Rupprecht-Gymnasium München *Abiturjahrgang 2026*

S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Mathematik löst Probleme des Alltags

Leitfach: Mathematik

Thema der Arbeit:

Grundlegende Erkenntnisse zu neuronalen Netzen als Grundlage für künstliche Intelligenz, ergänzt durch die beispielhafte Erstellung eines solchen Netzes zur Klassifizierung von handgezeichneten geometrischen Formen

Verfasser: Till Brügmann Kursleiter: S. Linder

Abgabetermin: 11. November 2025

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 61 (7) GSO = Summe **: 2** (gerundet)	

Datum und Unterschrift des Kursleiters

Inhalt

I Geschichte und Anwendung künstlicher Intelligenz 2

II Entwicklung eines neuronalen Netzes 3

1 Das Neuron 3

1.1 Aufbau 3

1.2 Aktivierungsfunktionen 4

1.2.1 Heaviside-Funktion 4

1.2.2 Sigmoid-Funktion 4

1.3 Das XOR-Problem 5

2 Das vorwärtsgerichtete neuronale Netz 6

2.1 Architektur 6

2.1.1 Schichtenaufbau 7

2.1.2 Initialisierung und Vorhersage 8

2.2 Trainingsprozess 9

2.2.1 Ziel des Trainings 9

2.2.2 Backpropagation 9

2.2.3 Iteratives Lernen 12

3 Formenklassifizierung als Anwendungsbeispiel 12

3.1 Zielsetzung und Vorgehensweise 12

3.2 Anpassung von Trainingsparametern 13

3.2.1 Neuronenanzahl der versteckten Schicht 13

3.2.2 Lernrate 15

3.3 Auswertung 15

III Bewertung neuronaler Netze und Ansatz zur Weiterentwicklung der Formenklassifikation 16

Literaturverzeichnis 17

Darstellungsverzeichnis 18

Anhang 19

Erklärung 24

I Geschichte und Anwendung künstlicher Intelligenz

Künstliche Intelligenz (*abgekürzt* KI; *engl.* Artificial Intelligence, AI) ist eine der derzeit am häufigsten diskutierten Zukunftstechnologien unserer Gesellschaft und gewinnt in unserem Alltag zunehmend an Bedeutung. Sie kann als „Hard- und Softwaresystem, welches ein intelligentes Problemlösungsverhalten zeigt“[1], beschrieben werden. Aufgrund ihrer Lernfähigkeit ermöglicht KI die Funktionsweise von großen Sprachmodellen (LLMs) wie ChatGPT oder autonom fahrenden Autos, wie sie aktuell von BMW oder Tesla entwickelt werden.

Doch der Einsatz von maschinellem Lernen ist keineswegs eine technologische Neuheit. „Die Dartmouth-Konferenz, die im Sommer 1956 am Dartmouth College in Hanover, New Hampshire, stattfand, gilt als Gründungsereignis […] der Künstlichen Intelligenz“[2]. Schon dort diskutierten Wissenschaftler unter der Leitung von John McCarthy über die Fähigkeit von Maschinen, menschliche Intelligenz simulieren zu können[3]. Seitdem wurde künstliche Intelligenz in verschiedenste Richtungen weiterentwickelt und kommt schon seit vielen Jahren in Smartphones, Tablets und PCs in Form von Autokorrekturen oder Spamfiltern in E-Mail-Postfächern zum Einsatz.

Eine besondere Anwendungsmöglichkeit ist die Klassifizierung von handgezeichneten geometrischen Formen, wie sie häufig in Programmen zur Erstellung von digitalen Notizen genutzt wird. Im Rahmen dieser Arbeit soll dafür eine künstliche Intelligenz entwickelt werden, die anschließend in meiner Notizen-App *Productivity Pro* Verwendung findet.

Zunächst wird dazu das Konzept des künstlichen Neurons erläutert, das in seiner Funktionsweise dem biologischen Neuron des menschlichen Gehirns nachempfunden ist. Anschließend wird eine Vielzahl dieser Neuronen zu einem neuronalen Netz (NN) verknüpft, das die Grundlage vieler künstlicher Intelligenzen bildet. Zudem wird das Verfahren der sogenannten Backpropagation erläutert, welches das Training des neuronalen Netzes ermöglicht. Dies erfolgt am Beispiel eines vorwärtsgerichteten neuronalen Netzes, der einfachsten Variante aller NN-Architekturen. Im Anschluss wird das neuronale Netz auf die Klassifizierung von handgezeichneten Formen trainiert, wobei der Einfluss der Anzahl der Neuronen sowie der Lernrate untersucht wird. Schließlich erfolgt eine Auswertung der Arbeit sowie ein Ausblick auf potenzielle Ansätze zur Weiterentwicklung der Formenklassifikation.

---

[1] Lämmel, Uwe / Cleve, Jürgen: Künstliche Intelligenz, 6. Auflage, München: Carl Hanser Verlag, 2023, S. 15.
[2] Krauss, Patrick: Künstliche Intelligenz und Hirnforschung, Berlin: Springer, 2023, S. 118.
[3] Vgl. Krauss, 2023, S. 118.

II Entwicklung eines neuronalen Netzes

1 Das Neuron

1.1 Aufbau

> Ein Neuron [(vgl. Abbildung 1)] ist eine Verarbeitungseinheit, die die über die gewichteten Verbindungen eingehenden Werte geeignet zusammenfasst […] und daraus mittels einer Aktivierungsfunktion […] einen Aktivierungszustand ermittelt. Aus dieser Aktivierung bestimmt eine Ausgabefunktion die Ausgabe des Neurons.[4]

Dabei erfolgt die Zusammenführung der Eingabewerte durch eine sogenannte Propagierungsfunktion net_j, die die gewichteten Eingangssignale x_k mit ihren zugehörigen Gewichten w_k multipliziert und die daraus resultierenden Produkte summiert[5]. Hierbei bestimmen die Gewichte „die Signalanteile [der Eingabeparameter], die das [Neuron in einem neuronalen] Netz in Vorwärtsrichtung weiterleitet [(vgl. Kapitel 2.1)], und die Fehleranteile, die das Netz in Rückwärtsrichtung durchlaufen“[6] (vgl. Kapitel 2.2.1). Gegebenenfalls wird zusätzlich ein Verzerrungswert b (*engl.* Bias) addiert, der es dem Neuron ermöglicht, die Entscheidungsgrenze unabhängig von den Eingabewerten zu verschieben[7]:

$$net_j = \sum_{k=1}^{n} x_k \cdot w_k + b = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_n \cdot w_n + b$$

Diese Entscheidungsgrenze stellt den Schwellenwert zwischen der Aktivierung (*Ausgabe*) und Nichtaktivierung (*keine Ausgabe*) eines Neurons dar[8], und hängt von der gewählten Aktivierungsfunktion $fac(net_j) = a_j$ ab[9] (vgl. Kapitel 1.2). Sie nimmt die Ausgabe der Propagierungsfunktion als Eingabe entgegen und entscheidet anhand dieser, ob und in welchem Ausmaß das Neuron aktiviert wird[10]. Schließlich wird die Ausgabe der Aktivierungsfunktion an die Ausgabefunktion $fou(a_j) = y$ übergeben, welche festlegt, in welcher Form das künstliche Neuron seinen berechneten Aktivierungswert nach außen weitergibt[11]. Die Ausgabefunktion wird meist mit $fou(a_j) = a_j$ beschrieben[12].

