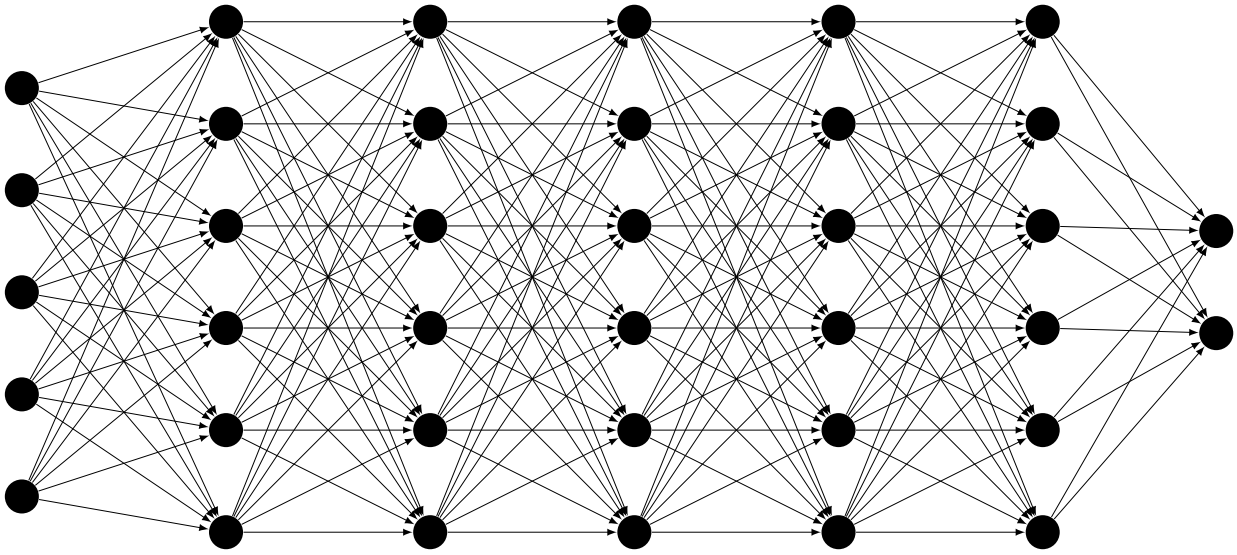
# Erstellung eines eigenen künstlichen neuronalen Netzwerks

Basil Rohner

31.10.2022



Kantonsschule Wohlen, G4E
Betreuung: Patric Rousselot
Co-Betreuung: Simon Reichmuth

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Seit ich denken kann, begeistere ich mich für Technik. Diese Faszination hat sich im Laufe der Zeit immer mehr in den Bereich der Informatik verlagert, was dazu geführt hat, dass ich seit einiger Zeit als Hobby programmiere. Auf die Idee meiner Maturaarbeit kam ich beim Lesen eines Blogs über generative Algorithmen [1]. Besonders fasziniert haben mich Anwendungen wie Deepfakes [2], sowie bildgenerierende Algorithmen wie Dall-E [3] und Midjourney [4]. Bei der Recherche zum Thema künstliche Intelligenz habe ich mich gefragt, wie künstliche Intelligenzen und künstliche neuronale Netzwerke eigentlich funktionieren. Aus dieser Frage heraus hat sich das Thema dieser Maturaarbeit ergeben. Neben der Kernfrage, was künstliche neuronale Netze sind und wie sie funktionieren, habe ich mir das Ziel gesetzt, herauszufinden, ob es möglich ist, ein eigenes künstliches neuronales Netzwerk von Grund auf zu programmieren und es dann an verschiedenen Datensätzen zu testen.

## 1.2 Zielsetzung

Im Rahmen dieser Maturaarbeit beschäftige ich mich mit dem Thema der künstlichen neuronalen Netzwerke. Ich habe mir das Ziel gesetzt, ein eigenes künstliches neuronales Netzwerk von Grund auf, in der Programmiersprache Python, zu programmieren. Zudem soll das künstliche neuronale Netzwerk mit mehreren Datensätzen unterschiedlicher Komplexität trainiert werden. Mithilfe der gewonnenen Daten und Informationen sollen anschliessend sinnvolle Optimierungen am Programm vorgenommen werden. Ein Vergleich meines künstlichen neuronalen Netzwerks mit einem modernen Machine-Learning-Modul schliesst den praktischen Teil der Arbeit ab. Im folgenden schriftlichen Teil dieser Arbeit werde ich die mathematischen Grundlagen und die Theorie eines künstlichen neuronalen Netzwerks behandeln und meinen Arbeitsprozess bei der Erstellung des Netzwerks dokumentieren. Des Weiteren werde ich im Kapitel *Resultate* die Erfolge und Grenzen meines Programms auflisten und interpretieren. Im letzten Teil werde ich mein Fazit zu dieser Arbeit ziehen.

[1] Vgl. Safar, Milad: Was ist generative KI und was kann sie? https://www.industry-of-things.de/was-ist-generative-ki-und-was-kann-sie-a-8faf44a80c7de6711d3b05875722c122/ (Abruf 27.10.2022).
[2] Vgl. Wikipedia: Deepfake. https://de.wikipedia.org/wiki/Deepfake (Abruf 27.10.2022).
[3] Vgl. OpenAI: Dall-E: Creating Images from Text. https://openai.com/blog/dall-e (Abruf 27.10.2022).
[4] Vgl. Wikipedia: Midjourney https://en.wikipedia.org/wiki/Midjourney (Abruf 27.10.2022).

# 2 Künstliche Neuronale Netzwerke

Künstliche neuronale Netze, wie sie in dieser Arbeit verwendet werden, sind eine besondere Ausprägung von Algorithmen des maschinellen Lernens, einem Zweig der Informatik, der sich mit den selbst lernenden Algorithmen beschäftigt. Der weitaus bekanntere Begriff der Künstlichen Intelligenz ist eng mit dem des künstlichen neuronalen Netzes verwandt. Laut Definition ist eine Künstliche Intelligenz ein System, das versucht, eine menschliche Intelligenz zu imitieren. Diese Definition deckt aber immer noch ein sehr breites Spektrum von Programmen ab. So können zum Beispiel Algorithmen, die ein menschliches Entscheidungsmuster simulieren, bereits dem Bereich der Künstlichen Intelligenz zugeordnet werden. Zoomt man nun in diesen riesigen Bereich der Künstlichen Intelligenz hinein, so findet man, neben zahlreichen anderen Unterbereichen, die Kategorie der maschinell lernenden Algorithmen. Maschinell lernende Algorithmen beschreiben Programme, die eine bestimmte Lernfähigkeit besitzen. Diese Lernfähigkeit beschränkt sich meist auf die Interpretation bestimmter Daten und die Erzeugung von Erkenntnissen. Künstliche neuronale Netze sind eine bestimmte Art von Algorithmen, die maschinelles Lernen betreiben. Mit künstlichen neuronalen Netzen werden Programme bezeichnet, die ihre Lernfähigkeit durch eine netzartige Struktur erlangen.[5]

## 2.1 Zweck von künstlichen neuronalen Netzwerken

Wie bereits erwähnt, sind künstliche neuronale Netze eine Art der Algorithmen des maschinellen Lernens und dienen dem Zweck, bestimmte Datensätze zu erlernen, um den ihnen zugrunde liegenden Algorithmus zu verstehen und selbständig Vorhersagen für weitere Daten des entsprechenden Datensatzes zu treffen. Grundsätzlich lassen sich künstliche neuronale Netze in zwei Kategorien einteilen. Zum einen gibt es Netze, die Datensätze sortieren und klassifizieren.[6] Andere Netze, die auf der Grundlage von Daten ein Spektrum von Werten vorhersagen, führen eine Regressionsanalyse durch.[7] Welche Methode der Vorhersage gewählt wird, hängt vom Datensatz ab. Es gibt jedoch viele Datensätze, die mit beiden Methoden gelöst werden können. Die beiden Methoden werden in den folgenden zwei Unterkapiteln erläutert.

### 2.1.1 Klassifikationsverfahren

Klassifikation ist eine Datenanalysemethode, welche, wie der Name preisgibt, Daten klassifiziert. Anhand von verschiedenen Eingaben bestimmt der Algorithmus, in welche Kategorie ein bestimmtes Objekt gehört. Ein Beispiel für ein Klassifikationsverfahren ist die Einteilung von Bildern von Haustieren. Ein Bild einer Katze wird dem Label *Katze* zugeordnet, während das Bild eines Hundes dem Label *Hund* zugeordnet wird. Mit dem Begriff Label bezeichnet man eine Kategorie, in welche Daten eingeteilt werden. Ein Klassifizierungsverfahren kommt dann ins Spiel, wenn klar ist, in welche Kategorien oder Label die Daten eingeordnet werden sollen. Wenn man jedoch nicht in der Lage ist, zu Beginn des Trainingsprozesses zu definieren, welche Klassen mögliche Ergebnisse sind, ist diese Methode nutzlos.[6]

[5] Vgl. Wikipedia: Künstliche Intelligenz. https://de.wikipedia.org/wiki/Künstliche_Intelligenz (Abruf 27.10.2022).
[6] Vgl. Wikipedia: Klassifikationsverfahren. https://de.wikipedia.org/wiki/Klassifikationsverfahren (Abruf 27.10.2022).
[7] Vgl. Wikipedia: Regressionsanalyse. https://de.wikipedia.org/wiki/Regressionsanalyse (Abruf 27.10.2022).

### 2.1.2 Regressionsanalyse

Ganz anders als das Klassifikationsverfahren arbeitet eine Regressionsanalyse. Diese nimmt keine Einteilung in Label oder Kategorien vor, sondern gibt eine Zahl aus einem fliessenden Spektrum an Zahlen aus. Das bedeutet, dass keine binäre Einteilung vorgenommen wird, sondern eine Vorhersage gemacht wird, welche sich nicht vordefinierten Kategorien anpassen muss. Beispiele für Datensätze, welche sinnvollerweise mit einer Regressionsanalyse verarbeitet werden, sind Aktienkurse oder Wettervorhersagen. Möchte man einen Aktienkurs vorhersagen, so ist es nicht sinnvoll, bestimmte Label im Vorhinein zu definieren. Das Netzwerk, welches auf den Aktienkurs trainiert ist, soll einen variierbaren Wert ausgeben, welcher als Kursvorhersage genutzt werden kann.[7]

# 3 Aufbau eines Künstlichen Neuronalen Netzwerks

Der Aufbau von künstlichen neuronalen Netzen ist stark von natürlichen neuronalen Netzen inspiriert, dessen bekanntestes Beispiel ist wohl das menschliche Gehirn. Genau wie dieses, ist ein künstliches neuronales Netzwerk aus verschiedenen Schichten von Neuronen, auch Neuronenlayer genannt, zusammengesetzt. Die verschiedenen Neuronenlayer sind in drei verschiedene Kategorien einteilbar: [8]

- Der Inputlayer bezeichnet die erste Schicht an Neuronen in dem Netzwerk. Dieser Layer ist insofern speziell, als dass nur ein einziger Wert in jedes der vorhandenen Neuronen eingegeben wird. Die Anzahl der Neuronen im Inputlayer wird bestimmt durch die Anzahl an Parameter, welche der zu trainierende Datensatz aufweist.[8]

- Outputlayer wird die letzte Schicht eines neuronalen Netzwerks genannt. Der Outputlayer ist die Schicht, welche am Schluss die Vorhersagen darstellt. Damit dieser Layer seinen Zweck richtig erfüllen kann, werden im Outputlayer andere Berechnungen durchgeführt als in den anderen Schichten. Nutzt ein Netzwerk das Klassifikationsverfahren, um seine Vorhersagen zu treffen, so entspricht die Anzahl der Neuronen im Outputlayer immer der Anzahl an Labeln, in welche der Datensatz kategorisiert werden soll. Bei der Ausgabe entspricht jedes Neuron des Outputlayers einem Label und gibt mit seinem Wert die Wahrscheinlichkeit für die Zugehörigkeit der Eingabe in der entsprechenden Kategorie an.[8]

- Hiddenlayer ist die Bezeichnung für alle Neuronenlayer, welche zwischen dem In- und Outputlayer liegen. Die Hiddenlayer bilden den grössten Anteil der Schichten in einem künstlichen neuronalen Netzwerk und können eine beliebige Grösse bzw. Anzahl an Neuronen besitzen. Obwohl die Grösse und Anzahl der Hiddenlayer grundsätzlich variabel ist, haben diese Parameter einen beträchtlichen Einfluss auf die Leistung des Netzwerks. Grundsätzlich wächst die ideale Anzahl und Grösse der Schichten mit der Komplexität des zu trainierenden Datensatzes.[8]

