Evaluation der Anwendbarkeit von praktischen Beispielen zur Dissemination von Machine Learning-Verfahren durch E-Learning

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Tobias Braun

[redacted]

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr. Larissa Putzar

Zweitprüfer: Jan Ruhnke

Hamburg, 07.03.2022

Zusammenfassung

Es gibt unterschiedlichste Lern-Materialien zum Thema Machine-Learning. Frei verfügbare Online-Tutorials erreichen oft nicht den Informationsgehalt anderer Medien. In dieser Arbeit werden Live-Tutorials für das E-Learning erstellt, die die mathematischen Grundlagen mithilfe von Praxis-Beispielen vermitteln. Dazu werden die Inhalte der Tutorials für Vortragende gesammelt und aufbereitet. Es wird überprüft, in welchem Umfang Praxis-Beispiele in Live-Tutorials eingesetzt werden können. Die Evaluation hat ergeben, dass Praxis-Beispiele zur Vermittlung von einfachen neuronalen Klassifizierern und Regressions-Verfahren zeitlich und technisch eingesetzt werden können. Tutorials zu Algorithmen neuronaler Netze werden aufwändiger. Praxis-Beispiele müssen inhaltlich gekürzt werden und der Nutzen der Beispiele zur Vermittlung der Algorithmen verschwindet.

Abstract

A wide variety of learning materials on the subject of machine learning exists. Freely available online tutorials often do not reach the information content of other media. In this work, live tutorials for e-learning are developed, to impart the mathematical basics using practical examples. For this purpose, the contents of the tutorials are collected and processed for lecturers. It is checked to what extent practical examples can be used in live tutorials. The evaluation has shown that practical examples can be used to teach simple neural classifiers and regression methods in terms of time and technology. Tutorials on neural network algorithms are becoming time-consuming. The content of the practical examples must be shortened and the use of the examples to convey the algorithms disappears.

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung und Motivation

Maschinelles Lernen, kurz ML, wurde in den 1960er Jahren entwickelt. Somit ist ML keine Erfindung der letzten Jahre. Einfache und leicht verständliche Lernmaterialien sind allerdings immer noch selten. Für eine Tutorentätigkeit für den Kurs „Adaptive Systems and Artificial Intelligence“(ASaAI) des Studienganges Media Systems an der HAW Hamburg habe ich mich am Anfang des Jahres 2020 in das Thema Machine Learning eingearbeitet. Dazu habe ich Bücher und frei verfügbare Videos aus dem Web genutzt. Das Durcharbeiten von Büchern ist aufgrund der Schriftform und der Schriftsprache zeitaufwändig. Viele Videos sind hingegen zu oberflächlich, zeigen nur die Verwendung von Funktionen aus Bibliotheken und gehen nicht auf die Algorithmen dahinter ins Detail ein. Je besser der Algorithmus wird, desto komplizierter ist er auch. Funktionalere Algorithmen besitzen immer mehr veränderbare Eigenschaften, Hyperparameter zum Einstellen der Modelle. Wer den Grundalgorithmus nicht verstanden hat, kann die Werte der Eigenschaften nur ausprobieren. Damit ich als Tutor die Algorithmen auch erklären und Fragen beantworten kann, sollte ich wissen, wie sie funktionieren. Ich habe hauptsächlich das Buch „Machine Learning mit Python und Scikit-learn und TensorFlow“(Raschka & Mirjalili 2018) genutzt und zusätzlich Videos angesehen, um mir dieses Wissen anzueignen. In dieser Bachelorarbeit will ich anderen Tutoren und ML-Interessierte mein Wissen über die Algorithmen und Erkenntnisse der Tutorials in kompakter Form weitergeben. Die gesammelten Grundlagen helfen jeden Interessierten in das Thema ML schnell einzusteigen, ohne aus unterschiedlichen Quellen die Inhalte suchen zu müssen.