---

[4] Lämmel, Uwe / Cleve, Jürgen: Künstliche Intelligenz, 3. Auflage, München: Carl Hanser Verlag, 2008 S. 197.
[5] Vgl. Lämmel / Cleve, 2008, S. 197.
[6] Rashid, Tariq: Neuronale Netze selbst programmieren, Heidelberg: O'Reilly, 2024, S. 225.
[7] Vgl. Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].
[8] Vgl. Steinwendner, Joachim / Schwaiger, Roland: Neuronale Netze programmieren mit Python, 2. Auflage, Bonn: Rheinwerk Verlag, 2020, S. 29.
[9] Vgl. Lämmel / Cleve, 2008, S. 197.
[10] Vgl. Lämmel / Cleve, 2008, S. 197.
[11] Vgl. Lämmel / Cleve, 2008, S. 197.
[12] Vgl. Lämmel / Cleve, 2008, S. 197.

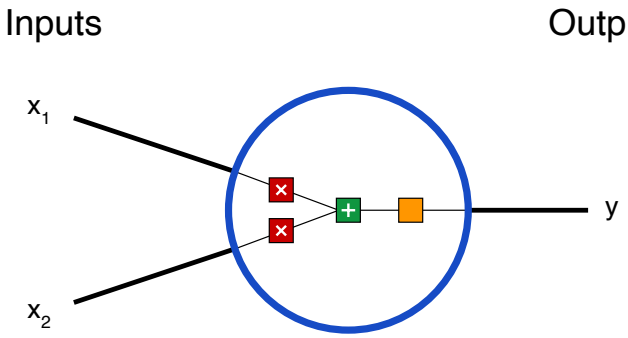

Abbildung 1: Das Neuron.
Quelle: Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].

1.2 Aktivierungsfunktionen

Nicht alle mathematischen Funktionen eignen sich gleichermaßen für den Einsatz als Aktivierungsfunktion[13]. Einige der in frühen Netzarchitekturen verwendeten Funktionen werden heute vor allem zu Lehrzwecken eingesetzt, da sie grundlegende Prinzipien veranschaulichen, jedoch nur eingeschränkt für komplexe Modelle geeignet sind[14]. Im Folgenden werden zwei zentrale Aktivierungsfunktionen vorgestellt und hinsichtlich ihrer mathematischen Eigenschaften sowie ihrer praktischen Eignung für künstlichen Neuronen bewertet[15].

1.2.1 Heaviside-Funktion

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$ [16]

Die Heaviside-Funktion, auch Stufenfunktion genannt, gilt als erste Aktivierungsfunktion in der Geschichte des künstlichen Neurons und bildet eine Eingabe auf zwei mögliche Ausgabewerte ab[17]. Die Ausgabe ist 0, wenn $x \in \mathbb{R}^-$ ist und 1, wenn $x \in \mathbb{R}_0^+$ ist. Aufgrund der Sprungstelle bei $x = 0$ ist die Heaviside-Funktion dort nicht differenzierbar. Da moderne neuronale Netze häufig die sogenannte Backpropagation als Trainingsmethode verwenden (vgl. Kapitel 2.2.1), die die Berechnung von Ableitungen und damit die Differenzierbarkeit einer Funktion voraussetzt, wird die Stufenfunktion mittlerweile selten als Aktivierungsfunktion eingesetzt[18].

1.2.2 Sigmoid-Funktion

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$ [19]

[13] Vgl. Trask, Andrew W.: Neuronale Netze und Deep Learning kapieren, Frechen: mitp Verlags GmbH & Co. KG, 2020, S. 232.
[14] Vgl. Rashid, 2024, S. 374f.
[15] Vgl. Lämmel / Cleve, 2023, S. 202.
[16] Vgl. Steinwendner / Schwaiger, 2020, S. 82.
[17] Vgl. Steinwendner / Schwaiger, 2020, S. 84.
[18] Vgl. Rojas, Rául: Neural Networks, Berlin: Springer, 1996, S. 149.
[19] Vgl. Rojas, 1996, S. 150.

Die Sigmoid-Funktion bildet jede beliebige reelle Zahl $x \in \mathbb{R}$ auf einen Wert im Bereich $\mathbb{W}_\sigma =]0\,;1[$ ab. Damit eignet sie sich besonders für Neuronen, deren Ausgabewerte als Wahrscheinlichkeiten interpretiert werden können[20]. Da die Funktion auf ihrem gesamten Definitionsbereich $\mathbb{D}_\sigma = \mathbb{R}$ differenzierbar ist, kann sie als Aktivierungsfunktion für neuronale Netze verwendet werden, die das Trainingsverfahren der Backpropagation nutzen[21].

1.3 Das XOR-Problem

Die XOR-Operation (eXclusive Or; *dt.* exklusives Oder; *Symbol* $\oplus$), auch Antivalenzfunktion genannt, ist eine logische Verknüpfung, die zwei boolesche Wahrheitswerte (*wahr* / *falsch*) verarbeitet und einen booleschen Ausgabewert liefert[22]. Dieser ist *wahr*, wenn sich die Eingabewerte unterscheiden, und *falsch*, wenn sie gleich sind[23]. Die Operation führt zu den in *Tabelle 1, Spalte 1-3* dargestellten Ergebnissen und wird mathematisch wie folgt beschrieben:

$$A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$$ [24].

Visualisiert man die möglichen Eingabeparameterkombinationen des XOR-Operators in einem Koordinatensystem, indem man Eingabeparameter A als x-Wert und Eingabeparameter B als y-Wert betrachtet, so ergibt sich ein Quadrat[25]. Die gegenüberliegenden Ecken dieses Quadrats weisen jeweils den gleichen Ausgabewert der Antivalenzfunktion auf (vgl. Abbildung 2).

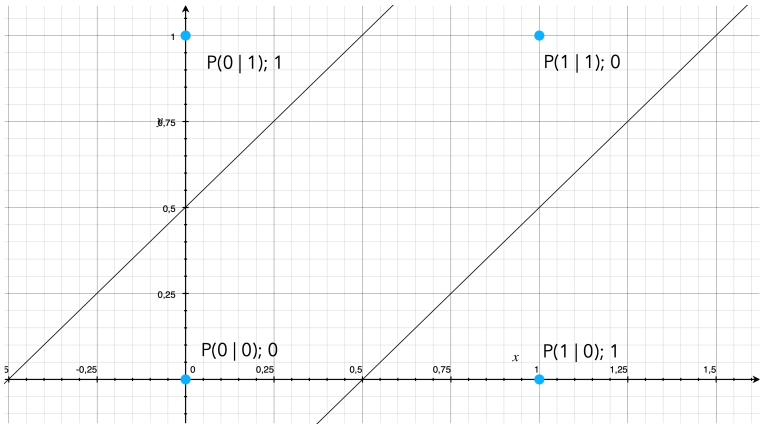

Abbildung 2: Darstellung der XOR-Eingabepaare im zweidimensionalen Raum mit zwei Entscheidungsgrenzen.
Quelle: Eigene Darstellung.

Diese Anordnung im zweidimensionalen Raum hat zur Folge, dass es nicht möglich ist, die verschiedenen Ausgabewerte mithilfe des Graphen einer einzelnen linearen Funktion, also der Entscheidungsgrenze eines Neurons, räumlich zu trennen[26]. Stattdessen sind mindestens zwei Entscheidungsgrenzen erforderlich, die zusammen eine nicht-lineare Trennung der beiden Klassen

[20] Vgl. Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].
[21] Vgl. Rojas, 1996, S. 149f.
[22] Vgl. Hoffmann, Dirk W.: Grundlagen der Technischen Informatik, München: Carl Hanser Verlag, 2016, S. 99.
[23] Vgl. Rashid, 2024, S. 65.
[24] Vgl. Hoffmann, 2016, S. 100.
[25] Vgl. Rashid, 2024, S. 62.
[26] Vgl. Rashid, 2024, S. 65f.

wahr (1) und *falsch* (0) ermöglichen[27] (vgl. Abbildung 2). Da die Propagierungsfunktion eines Neurons die Eingaben gewichtet und aufsummiert, ist dieses hingegen ausschließlich in der Lage, Muster linear zu separieren, das heißt, Entscheidungsgrenzen in Form einer einzigen Geraden zu erzeugen[28]. Die Ausgabe des Neurons stimmt folglich nicht mit dem gewünschten XOR-Verhalten überein (vgl. Tabelle 1).