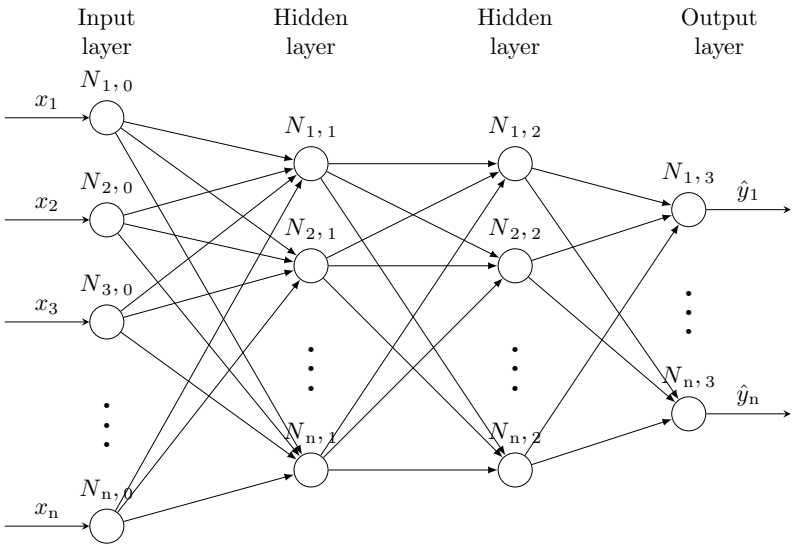



Abbildung 1: Allgemeines mehrschichtiges künstliches neuronales Netzwerk. (Eigene Abbildung)

[8] Wikipedia: Künstliches neuronales Netz. https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz (Abruf 27.10.2022)

In einem allgemeinen künstlichen neuronalen Netzwerk sind, wie in Abbildung 1 dargestellt, alle Neuronen einer Schicht mit allen Neuronen der nächsten und der vorherigen Schicht verbunden. Die künstlichen Neuronen können sich je nach Typ und Art der Neuronenschicht leicht unterscheiden, wobei der grundlegende Aufbau eines Neurons aber stets derselbe ist. Jedes künstliche Neuron ist grundsätzlich nur ein Modell, in welchem verschiedene mathematische Operationen ausgeführt werden. [8]

## 3.1 Netzeingabe und nicht-aktivierte Neuronenausgabe

Eine erste Operation, die jedes künstliche Neuron ausführt, ist die Berechnung der Netzeingabe $net_{k,j}$ (3.1). Diese berechnet sich wie folgt: jede Eingabe $x_{i,j}$, welche entweder im Fall $x_{i,j=0}$ der Eingabe des Netzwerks oder im Fall $x_{i,j>0}$ der Ausgabe eines Neurons $N_{i,j}$ entspricht, wird mit einem spezifischen Gewicht $w_{i,k,j+1}$ multipliziert. Das Gewicht definiert sich - anders als die Eingabe - durch zwei Neuronen, das Neuron $N_{i,j}$ und das Neuron $N_{k,j+1}$. Die Produkte aller Eingaben und Gewichte werden jeweils für alle Neuronen $N_{k,j+1}$ addiert und ergeben die Netzeingabe des entsprechenden Neurons. In einem weiteren Schritt wird ein für jedes Neuron individueller Schwellenwert $b_{k,j+1}$ addiert. Daraus resultiert der nicht-aktivierte Wert des Neurons $Z_{k,j}$ (3.2).[9] [8]

$$net_{k,j+1} = \sum_{i=1}^{n} x_{i,j} \cdot w_{i,k,j+1} \tag{3.1}$$

$$Z_{k,j} = net_{k,j} + b_{k,j} \tag{3.2}$$

Das ganze Konzept lässt sich mathematisch vereinfachen, indem man die Gewichte $w_{i,k,j}$, die Schwellwerte $b_{k,j}$ und die Eingaben $x_{i,j}$ als Vektoren darstellt. Der Vektor $\vec{x}_j$ beinhaltet alle Eingaben der Schicht $j$. Da es für jedes Neuron einer Schicht standardmässig nur eine Ausgabe gibt, ist dieser Vektor eindimensional. Der Vektor $\vec{b}_j$ setzt sich aus allen Schwellenwerten der Schicht $j$ zusammen. Auch er ist eindimensional, da jedes Neuron nur über einen Schwellenwert verfügt. Der Vektor $\vec{w}_j$, welcher die Gewichte beinhaltet, ist zweidimensional. Dies ist der Fall, da es für jede Verbindung von zwei Neuronen benachbarter Schichten jeweils ein spezifisches Gewicht gibt. Die Berechnung der nicht-aktivierten Werte aller Neuronen einer Schicht lässt sich damit deutlich einfacher berechnen (3.3)(3.4).[8] [9]

$$\vec{x}_j \cdot \vec{w}_{j+1} + \vec{b}_{j+1} = \vec{Z}_{j+1} \tag{3.3}$$

$$\begin{bmatrix} x_1 & x_2 & \dots & x_i \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \vdots & \vdots \\ w_{i,1} & w_{i,2} & \dots & w_{i,k} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \cdots & b_k \end{bmatrix} = \begin{bmatrix} Z_1 & Z_2 & \cdots & Z_k \end{bmatrix} \tag{3.4}$$

[9] Vgl. Wikipedia: Künstliches Neuron. https://de.wikipedia.org/wiki/Künstliches_Neuron (Abruf 27.10.2022).

Zudem ist es möglich, mehrere Ausgaben anhand multipler Eingaben gleichzeitig zu berechnen. In diesem Fall spricht davon, mehrere Samples durch das Modell laufen zu lassen. Ein Sample bezeichnet eine einzelne Eingabe eines Netzwerks. Mit der Samplegrösse bezeichnet man die Anzahl an Eingaben, welche ein Netzwerk gleichzeitig verarbeitet. Um dies zu ermöglichen, ergänzt man weitere Vektoren von Samples in der zweiten Dimension der Matrix, welche die Eingaben darstellt (3.5). Dazu müssen weder Änderungen am Vektor der Gewichte, noch am Vektor der Grenzwerte vorgenommen werden. Rechnet man in diesem Fall mit Vektoren, ist es wichtig, dass die Gewichte nicht mit den Eingaben multipliziert werden, sondern eine Operation namens Skalarprodukt durchgeführt wird.[10] Die mathematische Konstruktion des nicht-aktivierten Neurons lässt die Modellierung jeder beliebigen linearen mathematischen Funktion zu, indem nur die verschiedenen Parameter verändert werden.[8] [9]

$$\begin{bmatrix} x_{0,1} & x_{0,2} & \dots & x_{0,i} \\ x_{1,1} & x_{1,2} & \dots & x_{1,i} \\ x_{2,1} & x_{2,2} & \dots & x_{2,i} \\ x_{3,1} & x_{3,2} & \dots & x_{3,i} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \vdots & \vdots \\ w_{i,1} & w_{i,2} & \dots & w_{i,k} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \cdots & b_k \end{bmatrix} = \begin{bmatrix} Z_{0,1} & Z_{0,2} & \cdots & Z_{0,k} \\ Z_{1,1} & Z_{1,2} & \cdots & Z_{1,k} \\ Z_{2,1} & Z_{2,2} & \cdots & Z_{2,k} \\ Z_{3,1} & Z_{3,2} & \cdots & Z_{3,k} \end{bmatrix} \tag{3.5}$$

## 3.2 Aktivierungsfunktionen

Die zweite wichtige Operation, die in einem künstlichen Neuron stattfindet, ist die Anwendung einer Aktivierungsfunktion (3.6). Die Aktivierungsfunktion dient in erster Linie dazu, dem Netz zu erlauben, nichtlineare Funktionen zu modellieren. Eine Voraussetzung für Aktivierungsfunktionen ist daher, dass sie nichtlinear sind. Eine weitere Notwendigkeit für eine nichtlineare Aktivierungsfunktion findet sich in mehrschichtigen künstlichen Netzen. Betrachtet man das Verhalten der vorangegangenen linearen Berechnungen, die in einem mehrschichtigen Netz ausgeführt werden, so stellt man fest, dass das gesamte Netz als ein Netz mit nur einer einzigen Schicht dargestellt werden kann. Im Prinzip kann theoretisch fast jede nichtlineare Funktion als Aktivierungsfunktion verwendet werden. Da diese Funktion mitunter mehrere tausend Mal berechnet werden muss, ist es sinnvoll, eine Funktion zu verwenden, von der nicht nur die Funktion selbst, sondern auch ihre Ableitung einfach und zeitsparend berechnet werden kann. Eine weitere Eigenschaft, die für eine Aktivierungsfunktion unabdingbar ist, ist die Stetigkeit, auch konstante Differenzierbarkeit genannt. Im weiteren Verlauf dieser Arbeit wird deutlich, dass für den Trainingsprozess alle Funktionen des Netzes differenzierbar sein müssen. Es muss also gewährleistet sein, dass das Programm während des Trainings nicht auf Definitionslücken stösst. Neben dem Zweck, ein Neuron zu aktivieren, haben einige Aktivierungsfunktionen noch andere Zwecke. Die Aktivierungsfunktion der Ausgabeschicht hat z.B. auch die Aufgabe, die Ausgabe in ein geeignetes Format zu bringen.[11]

$$\vec{A}_j = \varphi(\vec{x}_{j-1} \cdot \vec{w}_j + \vec{b}_j) \tag{3.6}$$

[10] Vgl. Wikipedia: Skalarprodukt. https://de.wikipedia.org/wiki/Skalarprodukt (Abruf 27.10.2022).
[11] Vgl. Wikipedia: Activation function. https://en.wikipedia.org/wiki/Activation_function (Abruf 27.10.2022).

### 3.2.1 ReLU

Die Funktion namens Rectified Linear Unit oder kurz ReLU ist die beliebteste und am häufigsten verwendete Aktivierungsfunktion. Obwohl es nicht danach aussieht, ist die Funktion nicht linear. Die Einfachheit der Funktion (3.7) und ihrer Ableitung (3.8) ermöglicht eine schnelle Berechnung derselben. In dieser Arbeit wurden die Netze, sofern nicht anders angegeben, mit der ReLU-Funktion angepasst, um die Effizienz zu erhöhen.[11]

$$\varphi_{ReLU}(Z) = \begin{cases} Z & \text{falls } Z > 0 \\ 0 & \text{falls } Z \leq 0 \end{cases} \tag{3.7}$$

$$\varphi'_{ReLU}(Z) = \begin{cases} 1 & \text{falls } Z > 0 \\ 0 & \text{falls } Z \leq 0 \end{cases} \tag{3.8}$$

### 3.2.2 Sigmoid

Eine weitere sehr beliebte Aktivierungsfunktion ist die Sigmoid-Funktion (3.9). Diese wird aufgrund ihrer s-förmigen Natur anders als die meisten anderen Aktivierungsfunktionen oft mit einem kleinen griechischen s dargestellt. Die Sigmoidfunktion war vor der Entdeckung der ReLU-Funktion weit verbreitet und wird heute meist in Ausgangsschichten von Netzen zur Lösung von Regressionsproblemen verwendet. Die Sigmoidfunktion ist ausserdem einfach zu berechnen, ebenso wie ihre Ableitung (3.10). Ein Problem, mit dem die Sigmoidfunktion zu kämpfen hat, ist der sogenannte verschwindende Gradient.
Wenn man besonders grosse Werte in die Ableitung der Sigmoidfunktion einsetzt, wird der Wert der Ausgabe verschwindend klein, was dazu führt, dass der Trainingsprozess zum Erliegen kommt.[11] [12]

$$\sigma(Z) = \frac{1}{e^{-Z} + 1} \tag{3.9}$$

$$\sigma'(Z) = \sigma(Z) \cdot (1 - \sigma(Z)) \tag{3.10}$$

### 3.2.3 Tanh

Der hyperbolische Tangens oder Tanh (3.11) ist eine weitere bekannte Aktivierungsfunktion. Die Funktion ist der Sigmoid-Funktion sehr ähnlich, ist aber komplizierter zu berechnen. Aus diesem Grund wird die Sigmoid-Funktion oft bevorzugt. Die Berechnung der Ableitung der Tanh-Funktion (3.12) ist ebenfalls komplexer als die der Sigmoid-Funktion. Der Hauptunterschied der Tanh-Funktion zur Sigmoid-Funktion besteht darin, dass die Wertebereiche nicht zwischen null und eins, sondern zwischen minus eins und eins liegen. Dieser erweiterte Wertebereich ist für bestimmte Datensätze nützlich.[11]

$$\varphi_{Tanh}(Z) = \frac{e^{Z} - e^{-Z}}{e^{Z} + e^{-Z}} \tag{3.11}$$

$$\varphi'_{Tanh}(Z) = 1 - \varphi^2_{Tanh}(Z) \tag{3.12}$$

[12] Vgl. Wikipedia: Vanishing gradient problem. https://en.wikipedia.org/wiki/Vanishing_gradient_problem (Abruf 27.10.2022).

