y$$

Abbildung 2.1: Struktur einer Aktivierungsfunktion

mit $x, y \in \mathbb{R}$. Zur Erinnerung: x stellt die gewichtete Summe der Aktivierungen der vorherigen Schicht und des Bias da. Die Aktivierungsfunktion ϕ muss differenzierbar sein, da andernfalls keine Fehlerrückführung und damit kein Lernen möglich ist. Im Abschnitt zur Fehlerrückführung in künstlichen neuronalen Netzwerken wurde zur Vereinfachung diese Summe als z_n^i dargestellt - mit $n = \text{Neuronindex}$ und $i = \text{Schichtindex}$.

In der Praxis haben sich bestimmte Funktionen als besonders effizient erwiesen, das heißt, dass Sie entweder nicht besonders rechenaufwändig sind und/oder Netzwerke die, unter anderem diese Aktivierungsfunktionen verwenden, gute Ergebnisse beim Klassifizieren liefern. In dieser Arbeit wurden 4 verschiedene Aktivierungsfunktionen verwendet, die in den nächsten Abschnitten näher betrachtet werden, was uns im Vergleich helfen wird nachzuvollziehen, warum bestimmte Konfigurationen von neuronalen Netzwerken unterschiedlich reagieren, wenn diese Aktivierungsfunktionen ausgetauscht werden.

2.1 Relevanz

Da Aktivierungsfunktionen für jedes Neuron in einem Netzwerk aufgerufen werden, ist es sinnvoll, im Bezug auf die Geschwindigkeit, eine Funktion zu verwenden, die möglichst gut von Computern verarbeiten werden kann und möglichst wenige Terme und Operationen aufweist. Da während der Fehlerrückführung die Ableitung der Aktivierungsfunktion ebenfalls verwendet wird, ist es wichtig die Ableitung der gewählten Funktion zu betrachten. Unabhängig vom Rechenaufwand, sind für die Qualität der Ergebnisse eines Netzwerks einige Eigenschaften von Aktivierungsfunktionen hilfreich. So sollten positive und negative Werte abgebildet werden können, auch die Null. Der Vorteil wird klar, wenn man eine Aktivierungsfunktion wie ReLU für die Ausgabeneuronen eines Netzwerks verwendet, die keine negativen Ergebnisse produzieren kann. Warum ReLU diese Eigenschaft hat, werden ich in den nächsten Kapiteln näher behandeln. Sollte nach der potentiell vielschichtigen Forward Propagation, z_n^i in dem Ausgabeneuron negativ ausfallen, ist der gesammte Informationsgehalt verloren, sollte ReLU für das Ausgabeneuron verwendet worden sein, da ReLU alle negativen Werte als 0 abbildet. Dieser Informationsverlust führt dazu, dass das Netzwerk für bestimmte Eingabewerte eventuell schlechte Ergebnisse liefert, bzw. nicht so effektiv lernen kann, wie es eventuell möglich mit anderen Aktivierungsfunktionen möglich wäre.

2.2 Identität

Die lineare Identitätsfunktion ist die simpelste der verwendeten Aktivierungsfunktionen.

$$\phi(x) = x \quad (2.1)$$

$$\phi'(x) = 1 \quad (2.2)$$

Abbildung 2.2: Formel der linearen Aktivierungsfunktion: Identität

Wobei $\phi'(x)$ die Ableitung der Aktivierungsfunktion darstellt. Diese Funktion gibt die gewichteten Eingabewerte mit Bias an die nächste Schicht weiter, ohne diese zu manipulieren. Der große Vorteil ist die Recheneffizienz, da keine weitere Berechnung notwendig ist, sowohl in der Forward Propagation als auch der Fehlerrückführung, da die Ableitung der Funktion immer 1 ergibt. Da die Identitätsfunktion den gegebenen Wert identisch wiedergibt, können auch Ausreißer durch das Netz propagiert werden, die ggf. das gesamte Ergebnis in eine ungewollte Richtung lenken können. Des Weiteren

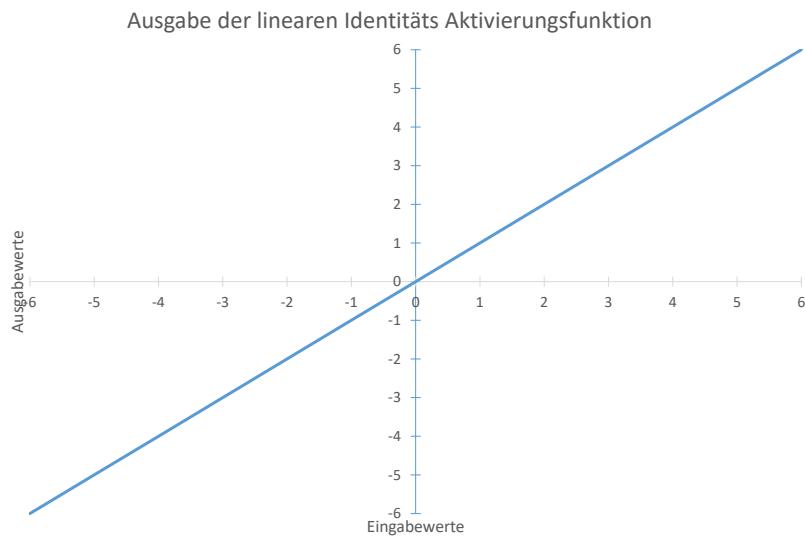


Abbildung 2.3: Graph der linearen Aktivierungsfunktion: Identität

bietet die Ableitung der Funktion keine Variationsmöglichkeiten, die bei dem Lernprozess zu flexibleren Lösungen führen kann. Ein Neuron kann also eine Aktivierung von 5000, oder -1 haben, was im Lernprozess beides zu 1 reduziert werden würde, was keinen Informationsgehalt besitzt. Die Identitätsfunktion ist nützlich als Aktivierungsfunktion für die Ausgabeschicht, da andere Aktivierungsfunktionen eventuell dem Ergebniss viel von seinem Informationsgehalt nehmen könnten, wie im Beispiel am Anfang dieses Kapitels beschrieben.

2.3 TanH

TanH, oder Tangens Hyperbolicus, beschreibt eine 'S'-Kurve zwischen den Werten -1 und 1 .

$$\phi(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = 1 - \frac{2}{e^{2x} + 1} \quad (2.3)$$

$$\phi'(x) = \tanh'(x) = 1 - \tanh^2(x) \quad (2.4)$$

Abbildung 2.4: Formel der Aktivierungsfunktion: TanH

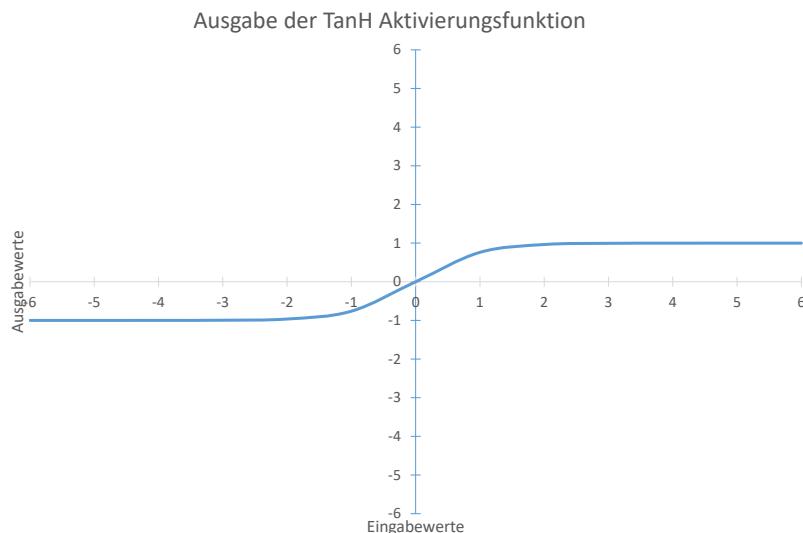


Abbildung 2.5: Graph der Aktivierungsfunktion: TanH

Im Vergleich zu der Identitätsfunktion, fällt TanH und seine Ableitung, sehr komplex aus, was die Rechenzeit erhöht. Die Rechenzeit bei der Fehlerrückführung kann jedoch reduziert werden, indem das Ergebnis der Aktivierungsfunktion bei der Forward Propagation zwischengespeichert wird, da dieser Wert in der Ableitung ein weiteres Mal verwendet wird. TanH kann nützlich sein, große Werte der Aktivierungen einzudämmen, was je nach Aufgabe der Eingabewerte allerdings auch dazu führen kann, dass Informationsgehalt der Eingabewerte potentiell stark reduziert werden kann. Um diesen Effekt zu sehen, nehmen wir an das $z_n^i = 5$, dann folgt $\tanh(5) = 0,999909204$. Wenn $z_n^i = 10$, verändert sich die Ausgabe von TanH nur minimal: $\tanh(10) =$

0,99999995. Obwohl sich der Eingabewert verdoppelt hat, kann dieser Unterschied nur sehr gering in der Ausgabe gefunden werden. Dieses Problem sorgt dafür, dass alle Eingabewerte über 1 und unter -1 sehr stark gegen 1 tendieren. TanH kann sowohl negative als auch positive Werte um dem Nullpunkt abbilden, ohne großen Informationsverlust, wenn auch nur in einem stark begrenzten Bereich.