Eingabe A	**Eingabe** B	**XOR** $A \oplus B$	$net_j = A + B - 1$ **mit** $w_A, w_B = 1,\ b = -1$	$H(x) = \begin{cases} 0, & net_j < 0 \\ 1, & net_j \geq 0 \end{cases}$
0	0	0	$0 + 0 - 1 = -1$	0
0	1	1	$0 + 1 - 1 = 0$	1
1	0	1	$1 + 0 - 1 = 0$	1
1	1	0	$1 + 1 - 1 = 1$	1

Tabelle 1: Die Ausgabe des XOR-Operators verglichen mit der Ausgabe eines Neurons.
Quelle: Eigene Darstellung.

Die Lösung des XOR-Problems erweist sich somit für ein einzelnes Neuron als nicht realisierbar und kann erst durch die Verknüpfung mehrerer Neuronen zu einem mehrschichtigen neuronalen Netz ermöglicht werden[29]. Dieses verfügt aufgrund seines Aufbaus über die Fähigkeit, mehrere Entscheidungsgrenzen zu erzeugen, wodurch auch nichtlinear trennbare Muster wie XOR abgebildet werden können[30].

2 Das vorwärtsgerichtete neuronale Netz

2.1 Architektur

Indem eine Vielzahl einzelner Neuronen, auch *Knoten* genannt, zu einem zusammenhängenden neuronalen Netz verknüpft wird, können diese mithilfe von Trainingsdaten auf ein bestimmtes Verhalten trainiert werden[31] [32]. „[A]nschließend [ist ein solches Netz] in der Lage, sowohl auf die Trainingssituationen als auch darüber hinaus auf neue Situationen angemessen zu reagieren“[33]. Je nach Verknüpfung der Neuronen ergeben sich unterschiedliche Netzarchitekturen, die für verschiedene Anwendungssituationen unterschiedlich gut geeignet sind[34] [35]. Als grundlegende und einfachste Netzarchitektur gilt das vorwärtsgerichtete neuronale Netz (*engl*. Feedforward Neural Network; *abgekürzt* FNN), das den Ausgangspunkt für die Entwicklung komplexerer Architekturen bildet[36].

[27] Vgl. Rashid, 2024, S. 67f.
[28] Vgl. Steinwendner / Schwaiger, 2020, S. 145.
[29] Vgl. Rashid, 2024, S. 67.
[30] Vgl. Rashid, 2024, S. 67.
[31] Vgl. Rashid, 2024, S. 81.
[32] Vgl. Lämmel / Cleve, 2008, S. 196.
[33] Lämmel / Cleve, 2008, S. 196.
[34] Vgl. Zhou, Victor: An Introduction to Recurrent Neural Networks for Beginners, in: Victor Zhou, 24.07.2019, [online] https://victorzhou.com/blog/intro-to-rnns [18.07.2025].
[35] Vgl. Zhou, Victor: CNNs, Part 1: An Introduction to Convolutional Neural Networks, in: Victor Zhou, 22.05.2019, [online] https://victorzhou.com/blog/intro-to-cnns-part-1 [18.07.2025].
[36] Vgl. Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].

Das FNN zeichnet sich vor allem durch den schichtweisen Aufbau aus Eingabeschicht, versteckter Schicht und Ausgabeschicht (vgl. Abbildung 3) und den vorwärtsgerichteten Datenfluss aus[37], bei dem Neuronen „nur in Richtung Ausgabeschicht miteinander verknüpft sind“[38].

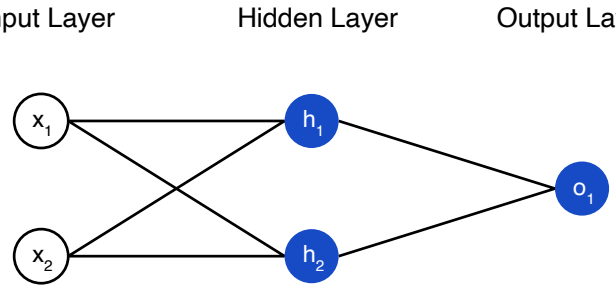

Abbildung 3: Das vorwärtsgerichtete neuronale Netz.
Quelle: Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].

2.1.1 Schichtenaufbau

In der Eingabeschicht (*engl.* Input Layer) werden die Eingabedaten in Form von numerischen Werten an das neuronale Netz übergeben[39] [40]. Jeder Knoten dieser Schicht nimmt dabei ein Merkmal der Eingabe entgegen, ohne selbst Berechnungen vorzunehmen[41]. Bei einem neuronalen Netz zur Klassifizierung von Bildern dienen Rastergrafiken fester Größe (z.B. 28 × 28 Pixel) als Eingabe, wobei jedes Neuron des Input Layers die Helligkeit eines einzelnen Pixels repräsentiert[42]. Alle Knoten der Eingabeschicht übergeben die vorbereiteten Werte anschließend an alle Neuronen der versteckten Schicht (*engl.* Hidden Layer), wo die eigentliche Verarbeitung beginnt[43]. Um ein symmetrisches Verhalten innerhalb des Hidden Layers zu vermeiden, werden die Gewichte der Neuronen vor dem Training des Netzes zufällig aus einer Wahrscheinlichkeitsverteilung, etwa der Normalverteilung mit definiertem Mittelwert und Standardabweichung, initialisiert[44]. Die optimale Anzahl an Neuronen in der versteckten Schicht ist mathematisch nicht exakt bestimmbar und muss stattdessen empirisch ermittelt werden[45]. Zu wenige Neuronen führen zu sogenanntem *Underfitting*, wodurch das neuronale Netz nicht alle relevanten Muster erlernen kann[46] [47]. Zu viele Neuronen hingegen verursachen *Overfitting*, bei dem das Netz zu genau an die Trainingsdaten angepasst ist und sich schlecht auf neue Eingabedaten verallgemeinern lässt[48] [49]. Schließlich werden die Ausgabedaten der versteckten Schicht an die Ausgabeschicht (*engl.* Output Layer) weitergege-

[37] Vgl. Lämmel / Cleve, 2023, S. 211.
[38] Lämmel / Cleve, 2023, S. 211.
[39] Vgl. Lämmel / Cleve, 2023, S. 205.
[40] Vgl. Trask, 2020, S. 51f.
[41] Vgl. Rashid, 2024, S. 89.
[42] Vgl. Rashid, 2024, S. 249.
[43] Vgl. Lämmel / Cleve, 2023, S. 205.
[44] Vgl. Rashid, 2024, S. 171f.
[45] Vgl. Lämmel / Cleve, 2023, S. 249f.
[46] Simon, Gyorgy / Aliferis, Constantin: Artificial Intelligence and Machine Learning in Health Care and Medical Sciences, Cham: Springer, 2024, S. 458.
[47] Vgl. Rashid, 2024, S. 276.
[48] Vgl. Rashid, 2024, S. 276.
[49] Vgl. Lämmel / Cleve, 2023, S. 249.

ben[50]. Die Anzahl der Neuronen in dieser Schicht hängt von der jeweiligen Anwendung ab[51]. Bei Klassifikationsaufgaben entspricht sie der Anzahl der Klassen, in die das neuronale Netz nach dem Training unterscheiden soll[52]. Die Ausgabe jedes Neurons repräsentiert dabei die geschätzte Wahrscheinlichkeit, dass die Eingabe zur jeweiligen Klasse gehört[53].

2.1.2 Initialisierung und Vorhersage

Im Folgenden werden die Initialisierung und Vorhersage des neuronalen Netzes im Hinblick auf eine praktische Umsetzung erläutert. Eine entsprechende Implementierung mithilfe der Programmiersprache Python, basierend auf den in den Kapiteln 3.1.2 und 3.2.2 beschriebenen Vorgehensweisen, ist in Teil A des Anhangs dargestellt. Das Verständnis der nachfolgenden Schritte und des anschließenden Trainingsverfahrens setzt grundlegende Kenntnisse der Matrizenrechnung voraus. Eine Einführung in dieses Thema findet sich in Teil B des Anhangs.