### 3.2.4 GCU

Die Funktion namens Growing Cosine Unit oder kurz GCU (3.13) ist eine Aktivierungsfunktion, die nur selten verwendet wird. Die Funktion hat Vorteile in Bezug auf bestimmte Datensätze, da sie eine oszillierende Funktion ist. Die Berechnung der Funktion selbst ist jedoch nicht sehr effizient, ebenso wie ihre Ableitung (3.14). Diese Funktion wird nur für sehr spezielle Anwendungen eingesetzt. In dieser Arbeit wird sie dennoch verwendet, da mit ihrer Hilfe die Unterschiede in der Genauigkeit und Geschwindigkeit von Mainstream-Aktivierungsfunktionen wie ReLU sehr gut mit der Genauigkeit von weniger verbreiteten Funktionen verglichen werden können.[13]

$$\varphi_{GCU}(Z) = \cos(Z) \cdot Z \tag{3.13}$$

$$\varphi'_{GCU}(Z) = \cos(Z) - \sin(Z) \cdot Z \tag{3.14}$$

### 3.2.5 Softmax

Eine Aktivierungsfunktion, die in Ausgangsschichten von Netzen, die klassifizieren, verwendet wird, ist die Softmax-Aktivierungsfunktion (3.15). Diese Funktion unterscheidet sich von den anderen Aktivierungsfunktionen dadurch, dass sie die Werte der Matrix vergleicht, die sie als Eingabe erhält. Die Aufgabe der Softmax-Funktion besteht darin, die Ausgaben eines Netzes in eine verständliche Form zu bringen. Die Softmax-Funktion setzt jeden Wert in ein Verhältnis zu allen anderen Werten der Matrix, führt also im Grunde eine Wahrscheinlichkeitsverteilung durch. Die Ausgabe gibt dann an, mit welcher Wahrscheinlichkeit die Datenprobe zu einer bestimmten Kategorie gehört, die durch je ein Neuron repräsentiert wird.[11]

$$\varphi_{Softmax}(Z)_i = \frac{e^{Z_i}}{\sum_n^{j=1} e^{Z_j}} \tag{3.15}$$

## 3.3 Fehler- und Kostenfunktionen

Wenn man ein künstliches neuronales Netz trainieren will, muss man zunächst wissen, wie genau oder ungenau die Vorhersage des Netzes im Moment ist. Eine Methode zur Bestimmung der Genauigkeit ist eine sogenannte Fehlerfunktion (3.16). Die Fehlerfunktionen berechnen einen Fehler, indem sie die Vorhersage des Netzes $\hat{y}_i$ mit dem gewünschten korrekten Wert $y_i$ vergleichen, dem sich die Vorhersage annähern sollte. Der Fehler ist ein Wert, der mit zunehmender Genauigkeit gegen null strebt.[14]

$$L = \lambda(\hat{y}_i, y_i) \tag{3.16}$$

[13] Vgl. Mithra Noel, Mathew; L, Arunkumar; Trivedi, Advait; Dutta, Praneet: Growing Cosine Unit: A Novel Oscillatory Activation Function That Can Speedup Training and Reduce Parameters in Convolutional Neural Networks. https://ui.adsabs.harvard.edu/abs/2021arXiv210812943M/abstract (Abruf 27.10.2022).

[14] Vgl. Wikipedia: Loss function. https://en.wikipedia.org/wiki/Loss_function (Abruf 27.10.2022).

Die Kostenfunktion gibt den Durchschnitt aller Fehler eines Netzes an (3.17). Der durch die Kostenfunktion berechnete Gesamtfehler oder Netzfehler gibt dann die Genauigkeit des gesamten Netzes an.[14]

$$C = \frac{\sum_{i=1}^{n} \lambda(\hat{y}_i, y_i)}{n} = \gamma(\vec{\hat{y}}, \vec{y}) \tag{3.17}$$

Wie bei den Aktivierungsfunktionen gibt es auch hier diverse anwendbare Fehlerfunktionen mit ihren eigenen Vor- und Nachteilen. Für Klassifizierungsprobleme, wie sie in dieser Arbeit behandelt werden, wird hauptsächlich die Fehlerfunktion namens Categorical-Cross-Entropy, kurz CCE, verwendet.[15] [14]

### 3.3.1 CCE

Die Categorical-Cross-Entropy-Funktion (3.18) ist die Fehlerfunktion, die in dieser Arbeit verwendet wird. Der Grund dafür ist die mathematische Natur der CCE-Funktion. Es ist nicht nur sehr einfach, diese Funktion zu berechnen, sondern es ist auch möglich, die Ableitung dieser Funktion mit der Ableitung der Softmax-Aktivierungsfunktion, die direkt vor der Fehlerfunktion angewendet wird, in einer sehr einfachen kombinierten Form zu vereinen. Dies spart eine Menge unnötiger Rechenleistung, da die Ableitung der Softmax-Funktion ohne diese Vereinfachungen ansonsten wesentlich komplizierter wäre.[15]

$$C = -\sum_{i=1}^{n} y_i \cdot log_e(\hat{y}_i) \tag{3.18}$$

[15] Vgl. Koech, Kiprono: Cross-Entropy Loss Function. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e (Abruf 27.10.2022).

## 3.4 Optimierer

Um das künstliche neuronale Netz zu trainieren, ist ein sogenannter Optimierer notwendig. Das Grundprinzip des Optimierers wird Fehlerrückführung genannt und beschreibt den Prozess der Anpassung der einzelnen Parameter in einer Weise, dass der Fehler des Netzes minimiert wird. Um dieses Konzept besser zu verstehen, hilft es, sich das gesamte neuronale Netz, dargestellt von Abbildung 2, als eine einzige Funktion vorzustellen (3.19).

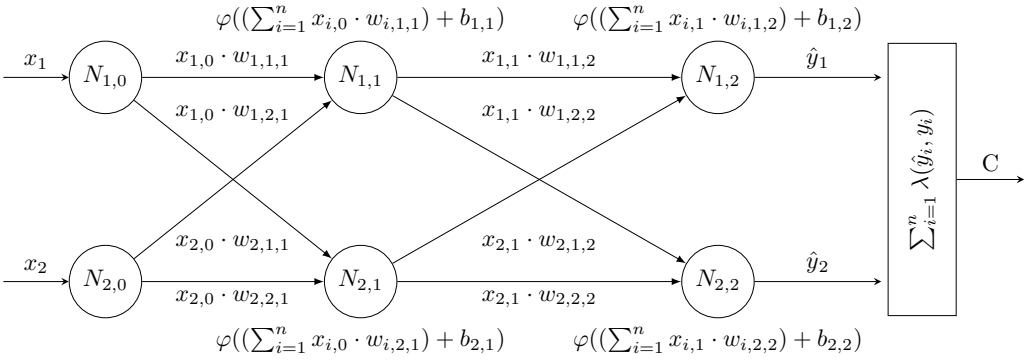



Abbildung 2: Schematische Darstellung eines mehrschichtigen künstlichen neuronalen Netzwerks mit sämtlichen Operationen der Forwärtspropagation. (Eigene Abbildung)

$$C = \frac{\sum_{j=1}^{n} \lambda(\forall_i^n \varphi((\sum_{i=1}^{n} \forall_h^n \varphi((\sum_{g=1}^{n} x_{g,1} \cdot w_{g,h,1}) + b_{h,1}) \cdot w_{h,i,2}) + b_{i,2}), y_j)}{n} \tag{3.19}$$

$$C = \gamma(\varphi(\varphi(\vec{x}_0 \cdot \vec{w}_1 + \vec{b}_1) \cdot \vec{w}_2 + \vec{b}_2), \vec{y}) \tag{3.20}$$

Um zu verstehen, wie die einzelnen Parameter variieren, ist es sinnvoll, die gesamte Gleichung mit Vektoren zu schreiben (3.20), um sie zu veranschaulichen. Aus der Gleichung geht hervor, dass die Gewichte und Grenzwerte einen Einfluss auf den Fehler des Netzes haben. Die Frage ist nun, wie die einzelnen Parameter verändert werden können, um die Funktion zu minimieren. Der Optimierer berechnet den Einfluss der einzelnen Parameter des Netzes auf die Netzfunktion. Dieser Einfluss wird als Gradient bezeichnet. Der Gradient wird berechnet, indem die Funktion nach den einzelnen Parametern differenziert wird. Mithilfe der Kettenregel lassen sich die Gleichungen für den Gradienten der Gewichte (3.21), den Gradienten der Grenzwerte (3.22) und den Gradienten der Eingaben (3.23) berechnen. [16]

$$\frac{\partial C}{\partial w_i} \underset{Kettenregel}{=} \frac{\partial C}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial w_i} \tag{3.21}$$

$$\frac{\partial C}{\partial b_i} \underset{Kettenregel}{=} \frac{\partial C}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial b_i} \tag{3.22}$$

$$\frac{\partial C}{\partial x_i} \underset{Kettenregel}{=} \frac{\partial C}{\partial A} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial Z}{\partial x_i} \tag{3.23}$$

[16] Vgl. Wikipedia: Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent (Abruf 27.10.2022).

Die verschiedenen Terme von diesen Formeln lassen sich in eine einfachere Form bringen, $\frac{\partial C}{\partial A}$ entspricht der Ableitung der Kostenfunktion $\gamma'(A)$, die Funktion $\frac{\partial A}{\partial Z}$ lässt sich zur Ableitung der Aktivierungsfunktion $\varphi'(Z)$ umformen. Im Weiteren lässt sich $\frac{\partial Z}{\partial w_i}$ als $x_i$, $\frac{\partial Z}{\partial b_i}$ als 1 und $\frac{\partial Z}{\partial x_i}$ als $w_i$ darstellen. Folgend können obige Gleichungen 3.21, 3.22 und 3.23 auf eine simplere Weise geschrieben werden als 3.24, 3.25 und 3.26. [15]

$$\frac{\partial C}{\partial w_i} = \gamma'(A) \cdot \varphi'(Z) \cdot x_i \tag{3.24}$$

$$\frac{\partial C}{\partial b_i} = \gamma'(A) \cdot \varphi'(Z) \tag{3.25}$$

$$\frac{\partial C}{\partial x_i} = \gamma'(A) \cdot \varphi'(Z) \cdot w_i \tag{3.26}$$

Subtrahiert man nun den Gradienten jedes Parameters von den entsprechenden Parametern der aktuellen Epoche $e$, erhält man die optimierten Parameter der nächsten Epoche $e + 1$. Mit diesem Verfahren bewegt sich die Netzfunktion in Richtung eines lokalen Minimums. Bevor der Gradient von seinem entsprechenden Parameter subtrahiert wird, wird er mit einer Lernrate $\alpha$ multipliziert. Die Lernrate bestimmt die Schrittweite, mit der sich dem Minimum genähert wird (3.27).[16]

$$p_{i,e+1} = p_{i,e} - \alpha \cdot \frac{\partial C}{\partial p_{i,e}} \tag{3.27}$$

Auf diese Weise passt der Optimierer die einzelnen Parameter Schritt für Schritt an. Während des Optimierungsprozesses werden Operationen der Subtraktion der Gradienten von den Parametern über mehrere Iterationen, auch Epochen genannt, durchgeführt. Dieser Prozess beginnt in der Ausgabeschicht und endet in der Eingabeschicht. Der Optimierungsprozess findet also von hinten nach vorne statt. Dieser Prozess muss auf diese Weise erfolgen, weil der Optimierer zu Beginn nur den Fehler des Netzes kennt. Sobald eine Neuronenschicht angepasst ist, wird die vorherige Schicht optimiert, wobei der Gradient der Eingabe $\frac{\partial C}{\partial x_i}$ des vorherigen Neurons als neuer Anfangswert verwendet wird. Optimierer, wie die Fehlerfunktionen und die Aktivierungsfunktionen, gibt es in verschiedenen Formen. In dieser Arbeit verwenden alle Optimierer ein Grundprinzip namens stochastic gradient descent.[16]