2.4 Sigmoid

Die logistische Sigmoidfunktion, beschreibt, wie TanH eine 'S'-Kurve, allerdings nur zwischen 0 und 1.

$$\phi(x) = \frac{1}{1 + e^{-x}} \quad (2.5)$$

$$\phi'(x) = \phi(x)(1 - \phi(x)) \quad (2.6)$$

Abbildung 2.6: Formel der Aktivierungsfunktion: Sigmoid

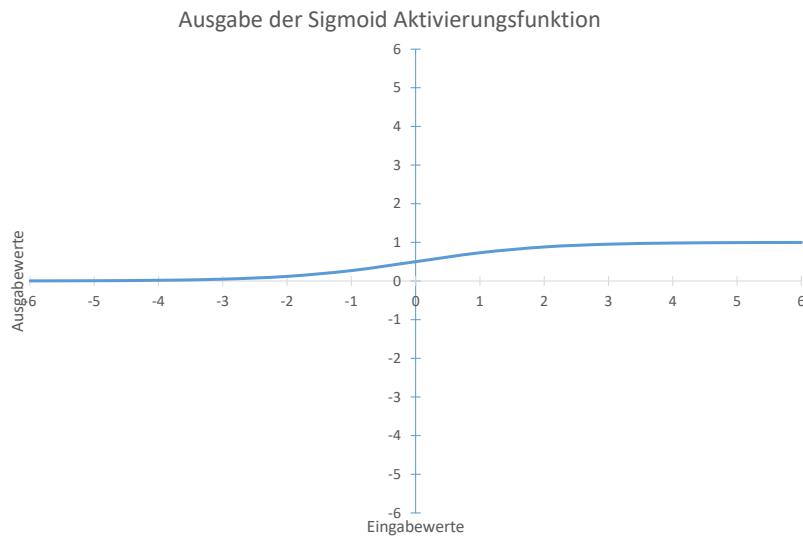


Abbildung 2.7: Graph der Aktivierungsfunktion: Sigmoid

Das hat den Nachteil, dass alle negativen Werte durch die Aktivierungsfunktion effektiv auf 0 abgebildet werden. Bei der Forward Propagation

kann dieser Umstand dazu führen, dass die entsprechenden Neuronen sich nicht mehr entwickeln können und auf 0 'stecken' bleiben, was die Qualität der Ergebnisse verschlechtern und die Effizienz des Lernprozesses verringern kann. Wie auch bei TanH kann das Ergebniss der Aktivierungsfunktion zwischengespeichert werden um den Rechenaufwand bei der Fehlerrückführung zu reduzieren. Bei der Forward Propagation bleibt jedoch der Rechenaufwand im Vergleich zu der Identitätsfunktion relativ hoch.

2.5 ReLU

ReLU, oder Rectified Linear Unit, ist eine stückweise lineare Aktivierungsfunktion, die im Jahr 2000 von Richard H. R. Hahnloser erstmals vorgestellt wurde [11]. Vor allem in den letzten Jahren erfreut sich ReLU von großer Beliebtheit [12] und kann in einer Reihe von Applikationen gefunden werden. [13] [1] [14]

$$\phi(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (2.7)$$

$$\phi'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (2.8)$$

Abbildung 2.8: Formel der Aktivierungsfunktion: ReLU

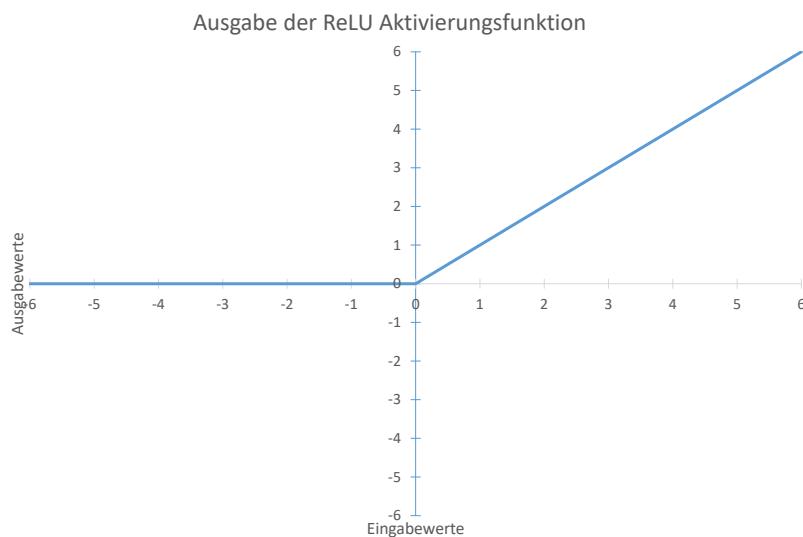


Abbildung 2.9: Graph der Aktivierungsfunktion: ReLU

In der Praxis hat sich ReLU als äußerst effizient beim Trainieren von Tiefen neuronalen Netzwerken erwiesen. In Ihrer Arbeit zur Objekterkennung in Bildern haben Alex Krizhevsky, Ilya Sutskever und Geoffrey E. Hinton zeigen können, dass ein Netzwerk mit ReLU Aktivierungsfunktionen um den Faktor 6 schneller war als ein identisches Netzwerk mit TanH Aktivierungsfunktionen, eine Fehlerrate von unter 25% zu erreichen. [1] ReLU scheint seine Effektivität seinen spärlichen Aktivierungen zu verdanken. Da alle Eingabewerte unter Null keine Reaktion, bzw. Ausgabe des Neurons konstituieren, erinnert das Verhalten von ReLU mehr an biologische Neuronen, deren Aktivierungen, laut Xavier Glorot, von einem Rectifier approximiert werden können. [15] In bestimmten Fällen führt das Verhalten von ReLU jedoch zu dem sogenannten "Dying ReLU" Problem, indem Neuronen aufgrund der Eingabewerte und Gewichte stets eine Null an die nächsten Neuronen ausgeben. Dieses Verhalten kann die Qualität der Ergebnisse des Netzwerks verringern, da ein Großteil "toter" Neuronen keinen Informationsgehalt darstellen kann. Um diesen Problem entgegenzuwirken, wurde das, erstmals 2013 von Andrew L. Maas verwendete, LeakyReLU (aus dem Englischen: Leaky = undicht) entworfen. Dazu mehr im Kapitel zu möglichen Verbesserungen.

Kapitel 3

Implementierung

Für diese Arbeit wurde der 1994 von Martin Riedmiller entwickelte Simulator für neuronale Netzwerke "N++" um die Aktivierungsfunktion ReLU erweitert. Nach der initialen Entwicklung, wurde N++ unter anderem von Thomas Gabel, Prof. an der Frankfurt University of Applied Sciences, weiterentwickelt und ist unter Prof. Gabel unter anderem in der simulierten Roboterfußballmannschaft "FraUNIited" in verschiedenen Bereichen verwendet worden, darunter das Steuern einzelner Agenten/Spieler und das Dekodieren gegnerischer Kommunikation. [16] In den folgenden Abschnitten werde ich auf die Fähigkeiten von N++ sowie auf die konkrete Erweiterung eingehen.

3.1 Fähigkeiten

Der in C++ geschriebene Simulator gibt dem Anwender die Möglichkeit mit wenigen Zeilen Code neuronale Netzwerke zu erstellen, konfigurieren, verwenden, abzuspeichern und gespeicherte Netzwerke wieder zu laden. Durch das Abspeichern des Netzwerks kann nach dem Trainingsprozess in den Gewichten nach Anomalien gesucht werden, wie extrem kleine oder große Gewichte. Die Fähigkeit Netzwerke zu laden, ermöglicht dem Anwender bereits trainierte Netzwerke wiederzuverwenden. N++ bietet zu dem Zeitpunkt der Anfertigung dieser Arbeit 2 Lernalgorithmen, sowie 3 Aktivierungsfunktionen. Aktivierungsfunktionen können für jedes Neuron individuell oder für jede Neuronenschicht definiert werden, das die Kombinationen von verschiedenen Aktivierungsfunktionen ermöglicht. Die verfügbaren Lernalgorithmen sind Backpropagation (BP) mit optionalen Weight Decay und Momentum, sowie Resilient Propagation (RPROP). Die Aktivierungsfunktionen sind die in dieser Arbeit bereits betrachteten TanH, Sigmoid und Identität.

3.2 Erweiterung

Der Simulator wurde um die Aktivierungsfunktion "ReLU" erweitert, die, wie bereits in dem Kapitel über Aktivierungsfunktionen beschrieben, eine stückweise lineare Funktion der folgenden Struktur darstellt:

$$\phi(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (3.1)$$

$$\phi'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (3.2)$$

Abbildung 3.1: Formel der Aktivierungsfunktion: ReLU

Wie sich ReLU in N++ im Vergleich zu den anderen Aktivierungsfunktionen verhält, werde ich in dem nächsten Kapitel genauer beschreiben.

Kapitel 4

ReLU im Vergleich

4.1 Trainingskonventionen – bekannte Probleme

Das Trainieren von künstlichen neuronalen Netzwerken, bzw. die oben beschriebene Fehlerrückführung, findet aus Performancegründen meist nicht pro Eingabewert statt, sondern in sogenannten Batches (aus dem Englischen: Batch = Stapel/Ladung). Ein Batch besteht aus einer Teilmenge der Trainingsdaten, die wiederum aus einer Reihe von Eingabewerten und dem gewünschten Ergebnis zu den Eingabewerten bestehen. Wenn den Trainingsdaten das richtige Ergebnis beiliegt, spricht man von "überwachtem maschinellem Lernen". Anstatt für jedes Trainingsbeispiel eine Fehlerrückführung vorzunehmen, wird beim Batch-Learning der Durchschnittsfehlerwert verwendet, nach dem jedes Trainingsbeispiel mit der Forward Propagation durch das Netzwerk gegangen ist. Mit diesem durchschnittlichen Fehlerwert wird am Ende jedes Batches die Fehlerrückführung durchgeführt. Die Größe eines Batches kann je nachdem wie oft der Lernprozess durchgeführt werden soll nur wenige Testeinheiten beinhalten oder die gesamte Trainingsmenge. Sollten alle Einheiten in der Trainingsmenge abgearbeitet worden sein, spricht man von einem Epoch (aus dem Englischen: Epoch = Epoche). Es ist üblich mehrere Epochs durchzuführen um das Netzwerk besser zu trainieren. Dieser Prozess ist analog zum Training beim Menschen - Übung macht den Meister. Ein Datensatz stellt die gesamte Menge an Trainingsdaten da, die sowohl für das Testen als auch das Trainieren eines Netzwerks verwendet werden. Um ein Netzwerk zu Trainieren und anschließen zu evaluieren, wird der Datensatz gemischt und in Trainingsmenge und Testmenge aufgeteilt. In dieser Arbeit wurden die verwendeten Datensätze stets in 80% Trainingsdaten und 20% Testdaten aufgeteilt. Dieser Vorgang soll sicherstellen, dass das trainierte Netzwerk auch bisher unbekannte Eingabewerte korrekt einordnen kann und damit flexibel genug ist. Sollte ein Netzwerk nur auf die trainierten Eingabewerte zugeschnitten sein, spricht man von Overfitting (aus dem Englischen: Überanpassung). Ein neuronales Netzwerk, das unter