Mit dem gesammeltem Wissen habe ich nach Absprache mit der Professorin Larissa Putzar Tutorials zu ML-Algorithmen erstellt und im Kurs ASaAI vorgetragen. Die Tutorials enthalten Programmierbeispiele der Algorithmen. Sie sollen dabei helfen den Algorithmus zu verstehen. So werden die mathematischen Operationen einer Funktion im Code auf mehrere Zeilen aufgespalten. Allerdings wird der Code immer länger, je komplexer der Algorithmus wird. In dieser Arbeit wurde überprüft, in welchem Rahmen und Umfang es möglich ist, Programmierbeispiele in Tutorials und Lernvideos zu nutzen. Die erstellten Tutorials können zum eigenen Lernen und die Erkenntnisse aus dem Erstellen der Tutorials zur Vermittlung an andere genutzt werden.

Das Ziel der Arbeit ist eine kompakte Sammlung der Machine Learning-Grundlagen, im Speziellen der Grundlagen von neuronalen Netzen. Diese Sammlung soll anderen

Tutoren das benötigte Wissen zur Verfügung stellen. Dieser Teil wird in der Arbeit stark beleuchtet. Mit diesem Wissen werden Tutorials zur Dissemination (Vermittlung/Verbreitung) von ML-Verfahren erstellt. Programmierbeispiele in den Tutorials sollen das Verstehen der Algorithmen einfacher machen. Daher werden in dieser Arbeit Tutorials analysiert und geprüft inwiefern Programmierbeispiele effektiv im E-Learning eingesetzt werden können. In dieser Arbeit wird davon ausgegangen, dass Programmierbeispiele einen positiven Effekt auf den Lernerfolg haben. Diese Behauptung ist die Grundlage dieser Arbeit.

1.2 Related Works und Abgrenzung

Es gibt viele verschiedene Lernmaterialien zum Thema Machine Leraning (ML). Verschiedene Anbieter nutzen verschiedene Medien und Techniken zum Vermitteln der Inhalte. Erhältlich sind Bücher, Online-Tutorials, Blogs und Lehrvideos. Vereinzelt stellen Lehrkräfte ihre Materialien online öffentlich zur Verfügung. Amazon bietet mehr als 7.000 „Fachbücher für Informatik“zum Thema ML an (Geprüft am 16.02.2022.). Die Bücher gehen unterschiedlich detailreich auf die Algorithmen ein. Einige beleuchten die mathematischen Funktionen der Algorithmen (Alpaydin 2019), andere nutzten darüber hinaus zur Verdeutlichung die Programmierung der Algorithmen (Raschka & Mirjalili 2018). ähnlich divers sind Online-Tutorials. Frei verfügbare Online-Tutorials in schriftlicher Form erklären mathematische Funktionen und enthalten Programmierbeispiele zu den Algorithmen (Tutorials Point). Online-Tutorials im Videoformat sind hingegen kostenpflichtig (Tutorials Point 2021). Weiterhin werden über den Websites von Programmier-Bibliotheken schriftliche Tutorials angeboten (Meta 2022), (PyTorch 2021), (TensorFlow 2021). Diese beschäftigen sich jedoch ausschließlich mit ihren eigenen Bibliotheken. Blogs beschäftigen sich ebenfalls oft mit der Handhabung der importierten Bibliotheken, und gehen nicht auf die Grundlagen ein (Brownlee 2021). 214.000.000 Videos zu diesem Thema gibt Google aus (Geprüft am 16.02.2022.). Diese sind wieder unterschiedlich detailliert, wobei die oberflächliche Betrachtung und die Programmierung im Vordergrund vieler Videos steht. Nur wenige Lehrkräfte stelle ihre Materialien der Öffentlichkeit zu Verfügung (Loviscach 2022).