### 3.4.1 SGD mit dynamische Lernraten und Momentum

SDG oder Stochastic Gradient Descent ist die effektivste Methode zur Optimierung der meisten Datensätze. Normalerweise bezieht sich der Begriff auf das Training mit einer einzigen Stichprobe des Datensatzes pro Iteration. Heutzutage wird der Begriff jedoch synonym mit den Begriffen Batch Gradient Descent und Mini Batch Gradient Descent verwendet. Batch Gradient Descent bezieht sich auf den Lernprozess mit dem gesamten Datensatz, d.h. in jeder Iteration werden alle Datenproben gleichzeitig optimiert. Mini Batch Gradient Descent bedeutet den Lernprozess mit einem Teil des gesamten Datensatzes. In dieser Arbeit wurde nur Batch Gradient Descent verwendet. Die Lernrate ist ein zentraler Bestandteil des Optimierers, sie ist verantwortlich für die Schrittweite. Ist die Lernrate zu hoch, werden Minima übersprungen und es findet keine Optimierung statt, ist sie zu niedrig, kann die Funktion in einem kleinen lokalen Minimum stecken bleiben, obwohl ein viel niedrigeres und damit besseres Minimum in der Nähe sein könnte. Neben der geschickten Wahl der Lernrate gibt es eine weitere Methode, dieses Problem zu vermeiden, die dynamische Lernrate. Eine dynamische Lernrate entspricht einer Funktion, die die Lernrate bei jeder Iteration verringert (3.28). Das Konzept hinter dieser Methode besteht darin, die Lernrate und damit die Schrittweite kontinuierlich zu verringern, wobei die Anpassung von der Ausgangs-Lernrate und einer Konstante $\delta_a$ abhängt. Dies soll eine Feinanpassung nach einer gewissen Zeit ermöglichen. [17] [16]

$$\alpha_e = \frac{\alpha_0}{1 + \delta_\alpha \cdot e} \tag{3.28}$$

Eine weitere Optimierung, welche am Optimierer vorgenommen werden kann, ist die Einführung eines Momentums (3.29). Das Konzept lässt sich am besten anhand folgender Analogie verstehen. Rollt man einen Ball einen Hügel hinunter, so wird man feststellen, dass der Ball je nach Verlauf seines vorherigen Weges einen Zuwachs an Geschwindigkeit gewinnt, der Ball wird über kleine Löcher und Unebenheiten des Untergrundes hinwegrollen und den tiefsten Punkt am Fusse des Hügels erreichen. Die Implementierung des Momentums ermöglicht es, eine parameterindividuelle Trägheit aufzubauen und kleinere lokale Minima zu durchqueren, um ein idealeres Minimum zu erreichen.[16]

$$v_{i,e+1} = \Gamma \cdot v_{i,e} - \alpha_e \cdot \frac{\partial A}{\partial p_{i,e}} \tag{3.29}$$

Dieses Momentum, welches hier berechnet wird, wird anstelle des Produkts des Gradienten mit der Lernrate zu den entsprechenden Parametern addiert (3.30). Die Formel eines SGD-Oprimierers mit dynamischer Lernrate und Momentum sieht demnach wie folgt aus. [16]

$$p_{i,e+1} = p_{i,e} + v_{i,e+1} \tag{3.30}$$

[17] Patrikar, Sushant: Batch, Mini Batch and Stochastic Gradient Descent. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a (Abruf 27.10.2022).

### 3.4.2 AdaGrad

Eine erste Evolution des grundlegenden SDG Optimierer wird AdaGrad genannt, AdaGrad steht für adaptive gradient. Bisher wurde eine funktionsübergreifende Lernrate verwendet, es stellt sich jedoch heraus, dass es sinnvoll sein kann, je nach Verhallten der einzelnen Parameter, die Lernrate für Parameter individuell anzupassen. AdaGrad reduziert die Grösse der Anpassungen der Parameter, abhängig von der Grösse der Summe aller vorherigen Änderungen. Dies bewirkt, dass die im Vergleich mit allen Neuronen relative Steigerung von weniger einflussreichen Neuronen stärkere Anpassungen erfahren. Das führt dazu, dass sich mehr Neuronen aktiv anpassen. Zur Umsetzung dieses Konzepts wird die Berechnung der neuen Parameter noch mit einer Division der Wurzel aus einem Cache und einer additiven Konstante erweitert (3.32). Der Cache wird berechnet, indem der Cache der letzten Epoche mit dem Quadrat der individuellen Parameteranpassung addiert wird, sodass sich der Cache mit dem Quadrat der Änderung des entsprechenden Parameters verhält (3.31). [16]

$$\Sigma_{i,e+1} = \Sigma_{i,e} + (\frac{\partial A}{\partial p_{i,e}})^2 \tag{3.31}$$

$$p_{i,e+1} = p_{i,e} - \frac{\alpha_e \cdot \frac{\partial A}{\partial p_{i,e}}}{\sqrt{\Sigma_{i,e+1}} + \varepsilon} \tag{3.32}$$

### 3.4.3 RMSprop

RMSprop, kurz für Root Mean Square Propagation, ist eine weitere Verbesserung des SGD-Optimierers. Wie der Adagrad-Optimierer folgt auch RMSprop dem Prinzip einer parameterindividuellen Lernrate. RMSprop versucht, diese Methode zu verfeinern, indem es einen Mechanismus hinzufügt, der dem Momentum-Prinzip gleicht (3.33). Dieses Momentum wird nun auf den Cache angewandt, wodurch er sich gleichmässiger verhält. Wenn sich der Cache abrupt ändert, kommt es nicht zu einem Ausbruch des Wertes, sondern die Trägheit sorgt dafür, dass sich die Wertanpassungen nicht schlagartig auf die Parameter auswirken. Die neue Konstante heisst Cache Memory Decay Rate und wird mit einem $\rho$ dargestellt. Die Formel zur Optimierung der Parameter ist gleich wie beim AdaGrad-Optimierer (3.34).[16]

$$\Sigma_{i,e+1} = \Sigma_{i,e} + \rho \cdot \Sigma_{i,e} + (1 - \rho) * (\frac{\partial A}{\partial p_{i,e}})^2 \tag{3.33}$$

$$p_{i,e+1} = p_{i,e} - \frac{\alpha_e \cdot \frac{\partial A}{\partial p_{i,e}}}{\sqrt{\Sigma_{i,e+1}} + \varepsilon} \tag{3.34}$$

### 3.4.4 Adam

Adam, kurz für Adaptive Momentum, ist einer der weit verbreitetsten Optimierer und basiert auf RMSprop. Dabei wird der RMSprop Algorithmus mit dem Konzept eines weiteren Momentums verbunden. Im weiteren führt der Adam Optimierer einen zusätzlichen Mechanismus ein. Dieser wird Verzerrung-Korrektur-Mechanismus oder Grenzwert-Korrektur-Mechanismus genannt und hat nichts mit den Grenzwerten des Netzwerks zu tun. Der Verzerrung-Korrektur-Mechanismus wird angewendet, um die anfänglichen Nullwerte im Cache und im Momentum zu Beginn des Trainingsvorganges zu kompensieren. Der korrigierte Wert des Momentums und des Caches wird mathematisch mit der Division der Komponenten mit einer weiteren Konstante $\beta$, welche jedoch zuerst von eins subtrahiert wird, berechnet (3.35)(3.36). [16]

$$v_{i,e}^{korrigiert} = \frac{v_{i,e}^{unkorrigiert}}{1 - \beta_1^{e+1}} \tag{3.35}$$

$$\Sigma_{i,e}^{korrigiert} = \frac{\Sigma_{i,e}^{unkorrigiert}}{1 - \beta_2^{e+1}} \tag{3.36}$$

Fügt man alles zusammen, so werden im Adam Optimierer grundlegend fünf Schritte ausgeführt. Als erstes wird das Momentum und der Cache beider Parameter berechnet (3.37)(3.38).

$$v_{i,e+1} = \beta_1 \cdot v_{i,e} + (1 - \beta_1) \cdot \frac{\partial A}{\partial p_{i,e}} \tag{3.37}$$

$$\Sigma_{i,e+1} = \beta_2 \cdot \Sigma_{i,e} + (1 - \beta_2) \cdot (\frac{\partial A}{\partial p_{i,e}})^2 \tag{3.38}$$

In den weiteren zwei Schritten wird jeweils das Momentum und der Cache in die entsprechende Korrekturfunktion eingesetzt (3.35)(3.36). Der letzte Schritt, welcher noch ausgeführt werden muss, beschreibt die Adaption der Parameter. Die Parameter werden gleich wie beim RMSprop Algorithmus angepasst, mit der Ausnahme, dass die Anpassung mithilfe des Momentums stattfindet (3.39). [16]

$$p_{i,e+1} = p_{i,e} - \frac{\alpha_e \cdot v_{i,e}^{korrigiert}}{\sqrt{\Sigma_{i,e}^{korrigiert}} + \varepsilon} \tag{3.39}$$

## 3.5 Overfitting

Ein Problem, mit dem viele künstliche neuronale Netze zu kämpfen haben, heisst Overfitting. Overfitting beschreibt die zu genaue Anpassung oder Überanpassung des Netzes an den Trainingsdatensatz. Grundsätzlich besteht das Ziel neuronaler Netze darin, einen hintergründigen Algorithmus zu erlernen. Wenn sich das Netz jedoch zu genau an die Trainingsdaten anpasst, lernt es die ihm zugeführten Daten lediglich auswendig. Ob ein Netz sich zu genau anpasst, kann überprüft werden, indem die Genauigkeit des Netzes mit weiteren untrainierten Daten des Datensatzes bestimmt und mit der Genauigkeit der trainierten Daten verglichen wird. Die Daten, die für diesen Vergleich verwendet werden, nennt man Validierungsdaten. Das Problem des Overfitting kann auf viele verschiedene Arten gelöst werden, wobei die Wirksamkeit der verschiedenen Methoden unterschiedlich ist. Eine erste Massnahme besteht darin, die Grösse und Anzahl der Hiddenlayers anzupassen. Im Prinzip sind diese Parameter frei wählbar, aber es ist wichtig, die Komplexität des Netzes proportional zur Komplexität des Datensatzes zu wählen. Je mehr Neuronen ein Netz hat, desto genauer werden die Anpassungen auf der Grundlage des Trainingsdatensatzes vorgenommen, was zu einer Überanpassung des Modells führt. Wenn ein Netz jedoch zu wenige Neuronen hat, kann es sich nicht richtig an die Daten anpassen und lernt daher schlechter. [18]

### 3.5.1 L1 und L2 Regularisierer

Eine deutlich konstantere und effektivere Methode das Problem zu lösen wird Regularisierung genannt. Die Regularisierung beruht auf der Annahme, dass es besser ist, wenn Werte der Parameter des künstlichen neuronalen Netzwerkes nicht allzu gross werden. Je einheitlicher die Werte eines Netzwerks sind, desto mehr werden die Daten verallgemeinert und desto weniger Overfitting findet statt. Um dieses Konzept umzusetzen, erweitert man den Gesamtfehler des Netzwerks um den sogenannten Regularisierungsfehler. Als Regularisierungsfehler wird ein Wert bezeichnet, der zu dem Fehler dazuaddiert wird und abhängig von der Grösse der Parameter ist (3.40). Populär sind zwei Regularisierer, genannt Lambda 1 und Lambda 2.[19]

$$C_{Netzwerk} = C(\hat{y}, y) + C_{L1} + C_{L2} \tag{3.40}$$

Lambda 1 entspricht dem Wert der Summe aller Beträge aller Parameter multipliziert mit einer Regularisierungskonstante $\Lambda_1$ (3.41). [18]

$$L_1 = \Lambda_1 \cdot \sum_{i=1} |p_i| \tag{3.41}$$

Lambda 2 entspricht dem Wert der Summe aller quadrierten Parameter multipliziert mit einer Regularisierungskonstante $\Lambda_2$ (3.42). [18]

$$L_2 = \Lambda_2 \cdot \sum_{i=1} p_i^2 \tag{3.42}$$

[18] Vgl. Wikipedia: Overfitting. https://en.wikipedia.org/wiki/Overfitting (Abruf 27.10.2022).

[19] Vgl. Tyagi, Neelam: L2 vs L1 Regularization in Machine Learning. https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning (Abruf 27.10.2022).

# 4 Datensätze

Bei der Programmierung eines künstlichen neuronalen Netzes ist die Programmierung des Netzes nicht die einzige Aufgabe, die es zu lösen gilt. Ein weiterer zentraler Teil dieser Arbeit ist das eigentliche Training des Netzes. Datensätze, die für das Training eines künstlichen neuronalen Netzes verwendet werden, müssen eine grosse Menge an Daten enthalten, die korrekt beschriftet werden müssen. Glücklicherweise gibt es viele vorgefertigte Datensätze im Internet, die leicht heruntergeladen und angewendet werden können. Die in dieser Arbeit verwendeten Datensätze decken ein relativ breites Spektrum an Komplexität ab und ermöglichen es, die Grenzen, Stärken und Schwächen dieses Netzes zu ermitteln.

## 4.1 Spiral-Datensatz

Der erste Datensatz, auf dem ein künstliches neuronales Netz trainiert wird, ist in der Regel recht einfach und überschaubar. Der hier verwendete Spiral-Datensatz kann mit einer einfachen Funktion erzeugt werden, die es ermöglicht, die Anzahl der Klassen und Datenproben mit zwei Parametern anzupassen (Abbildung 3). Ein weiterer Vorteil dieses Datensatzes ist die Übersichtlichkeit. Man kann sich die Entscheidungsfunktion, die das Netz bilden muss, leicht vorstellen. Da jeder Punkt der Spirale mit zwei Variablen definiert werden kann, ist die Grösse des Trainingsdatensatzes relativ klein, was zu einem schnelleren Trainingsprozess führt. Ein Nachteil dieses Datensatzes ist, dass eine Entscheidungsgrenze nicht immer eindeutig gezogen wird. Dies liegt daran, dass sich die Bereiche, in denen die Punkte erzeugt werden, teilweise leicht überschneiden.