Overfitting leidet, verfügt über eine täuschend niedrige Fehlerrate für den verwendeten Datensatz. Konfrontiert man das Netzwerk jedoch mit relativ unbekannten Kombinationen von Eingabewerten, versagt das Netz. Sollte ein Netzwerk zu wenig oder mit schlechen Daten trainiert sein, kann das Netzwerk keine aussagekräftigen Ergebnisse liefern - man spricht hier von Underfitting (aus dem Englischen: Underfitting = Unteranpassung). Je nach Implementierung der Aufteilung in Trainings- und Testdaten muss der Datensatz vorher gemischt werden, für den Fall dass der Datensatz sortiert ist. Sollte die Trainingsmenge nicht aus Einträgen mit angemessener Diversität bestehen, droht Overfitting. Der in dieser Arbeit verwendete Ansatz spaltet den eingelesenen Datensatz auf, indem die ersten 80% der Einträge der Trainingsmenge und der Rest der Testmenge zugewiesen werden. Deswegen ist hier ein präventives mischen notwendig.

4.2 Ziele

Ziel dieser Arbeit ist es, die Qualität und Geschwindigkeit von ReLU, im Vergleich zu TanH, Sigmoid und Identität, in mit N++ erstellten neuronalen Netzwerken zu messen. Dazu werden 3 verschiedene Datensätze verwendet, die jeweils eine unterschiedliche Menge von Ein- und Ausgabewerten aufweisen und verschiedene Szenarien darstellen. Von der Kategorisierung von Blumenarten anhand ihrer Größe, über die Qualität von Rotwein anhand des Säure- und Alkoholgehalts etc. bis hin zu Einschätzungen, ob ein Bankkunde an einem Finanzprodukt Interesse hat anhand von Alter, Einkommen und vielem mehr. Zudem soll die Geschwindigkeit von ReLU-Netzwerken in N++ im Vergleich zu den anderen Aktivierungsfunktionen getestet werden.

4.3 Versuchsaufbau

Jede Kombination von Lernalgorithmus und Aktivierungsfunktion wurde mit drei verschiedenen Datensätzen jeweils 10x getestet. Pro Durchlauf wurden die Netzwerke und die Datensätze unterschiedlich initialisiert, was in diesem Fall bedeutet, dass initial die Gewichte andere Werte besitzen und dass die Trainings- und Testmengen unterschiedliche Elemente enthalten. Dazu wurde in C++ die "rand(Seed)" Funktion verwendet, mit der man den Ursprungswert für die Zufallszahlenerzeugung definieren kann. So können vergleichbare Ergebnisse für die einzelnen Netzwerke erzielt werden, da für jede Kombination von Lernalgorithmus und Aktivierungsfunktion die selben 10 Seeds (aus dem Englischen: Seed = Samen) verwendet wurden. Jede der Kombinationen wurde 10x mit verschiedenen Seeds getestet, um die Auswirkungen von Anomalien im Trainings- und Testprozess zu reduzieren. Jedes Netzwerk wurde mit 150 Epochs mit der gegebenen Trainingsmenge trainiert. Die Netzwerke bestanden stets aus einer Eingabeschicht mit

n Neuronen, wobei n die Anzahl von verschiedenen Eingabewerten für den Datensatz darstellt, einem Hidden Layer mit $n * 2$ Neuronen und einer Ausgabeschicht mit einem Neuron pro möglichem Ausgabewert. Das Ausgabe neuron, das den höchsten Wert aufweist, ist das vom Netzwerk ausgewählte Ergebnis für die gegebenen Eingabewerte.

Da die Eingabewerte sich in Ihrer Größe stark unterscheiden können, wurde beim Einlesen der Datensätze jeder Eingabewert normalisiert. Das kann zur Folge haben, dass Aktivierungsfunktionen wie Sigmoid effizienter lernen können als ohne Normalisierung, da die Ableitung von Sigmoid große Werte schnell gegen 0 tendieren lässt und damit den Lernprozess nicht voran treibt. Um mit der Normalisierung den Informationsgehalt der Eingabewerte möglichst wenig zu reduzieren, wurde die folgende Funktion verwendet:

$$\mathbb{Z} = \frac{X - \mu}{\sigma} \quad (4.1)$$

$$(4.2)$$

Abbildung 4.1: Formel der Aktivierungsfunktion: Sigmoid

Wobei Z den transformierten Wert, X den zu transformierenden Wert, μ den arithmetischen Mittelwert und σ die Standardabweichung darstellt. Für jeden Trainingsdurchlauf wurde der quadratische Fehler des Netzwerks notiert, sowie die Anzahl von richtigen und falschen Ergebnissen beim Testdurchlauf. Für die Geschwindigkeitsmessung wurde der größte Datensatz verwendet und über 1000 Epochen das Netzwerk trainiert. Dieser Prozess wurde für Netzwerke mit einem, zwei und drei Hidden Layern durchgeführt.

Für die Backpropagation wurde weder Weight-Decay noch Momentum verwendet, jedoch unterschiedliche Lernraten für unterschiedliche Aktivierungsfunktionen und Datensätze:

Datensatz	Identität	Sigmoid	TanH	ReLU
Iris	0,0002	0,0001	0,0001	0,0001
Rotwein-Qualität	0,0001	0,0001	0,0001	0,0001
Bank	0,000001	0,00001	0,00001	0,00001

Tabelle 4.1: Lernraten für die Datensätze für Backpropagation

Diese Werte haben sich durch die Durchführung von Testläufen ergeben. Es besteht die Möglichkeit, dass es Parameter für die Netzwerke gibt, die bessere Ergebnisse erzielen könnten. Dieses trifft auch auf die Konfiguration der Netzwerke sowie die Art der Normalisierung der Eingabewerte zu. Diese Parameter können für jeden Datensatz unterschiedlich sein. Die Ergebnisse dieser Arbeit bleiben jedoch vergleichbar, wenn auch nicht optimal.

4.4 Erwartungen

Anhand der Popularität, Errungenschaften und Eigenschaften von ReLU, ist die Erwartung, dass die Netzwerke die mit ReLU trainiert wurden mit weniger Iterationen die Fehlerrate reduzieren können als Netzwerke, die mit anderen Aktivierungsfunktionen trainiert wurden. Ein Geschwindigkeitstest sollte einen erkennbaren Unterschied zwischen ReLU-Netzwerken und TanH-/Sigmoid-Netzwerken zeigen. Generell sollten die ReLU-Netzwerke im Durchschnitt nicht wesentlich schlechter, wenn nicht besser als die anderen Netzwerke sein, unabhängig vom verwendeten Datensatz.

4.5 Datensätze

In dieser Arbeit wurden 3 Datensätze aus dem UCI Machine Learning Repository [17] verwendet.

1. Iris [18]
2. Rotwein-Qualität [19]
3. Bank [20]

Eigenschaften der Datensätze

Datensatzname	Anzahl der Einheiten	Anzahl der Eingabewerte	Anzahl der Ausgabewerte
Iris	150	4	3
Rotwein-Qualität	1599	12	11
Bank	45211	17	2

Tabelle 4.2: Eigenschaften der verwendeten Datensätze

Mit dem Iris-Datensatz soll das Netzwerk lernen, 3 verschiedene Arten von Irisblumen anhand der Größe zu erkennen. Netze, die mit dem Rotwein-Qualität Datensatz trainiert wurden, sollen die Fähigkeit besitzen, anhand von Säuregehalt, Alkoholgehalt, Zucker etc. die Qualität des Weins auf einer Skala von 0-10 zu erkennen, wobei 0 der schlechteste und 10 der beste Werte ist. Der Bank-Datensatz enthält Informationen über Bankkunden, wie das Alter, Beruf, ob ein Kredit aufgenommen wurde und viele mehr. Anhand dieser Informationen soll vorhergesagt werden können, ob der Kunde Interesse an einem Finanzprodukt hat. Die Antwortmöglichkeiten sind hierbei "Ja" oder "Nein".

4.6 Vergleich

Für den Vergleich von ReLU mit den anderen Aktivierungsfunktionen, gehe ich zuerst auf die Qualität der Ergebnisse ein, die mit den trainierten Netzwerken erzielt wurden. Dabei werden für jeden verwendeten Datensatz die Ergebnisse mit Backpropagation gefolgt von RPROP vorgestellt. Die Analyse dieser Ergebnisse findet anschließend im Evaluationsabschnitt statt.