Um das Netz möglichst wiederverwendbar zu gestalten, erhält die Klasse `NeuralNetwork` (vgl. Anhang A.1) bei der Initialisierung die Anzahl an Knoten in Input, Hidden und Output Layer[54]. Auf dieser Grundlage erzeugt sie anschließend Matrizen, welche die erforderlichen Gewichte und Biaswerte zwischen den einzelnen Schichten speichern[55]. Jede Zeile einer Gewichtsmatrix W repräsentiert ein Neuron der entsprechenden Schicht, während jede Spalte einem Eingabewert entspricht (vgl. Tabelle 2)[56]. Das Matrixelement w_{mn} steht somit für das Gewicht zwischen der m-ten Eingabe und dem n-ten Neuron.

	Eingabe 1	Eingabe 2	Eingabe m
Neuron 1	w_{11}	w_{12}	w_{1m}
Neuron 2	w_{21}	w_{22}	w_{2m}
Neuron n	w_{n1}	w_{n2}	w_{nm}

Tabelle 2: Aufbau der Gewichtsmatrix einer Schicht des neuronalen Netzes.
Quelle: Eigene Darstellung.

Analog lässt sich auch eine Biasmatrix b darstellen. Da der Bias für alle Eingaben eines Neurons identisch ist, wird die Biasmatrix in Form eines Vektors angegeben. Die initialen Werte der Gewichtsmatrizen und Biasvektoren stammen aus einer Normalverteilung mit dem Mittelwert 0 und einer Standardabweichung von $n^{-0,5}$, wobei n der Anzahl der Neuronen in der jeweiligen Schicht entspricht[57].

[50] Vgl. Lämmel / Cleve, 2023, S. 205.
[51] Vgl. Rashid, 2024, S. 265.
[52] Vgl. Rashid, 2024, S. 276.
[53] Vgl. Lämmel / Cleve, 2023, S. 346f.
[54] Vgl. Rashid, 2024, S. 221f.
[55] Vgl. Rashid, 2024, S. 225.
[56] Vgl. Rashid, 2024, S. 108.
[57] Vgl. Rashid, 2024, S. 171-174.

Mithilfe der Methode `query` (vgl. Anhang A.2) kann das neuronale Netz nun eine Vorhersage treffen[58]. Dazu wird die an die Funktion übergebene Liste aller Eingaben zunächst in einen Spaltenvektor x transformiert, um die anschließende Matrizenmultiplikation mit der Gewichtsmatrix der versteckten Schicht zu ermöglichen[59]. Danach wird die gewählte Aktivierungsfunktion, eine Sigmoid-Funktion, auf alle Elemente der entstandenen Matrix h angewendet[60]. Im nächsten Schritt wird das Verfahren mit der Ausgabe der versteckten Schicht und der Gewichtsmatrix der Ausgabeschicht wiederholt[61]. Als Ergebnis gibt die Methode schließlich einen Ausgabevektor o_{out} zurück, der für jedes Neuron der Ausgabeschicht ein Element enthält. Die Ausgabe kann nun der Anwendungssituation entsprechend interpretiert und weiterverwendet werden.

2.2 Trainingsprozess

2.2.1 Ziel des Trainings

Das neuronale Netz kann zum jetzigen Zeitpunkt bereits auf Grundlage einer Eingabe eine Vorhersage erzeugen, doch sind diese Ergebnisse noch nicht aussagekräftig, da das Netz bislang nicht an eine konkrete Anwendungssituation angepasst wurde[62]. Erst durch ein gezieltes Training mit zuvor generierten Trainingsdaten und unter Verwendung der sogenannten Backpropagation (Backpropagation of Error; *dt.* Fehlerrückführung) wird das Netz in die Lage versetzt, brauchbare und zuverlässige Vorhersagen zu liefern[63] [64] [65].
Die Backpropagation, umgesetzt in der Methode `train` (vgl. Anhang A.3), ermöglicht es, die Vorhersagefehler eines neuronalen Netzes auf seine Gewichtungen und Biaswerte zurückzuführen und diese schrittweise anzupassen[66].

2.2.2 Backpropagation

Der erste Schritt der Backpropagation ist die Berechnung einer Vorhersage für einen Trainingsdatensatz und entspricht der in Kapitel 3.1.3 definierten `query` Methode[67]. Im Anschluss wird die vom Netzwerk erzeugte Ausgabe o_{out} mit dem Erwartungswert t verglichen, der die ideale Ausgabe des neuronalen Netzes darstellt[68]. Die Differenz zwischen der tatsächlichen Ausgabe und dem Erwartungswert definiert den Fehler e_{output} der Ausgabeschicht[69]:

$$e_{output} = t - o_{out}$$ [70]

[58] Vgl. Rashid, 2024, S. 229.
[59] Vgl. Rashid, 2024, S. 230.
[60] Vgl. Rashid, 2024, S. 231.
[61] Vgl. Rashid, 2024, S. 233.
[62] Vgl. Rashid, 2024, S. 238.
[63] Vgl. Lämmel / Cleve, 2023, S. 220.
[64] Vgl. Rashid, 2024, S. 30.
[65] Vgl. Rashid, 2024, S. 240.
[66] Vgl. Lämmel / Cleve, 2023, S. 221.
[67] Vgl. Rashid, 2024, S. 239.
[68] Vgl. Lämmel / Cleve, 2023, S. 221.
[69] Vgl. Rashid, 2024, S. 241.
[70] Vgl. Rashid, 2024, S. 241.

Dieser Fehler wird anschließend benötigt, um die Gewichtungen und Biaswerte zwischen versteckter und Ausgabeschicht anzupassen[71]. Darüber hinaus muss bestimmt werden, in welchem Maße die Neuronen der versteckten Schicht zum Gesamtfehler des Netzwerks beigetragen haben, um anschließend auch die Gewichtungen zwischen Eingabe- und versteckter Schicht gezielt anpassen zu können[72]. Dazu werden die Spalten und Zeilen der Gewichtsmatrix $W_{hidden-output}$ zwischen versteckter und Ausgabeschicht vertauscht, wodurch eine neue Matrix $W^{\top}_{hidden-output}$ entsteht[73]. Diese Operation wird als *Transponierung* bezeichnet[74]. Sie ist notwendig, da der Fehler das neuronale Netz nicht vorwärtsgerichtet von der versteckten zur Ausgabeschicht durchläuft, sondern rückwärtsgerichtet von der Ausgabeschicht zur versteckten Schicht weitergegeben wird[75].
Anschließend kann für jedes Neuron des Hidden Layers berechnet werden, in welchem Maße es zum Fehler der Ausgabeschicht beigetragen hat[76]:

$$e_{hidden} = W^{\top}_{hidden-output} \cdot e_{output}$$ [77]

Zusätzlich wird der mittlere quadratische Fehler (*engl.* Mean Squared Error; *abgekürzt MSE*) berechnet[78]. Er ist für das Training selbst nicht weiter erforderlich, dient in Kapitel 4 jedoch zur Beurteilung der Trainingsergebnisse[79]:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (t_i - o_{out,\,i})^2$$ [80]

Im zweiten Schritt des Trainings werden die Gewichte und Biaswerte zwischen den einzelnen Schichten auf Grundlage der zuvor berechneten Fehler angepasst[81]. Dazu wird zunächst die Ableitung der Aktivierungsfunktion benötigt[82]. Sie beschreibt, wie stark sich die Ausgabe der Aktivierungsfunktion verändert, wenn sich die Eingabe – und damit auch die zugehörigen Gewichte (vgl. Kapitel 1.1) – minimal ändert. Handelt es sich bei der Aktivierungsfunktion wie in dieser Implementierung um eine Sigmoid-Funktion, so ergibt sich folgende Ableitung:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Multipliziert man nun den Fehler e_{output} der Ausgabeschicht mit der Ableitung der Aktivierungsfunktion, so erhält man den Fehlerterm δ_{output}[83]. Er gibt die Änderungsrate an, die nötig ist, um den Fehler des Neurons zu verringern[84]:

---

[71] Vgl. Lämmel / Cleve, 2023, S. 221.
[72] Vgl. Lämmel / Cleve, 2023, S. 223.
[73] Vgl. Rashid, 2024, S. 135f.
[74] Vgl. Rashid, 2024, S. 135.
[75] Vgl. Lämmel / Cleve, 2023, S. 223.
[76] Vgl. Lämmel / Cleve, 2023, S. 223.
[77] Vgl. Rashid, 2024, S. 136.
[78] Vgl. Lämmel / Cleve, 2023, S. 232.
[79] Vgl. Lämmel / Cleve, 2023, S. 235.
[80] Vgl. Lämmel / Cleve, 2023, S. 223.
[81] Vgl. Lämmel / Cleve, 2023, S. 221.
[82] Vgl. Lämmel / Cleve, 2023, S. 223.
[83] Vgl. Lämmel / Cleve, 2023, S. 223.
[84] Vgl. Lämmel / Cleve, 2023, S. 223.

$$\delta_{output}(h_n) = e_{output} \cdot \sigma'(h_n)$$ [85]

Mithilfe des Fehlerterms ist es nun möglich, die Gewichtsmatrix $W_{hidden-output}$ und den Biasvektor b_{output} der Ausgabeschicht zu aktualisieren[86]. Der Fehlerterm wird hierfür auf den Vektor h angewendet, der aus den Eingaben aller Neuronen der Ausgabeschicht besteht (vgl. Kapitel 2.1.2). Der entstandene Vektor δ_{output} wird anschließend mit der Lernrate η und der Transponierung $h^\top$ von h multipliziert[87].

Die Lernrate η bestimmt die Schrittweite, mit der die Gewichte eines neuronalen Netzes während des Trainings angepasst werden[88]. Eine hohe Lernrate führt zu schnellen, aber ungenauen Anpassungen, während eine zu niedrige Lernrate das Lernen verlangsamt[89]. Da ihr optimaler Wert von den Trainingsdaten abhängig ist, muss die Lernrate in der Regel empirisch bestimmt werden[90]. Schließlich wird die resultierende Matrix zu $W_{hidden-output}$ addiert, sodass jedes Gewicht w_{mn} proportional zum Fehler des Ausgabeneurons, der Ausgabe des Neurons der versteckten Schicht und der Lernrate aktualisiert wird[91]:

$$W_{hidden-output} = W_{hidden-output} + \eta \cdot \delta_{output} \cdot h^\top$$ [92]

Der Biasvektor b_{output} der Ausgabeschicht wird auf ähnliche Weise angepasst[93]. Hierbei wird die Lernrate η mit dem Fehlervektor δ_{output} multipliziert und zu b_{output} addiert[94]:

$$b_{output} = b_{output} + \eta \cdot \delta_{output}$$ [95]

Analog zum vorherigen Schritt werden im letzten Schritt des Trainings die Gewichtsmatrix $W_{input-hidden}$ und der Biasvektor b_{hidden} der versteckten Schicht angepasst[96]. Zunächst wird der Fehlerterm der versteckten Schicht δ_{hidden} mithilfe des zu Beginn des Trainings berechneten Fehlers e_{hidden} und der Ableitung der Aktivierungsfunktion bestimmt[97]:

$$\delta_{hidden}(x_n) = e_{hidden} \cdot \sigma'(x_n)$$ [98]

---

[85] Vgl. Lämmel / Cleve, 2023, S. 223.
[86] Vgl. Lämmel / Cleve, 2023, S. 226.
[87] Vgl. Rashid, 2024, S. 242.
[88] Vgl. Lämmel / Cleve, 2023, S. 221.
[89] Vgl. Rashid, 2024, S. 288.
[90] Vgl. Rashid, 2024, S. 288.
[91] Vgl. Rashid, 2024, S. 242f.
[92] Vgl. Rashid, 2024, S. 242f.
[93] Vgl. Lämmel / Cleve, 2023, S. 226.
[94] Vgl. Lämmel / Cleve, 2023, S. 226.
[95] Vgl. Lämmel / Cleve, 2023, S. 226.
[96] Vgl. Rashid, 2024, S. 243.
[97] Vgl. Lämmel / Cleve, 2023, S. 223.
[98] Vgl. Lämmel / Cleve, 2023, S. 223.

Anschließend werden $W_{input-hidden}$ und b_{hidden} anhand des Fehlerterms und der Lernrate aktualisiert[99]:

$$W_{input-hidden} = W_{input-hidden} + \eta \cdot \delta_{hidden} \cdot x^{\top} \text{ [100]}$$

$$b_{hidden} = b_{hidden} + \eta \cdot \delta_{hidden} \text{ [101]}$$

2.2.3 Iteratives Lernen

Indem die Backpropagation mehrfach für alle Trainingsdaten durchgeführt wird, passen sich die Gewichte und Biaswerte des neuronalen Netzes schrittweise an die jeweilige Anwendungssituation an[102]. Der iterative Lernprozess läuft so lange, bis die Fehlerfunktion einen ausreichend kleinen Wert erreicht oder eine vorher festgelegte Anzahl an Durchläufen abgeschlossen ist[103]. Nach ausreichend Training ist das neuronale Netz anschließend auch in der Lage, für unbekannte Eingaben korrekte Vorhersagen zu treffen[104].

3 Formenklassifizierung als Anwendungsbeispiel

3.1 Zielsetzung und Vorgehensweise

Das vorwärtsgerichtete neuronale Netz soll nun auf das Anwendungsbeispiel der Erkennung von handgezeichneten geometrischen Formen angepasst und anschließend in die Notizensoftware *Productivity Pro* eingesetzt werden. Dabei soll das neuronale Netz in die Lage versetzt werden, zwischen den Klassen *Kreis*, *Dreieck* und *Rechteck* zu unterscheiden. Ziel ist es, ein fertig trainiertes Modell zu erhalten, das die gelernten Gewichtungen und Biaswerte speichert und möglichst präzise Vorhersagen trifft. Um ein solches Modell zu entwickeln, werden die Anzahl der Neuronen in der versteckten Schicht und die Lernrate durch empirische Versuche angepasst. Bei den verwendeten Trainingsdaten handelt es sich um graustufige Rastergrafiken der Größe 28 × 28 Pixel. Jeder Pixel besitzt dabei eine Helligkeit zwischen 0 und 255 (vgl. Abbildung 4), die auf den Bereich zwischen $0{,}1$ und $0{,}99$ normalisiert wird. Diese Normalisierung erfolgt aufgrund der gewählten Sigmoid-Aktivierungsfunktion, deren Wertebereich zwischen 0 und 1 liegt (vgl. Kapitel 1.2.2). Das neuronale Netz wird mit 90 Bildern pro Klasse trainiert, was bei drei Klassen einer Gesamtzahl von 270 Trainingsbeispielen entspricht. Die Trainingsbilder stammen dabei von fünf verschiedenen Personen, die jeweils 3 × 18 Formen gezeichnet haben, um eine ausreichende Variation innerhalb der Klassen zu gewährleisten. Diese Variation ist notwendig, damit das FNN die charakteristischen Merkmale der verschiedenen Formen unabhängig von individuellen Zeichenstilen erlernt.

[99] Vgl. Lämmel / Cleve, 2023, S. 226.
[100] Vgl. Lämmel / Cleve, 2023, S. 226.
[101] Vgl. Lämmel / Cleve, 2023, S. 226.
[102] Vgl. Rashid, 2024, S. 167.
[103] Vgl. Rashid, 2024, S. 146.
[104] Vgl. Lämmel / Cleve, 2023, S. 209.