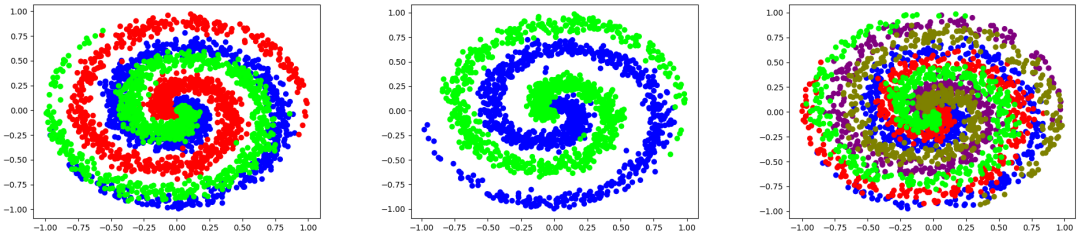
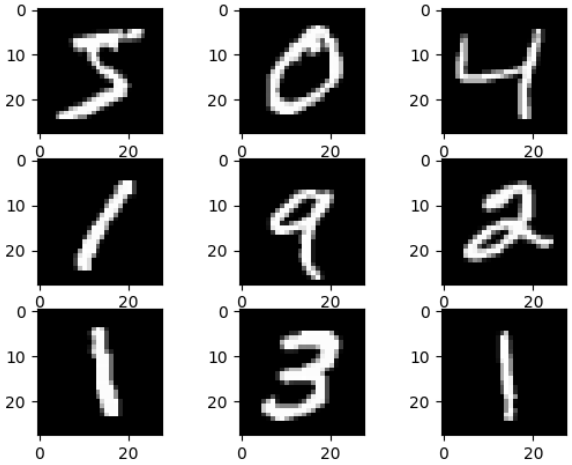


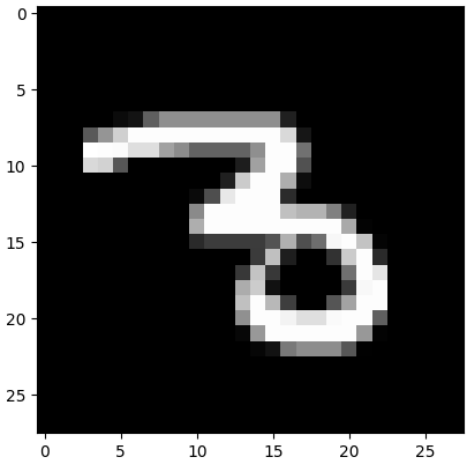
Abbildung 3: Verschiedene Variationen des Spiral-Datensatz mit unterschiedlicher Anzahl an Samples und Kategorien. (Eigene Abbildung)

## 4.2 MNIST-Datensatz

Ein bekannter Datensatz ist der MNIST-Datensatz, der aus Tausenden von handgezeichneten Ziffern besteht (Abbildung 4), die von null bis neun beschriftet sind. Dieser Datensatz ist viel komplexer als der Spiral-Datensatz, da das Netz 784 Eingaben pro Datenprobe erhält. Dennoch ist es einer der einfachsten Datensätze und wird oft nach der Erstellung des neuronalen Netzes verwendet, um dessen Funktionalität zu testen. Ein grosser Vorteil des MNIST-Datensatzes ist, dass die Daten bereits ideal aufbereitet sind. Normalerweise müssen beim Training von Bildern in einem künstlichen neuronalen Netz zunächst verschiedene Programme auf die Daten angewendet werden, um sie für das Training vorzubereiten. Der Datensatz kann in einem Format importiert werden, welches ermöglicht, die Daten ohne weitere Nachbearbeitung für den Trainingsvorgang zu verwenden. [20]



Mehrere Trainingssamples des Datensatzes MNIST



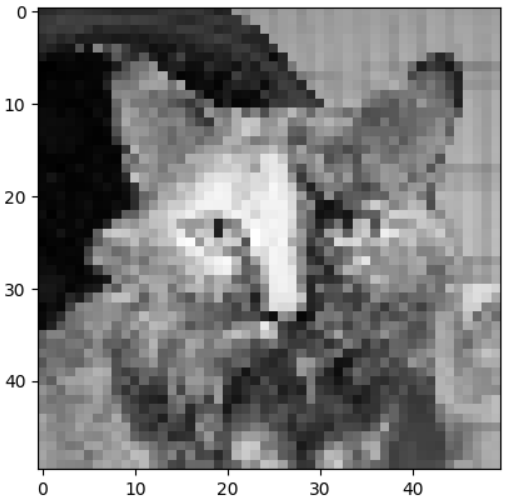
Einzelnes Trainingssample des Datensatzes MNIST

Abbildung 4: Datensamples des MNIST-Datensates. https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/ (Abrufdatum: 30.10.2022)

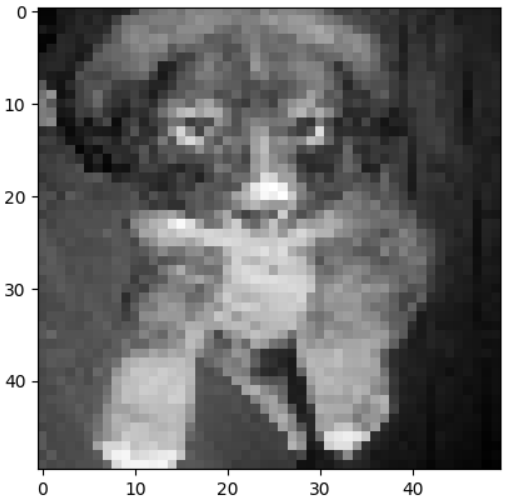
[20] Deeplake: MNIST. https://datasets.activeloop.ai/docs/ml/datasets/mnist/ (Abruf 27.10.2022).

## 4.3 Katzen-Hunde-Datensatz

Der letzte und zugleich komplexeste Datensatz, der in dieser Arbeit behandelt wird, ist der Katzen-Hund-Datensatz. Dieser Datensatz besteht aus heruntergeladenen, unbearbeiteten Bildern von Hunden und Katzen (Abbildung 5). Reine Bilder können nicht in ein neuronales Netz eingespeist werden, da zunächst zahlreiche Verarbeitungsschritte erforderlich sind, um jedes Bild in eine Zahlenfolge mit konstanter Grösse umzuwandeln. Ausserdem zeigen diese Bilder nicht nur die Haustiere selbst, sondern auch ihre Umgebung. Das künstliche neuronale Netze muss also in der Lage sein, das Tier vom Hintergrund zu unterscheiden. Spätestens bei der Betrachtung der einzelnen Trainingsbeispiele wird deutlich, dass ein Teil der Daten selbst für einen Menschen unkenntlich sind und somit komplexer sind als die beiden vorherigen Datensätze.




Trainingssample mit dem Label "Katze"




Trainingssample mit dem Label "Hund"

Abbildung 5: Datensamples des Katze-Hund Datensatz. (Eigene Abbildung)

# 5 Praktische Implementierung des künstlichen neuronalen Netzwerks

In diesem Abschnitt soll nun die praktische Umsetzung der besprochenen theoretischen Konzepte erläutert werden. Bei der Erstellung der praktischen Arbeit, dem Programm des künstlichen neuronalen Netzes, wurde auf die folgenden Punkte Wert gelegt:

- Modularer Aufbau des Programms: Ein neuronales Netz muss man sich als eine Abfolge verschiedener Algorithmen vorstellen, es ist wichtig, dass man als Entwickler einzelne Komponenten austauschen oder hinzufügen kann, ohne ein komplett neues Netz erstellen zu müssen. Um diesen modularen Aufbau zu erreichen, wurde viel Wert auf die Autarkie der einzelnen Funktionen gelegt.
- Klare Struktur des Programms: Eine konstante und klare Syntax ist essenziell für einen effizienten Arbeitsprozess.

Der gesamte besprochene Quelltext des künstlichen neuronalen Netzes und den verschiedenen Datensätzen ist unter dem QR-Code (Abbildung 6) verfügbar.

Abbildung 6: QR-Code zu dem gesammten Quelltext der praktischen Arbeit. Auch mit folgendem Link aufrufbar: https://github.com/BasilRohner/MAR. (Eigene Abbildung)

## 5.1 Implementierung des künstlichen neuronale Netzwerks

Der erste Teil der praktischen Arbeit befasste sich mit der Implementierung des Programms für das künstliche neuronale Netz. Schritt für Schritt wurden verschiedene Funktionalitäten implementiert und getestet.

### 5.1.1 Neuronenschichten

Die erste Implementierung, die in der praktischen Arbeit vorgenommen wurde, ist die der grundlegenden Neuronenschicht. Bei der Instanziierung einer Neuronenschicht muss sowohl die Anzahl der Neuronen der Schicht als auch die Anzahl der Eingänge, d.h. die Grösse der vorhergehenden Schicht, angegeben werden. Im weiteren muss festgelegt werden, welche Aktivierungsfunktion und ggf. auch Fehlerfunktion verwendet werden soll. Die einzelnen Parameter werden standardmäßig zufällig generiert. Neben den grundlegenden Parametern wie Gewichte und Grenzwerte werden in dieser Klasse auch weitere, für die Neuronenschicht spezifische Parameter wie Regularisierungsparameter, Momentum und Caches gespeichert. Die Klasse einer allgemeinen Neuronenschicht besteht aus zwei Methoden, wobei die erste Methode für den sogenannten Vorwärtspass zuständig ist. Der Vorwärtspass bezeichnet alle Berechnungen eines neuronalen Netzes bis hin zur Berechnung des Gesamtfehlers des Netzes. Wenn diese Funktion von einer Instanz ausgeführt wird, welche die Eingabeschicht bildet, dann ist es notwendig, die Eingabe des Netzes anzugeben. Möchte man den Fehler berechnen, so muss der letzten Neuronenschicht noch eine Matrix mit den Label des Trainingsdatensatzes hinzugefügt werden. Die zweite Methode ist für den Rückwärtspass zuständig. Im Gegensatz zum Vorwärtspass ist der Rückwärtspass für die Optimierung der einzelnen Parameter, also den Optimierungsprozess, zuständig.

### 5.1.2 Aktivierungsfunktionen und Fehlerfunktionen

Sowohl die Aktivierungsfunktionen als auch die Fehlerfunktionen müssen bei der Instanziierung einer Neuronenschicht als Argumente angegeben werden und wurden entsprechend als einfache Funktionen implementiert. Dieser Ansatz erlaubt den modularen Austausch der einzelnen Funktionen und sogar die Implementierung einer völlig neuen Funktion, ohne diese vorher im Programm implementieren zu müssen. Da in dieser Arbeit ausschliesslich die CCE-Fehlerfunktion angewendet wurde, wurde auf die Implementierung der Ableitung der Softmax-Aktivierungsfunktion verzichtet. Diese ist bereits mit der Ableitung der Fehlerfunktion verbunden.

### 5.1.3 Optimierer

Im Gegensatz zu den Fehlerfunktionen und den Aktivierungsfunktionen wurden die verschiedenen Optimierer als eigene Schicht im neuronalen Netz implementiert. Obwohl der Optimierer theoretisch keine eigenen Neuronen hat, ist es sinnvoll, ihn als letzte Schicht des Netzes zu betrachten. Diese Art der Implementierung mag seltsam erscheinen, insbesondere im Vergleich zu dem viel einfacheren Ansatz der Implementierung der Aktivierungs- und Fehlerfunktionen. Der Grund für diese Komplikation ist, dass eine solche Implementierung die Definition der für den Optimierer speziefischen Parameter zum Zeitpunkt der Netzerstellung ermöglicht.

### 5.1.4 Netzwerk

Der Kern des Programms ist eine Klasse namens Netzwerk. Diese Klasse kann als Gerüst gesehen werden, auf dem das künstliche neuronale Netz aufgebaut ist. Die Klasse Netzwerk definiert die Interaktionen der einzelnen Schichten untereinander und sorgt so für die Verbindung der einzelnen Schichten. Nicht nur die einzelnen Funktionen des Netzes, wie z.B. der Trainingsprozess, werden durch Methoden dieser Klasse gesteuert. Die Klasse sorgt auch für die Dokumentation und Darstellung der Ergebnisse aller Trainingsepochen. In der Netzklasse sind wie in der Klasse der Neuronschicht zwei Methoden angelegt. Einerseits eine Methode, die dafür zuständig ist, einen einfachen Vorwärtspass durchzuführen und eine zweite Methode, in welcher der Trainingsprozess implementiert ist.