4.6.1 Iris

Backpropagation

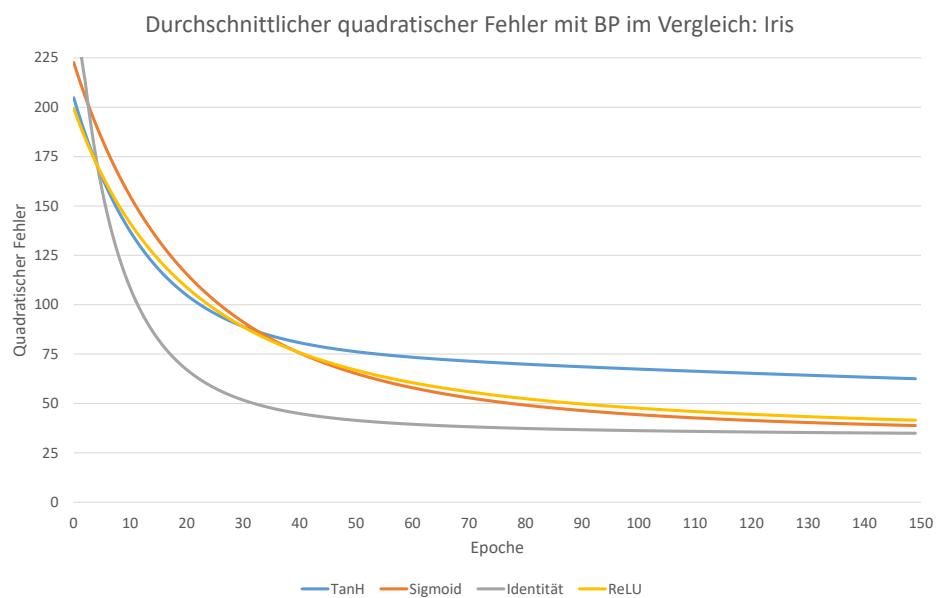


Abbildung 4.2: Durchschnittliche quadratische Fehler mit Backpropagation: Iris

Der Iris-Datensatz hat aufgrund der Normalisierung im Durchschnitt Werte zwischen -1,5 und 1,5 angenommen, was dem Lernprozess von Sigmoid und TanH zugute kommt. Trotzdem lassen die Ergebnisse erkennen, dass TanH mit Backpropagation für diesen Datensatz nicht die lokalen Minima finden kann, die bei den anderen Aktivierungsfunktionen gefunden wurden. Nach 150 Epochs haben, bis auf TanH, alle Aktivierungsfunktionen sich demselben Wert genähert. Die Identitätsfunktion hat dabei im Durchschnitt den schnellsten Fall des quadratischen Fehlers realisieren können, welche sich nach ca. 50 Epochs jedoch nicht mehr relevant verbessert. Die begünstigte Sigmoidfunktion beschreibt eine nahezu identische Kurve wie ReLU.

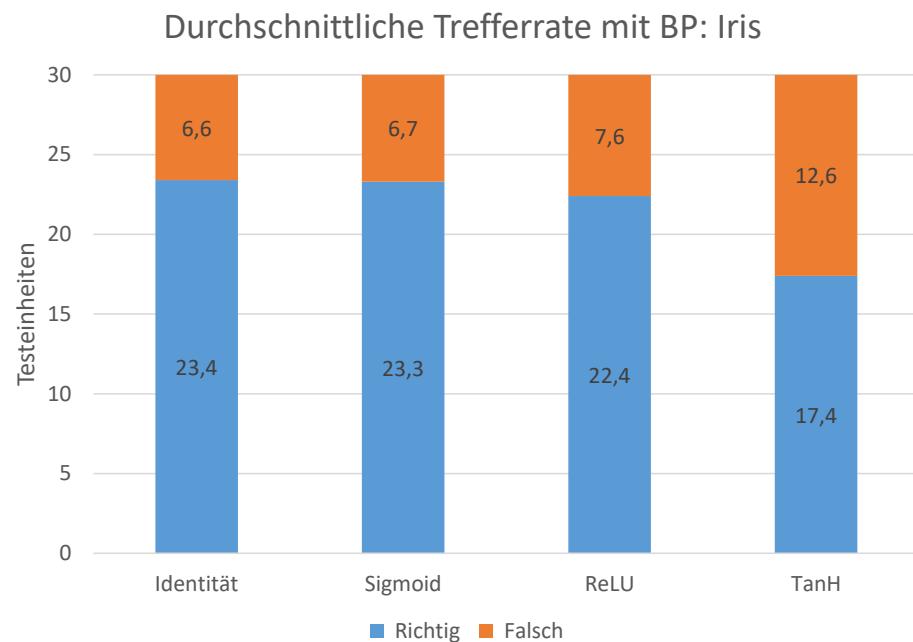
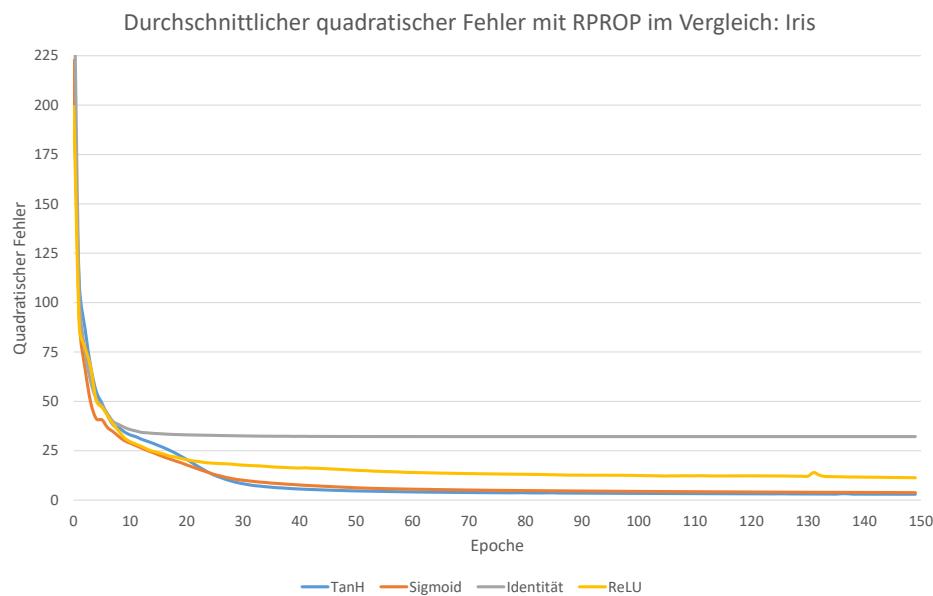


Abbildung 4.3: Durchschnittliche Trefferrate mit Backpropagation: Iris

Die trainierten ReLU-Netzwerke haben mit Backpropagation im Durchschnitt eine 74,6% Trefferrate, was knapp hinter Sigmoid mit 77,6 % liegt.

RPROP**Abbildung 4.4:** Durchschnittliche quadratische Fehler mit RPROP: Iris

Mit RPROP nähert sich die durchschnittliche Lernkurve wesentlich mehr an das Optimum (0 auf der y-Achse). Wie auch mit Backpropagation, kann TanH nicht die Fehlerwerte der restlichen Aktivierungsfunktionen erreichen. Der Fehlerwert von ReLU befindet sich nach 150 Epochs ebenfalls über dem von Sigmoid und Identität, das jedoch in diesem Fall nichts über die durchschnittliche Trefferrate aussagt.

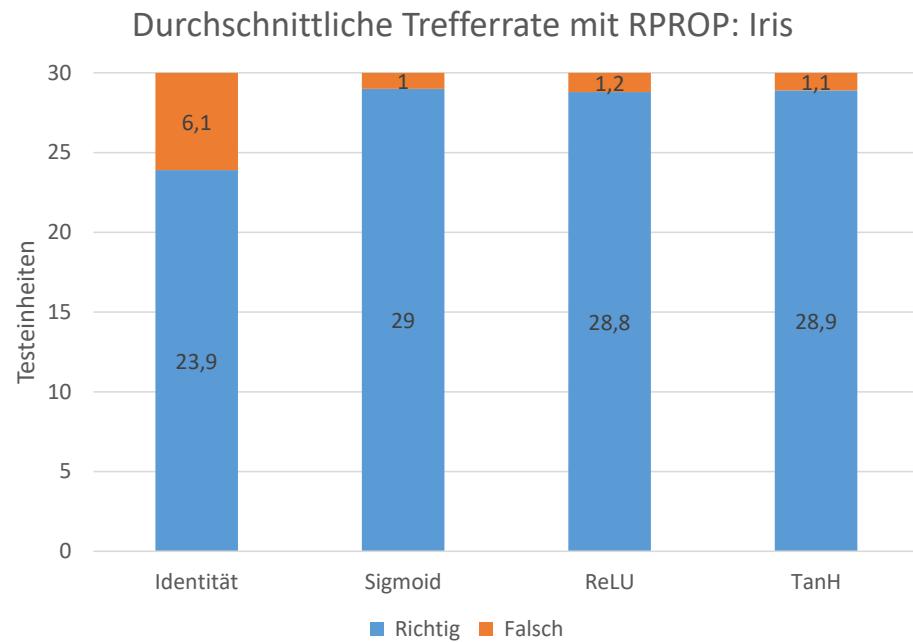


Abbildung 4.5: Durchschnittliche Trefferrate mit RPROP: Iris

Im Durchschnitt schaffen es die Rprop-Netzwerke mit Idenitätsfunktion 79,6% der Eingabewerte richtig zu kategorisieren. Klar hinter ReLU mit 96,0% und Sigmoid mit 96,6%.

4.6.2 Rotwein-Qualität

Backpropagation

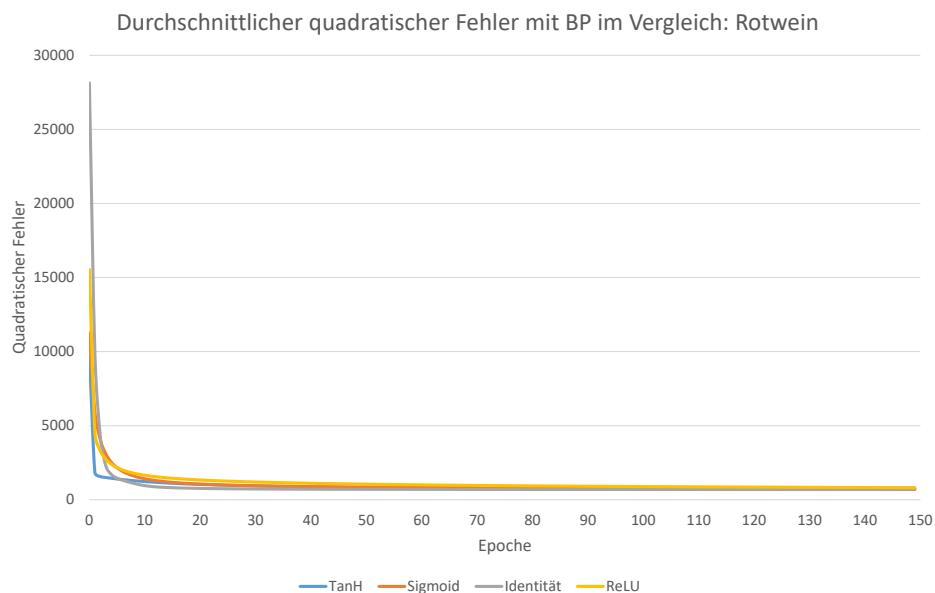


Abbildung 4.6: Durchschnittliche quadratische Fehler mit Backpropagation: Rotwein-Qualität

Der Lernprozess stößt unter Backpropagation schnell an seine Grenzen. Nach ca. 20 Epochs verändert sich die durchschnittliche Fehlerrate nur noch geringfügig, unabhängig von der Aktivierungsfunktion. Jedoch schaffen es die TanH-Netzwerke im Durchschnitt schneller dieses Plateau zu erreichen. Gefolgt von Identität und ReLU.