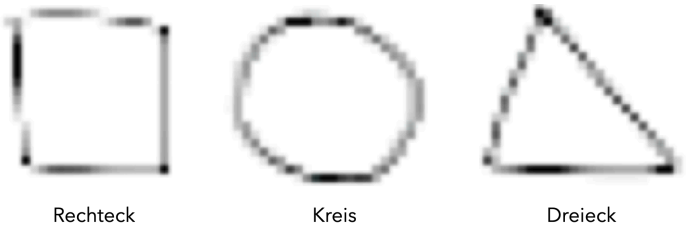

Abbildung 4: Beispielhafte Trainingsdaten zu jeder Klasse.
Quelle: Eigene Darstellung.

Die Genauigkeit des neuronalen Netzes wird im Folgenden auf zwei verschiedene Arten ermittelt. Zum einen erfolgt die Bewertung über den in Kapitel 2.2.2 eingeführten mittleren quadratischen Fehler, der die Abweichung zwischen den vorhergesagten Werten und den tatsächlichen Zielwerten während des Trainings angibt. Zum anderen wird die Genauigkeit über die Erkennungsrate A bestimmt. Dabei werden 45 von den Trainingsdaten unabhängige Testdaten verwendet, wobei A als Verhältnis der korrekt klassifizierten zu allen getesteten Beispielen berechnet wird:

$$A = \frac{N_{korrekt}}{N_{gesamt}}$$

Für eine möglichst zuverlässige Auswertung des Trainings werden Verlust und Erkennungsrate für jedes neuronale Netz als Durchschnitt über 50 Trainingsläufe ohne Änderung der Parameter berechnet. So wird ausgeschlossen, dass die Genauigkeit des Netzes von der zufälligen Initialisierung der Gewichte abhängt.

3.2 Anpassung von Trainingsparametern

3.2.1 Neuronenanzahl der versteckten Schicht

Basierend auf der Größe der Eingabedaten und der Anzahl der zu unterscheidenden Klassen werden 784 Eingabeknoten und drei Ausgabeknoten gewählt. Zudem wird die Lernrate zunächst auf den Wert $\eta = 0{,}3$ gesetzt. Dieser orientiert sich an dem in Tariq Rashids Buch *„Neuronale Netze selbst programmieren“* verwendeten Anfangswert, ohne dass hierfür ein weiterer Hintergrund besteht[105]. Die Anzahl der Neuronen in der versteckten Schicht wird variiert, um ihren Einfluss auf die Trainingsleistung zu untersuchen. Dabei werden die Werte $3, 30, 60, 120, 240, 400$ und 784 getestet:

[105] Vgl. Rashid, 2024, S. 222.

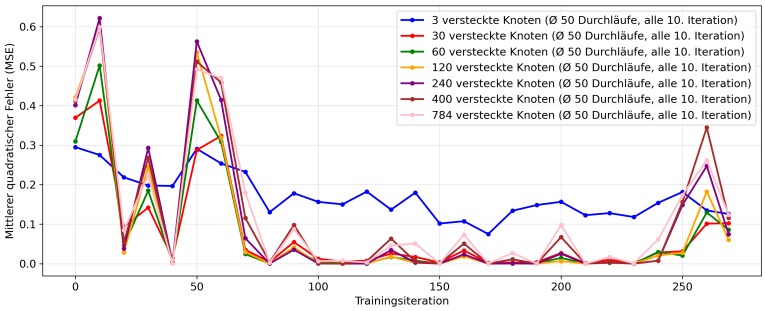

Abbildung 5: Entwicklung des Verlusts für 3, 30, 60, 120, 240, 400 und 784 Knoten in der versteckten Schicht.
Quelle: Eigene Darstellung.

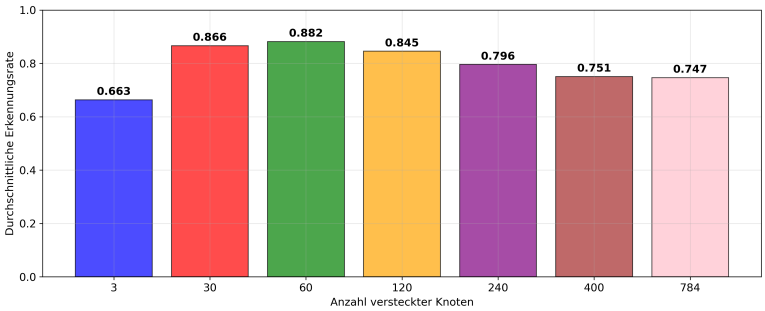

Abbildung 6: Durchschnittliche Erkennungsraten für 3, 30, 60, 120, 240, 400 und 784 Knoten in der versteckten Schicht.
Quelle: Eigene Darstellung.

Mit einem finalen Verlust von 0,125685 und einer Erkennungsrate von 0,663 erzielt das neuronale Netz mit drei Neuronen in der versteckten Schicht die schlechteste Leistung. Dies entspricht dem in Kapitel 2.1.1 beschriebenen Underfitting, bei dem die Anzahl der Neuronen zu gering ist, um Zusammenhänge in den Trainingsdaten ausreichend zu erfassen.

Betrachtet man den Verlust in Abbildung 5, zeigt das neuronale Netz mit 120 Neuronen die beste Leistung, während die Erkennungsrate in Abbildung 6 mit 0,882 für 60 Neuronen am höchsten ist. Dieser scheinbare Widerspruch ergibt sich aus den unterschiedlichen Aussagen der Bewertungsmethoden: Während der Verlust die Anpassung an die Trainingsdaten misst, spiegelt die Erkennungsrate die Leistungsfähigkeit des Modells bei neuen Daten wider und ist damit das aussagekräftigere Beurteilungskriterium. Auf Grundlage dieser Beobachtung wird die Anzahl der Neuronen in der versteckten Schicht daher auf 60 festgelegt.

Darüberhinaus sind die Sprünge des Verlusts in Abbildung 6 zu Beginn und gegen Ende des Trainings auffällig. Zu Beginn entstehen sie, „[d]a in den zufälligen Mustern [der Trainingsdaten zunächst] keine Regeln erkennbar sind“[106]. Später können die Sprünge aufgrund von Overfitting (vgl. Kapitel 2.1.1) auftreten[107].

[106] Lämmel / Cleve, 2023, S. 251.
[107] Vgl. Rashid, 2024, S. 293.

3.2.2 Lernrate

Das Vorgehen zur Ermittlung einer geeigneten Anzahl an Neuronen in der versteckten Schicht wird nun mit der Lernrate η wiederholt. Es werden die Werte $0{,}01$, $0{,}05$, $0{,}1$, $0{,}2$, $0{,}3$, $0{,}5$ und $1{,}0$ getestet:

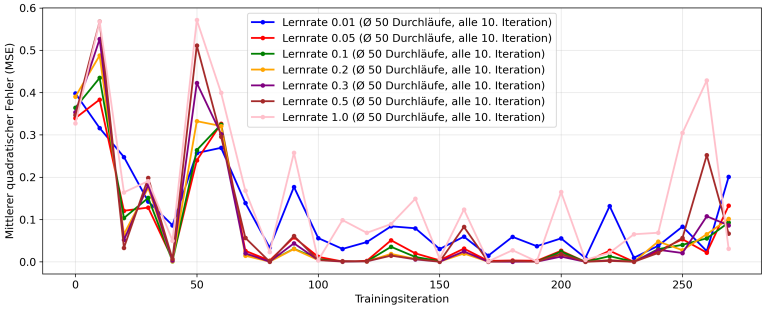

Abbildung 7: Entwicklung des Verlusts bei einer Lernrate von $0{,}01$, $0{,}05$, $0{,}1$, $0{,}2$, $0{,}3$, $0{,}5$ und $1{,}0$.
Quelle: Eigene Darstellung.