## 5.2 Implementierung der Datensätze

In einem weiteren Teil dieser Arbeit wurden die verschiedenen Datensätze implementiert und getestet. Datensätze wie der Spiral-Datensatz konnten als einfache Funktion im Programm modelliert werden, während der MNIST-Datensatz über das Modul Keras importiert werden konnte und der Katzen-Hund-Datensatz in Form eines Ordners mit zahlreichen Bildern aus dem Internet heruntergeladen werden musste.

### 5.2.1 Spiral-Datensatz

Der erste implementierte Datensatz ist der Spiral-Datensatz, der durch eine Funktion erzeugt wird. Die Funktion ist so konstruiert, dass sowohl die Anzahl der Datenproben als auch die Anzahl der Kategorien frei gewählt werden kann. Dies ermöglicht eine gewisse Variabilität, was die Komplexität des Datensatzes angeht.

### 5.2.2 MNIST-Datensatz

Der zweite implementierte Datensatz ist der MNIST-Datensatz. Erfreulicherweise ist es möglich, den Datensatz direkt aus der Keras-Bibliothek zu importieren, wodurch der separate Download der zahlreichen Bilder vermieden wird. Die einzelnen Matrizen werden vor der Eingabe in das Netzwerk normalisiert, was bedeutet, dass der Wertebereich auf Werte von null bis eins reduziert wurde.

### 5.2.3 Katze-Hund-Datensatz

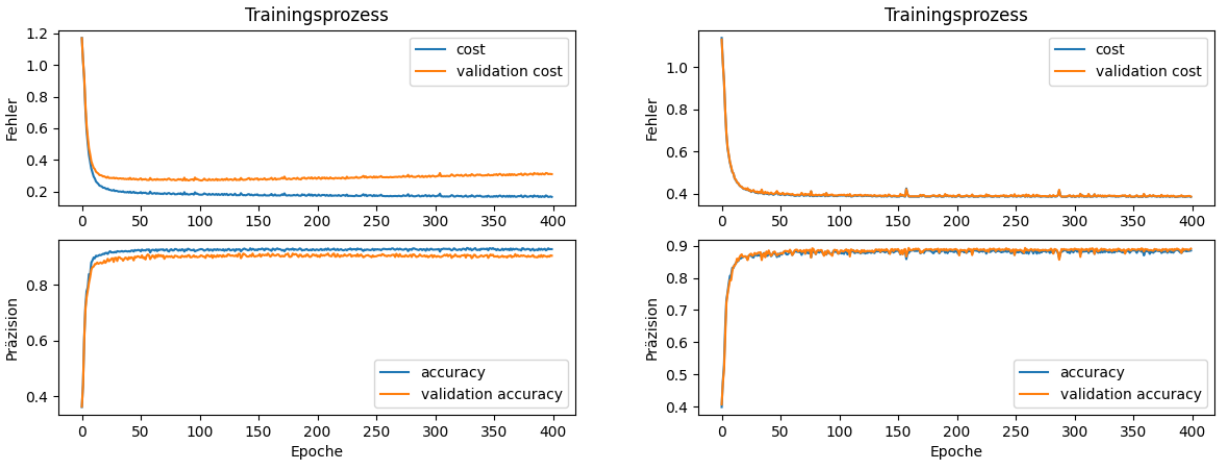
Der komplizierteste Datensatz in der Implementierung ist der Katzen-Hund-Datensatz. Die vom Internet heruntergeladenen Bilder wurden zunächst in ein bestimmtes Format gebracht und mit einem Schwarz-Weiss-Filter bearbeitet. Ein weitere Funktion wandelte die Bilder in eine Zahlenmatrix um, welche sich durch die Helligkeit der einzelnen Pixel des entsprechenden Bildes definiert.

# 6 Resultate

In diesem Abschnitt werden die Ergebnisse des Trainingsprozesses aufgelistet und gleichzeitig interpretiert. Für jeden Datensatz werden Vergleiche der verschiedenen Aktivierungsfunktionen, Optimierer und Regularisierer durchgeführt, wobei der Fokus auf der Leistung des Netzwerks bezogen auf den jeweiligen Datensatz liegt. Darüber hinaus wird das selbst erstellte künstliche neuronale Netzwerk in einem weiteren Schritt mit einem populären Modul namens Keras verglichen.

## 6.1 Spiral-Datensatz

Der Spiral-Datensatz ist der erste Datensatz, der trainiert wurde. Die ersten beiden Trainingsläufe wurden jeweils mit der Aktivierungsfunktion ReLU und dem Optimierer Adam durchgeführt. Die beiden Durchläufe unterscheiden sich dadurch, dass ein Durchlauf mit einem Regularisierer durchgeführt wurde, während der andere Durchlauf als Kontrolle diente und keinen Regularisierer verwendete (Abbildung 7). Aus den beiden Ergebnissen lassen sich einerseits Rückschlüsse auf die Leistungsfähigkeit des Netzwerks, aber auch auf die Effizienz des Regularisierers ziehen.



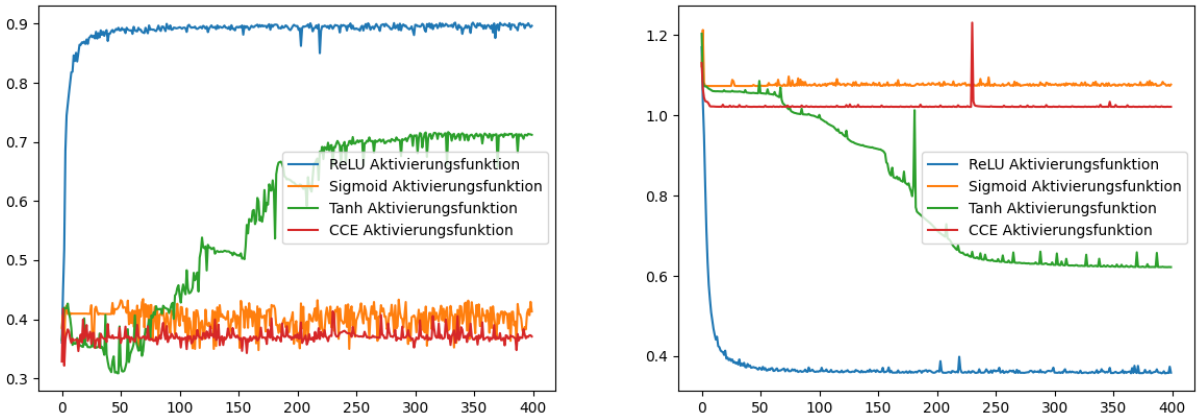

Trainingprozesses ohne Regularisierer

Trainingsprozess mit Regularisierer

Abbildung 7: Resultate des Trainingsprozesses des Spiral-Datensatz mit und ohne Regularisierer mit ReLU Aktivierungsfunktion, CCE Fehlerfunktion und Adam Optimierer. (Eigene Abbildung)

Der erste Trainingslauf zeigte anfänglich einen starken Anstieg der Genauigkeit und gleichzeitig einen proportionalen Rückgang des Fehlers des Netzwerks. Tatsächlich erreichte das Netzwerk bereits in der 25. Epoche eine Genauigkeit von mehr als 90 Prozent gegenüber dem Trainingsdatensatz. In den weiteren 375 Epochen des Trainingslaufs fand jedoch keine weitere Steigerung der Genauigkeit statt. Im Gegensatz zum Trainingsdatensatz fiel die Genauigkeit des Netzwerks mit dem Validierungsdatensatz deutlich schwächer aus. Dieser Leistungsunterschied ist vor allem beim Netzwerkfehler zu beobachten. Das Phänomen des Overfitting ist in diesem Trainingsdurchgang ohne Regularisierung deutlich erkennbar.

Ähnlich wie im ersten Trainingsdurchgang kann auch im zweiten Trainingsdurchgang zunächst ein starker Anstieg der Genauigkeit bei gleichzeitiger Abnahme des Fehlers beobachtet werden. Doch im Gegensatz zum ersten Durchgang gab es kaum einen Unterschied in der Leistung des Netzwerks bezogen auf den Trainings- und den Validierungsdatensatz. Es fiel jedoch auf, dass die Gesamtleistung des Netzwerks etwas nachgelassen hat. So konnte das Netzwerk nur noch knapp eine Genauigkeit von 90 Prozent erreichen und auch der Gesamtfehler erreichte einen deutlich grösseren Wert als im ersten Durchgang. Obwohl der Regularisierer das Overfitting-Problem beseitigen konnte, scheinen sich sowohl die Genauigkeit als auch der Fehler im Vergleich zum ersten Durchlauf nicht verbessert zu haben.

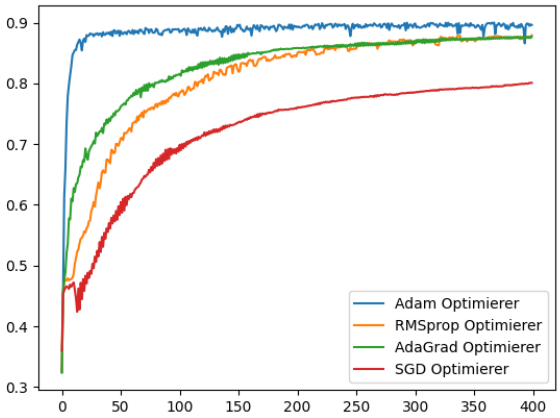



Verlauf der Präzision

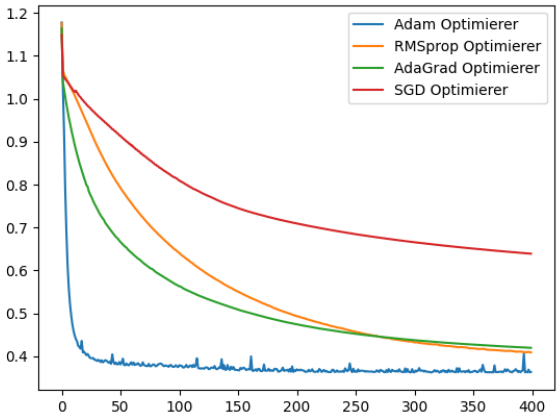
Verlauf des Fehlers

Abbildung 8: Resultate des Trainingsprozesses des Spiral-Datensatz mit verschiedenen Aktivierungsfunktionen, CCE Fehlerfunktion und Adam Optimierer. (Eigene Abbildung)

In vier weiteren Trainingsdurchgängen wurde die Präzision und der Netzwerkfehler bei Netzwerken verschiedener Aktivierungsfunktionen verglichen (Abbildung 8). Die ReLU-Aktivierungsfunktion zeigte sehr früh im Trainingsprozess eine starke Optimierung der Genauigkeit und des Netzwerkfehlers und stabilisiert sich dann in den folgenden Epochen. Die Tanh-Aktivierungsfunktion zeigte zunächst einen kleinen Einbruch, gefolgt von einem konstanten Anstieg der Genauigkeit im weiteren Verlauf des Trainingsprozesses. Selbst nach 400 Epochen erreichte das Netzwerk mit der Tanh-Aktivierungsfunktion nur eine maximale Genauigkeit von etwa 70 Prozent. Sowohl die Sigmoid- als auch die GCU-Aktivierungsfunktion zeigten im gesamten Verlauf des Trainings keine konstanten Optimierungen bezogen auf die Genauigkeit oder den Netzwerkfehler. Es waren jedoch starke Schwankungen der Genauigkeit zwischen 40 und 30 Prozent zu beobachten. Im Gesamtvergleich schnitt die ReLU-Aktivierungsfunktion am besten ab, das ReLU-Netzwerk ist in der Lage, Vorhersagen zu treffen, die um mehr als 20 Prozent genauer sind als Netzwerke mit den anderen Funktionen. Am überraschendsten war das schlechte Abschneiden der Sigmoid-Funktion, die trotz ihres guten Rufs nicht einmal die Hälfte aller Vorhersagen richtig traf.