Gegenüber Büchern besitzen Videos den großen Vorteil, dass das Geschehen verbal erläutert wird. Ebenso kann im Video genau wie in einer Präsenz-Veranstaltung der Inhalt aktiv erarbeitet werden. Die Vermeidung der Schriftsprache ist im Allgemeinen vorteilhaft. Besonders lese-schwache Personen werden somit nicht mehr vom vermittelten Wissen ausgeschlossen. Der Nachteil von Videos ist die zeitliche Begrenzung auf wenige Minuten bis zu einer Stunde. Dadurch können die Videos nicht so viele Inhalte fassen wie zum Beispiel Bücher. Mehrere und umfangreiche Inhalte müssen auf mehrere Videos aufgespalten werden. Der Informationsgehalt der Videos bleibt dennoch meist hinter den von Büchern. So wird in einem Video zur mathematischen Funktion nicht zusätzlich ein Programmierbeispiel genutzt. Dabei können

Programmierbeispiele das Verstehen der Algorithmen vereinfachen, indem sie sie auf eine andere Art darstellen: Die mathematische Formel wird im Beispiel als Code dargestellt. Zum Anderen können in Videos große Zeiträume rausgeschnitten werden und die Quintessenz dem Zuschauer kompakt präsentiert werden. Dies ist in einer Live-Veranstaltung wie einer Präsenz-Vorlesung nicht möglich.

Der große Vorteil von Büchern ist, dass sie mehrere komplexe Verfahren erläutern können. Die Informationen kommen alle aus einer Quelle. Sind Videos und Online-Tutorials zu mehreren Verfahren gewünscht, müssen sie einzeln aus meist mehreren Quellen gesucht und gesammelt werden. Dies erschwert das Online-Lernen.

In dieser Arbeit sollen die Grundlagen zu bestimmten ML-Algorithmen gesammelt und aus ihnen Live-Tutorials erstellt werden. Die Tutorials sollen live und online veranstaltet werden können. Die Vorteile der unterschiedlichen Medien sollen vereint werden. So sollen sie nicht nur die mathematischen Grundlagen vermitteln, sondern auch Programmierbeispiele beinhalten. Die Inhalte sollen in den Tutorials live erarbeitet werden.

1.3 Aufbau der Arbeit

Die weitere Arbeit ist in drei Kapitel unterteilt. In Kapitel 2 werden theoretische Grundlagen des maschinellen Lernens erklärt und vertieft. Angefangen wird bei einfachen neuronalen Klassifizierern. Danach wird auf komplexe neuronale Netze eingegangen. In Kapitel 3 geht es um die praktische Anwendung der Algorithmen des maschinellen Lernens. Dabei werden Praxisbeispiele eingeführt und auf ihre Anwendbarkeit in Tutorials überprüft, indem die benötigte Laufzeit der Algorithmen gemessen wird. Ebenso wird in diesem Kapitel auf Möglichkeiten der Optimierung der Praxisbeispiele eingegangen, um sie in einem Live-Tutorial nutzen zu können. Das letzte Kapitel fasst im Anschluss die gewonnen Erkenntnisse zusammen und stellt klar, auf welche Weise Praxisbeispiele im Rahmen eines Tutorials eingesetzt werden können.

2 Maschinelles Lernen

Das Verständnis der Algorithmen ist die Voraussetzung, um selbst Tutorials über die Algorithmen abhalten zu können. Deshalb werden in diesem Kapitel die in den Tutorials genutzten Algorithmen vorgestellt.

Da es viele unterschiedliche ML-Algorithmen gibt, kann im Rahmen dieser Arbeit nur auf einige dieser Algorithmen eingegangen werden. Zur Abgrenzung zu anderen ML-Algorithmen folgt daher dieser Einleitung eine grobe Übersicht über ML-Algorithmen 2.1. Im darauf folgenden Abschnitt 2.2 wird auf die Algorithmen selbst eingegangen. Die vorgestellten Algorithmen werden nicht wie üblich nach ihren Aufgabenstellungen in Klassifikation und Regression unterteilt. Stattdessen wurden die Algorithmen neu sortiert, sodass sie logisch aufeinander aufbauen. Die Reihenfolge orientiert sich an der Komplexität der Algorithmen. Es wird mit einem einfachen Algorithmus angefangen, um diesen in mehreren Schritten zu komplexeren Algorithmen zu erweitern. Damit soll ersichtlich werden, wie die unterschiedlichen Algorithmen aufeinander aufbauen und welcher Algorithmus eine Erweiterung eines vorangegangenen ist. Das Ziel ist ein Überblick der Algorithmen von einfachen neuronalen Klassifizierern bis hin zu komplexen tiefen neuronalen Netzen im Abschnitt 2.3.