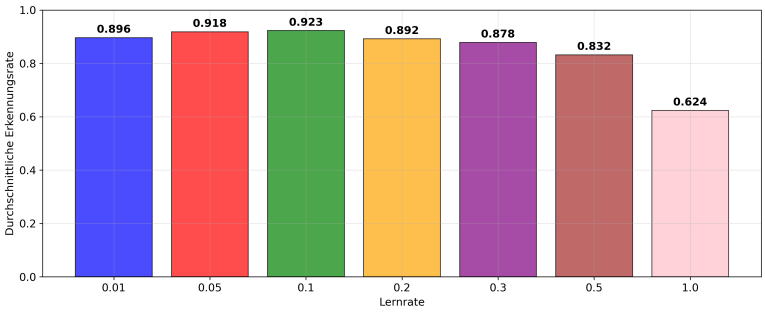

Abbildung 8: Durchschnittliche Erkennungsraten bei einer Lernrate von $0{,}01$, $0{,}05$, $0{,}1$, $0{,}2$, $0{,}3$, $0{,}5$ und $1{,}0$
Quelle: Eigene Darstellung.

Bei der Anpassung der Lernrate zeigt sich deutlich der Effekt des Overfittings: Eine Lernrate von 1 erzielt zwar den geringsten finalen Trainingsverlust (vgl. Abbildung 7), führt jedoch gleichzeitig auch zur niedrigsten Erkennungsrate auf den Testdaten (vgl. Abbildung 8), da das neuronale Netz zu stark an die Trainingsdaten angepasst ist. Eine geeignete Wahl der Lernrate für das Netz ist hingegen $0{,}1$. Zwar fällt der Trainingsverlust mit $0{,}097886$ hierbei höher aus, das Modell zeigt mit einer Erkennungsrate von $0{,}923$ jedoch die beste Generalisierbarkeit auf unbekannten Daten.

3.3 Auswertung

Die Untersuchungen zeigen, dass neuronale Netze durch die gezielte Anpassung von Trainingsparametern wie der Anzahl der Neuronen und der Lernrate deutlich optimiert werden können. So ließ sich die Erkennungsrate des Netzes von $0{,}663$ auf $0{,}923$ steigern, ohne die Menge oder Art der Trainingsdaten zu verändern. Aufbauend darauf können sowohl die Anzahl der Neuronen in der versteckten Schicht als auch die Lernrate weiter verfeinert werden, indem die Parameter in kleineren Schritten variiert werden.

III Bewertung neuronaler Netze und Ansatz zur Weiterentwicklung der Formenklassifikation

Abschließend lässt sich festhalten, dass der Einsatz künstlicher Intelligenz – und damit auch der Einsatz neuronaler Netze – insbesondere dort sinnvoll ist, wo klassische deterministische Algorithmen an ihre Grenzen stoßen und stattdessen Systeme erforderlich sind, die aus Daten lernen und sich an komplexe Muster anpassen können. Der Einsatz solcher Netze beschränkt sich dabei nicht nur auf groß angelegte Anwendungen wie große Sprachmodelle oder autonome Fahrzeuge, sondern erweist sich bereits in kleineren Projekten, wie zur Klassifikation von handgezeichneten geometrischen Formen, als äußerst hilfreich.

Zwar ist das in dieser Arbeit entwickelte neuronale Netz in der Lage, zuverlässige Ergebnisse zu liefern und eignet sich damit durchaus für den Einsatz in *Productivity Pro*, dennoch bestehen weiterhin Möglichkeiten zur Verbesserung. Anstelle einer reinen Anpassung der Trainingsparameter, wie sie in dieser Arbeit vorgenommen wurde, könnte eine Weiterentwicklung durch die Veränderung der gewählten Netzwerkarchitektur erzielt werden. Besonders geeignet wäre hierbei das sogenannte Convolutional Neural Network (*dt.* faltungsbasiertes neuronales Netzwerk)[108]. Es zeichnet sich vor allem durch die effiziente Analyse von Bilddaten aus, wie sie auch in *Productivity Pro* als Eingabe dienen[109].

Doch auch unabhängig von diesem konkreten Anwendungsbereich besteht in der Forschung zu künstlicher Intelligenz weiterhin ein großes Entwicklungspotenzial. Nahezu täglich werden derzeit neue Erkenntnisse veröffentlicht, insbesondere im Bereich der generativen KI. Es bleibt spannend zu beobachten, welche Fortschritte und Möglichkeiten sich in den kommenden Jahren daraus ergeben werden.

[108] Vgl. Zhou, Victor: CNNs, Part 1: An Introduction to Convolutional Neural Networks, in: Victor Zhou, 22.05.2019, [online] https://victorzhou.com/blog/intro-to-cnns-part-1 [30.10.2025].
[109] Vgl. Zhou, Victor: CNNs, Part 1: An Introduction to Convolutional Neural Networks, in: Victor Zhou, 22.05.2019, [online] https://victorzhou.com/blog/intro-to-cnns-part-1 [30.10.2025].

Literaturverzeichnis

Buchquellen

- Hoffmann, Dirk W.: Grundlagen der Technischen Informatik, München: Carl Hanser Verlag, 2016.
- Kirchgessner, Karsten / Schreck, Marco: Vektor- und Matrizenrechnung für Dummies, 1. Auflage, Weinheim: WILEY-VCH Verlag, 2013.
- Krauss, Patrick: Künstliche Intelligenz und Hirnforschung, Berlin: Springer, 2023.
- Lämmel, Uwe / Cleve, Jürgen: Künstliche Intelligenz, 3. Auflage, München: Carl Hanser Verlag, 2008.
- Lämmel, Uwe / Cleve, Jürgen: Künstliche Intelligenz, 6. Auflage, München: Carl Hanser Verlag, 2023.
- Rashid, Tariq: Neuronale Netze selbst programmieren, Heidelberg: O'Reilly, 2024.
- Rojas, Rául: Neural Networks, Berlin: Springer, 1996.
- Simon, Gyorgy / Aliferis, Constantin: Artificial Intelligence and Machine Learning in Health Care and Medical Sciences, Cham: Springer, 2024.
- Steinwendner, Joachim / Schwaiger, Roland: Neuronale Netze programmieren mit Python, 2. Auflage, Bonn: Rheinwerk Verlag, 2020.
- Trask, Andrew W.: Neuronale Netze und Deep Learning kapieren, Frechen: mitp Verlags GmbH & Co. KG, 2020.

Internetquellen

- Zhou, Victor: An Introduction to Recurrent Neural Networks for Beginners, in: Victor Zhou, 24.07.2019, [online] https://victorzhou.com/blog/intro-to-rnns [18.07.2025].
- Zhou, Victor: CNNs, Part 1: An Introduction to Convolutional Neural Networks, in: Victor Zhou, 22.05.2019, [online] https://victorzhou.com/blog/intro-to-cnns-part-1 [18.07.2025].
- Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.com/blog/intro-to-neural-networks [04.02.2025].

Darstellungsverzeichnis

Abbildungen

- Abbildung 1: Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.-com/blog/intro-to-neural-networks [04.02.2025].
- Abbildung 2: Eigene Darstellung.
- Abbildung 3: Zhou, Victor: Machine Learning for Beginners: An Introduction to Neural Networks, in: Victor Zhou, 03.03.2019, [online] https://victorzhou.-com/blog/intro-to-neural-networks [04.02.2025].
- Abbildung 4: Eigene Darstellung.
- Abbildung 5: Eigene Darstellung.
- Abbildung 6: Eigene Darstellung.
- Abbildung 7: Eigene Darstellung.
- Abbildung 8: Eigene Darstellung.

Tabellen

- Tabelle 1: Eigene Darstellung.
- Tabelle 2: Eigene Darstellung.