Verlauf der Präzision



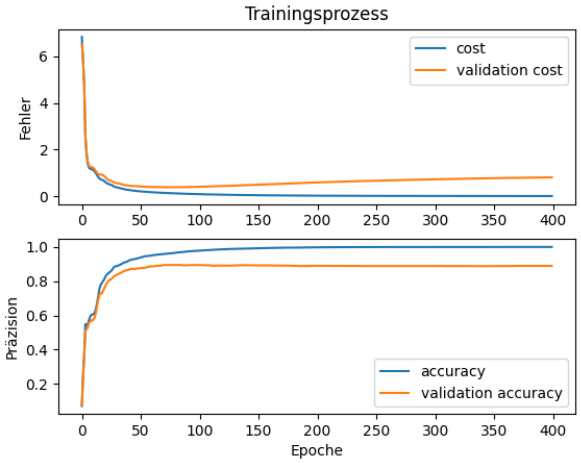

Verlauf des Fehlers

Abbildung 9: Resultate des Trainingsprozesses des Spiral-Datensatz mit ReLU Aktivierungsfunktion, CCE Fehlerfunktion und verschiedenen Optimierer. (Eigene Abbildung)

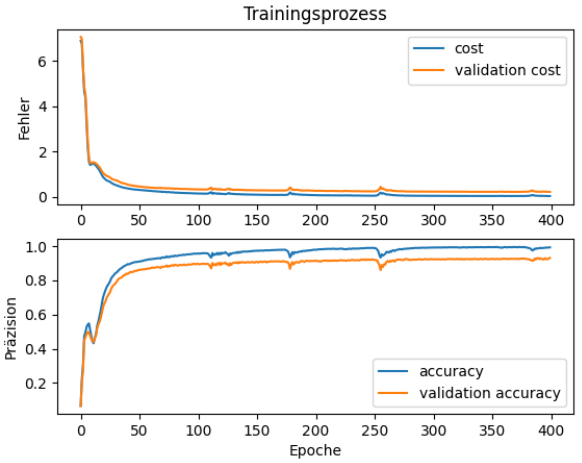
Analog zu den vier Trainingseinheiten mit unterschiedlichen Aktivierungsfunktionen wurden vier weitere Trainingsverfahren durchgeführt. Die Leistung der einzelnen Optimierer wurde dabei getestet und verglichen (Abbildung 9). Die Ergebnisse dieser Trainingssitzungen mit Netzwerken verschiedener Optimierer zeigen deutlich, dass jeder Optimierer in der Lage ist, die Genauigkeit und den Fehler des Netzwerks zu optimieren. Der effizienteste Optimierer für diesen Trainingsprozess ist der Adam-Optimierer. Nach weniger als zehn Epochen erreichte das Netzwerk mit dem Adam-Optimierer eine Präzision von fast 90 Prozent. Die Optimierer AdaGrad, RMSprop und SGD hatten eine deutlich langsamere Entwicklung der Präzision und des Fehlers, zeigten aber immer noch die gleiche Tendenz wie der Adam-Optimierer. Am Anfang des Trainingsvorgangs war ein Anstieg gefolgt von einem langsamen Abflachen zu erkennen. Dies ist auf die dynamische Lernrate zurückzuführen, die während des Trainingsprozesses abnimmt und somit zu immer kleineren Änderungen in den Anpassungen führt.

## 6.2 MNIST-Datensatz

Die Vorgehensweise bei der Analyse des MNIST-Datensatzes entspricht jener des Spiral-Datensatzes. Zunächst wurden zwei Trainingsdurchläufe durchgeführt, einer ohne und ein zweiter mit Regularisierer (Abbildung 10). Anschließend wurden die Aktivierungsfunktionen im Trainingsprozess getestet und verglichen. Im letzten Schritt wurden die vier verschiedenen Optimierer auf ihre Leistungsfähigkeit getestet und verglichen.



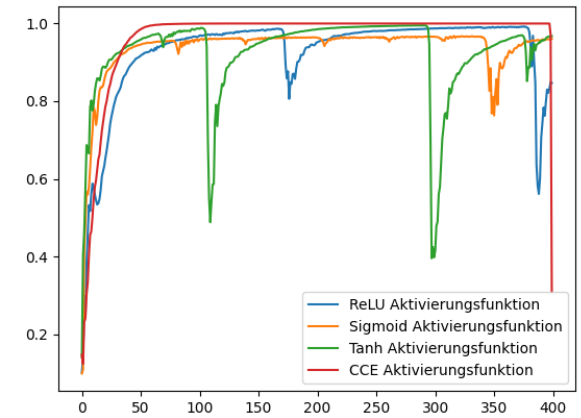

Trainingprozesses ohne Regularisierer



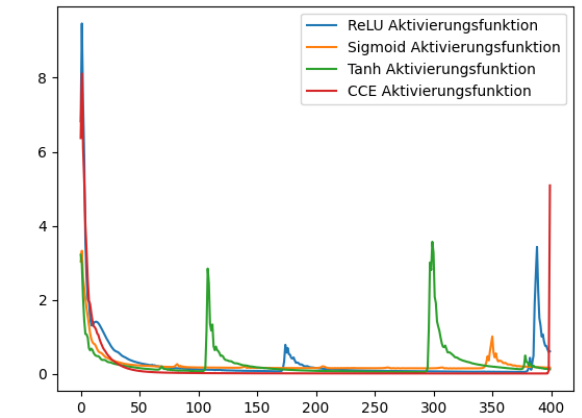

Trainingsprozess mit Regularisierer

Abbildung 10: Resultate des Trainingsprozesses des MNIST-Datensatz mit und ohne Regularisierer mit ReLU Aktivierungsfunktion, CCE Fehlerfunktion und Adam Optimierer. (Eigene Abbildung)

Die Genauigkeit am Ende des Trainingsprozesses dieses Datensatzes war überraschenderweise besser als die Genauigkeit des Netzwerks, das auf den Spiral-Datensatz trainiert wurde. Dies ist darauf zurückzuführen, dass die von diesem Netzwerk gezogenen Entscheidungsgrenzen deutlicher sind als im Spiral-Datensatz. Auch hier ist im Trainingslauf ohne Regularisierer eine Divergenz der Ergebnisse des Trainingsdatensatzes im Vergleich zu den Ergebnissen des Validierungsdatensatzes zu beobachten: Der Regularisierer konnte das Overfitting drastisch reduzieren und damit die Genauigkeit der Validierungsdaten näher an die der Trainingsdaten heranführen. Eine Verschlechterung der Gesamtleistung des Netzwerks ist nicht zu beobachten. Das bedeutet, dass der Regularisierer den Lernprozess nicht beeinträchtigt hat.



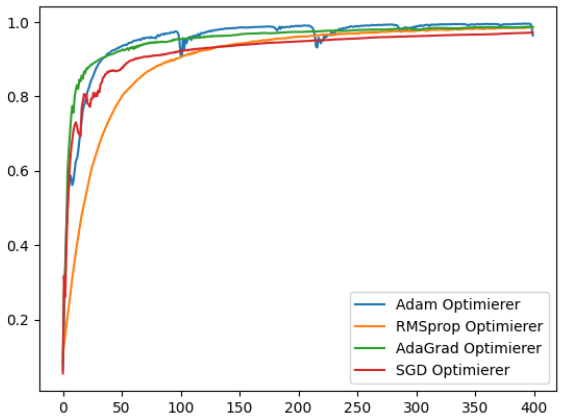

Verlauf der Präzision



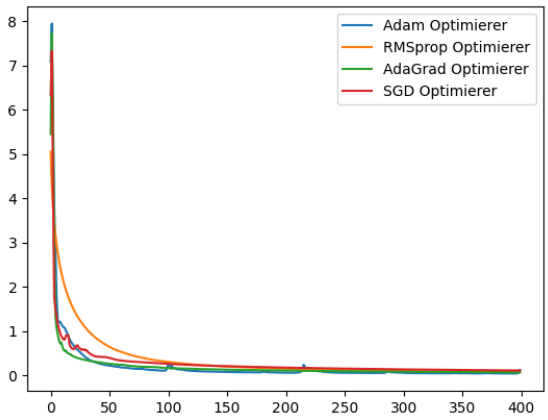

Verlauf der Fehlers

Abbildung 11: Resultate des Trainingsprozesses des MNIST-Datensatz mit verschiedenen Aktivierungsfunktionen, CCE Fehlerfunktion und Adam Optimierer. (Eigene Abbildung)

Wie beim vorherigen Datensatz wurden vier verschiedene Netzwerke trainiert und verglichen, welche sich nur in ihrer Aktivierungsfunktion unterscheiden (Abbildung 11). Im Vergleich zu den verschiedenen Netzwerken, die den Spiral-Datensatz trainierten, waren die Ergebnisse deutlich homogener. Alle Aktivierungsfunktionen erreichten beim Training eine maximale Genauigkeit von mehr als 90 Prozent. Die Aktivierungsfunktion, die im Durchschnitt die beste Genauigkeit erzielte, ist überraschenderweise die CCE-Aktivierungsfunktion. Es ist schwierig, dieses Ereignis zu interpretieren, da es für einen Menschen unmöglich ist, die Netzwerkfunktion und ihre Abhängigkeiten auf diesem Komplexitätsniveau zu verstehen. Die schlechtesten Ergebnisse erzielte das Netzwerk, das mit der Sigmoid-Aktivierungsfunktion lernte. Aber auch dieses erreichte eine maximale Genauigkeit von über 90 Prozent. Die verschiedenen Einbrüche in der Genauigkeit im Laufe des Trainings deuten darauf hin, dass mehrere lokale Minima durchquert wurden, um ein grösseres Minimum zu finden. Diese Eigenschaft wird bei Trainingsprozessen geschätzt, da sie in der Regel dazu führt, dass ein Netzwerk bessere Ergebnisse erzielt.




Verlauf der Präzision



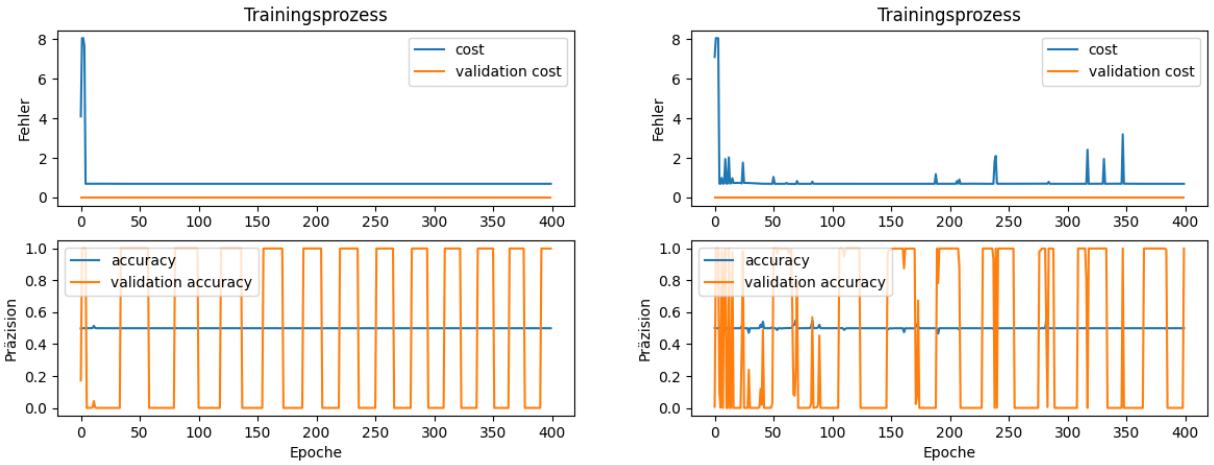

Verlauf des Fehlers

Abbildung 12: Resultate des Trainingsprozesses des MNIST-Datensatz mit ReLU Aktivierungsfunktion, CCE Fehlerfunktion und verschiedenen Optimierer. (Eigene Abbildung)

Die weiteren vier Trainingsprozesse wurden mit Netzwerken ausgeführt, welche sich in den Optimierern unterscheiden (Abbildung 12). Die verschiedenen Netzwerke zeigen sehr ähnliche Entwicklungen der Resultate während dem Trainingsprozess. Alle Netzwerke optimierten Präzision und Netzwerkfehler sehr schnell und erreichten eine Präzision von fast 100 Prozent. Das Netzwerk, welches sich dem RMSprop-Optimierer bedient, entwickelte sich langsamer. Feststellbar sind ausserdem kleine Einbrüche der Leistung während des Trainingsvorgangs mit dem Adam Optimierer. Der Grund dafür lässt sich erneut auf die Überquerung kleinerer lokaler Minima zurückführen. Bezüglich der Leistung der verschiedenen Optimierer beim Trainingsprozess mit dem MNIST-Datensatz lassen sich folgende Aussagen treffen: ähnlich wie bei der Leistungsverteilung der verschiedenen Aktivierungsfunktionen ist die Leistung der verschiedenen Optimierer relativ einheitlich. Der Optimierer Adam zeigt die höchste Präzision, dabei wird er zeitweise von SGD und AdaGrad überholt. Eine schwächere Leitung zeigt in diesem Durchgang der RMSprop-Optimierer.

## 6.3 Katzen- und Hunde-Datensatz

Der letzte der drei zu trainierenden Datensätze war der Katzen-Hund-Datensatz. Im Gegensatz zu seinen beiden Vorgängern wurden bei diesem Datensatz nur zwei verschiedene Trainingsverfahren analysiert. Dabei unterschieden sich die Trainingsdurchgänge wieder durch das Vorhandensein eines Regularisierers (Abbildung 13).