Dieses Kapitel ist in drei Kategorien unterteilt:

2.1 Grundbegriff maschinelles Lernen

2.2 Künstliche Neuronen

2.3 Künstliche neuronale Netze

Zum Verständnis der Algorithmen gehört auch das Verständnis ihrer mathematischen Funktionsweise. Dies ist wichtig, um tiefe neuronale Netze modellieren und einstellen zu können. Zudem haben mathematische Funktionen den Vorteil, dass sie leicht zu programmieren sind. Daher wird in dieser Arbeit stark auf die mathematischen Funktionen der Algorithmen eingegangen. Größtenteils wurden die Funktionen aus dem Buch „Machine Learning mit Python und Scikit-learn und Tensorflow“(Raschka & Mirjalili 2018) entnommen. Die verwendeten Rechenbeispiele wurden im Rahmen dieser Arbeit entwickelt.

2.1 Grundbegriff maschinelles Lernen

Maschinelles Lernen ist ein Teilbereich der Künstlichen Intelligenz. Bei dem maschinellen Lernen werden mathematische Verfahren genutzt um bestehende Daten zu

analysieren. Das Analyseergebnis wird verwendet, um neue Erkenntnisse aus den Daten zu erhalten oder um neue Daten schneller mit dem Computer verarbeiten zu können. Die Verfahren benötigen Daten zum maschinellen Lernen. Bei der Analyse werden Eigenschaften der Datenelemente miteinander verglichen. Die Daten müssen geeignet für die Aufgabe sein und aufbereitet werden. Sind wenige und unausgewogene Daten vorhanden, kann die Analyse ungenau sein. (Alpaydin 2019: 1-4)

Durch den Vergleich können die Elemente kategorisiert werden (Klassifikation und Clustering), Vorhersagen über neue Daten gemacht werden (Regression) und Anomalien in den Daten erkannt werden (Anomaly Detection).

Es gibt viele verschiedene ML-Algorithmen. Sie können in unterschiedliche Kategorien eingeteilt werden. Es folgen drei Arten zur Unterscheidung von ML-Algorithmen:

2.1.1 Trennung nach Aufgaben

2.1.2 Überwachte, unüberwachte und bestärkende Lernmethoden

2.1.3 Parametrische und nichtparametrische Methoden

Um die in dieser Arbeit vorgestellten Algorithmen in die Reihe von ML-Algorithmen einordnen zu können, müssen die Kategorien bekannt sein. Ebenfalls dient dieser Abschnitt dazu, zu zeigen welche anderen Verfahren es abseits der vorgestellten Algorithmen gibt. In den folgenden Kapiteln wird nur auf neuronale Modelle und neuronale Netze eingegangen. Für diese Algorithmen werden in Kapitel 3 Tutorials erstellt.

2.1.1 Trennung nach Aufgaben

Jeder ML-Algorithmus dient einem bestimmten Zweck. Dadurch können sie nach ihren Aufgaben aufgeteilt werden. Mit ML-Algorithmen lassen sich folgende Aufgaben umsetzten: Klassifizierung bekannter Gruppen, Erkennen unbekannter Gruppen (Clustering), Trenderkennung (Regression), und Anomalieerkennung (Anomaly detection).

Klassifizierung

Sind zu den Datenobjekten schon Gruppenzugehörigkeiten bekannt, können mit diesen Daten Modelle trainiert werden. Die trainierten Modelle können neue Datenobjekte den vorher bekannten Klassen zuordnen. Ein Beispiel ist die Klassifizierung von Neukunden in vorher bekannte Kundengruppen. Dadurch kann zum Beispiel der Service für die Neukunden an den Kundengruppen ausgerichtet werden. Die automatische Erkennung von Bildinhalten gehört auch zur Klassifizierung.