Anhang

A Implementierung eines neuronalen Netzes

```
class NeuralNetwork:
    def __init__(
        self, inputnodes, hiddennodes,
        outputnodes, learningrate
    ):
        # Anzahl der Neuronen in den einzelnen Schichten
        self.inputnodes = inputnodes
        self.hiddennodes = hiddennodes
        self.outputnodes = outputnodes

        # Lernrate: Schrittgröße bei der Gewichtsanpassung
        self.learningrate = learningrate

        # Gewichtsmatrizen: Normalverteilung mit μ = 0, σ = n^-0,5
        self.weights_input_hidden = numpy.random.normal(
            0.0, pow(self.hiddennodes, -0.5),
            (self.hiddennodes, self.inputnodes)
        )
        self.weights_hidden_output = numpy.random.normal(
            0.0, pow(self.outputnodes, -0.5),
            (self.outputnodes, self.hiddennodes)
        )

        # Biasmatrizen: Normalverteilung mit μ = 0, σ = n^-0,5
        self.bias_hidden = numpy.random.normal(
            0.0, pow(self.hiddennodes, -0.5),
            (self.hiddennodes, 1)
        )
        self.bias_output = numpy.random.normal(
            0.0, pow(self.outputnodes, -0.5),
            (self.outputnodes, 1)
        )

        # Sigmoid als Aktivierungsfunktion
        self.activation_function = lambda x: 1.0 / (
            1.0 + numpy.exp(-x))
```

Anhang A.1: Initialisierung eines vorwärtsgerichteten neuronalen Netzes.
Quelle: Eigene Darstellung.

```python
    def query(self, inputs_list):
        # Vorbereitung der Eingabedaten (Input Layer)
        inputs = numpy.array(inputs_list, ndmin=2).T

        # Eingabe und Ausgabe des Hidden Layers
        inputs_of_hidden = numpy.dot(
            self.weights_input_hidden, inputs
        ) + self.bias_hidden
        outputs_of_hidden = self.activation_function(
            inputs_of_hidden
        )

        # Eingabe und Ausgabe des Output Layers
        inputs_of_output = numpy.dot(
            self.weights_hidden_output, outputs_of_hidden
        ) + self.bias_output
        outputs_of_output = self.activation_function(
            inputs_of_output
        )

        # Ausgabe des Netzwerks als Matrix
        return outputs_of_output
```

Anhang A.2: Implementierung der Methode `query`.
Quelle: Eigene Darstellung.

```
def train(self, inputs_list, targets_list):
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # Eingabesignale in die versteckte Schicht (mit Bias)
    inputs_of_hidden = numpy.dot(
        self.weights_input_hidden, inputs
    ) + self.bias_hidden

    # Ausgabesignale der versteckten Schicht
    outputs_of_hidden = self.activation_function(
        inputs_of_hidden
    )

    # Eingabesignale in die Ausgabeschicht (mit Bias)
    inputs_of_output = numpy.dot(
        self.weights_hidden_output, outputs_of_hidden
    ) + self.bias_output
    # Ausgabesignale der Ausgabeschicht
    outputs_of_output = self.activation_function(
        inputs_of_output
    )

    # Verlustberechnung der Ausgabeschicht (target - actual)
    output_errors = targets - outputs_of_output

    # Verlustberechnung der versteckten Schicht
    hidden_errors = numpy.dot(
        self.weights_hidden_output.T, output_errors
    )

    # Gewichte zwischen versteckter und Ausgabeschicht updaten
    self.weights_hidden_output += self.learningrate * numpy.dot(
        output_errors * outputs_of_output * (
            1.0 - outputs_of_output
        ), numpy.transpose(outputs_of_hidden)
    )
    self.bias_output += self.learningrate * (
        output_errors * outputs_of_output * (
            1.0 - outputs_of_output
        )
    )

    # Gewichte zwischen Eingabe- und versteckter Schicht updaten
    self.weights_input_hidden += self.learningrate * numpy.dot(
        hidden_errors * outputs_of_hidden * (
            1.0 - outputs_of_hidden
        ), numpy.transpose(inputs)
    )
    self.bias_hidden += self.learningrate * (
        hidden_errors * outputs_of_hidden * (
            1.0 - outputs_of_hidden
        )
    )

    # Verlustgraph
    self.losses.append(((targets - outputs_of_output) ** 2)
       .mean())
```

Anhang A.3: Training eines Feedforward Neural Networks.
Quelle: Eigene Darstellung.

B.1 Struktur der Matrizen

Unter einer Matrix versteht man eine rechteckige Anordnungen von Zahlen, Symbolen oder Funktionen, die in Zeilen und Spalten eingetragen sind (vgl. Abbildung 4)[110]. Formal wird eine Matrix A mit m Zeilen und n Spalten als $A = \left[a_{ij}\right]$ mit $i = 1,\ldots,m$ und $j = 1,\ldots,n$ beschrieben, wobei a_{ij} das Element in der i-ten Zeile und j-ten Spalte bezeichnet[111]:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}_{m \times n}$$

Eine Sonderform der Matrix ist die $m \times 1$-Matrix, die meist als Vektor in der analytischen Geometrie zum Einsatz kommt[112]:

$$V = \begin{bmatrix} v_{11} \\ v_{21} \\ \vdots \\ v_{m1} \end{bmatrix}_{m \times 1}$$

B.2 Addition von Matrizen

Die Addition zweier Matrizen dient zur Kombination gleichartiger Einträge (z.B. zwei Gewichtsmatrizen)[113]. Dabei müssen die zu addierenden Matrizen $A = \left[a_{ij}\right]$ und $B = \left[b_{ij}\right]$ die gleiche Dimension $m \times n$ aufweisen; die Anzahl ihrer Zeilen und Spalten muss folglich jeweils übereinstimmen[114]. Ist diese Bedingung erfüllt, so ergibt sich die Summenmatrix $C = A + B = \left[c_{ij}\right]$.

Die Addition erfolgt dabei elementweise: Jedes Element der Matrix A in der i-ten Zeile und der j-ten Spalte wird mit dem entsprechenden Element der Matrix B addiert[115].

B.3 Multiplikation von Matrizen

Mithilfe der Matrizenmultiplikation können zwei Matrizen mit unterschiedlichen Bedeutungen (z.B. eine Gewichtsmatrix und eine Matrix mit Eingabewerten) strukturell miteinander verknüpft werden[116]. Dazu ist zunächst erforderlich, dass die Zeilenanzahl einer Matrix A mit der Spaltenanzahl einer Matrix B

[110] Kirchgessner, Karsten / Schreck, Marco: Vektor- und Matrizenrechnung für Dummies, 1. Auflage, Weinheim: WILEY-VCH Verlag, 2013, S. 41.
[111] Vgl. Kirchgessner / Schreck, 2013, S. 41f.
[112] Vgl. Kirchgessner / Schreck, 2013, S. 47.
[113] Vgl. Kirchgessner / Schreck, 2013, S. 44.
[114] Vgl. Kirchgessner / Schreck, 2013, S. 44.
[115] Vgl. Kirchgessner / Schreck, 2013, S. 44.
[116] Vgl. Kirchgessner / Schreck, 2013, S. 44f.

übereinstimmt[117]. Ist $A \in \mathbb{K}^{m \times n}$ und $B \in \mathbb{K}^{n \times p}$, so ergibt sich das Produkt $C = A \cdot B$ als Matrix der Dimension $m \times p$[118].

Die Elemente der resultierenden Matrix C werden durch die Summe der Produkte entsprechender Elemente der Zeilen von A und der Spalten von B gebildet[119]:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \quad 1 \leq i \leq m, 1 \leq j \leq p$$

[117] Vgl. Kirchgessner / Schreck, 2013, S. 44f.
[118] Vgl. Kirchgessner / Schreck, 2013, S. 44f.
[119] Vgl. Kirchgessner / Schreck, 2013, S. 44.