Trainingprozesses ohne Regularisierer

Trainingsprozess mit Regularisierer

Abbildung 13: Resultate des Trainingsprozesses des Katzen-Hunde-Datensatz mit und ohne Regularisierer mit ReLU Aktivierungsfunktion, CCE Fehlerfunktion und Adam Optimierer. (Eigene Abbildung)

Die Ergrebnisse der ersten beiden Trainingsverfahren überzeugen nicht. Die Ergebnisse zeigen, dass das Netzwerk nicht in der Lage war, Anpassungen vorzunehmen, die zu einer Erhöhung der Genauigkeit geführt hätten. Das betrachtete Muster zeigt, dass das künstliche neuronale Netzwerk scheinbar zufällige Vermutungen angestellt hat und entweder eine Genauigkeit von hundert Prozent oder null Prozent erreichte. Auch bei dem Trainingslauf, bei dem ein Regularisierer eingesetzt wurde, sind die Ergebnisse nicht besser. Der Datensatz zeigt die Grenzen dieses künstlichen neuronalen Netzwerks auf. Er ist ohne weitere Optimierung des Netzwerks nicht lösbar. Der Grund dafür ist, dass die Bilder, die dem Netzwerk zugeführt wurden, viele Störelemente enthalten. Dies ist auch der Grund, warum kein weiteres Training durchgeführt wurde. Das Netzwerk verfügt noch nicht über eine Methode zur Eliminierung dieser Störelemente und kann diesen Datensatz daher nicht lösen, unabhängig von der Aktivierungsfunktion und dem Optimierer.

## 6.4 Vergleich mit Keras

Nach der Bewertung des Netzwerks ist klar, dass das Netzwerk innerhalb eines bestimmten Rahmens funktioniert. Es stellt sich nun die Frage, wie leistungsfähig das künstliche neuronale Netzwerk im Vergleich zu einem bewährten state-of-the-art Netzwerk ist. Um festzustellen, ob das selbst erstellte Netzwerk mit seinen Konkurrenten auf dem Markt mithalten kann, wird das Netzwerk im folgenden Abschnitt mit der Bibliothek namens Keras verglichen. Keras ist eine bekannte Python-Bibliothek, die zur Erstellung künstlicher neuronaler Netzwerke verwendet wird. Es ist wichtig zu beachten, dass jedes mit Keras erstellte Netzwerk die gleiche Struktur wie die entsprechenden selbst erstellten Netzwerke aufweist und nicht auf zusätzliche Funktionalitäten zurückgreift, die theoretisch im riesigen Toolkit von Keras zur Verfügung stehen würden.

### 6.4.1 Spiral-Datensatz

Die Ergebnisse des ersten Trainingslaufs mit dem Spiral-Trainingsdatensatz, gezeigt durch Abbildung 14, sind erstaunlich. Die Ergebnisse zeigen, dass das auf Keras basierende künstliche neuronale Netzwerk den Algorithmus hinter dem einfachen Datensatz nicht gelernt hat. Die Genauigkeit der Vorhersage der Trainingsdaten wurde während des Trainings deutlich erhöht. Da aber die Genauigkeit unverändert blieb, wenn Validierungsdaten eingegeben wurden, bedeutet dies, dass das Netzwerk nur die Trainingsdaten gelernt und den zugrunde liegenden Algorithmus ignoriert hat. Dieser Fehler ist schwer zu erklären, da das Training mit anderen Datensätzen problemlos funktionierte.

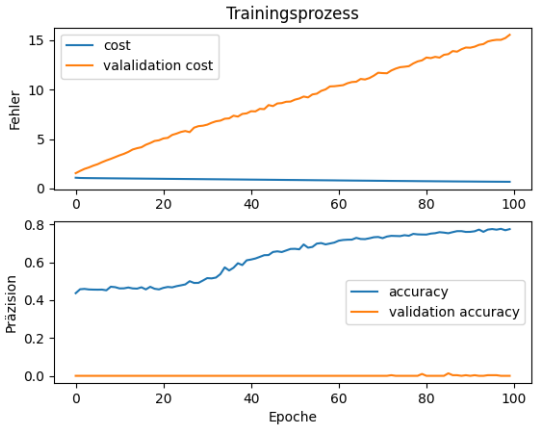



Abbildung 14: Leistung des Spiral-Datensatzes in einem mit Keras erstelltem Modell eines künstlichen neuronalen Netzwerks. (Eigene Abbildung)

### 6.4.2 MNIST-Datensatz

Einleuchtender sind die Ergebnisse eines Trainingslaufs mit dem MNIST-Datensatz. Die Ergebnisse, dargestellt in Abbildung 15, zeigen eine deutliche Steigerung der Vorhersagegenauigkeit während des Trainingsprozesses. Neben der zunehmenden Genauigkeit der Vorhersagen auf den Trainingsdaten ist auch eine anfängliche Zunahme der Genauigkeit auf den Validierungsdaten zu erkennen. Allerdings ist auch hier relativ früh eine Manifestation des Overfittings zu beobachten. Da in diesem Modell keine Methode zur Verhinderung von Overfitting verwendet wurde, ist ein Unterschied in der Genauigkeit von Trainings- und Validierungsdaten zu erwarten. Im Vergleich zum eigenen Netzwerkmodell dieser Arbeit betrieb das Netzwerk von Keras deutlich mehr Overfitting, war aber in der Lage, eine deutlich höhere Präzision bezogen auf die Trainingsdaten in weniger Epochen zu erreichen.

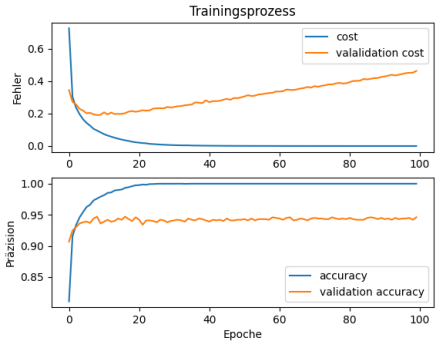



Abbildung 15: Leistung des MNIST-Datensatzes in einem mit Keras erstelltem Modell eines künstlichen neuronalen Netzwerks. (Eigene Abbildung)

### 6.4.3 Katzen und Hunde-Datensatz

Der Katz-Hund-Datensatz zeigt überraschende Ergebnisse (Abbildung 16). Der Datensatz, der von dem im Rahmen dieser Arbeit erstellten neuronalen Netzwerk nicht gelernt werden konnte, konnte in begrenztem Umfang von dem Netzwerk des Moduls Keras mit der gleichen Struktur gelernt werden. Die Genauigkeit in Bezug auf die Trainingsdaten verbesserte sich, als ob das Netzwerk einen Algorithmus lernen würde. Wurde das Netzwerk jedoch unbekannten Bildern von Hunden und Katzen ausgesetzt, variierte seine Präzision während des Trainings stark. Das Netzwerk war also in der Lage, die Trainingsdaten zu lernen, konnte aber keinen zugrunde liegenden Algorithmus ermitteln.

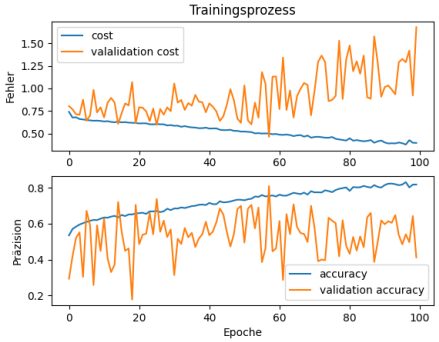



Abbildung 16: Leistung des Katzen-Hunde Datensatzes in einem mit Keras erstelltem Modell eines künstlichen neuronalen Netzwerks (Eigene Abbildung)

# 7 Fazit

Am Ende dieser Arbeit kann ich mit Stolz sagen, dass es mir gelungen ist, mein eigenes künstliches neuronales Netz zu programmieren. Obwohl das Netz nicht perfekt ist und wahrscheinlich nie perfekt sein kann, funktioniert es. Dies wird durch die Ergebnisse des Spiral-Datensatzes und des MNIST-Datensatzes bewiesen. Während der Zeit, in der ich an diesem Projekt gearbeitet habe, hatte ich viele Erfolge, aber auch viele Misserfolge. Im Grunde genommen hatte ich eine Menge Spass, vor allem bei der Erstellung des Programms. Während der Arbeit habe ich nicht nur eine Menge mathematischer Konzepte gelernt, sondern auch mein Verständnis über die Programmiersprache Python erweitert. Je tiefer ich jedoch in das Thema einstieg, desto mehr wurde mir klar, dass dieses Wissen nur die Spitze des Eisbergs ist. Das ist einerseits motivierend, da man sich der zahlreichen Möglichkeiten dieses Bereichs bewusst wird, andererseits aber auch manchmal etwas überwältigend. So endete die Recherche zu bestimmten Themen oftmals schnell in einer Sackgasse oder in einem viel zu komplexen Bericht eines Universitätsprofessors. Ich bedaure besonders, dass es mir aus Zeitmangel nicht möglich war, einen erfolgreichen Trainingsprozess für den Katzen-Hund-Datensatz durchzuführen. Generell gibt es noch viele Macken im Programm. In der aktuellen Version könnten sicherlich noch viele Funktionen sinnvoller gestaltet werden, um die Verständlichkeit des Programms zu verbessern.
Abschliessend kann ich sagen, dass ich froh bin, dass sich diese Arbeit nun dem Ende zuneigt. Obwohl ich viel Spass bei der Beschäftigung mit diesem Thema hatte, steckt auch viel Frust in dieser Arbeit und das Gefühl, diese Arbeit abgeschlossen zu haben, ist unbeschreiblich.

# Literatur


[1] Safar, Milad: Was ist generative KI und was kann sie?
https://www.industry-of-things.de/was-ist-generative-ki-und-was-kann-sie-a-8faf44a80c7de6711d3b05875722c122/
(Abruf 27.10.2022).

[2] OpenAI: Dall-E: Creating Images from Text.
https://openai.com/blog/dall-e
(Abruf 27.10.2022).

[3] Wikipedia: Deepfake
https://de.wikipedia.org/wiki/Deepfake.
(Abruf 27.10.2022).

[4] Wikipedia: Künstliche Intelligenz.
https://de.wikipedia.org/wiki/Künstliche_Intelligenz
(Abruf 27.10.2022).

[5] Wikipedia: Midjourney.
https://en.wikipedia.org/wiki/Midjourney
(Abruf 27.10.2022).

[6] Wikipedia: Klassifikationsverfahren.
https://de.wikipedia.org/wiki/Klassifikationsverfahren
(Abruf 27.10.2022).

[7] Wikipedia: Regressionsanalyse.
https://de.wikipedia.org/wiki/Regressionsanalyse
(Abruf 27.10.2022).

[8] Wikipedia: Skalarprodukt.
https://de.wikipedia.org/wiki/Skalarprodukt
(Abruf 27.10.2022).

[9] Wikipedia: Activation function.
https://en.wikipedia.org/wiki/Activation_function
(Abruf 27.10.2022).

[10] Wikipedia: Vanishing gradient problem.
https://en.wikipedia.org/wiki/Vanishing_gradient_problem
(Abruf 27.10.2022).

[11] Mithra Noel, Mathew; L, Arunkumar; Trivedi, Advait; Dutta, Praneet: Growing Cosine Unit: A Novel Oscillatory Activation Function That Can Speedup Training and Reduce Parameters in Convolutional Neural Networks.
https://ui.adsabs.harvard.edu/abs/2021arXiv210812943M/abstract
(Abruf 27.10.2022).

[12] Koech, Kiprono: Cross-Entropy Loss Function.
https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e
(Abruf 27.10.2022).

[13] Wikipedia: Loss function.
https://en.wikipedia.org/wiki/Loss_function
(Abruf 27.10.2022).

[14] Wikipedia: Stochastic gradient descent.
https://en.wikipedia.org/wiki/Stochastic_gradient_descent
(Abruf 27.10.2022).

[15] Wikipedia: Overfitting.
https://en.wikipedia.org/wiki/Overfitting
(Abruf 27.10.2022).

[16] Tyagi, Neelam: L2 vs L1 Regularization in Machine Learning.
https://www.analyticssteps.com/blogs/l2-and-l1-regularization-machine-learning
(Abruf 27.10.2022).

[17] Deeplake: MNIST.
https://datasets.activeloop.ai/docs/ml/datasets/mnist/ (Abruf 27.10.2022).

[18] Wikipedia: Künstliches Neuron.
https://de.wikipedia.org/wiki/Künstliches_Neuron
(Abruf 27.10.2022).

[19] Wikipedia: Künstliches neuronales Netz.
https://de.wikipedia.org/wiki/Künstliches_neuronales_Netz
(Abruf 27.10.2022).

[20] Patrikar, Sushant: Batch, Mini Batch  Stochastic Gradient Descent.
https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a
(Abruf 27.10.2022).

# Abbildungsverzeichnis