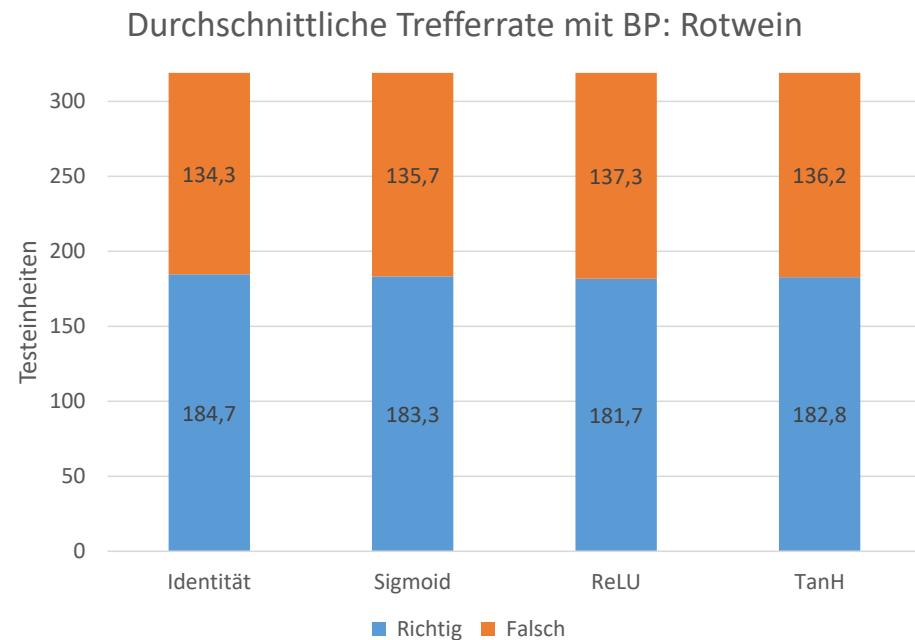


Abbildung 4.7: Durchschnittliche Trefferrate mit Backpropagation: Rotwein-Qualität

Die trainierten Netzwerke scheinen mit dem Rotwein-Qualität-Datensatz unter Backpropagation, als auch unter RPROP nur geringfügig aussagekräftige Ergebnisse liefern zu können, unabhängig von der verwendeten Aktivierungsfunktion. Eine eventuelle Erklärung betrachten ich in dem Evaluationsabschnitt. (Abschnitt 4.7) Mit Backpropagation konnten die Identitäts-Netzwerke im Durchschnitt 57,8% der Testeingaben korrekt kategorisieren. Knapp darunter befindet sich die Sigmoidfunktion mit 57,8%. ReLU befindet sich auf dem letzten Platz mit 56,9%.

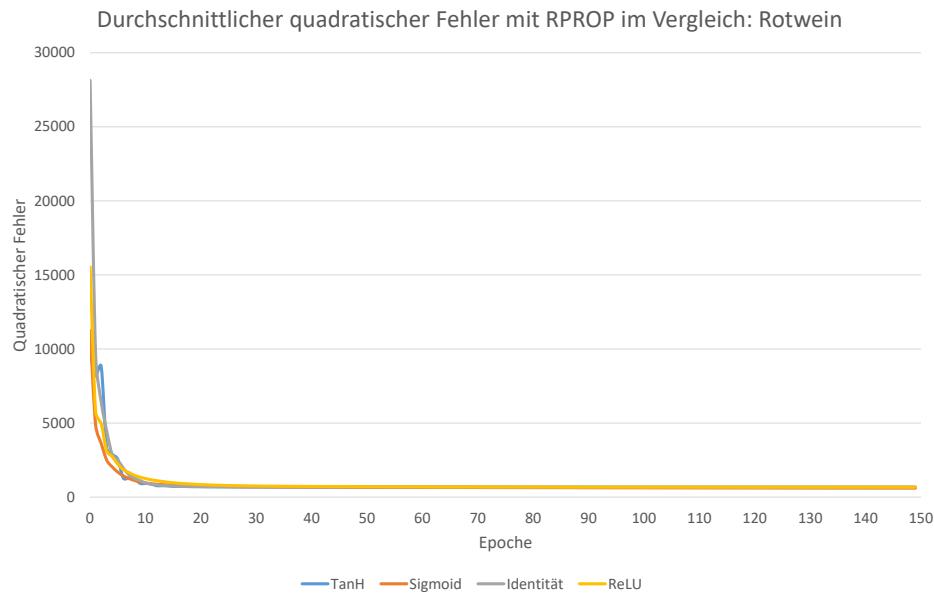
RPROP

Abbildung 4.8: Durchschnittliche quadratische Fehler mit RPROP: Rotwein-Qualität

Der Verlauf des durchschnittlichen quadratischen Fehlers mit RPROP, zeigt bei TanH initial ein leichtes Stottern, bei dem der Fehler im dritten Epoch wieder steigt. Jedoch konvergieren alle Netzwerke nach ungefähr 25 Epochs auf einen durchschnittlichen quadratischen Fehlerwert, der in den restlichen 125 Epochs nur extrem geringfügig abnimmt.

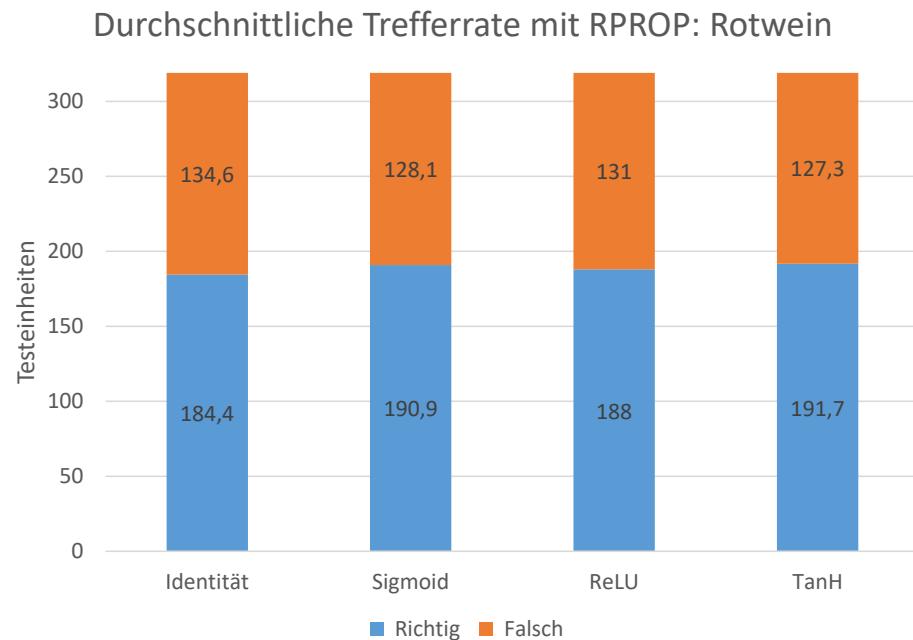


Abbildung 4.9: Durchschnittliche Trefferrate mit RPROP: Rotwein-Qualität

Mit RPROP steigt die durchschnittliche Trefferquote für jede Aktivierungsfunktion um ca. 1%. TanH und Sigmoid kategorisieren im Durchschnitt weiterhin besser als Identität und ReLU, jedoch rutscht Identität mit RPROP auf den letzten Platz mit einer durchschnittlichen Trefferquote von 57,8%. ReLU hingegen steht mit 58,9% knapp über Identität und 1,1% unter TanH mit 60,0%.

4.6.3 Bank

Backpropagation

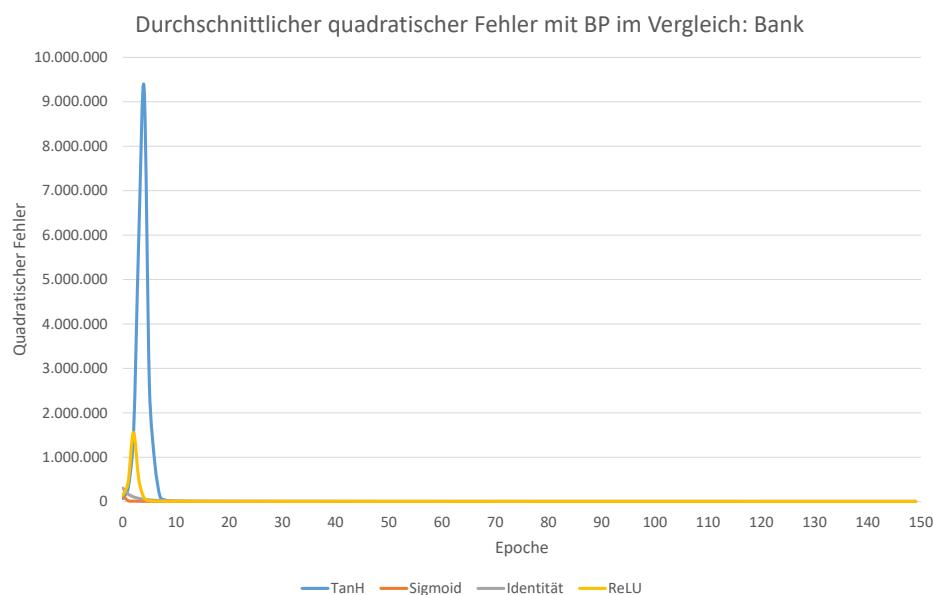


Abbildung 4.10: Durchschnittliche quadratische Fehler mit Backpropagation: Bank

Sowohl TanH als auch ReLU - wenn auch bei weitem nicht so extrem - beschreiben einen beachtlichen Hügel in ihrem durchschnittlichen quadratischen Fehler. Wie auch bei dem Rotwein-Qualitäts-Datensatz nähern sich alle Aktivierungsfunktionen nach wenigen Epochen einem Fehlerwert, von dem nur noch minimal abgewichen wird. In diesem Fall haben alle Aktivierungsfunktionen nach ca. 15 Epochs ungefähr denselben durchschnittlichen quadratischen Fehlerwert von ca. 6000.