Klassifizierungs-Algorithmen sind neuronale Klassifizierer wie das Perzeptron und die logistische Regression, neuronale Netze, k-nearest-neighbor (kNN) Support-Vector-Machines (SVM) und Entscheidungsbäume. Bei neuronalen Algorithmen wird zu jedem Merkmal (Feature) der Datenobjekte ihre Relevanz für eine Klasse berechnet. Die Features werden gewichtet. Aufgrund der Gewichtung und den Werten der Objekte,

können sie einer Klasse zugeordnet werden. Bei kNN wird die Klasse des neuen Datenobjektes anhand ihrer Entfernung zu den vorhandenen Datenobjekten bestimmt. Objekte gehören zu einer Gruppe oder Klasse, wenn sie einen möglichst geringen Abstand zueinander besitzen. Unterscheiden sich die Werte der Features stark, ist der Abstand groß und sie gehören einer anderen Klassen an. SVMs suchen eine Linie, die die Klassen trennt. Dabei soll die gesuchte Linie den möglichst breitesten Rand zu den Objekten der unterschiedlichen Klassen besitzen. Entscheidungsbäume lernen Schwellenwerte für die relevantesten Features. Dabei werden die Schwellenwerte der Features hierarchisch nach ihrer Relevanz notiert. (Raschka & Mirjalili 2018: 81, 119)(Alpaydin 2019: 361, 217)

Clustering

Wenn keine Klassenzugehörigkeiten vorher bekannt sind, können Clustering-Verfahren eingesetzt werden. Sie erkennen automatisch Strukturen in den Daten. Die Clustering-Verfahren berechnen die Ähnlichkeit der Objekte zueinander. Anhand ihrer Ähnlichkeit werden die Daten in Gruppen aufteilen. So können zum Beispiel vorher nicht bekannte Gruppen von Kunden erkannt werden, um den Service besser auf die Gruppen zuzuschneiden. Diese Gruppen können wiederum für die Klassifikation genutzt werden. Ein Clustering-Verfahren ist der k-means-Algorithmus. Zur Erkennung der Gruppen werden Punkte gesucht, zu denen die Objekte den geringsten Abstand besitzen. Also wird der Median der Datenobjekte gesucht, zu dem die Datenobjekte den geringsten Abstand besitzen. (Raschka & Mirjalili 2018: 353)

Regression

Mit der Regression können Trends in Daten erkannt werden. Dazu werden die Datenobjekte zu einer Formel vereinfacht. Die Formel passt sich den Datenobjekten automatisch an. An ihr kann ein Trend in den Daten abgelesen werden. Beispielweise können durch die Regressions-Analyse Preissteigerungen über einen Zeitraum erkannt werden. Ein Regressions-Verfahren ist die lineare Regression. (Alpaydin 2019: 10)

Anomaly detection

Die Anomaly detection dient dem Aufspüren von aus den Daten heraus stechenden Objekten. Diese können auf Fehler in den Daten zurückzuführen sein. So können die fehlerhaften Daten behandelt oder entfernt werden, bevor einer der vorherigen Verfahren angewandt wird. Bei dem Clustering könnten fehlerhafte Anomalien die Erkennung der Gruppen negativ beeinflussen. Zur Erkennung von Anomalien können unterschiedliche Algorithmen eingesetzt werden. So z.B. kNN oder ein Clustering-Verfahren. (Alpaydin 2019: 9)

2.1.2 Überwachte, unüberwachte und bestärkende Lernmethoden

ML-Algorithmen werden in überwachte, unüberwachte und bestärkende Lernmethoden unterschieden. Bei überwachten Lernmethoden sind zu den Datenobjekten mit ihren Features jeweils eine Gruppenzugehörigkeit, eine Klasse bekannt. Die Daten sind markiert, gelabelt. Mit den gelabelten Daten kann ein ML-Modell angepasst und trainiert werden. Gelabelte Daten werden für die Klassifikation benötigt. Somit gehört die Klassifikation zu den überwachten Lernmethoden.