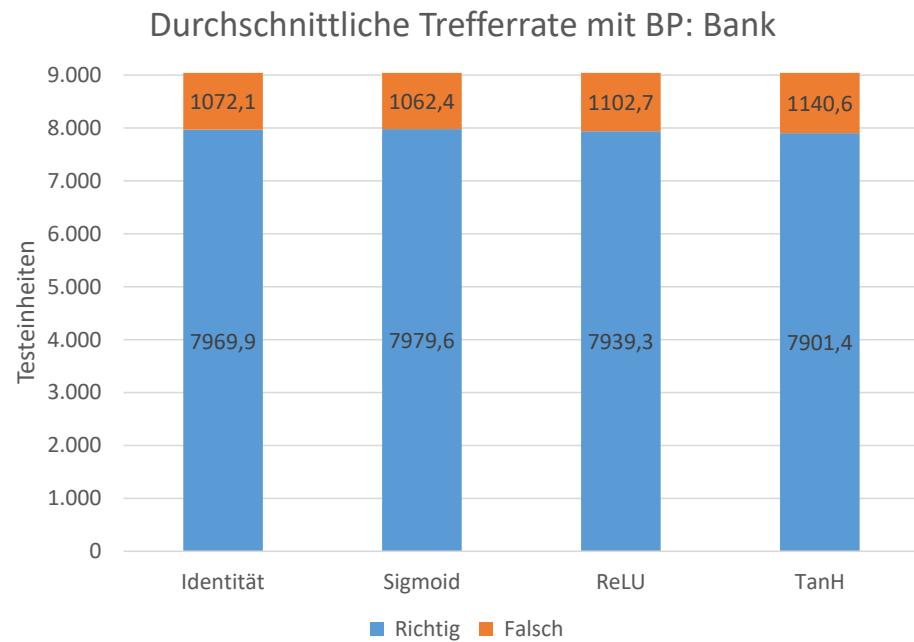
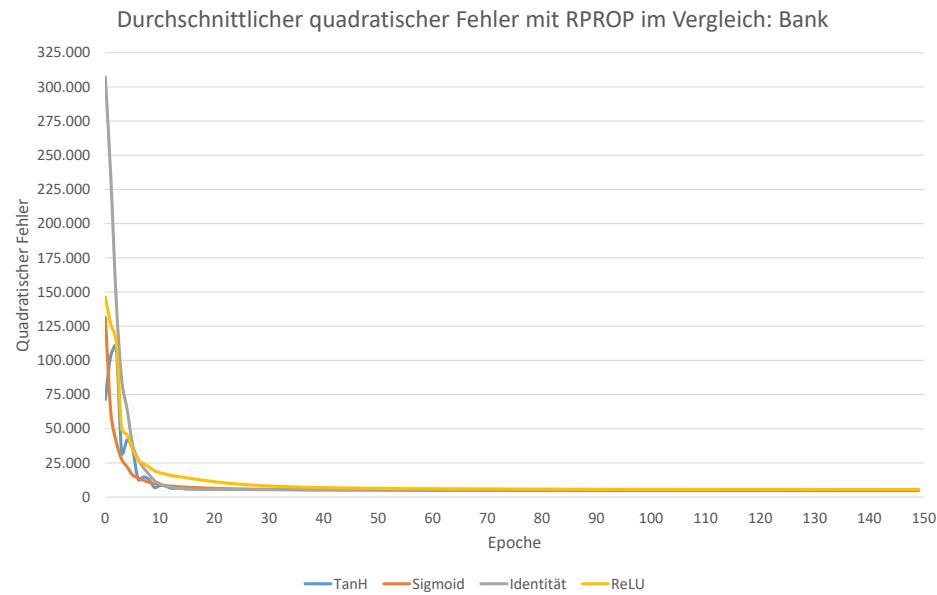


Abbildung 4.11: Durchschnittliche Trefferrate mit Backpropagation: Bank

Alle trainierten Netzwerke liefern ungefähr gleich gute Ergebnisse, wenn es zu der Testmenge kommt. ReLU landet auf dem dritten Platz mit einer durchschnittlichen Trefferrate von 88,0%, wohingegen Sigmoid mit 88,2% mit nur wenig Abstand auf Platz 1 landet.

RPROP**Abbildung 4.12:** Durchschnittliche quadratische Fehler mit RPROP: Bank

Mit RPROP kann das Ausschlagen des durchschnittlichen quadratischen Fehlerwerts bei ReLU verhindert werden. TanH bricht in den ersten 10 Epochs mehrmals aus, gefolgt von steilen Verbesserungen. Während sich die restlichen Aktivierungsfunktionen nach ungefähr 15 Epochs aneinander angenähert haben, benötigt ReLU noch ca. 45 weitere Epochs um in Reichweite der anderen Aktivierungsfunktionen zu gelangen.

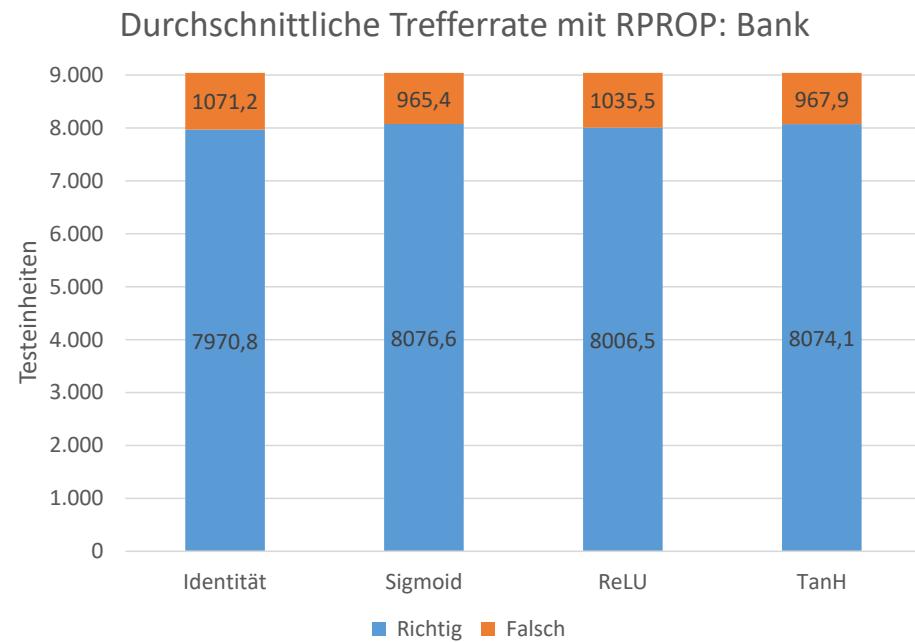


Abbildung 4.13: Durchschnittliche Trefferrate mit RPROP: Bank

Die langsame Annäherung von ReLU und die restlichen Aktivierungsfunktionen spiegelt sich nicht klar in den Testdurchläufen wieder. Alle Aktivierungsfunktionen produzieren auch mit RPROP sehr ähnliche Ergebnisse, mit Sigmoid an erster Stelle mit einer durchschnittlichen Trefferrate von 89,3%. ReLU befindet sich auf Platz 3 mit 88,5%.

4.6.4 Geschwindigkeit

Die Annahme, dass ReLU einen Geschwindigkeitsvorteil gegenüber TanH und Sigmoid besitzt konnte sich durch die, für diese Arbeit durchgeführten Experimente beweisen. Jedoch stellt sich der Unterschied für kleine Datensätze und kleine Netzwerke als vernachlässigbar heraus.

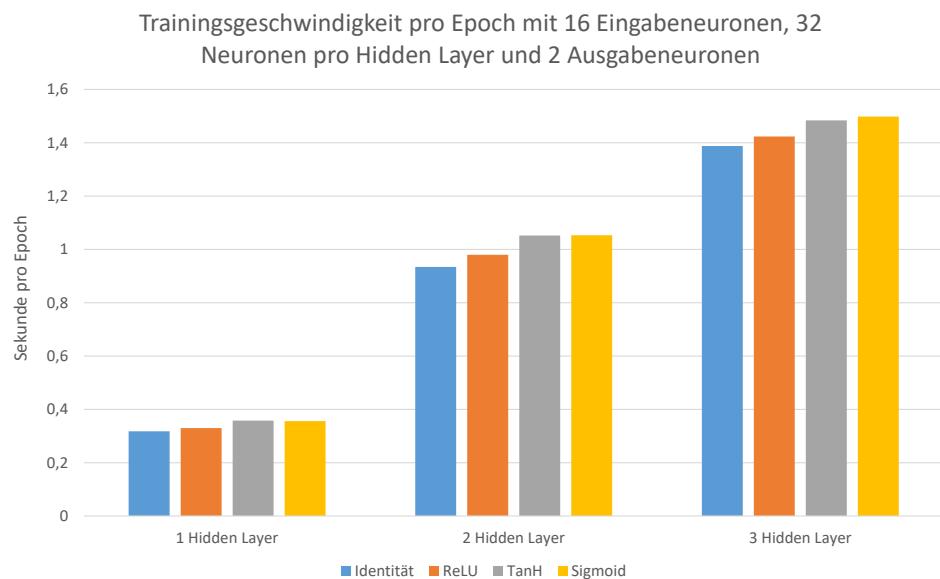


Abbildung 4.14: Durchschnittliche Trainingsgeschwindigkeit pro Epoch mit dem Bank Datensatz

Im Durchschnitt benötigt ReLU in einem Netzwerk mit 16 Eingabeneuronen, einem Hidden Layer mit 32 Neuronen und 2 Ausgabeneuronen pro Epoch ca. 0,028 Sekunden weniger als TanH und Sigmoid. Fügt man dem Netzwerk ein weiteres Hidden Layer mit gleich Neuronenanzahl hinzu, ist ReLU im Durchschnitt ca. 0,071 Sekunden schneller.

4.7 Evaluierung

Mit den verwendeten Kombinationen von Netzwerkaufbau (Anzahl von Hidden Layer und Hidden Layer Neuronen), Datensätzen und Konfiguration der Lernalgorithmen, konnte ReLU sich zwar beweisen, übertraf jedoch nicht die bereits vorhandenen Aktivierungsfunktionen in Hinsicht auf die Qualität der Ergebnisse. Dieser Umstand kann mehrere Gründe haben, auf die ich in den nächsten Seiten eingehen werde.

4.7.1 Iris

Das Sigmoid minimal besser mit dem Iris-Datensatz umgehen konnte, kann dem Aufbau der Eingabewerte zugeschrieben werden. ReLU hat gegenüber Sigmoid den Vorteil auch große Eingabewerte zu berücksichtigen, die bei Sigmoid aufgrund des "Vanishing gradient"-Problems ihren Informationsgehalt verlieren. Das vanishing-gradient-Problem (aus dem Englischen: Vanishing = verschwinden und Gradient = Gradient/Steigung) beschreibt den Effekt, den Sigmoid auf kleine und große Eingabewerte hat. Diese Eingabewerte werden von Sigmoid sehr nah an 0 und 1 abgebildet. Demnach ist Sigmoid effektiv wenn die Eingabewerte sich zwischen -2,5 und 2,5 bewegen. Dieser Umstand ist im Iris-Datensatz gewährleistet und durch die Normalisierung der Eingabewerte optimiert.

4.7.2 Rotwein-Qualität

Am Rotwein-Datensatz war zu erkennen, dass keine der verwendeten Aktivierungsfunktionen besonders gut abgeschnitten haben. Das kann an dem Datensatz liegen. Der Datensatz beschreibt die Qualität von Rotwein anhand von Säurewerten, Alkoholgehalt etc. Das Problem könnte hier daran liegen, dass die Ausgabewerte - eine Punktzahl von 0 bis 10 - keinem effektiv trainierbaren Muster folgen, da sie das Produkt von subjektiven Weinverkostungen sind. Es sollte nahezu unmöglich sein, eine hinreichend objektive Kategorisierung von subjektiven Trainingsdaten zu realisieren. Der Datensatz umfasst 1599 Einheiten. Selbst wenn die Verkostung der Weine von der selben Person durchgeführt werden sollte und sich dadurch ein Geschmacksmuster bilden könnte, ist anzunehmen, dass das Verkosten von 1599 Weinen die Geschmackswahrnehmung beeinflusst. Sollten die Weine über einen großen Zeitraum verkostet worden sein, kann sich in dieser Zeit ebenfalls die Geschmackswahrnehmung verändern. Die trainierten Netzwerke scheinen dieses Verhalten wiederzuspiegeln, da im Schnitt nur ca. 60% der Eingaben richtig kategorisiert werden konnten. Das Hinzufügen von extra Hidden Layern konnte ebenfalls keine relevante Verbesserung der Ergebnisse produzieren. Dennoch ist zu sehen, dass ReLU mit Backpropagation und den verwendeten Werten für die Lernrate im Durchschnitt nie die Fehler-

werte erreicht wie z.B. Sigmoid. ReLU kann jedoch ebenso gute Ergebnisse wie Sigmoid für den Rotwein-Datensatz erzielen, was die Qualität von ReLU zu bestätigen scheint.

4.7.3 Bank

Der Bank-Datensatz verfügt nach der Normalisierung der Eingabewerte über viele negative Werte, die von ReLU "ignoriert" werden. Dieser Umstand könnte die Leistung von ReLU verringern, jedoch kann man in den Ergebnissen zu dem Bank-Datensatz erkennen, dass ReLU sich mit Aktivierungsfunktionen wie Sigmoid durchaus messen kann und gleichzeitig weniger Zeit für die Trainingsphase verwendet werden muss. ReLU scheint mit dem Bank-Datensatz jedoch keinen gravierenden Vorteil, wenn überhaupt, zu erzielen. Das RPROP in jedem Fall bessere Ergebnisse als Backpropagation produziert hat, kann zum Teil an den verwendeten Parametern für die Backpropagation liegen. So konnte für diese Arbeit eine Reihe von möglichen Werten für die Lernrate gefunden werden. Ob diese Werte jedoch für den jeweiligen Datensatz optimal sind, kann nur mit weiteren Tests gezeigt werden. Eine gute Konfiguration zeichnet sich durch gute Leistungen des trainierten Netzwerks aus.

Kapitel 5

Mögliche Verbesserungen

5.1 Leaky ReLU

Um negativen Effekten wie dem Dying ReLU Problem entgegen zu wirken, kann in weiteren Erweiterungen von N++ ReLU verbessert werden. Darunter die sogenannte "Leaky ReLU" (oder LReLU) Erweiterung. Wie in dem Kapitel zu Aktivierungsfunktionen beschrieben, werden Neuronen beim Dying ReLU Problem "deaktiviert", in dem die Eingabewerte für ReLU über die Gewichte und Bias stets im negativen Bereich oder auf Null landen. Alle negative Eingabewerte werden von ReLU als Null abgebildet. Damit ist bei der Fehlerrückführung der Gradient des Neurons stets Null, was den Lernprozess zum Halten bringt. Um zu verhindern dass Neuronen mit ReLU Aktivierungsfunktionen "sterben", kann ReLU wie mit einer Steigung a erweitert werden:

$$\phi(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0 \end{cases} \quad (5.1)$$

$$\phi'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ a & \text{if } x < 0 \end{cases} \quad (5.2)$$

Abbildung 5.1: Formel der Aktivierungsfunktion: Leaky ReLU

Sei $a = 0,1$, so würde der Funktionsgraph wie folgt aussehen:

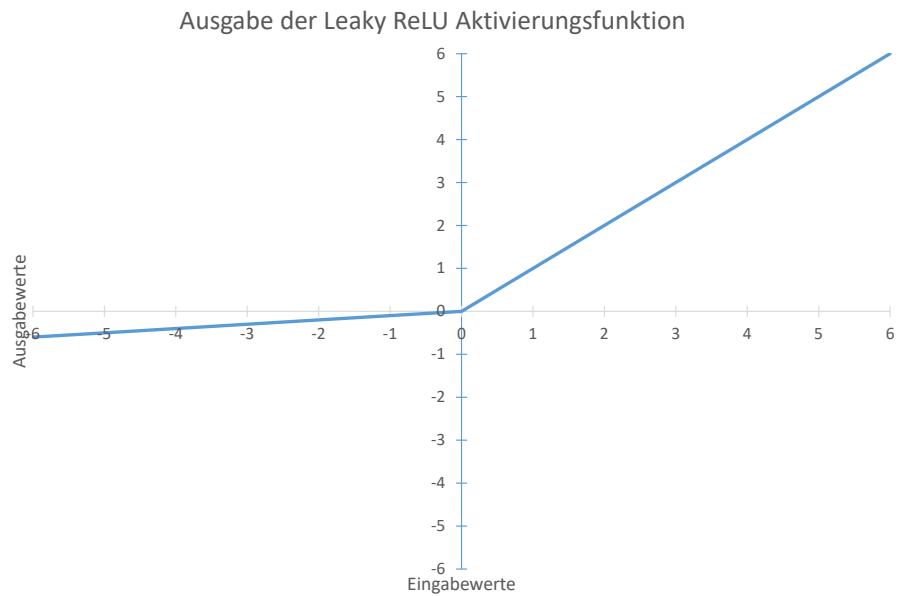


Abbildung 5.2: Graph der Aktivierungsfunktion: Leaky ReLU

Die entstandene Steigung sorgt dafür, dass während der Fehlerrückführung statt Null, die Steigung a verwendet wird. Somit kann auch bei negativen Werten das Neuron lernen, ohne ggf. auf Null "stecken zu bleiben". Um diese Verbesserung von ReLU zu testen, wurden dieselben Datensätze wie für den Vergleich mit anderen Aktivierungsfunktionen verwendet, ebenso wie die selben Netzwerkkonfigurationen und Zufallszahlen-Seed. Weiter unten werde ich auf den Unterschied zwischen ReLU und LReLU im Hinblick auf die Qualität der produzierten Ergebnisse eingehen.

5.2 Vergleich

Leaky ReLU mit einer Steigung von 0,01 konnte in 100% der durchgeföhrten Tests, sowohl mit Backpropagation als auch RPROP bessere Ergebnisse erzielen als reguläres ReLU. Auch wenn diese Verbesserung nur sehr gering ausfällt, kann z.B. beim Trainieren mit dem Bank-Datensatz und RPROP ein klarer Unterschied im Verlauf des Gradienten erkannt werden.

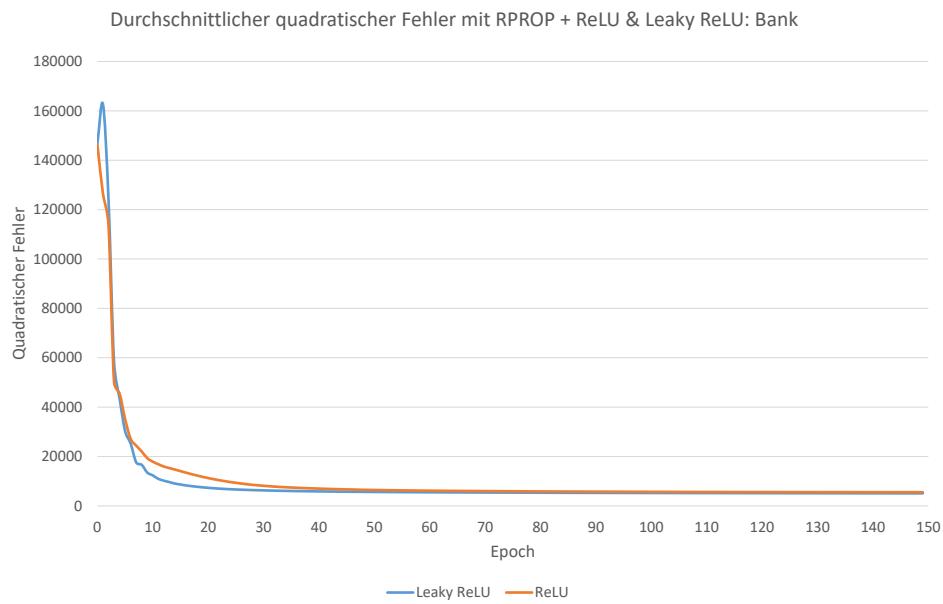


Abbildung 5.3: Durchschnittliche quadratische Fehler mit RPROP + ReLU und Leaky ReLU: Bank

Leaky ReLU schafft es im Durchschnitt ca. 20 Epochs schneller als ReLU auf einen Fehlerwert zu kommen. Ein ähnliches Verhalten kann mit dem Iris-Datensatz und RPROP beobachtet werden.

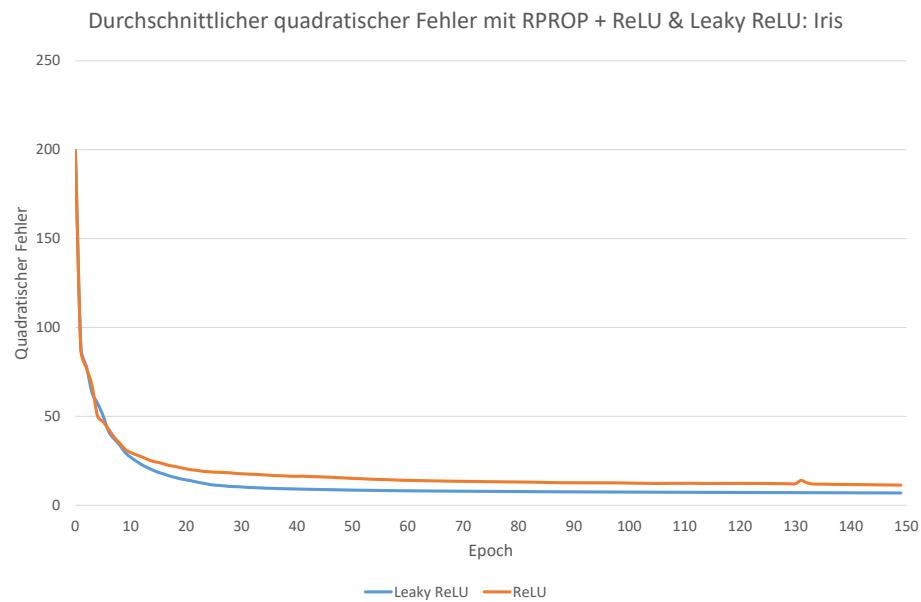


Abbildung 5.4: Durchschnittliche quadratische Fehler mit RPROP + ReLU und Leaky ReLU: Iris

In diesem Fall schafft reguläres ReLU nicht, den Fehlerwert von Leaky ReLU zu erreichen. Betrachtet man jedoch die Trefferraten der mit trainierten Netzwerken wird klar, dass die Verbesserung sich in diesen Fällen nur geringfügig in den Ergebnissen wiederfinden, unabhängig von BP und RPROP.

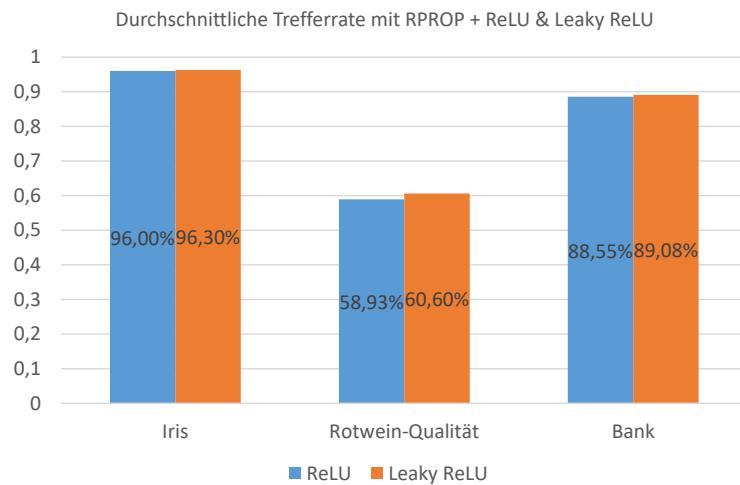


Abbildung 5.5: Durchschnittliche Trefferrate mit RPROP + ReLU und Leaky ReLU

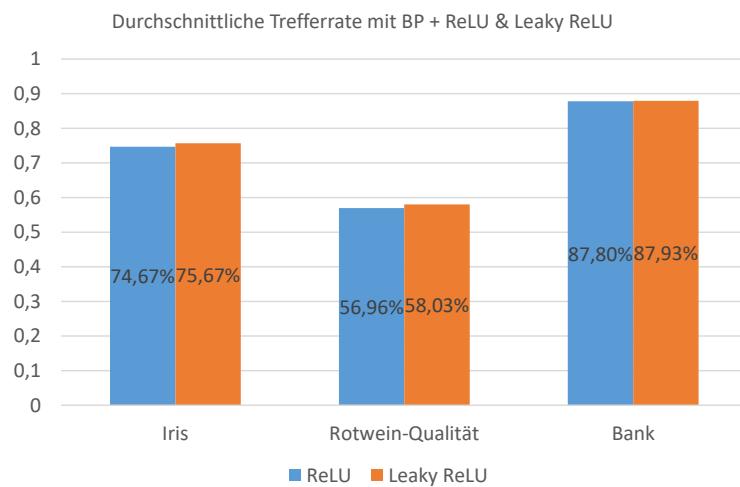


Abbildung 5.6: Durchschnittliche Trefferrate mit BP + ReLU und Leaky ReLU

So erzielt Leaky ReLU stets bessere Ergebnisse als ReLU, jedoch bewegt sich der Unterschied bei unter 1%.

Kapitel 6

Fazit

Abschließend kann man sagen, dass die Implementierung von ReLU und Leaky ReLU eine effektive Addition für N++ darstellt. So konnten zwar mit den verwendeten Datensätzen und Konfigurationen keine bahnbrechenden Qualitätssteigerungen erreicht werden, jedoch wurden ein Großteil der gestellten Erwartungen erfüllt. Es wurde gezeigt, dass ReLU schneller als TanH und Sigmoid ist und dennoch in der selben Anzahl von Trainingsepochs ungefähr gleichgute Ergebnisse liefern kann. Die Leaky Erweiterung von ReLU durch das Einführen einer Steigung für negative Werte konnte diese Qualität von ReLU noch weiter verbessern. Die in dieser Arbeit produzierten Ergebnisse zeigen zusätzlich, dass die ReLU-Aktivierungsfunktion nicht in jedem Fall bessere Ergebnisse liefert als z.B. die Sigmoidfunktion. Abhängig von Eingabewerten, Normalisierung der Eingabewerte, Netzaufbau und verwendetem Lernalgorithmus schneidet ReLU besser oder schlechter ab. Für die in dieser Arbeit verwendeten Datensätze konnten ReLU und Leaky ReLU ähnliche Ergebnisse produzieren wie ihre Konkurrenten, das jedoch nicht bedeutet, dass ReLU und Leaky ReLU nicht für andere Datensätze und Netzkonfigurationen eventuell bessere Ergebnisse erzielen könnten. Des Weiteren hat sich Leaky ReLU in jedem Experiment als effektiver als reguläres ReLU erwiesen. Die in dieser Arbeit durchgeführten Test konnten zeigen, dass die Datensätzen, die eine hohe Anzahl von negativen Eingabewerten aufwiesen durch Leaky ReLU besser verarbeitet werden konnten als durch ReLU.

Literaturverzeichnis

- [1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [2] Thomas Gabel, Philipp Klöppner, and Eicke Godehardt. Fra-united - team description 2018, 2018.
- [3] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
- [4] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.
- [5] Paul Werbos. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, 1974.
- [6] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.
- [7] Keras. <https://keras.io/getting-started/functional-api-guide/>.
- [8] Pytorch. <https://pytorch.org/>.
- [9] Martin Riedmiller and Heinrich Braun. Rprop-a fast adaptive learning algorithm. In *Proc. of ISCIS VII), Universitat*. Citeseer, 1992.
- [10] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Proceedings of the IEEE international conference on neural networks*, volume 1993, pages 586–591. San Francisco, 1993.
- [11] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

- [12] Dan Becker. Rectified linear units (relu) in deep learning. <https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning>.
- [13] Matthew D Zeiler, M Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521. IEEE, 2013.
- [14] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198. ACM, 2016.
- [15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.
- [16] Thomas Gabel, Philipp Klöppner, Eicke Godehardt, et al. Communication in soccer simulation: On the use of wiretapping opponent teams, 2018.
- [17] Uci machine learning repository. <https://archive.ics.uci.edu/ml/index.php>.
- [18] Iris data set. <https://archive.ics.uci.edu/ml/datasets/Iris>.
- [19] Wine quality data set. <https://archive.ics.uci.edu/ml/datasets/Wine+Quality>.
- [20] Bank marketing data set. <https://archive.ics.uci.edu/ml/datasets/Bank+Marketing>.