Sind hingegen keine gelabelten Daten vorhanden, können Algorithmen des unüberwachten Lernens angewendet werden. Clustering-Verfahren sind typische unüberwachte Lernmethoden (Alpaydin 2019: 12). Ebenso kann mit unüberwachten Lernmethoden die Anzahl der Features reduziert werden (Komprimierung/Dimensionsreduktion) (Raschka & Mirjalili 2018: 32). Mit beiden Lernmethoden können Anomalien in den Daten erkannt werden und eine Regressions-Analyse durchgeführt werden.

Eine weitere Lernmethode ist das Bestärkende Lernen. Bestärkendes Lernen kann genutzt werden, um Roboter bzw. Agenten bestimmte Aktionen anzutrainieren (Alpaydin 2019: 14).

2.1.3 Parametrische und nichtparametrische Methoden

ML-Algorithen können zwischen parametrische und nichtparametrische Methoden unterschieden werden. Bei parametrischen Methoden werden Parameter gesucht, die die Datenelemente beschreiben. Diese Parameter können genutzt werden, um die Daten zu klassifizieren oder Vorhersagen aus den Daten abzuleiten. Die Parameter können als Funktion dargestellt werden. Bevor das Modell an die Daten angepasst wird, werden die Parameter und ihre Anzahl festgelegt. Die Anzahl kann nicht verändert werden, ohne das Modell erneut an die Daten anzupassen. Zu den parametrischen Methoden gehören zum Beispiel neuronale Klassifizierer und Regressionsmodelle, sowie neuronale Netze und Support Vector Machines (SVM).

Nichtparametrische Methoden benötigen keine Festlegung der Parameteranzahl. Dem Modell werden alle verfügbaren Parameter übergeben. Zur Vorhersage werden die Werte der Features selbst genutzt. Das Modell kann jederzeit neue Features berücksichtigen und benötigt keine Festlegung der Parameteranzahl. Zu den nichtparametrischen Methoden gehören zum Beispiel kNN, K-Means und Entscheidungsbäume (Alpaydin 2019: 189, 217)(Aunkofer 2018).

2.2 Künstliche Neuronen

In diesem Abschnitt wird auf einfache neuronale Klassifizierer und Regressions-Verfahren eingegangen. Das Verständnis der Funktionsweise von neuronalen Einheiten ist Voraussetzung, um komplexe Modelle wie ein neuronales Netz zu verstehen. Da in Kapitel 3 die Algorithmen programmiert werden sollen, muss auf die mathematische Funktionsweise der Algorithmen eingegangen werden. In dieser Arbeit wird darauf geach-

tet die mathematischen Funktionen zu erklären und auf vorangegangene Funktionen aufzubauen. Durch das Aufeinanderaufbauen der Algorithmen kommt es in dieser Arbeit zu der Eingliederung der linearen Regression in die Reihe der Klassifizierungs-Algorithmen Perzeptron und Adaline. Dies ist insofern unüblich, da die lineare Regression ein Regressions-Verfahren darstellt. Somit wird in dieser Folge der Algorithmen ein Regressions-Verfahren als Zwischenschritt zwischen Klassifizierungs-Algorithmen genutzt.

Die vorgestellten Algorithmen sind wie folgt:

2.2.1 McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

2.2.2 Lineare Regression

2.2.3 Adaline

2.2.4 Logistische Regression

Künstliche Neuronen können zur Klassifizierung oder zur Regressions-Analyse verwendet werden. Bei der Klassifizierung wird ein Objekt an ein trainiertes Neuronen-Modell übergeben und das Neuronen-Modell gibt eine Klassenbezeichnung aus. Bei der Regressions-Analyse handelt es sich um eine Trend-Erkennung aus dem Datensatz. In beiden Fällen muss das Neuronen-Modell mit Daten trainiert werden. Die Trainingsdaten benötigen Klassenzugehörigkeiten zu den Objekten. Die Daten müssen gelabelt sein. Durch die benötigten Klassen der Trainingsdaten gehören Neuronen-Modelle zu der Klasse des überwachten Lernens.

2.2.1 Das McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

Der einfachste neuronale Algorithmus ist das von Warren McCulloch und Walter Pitts 1943 veröffentlichte Perzeptron-Modell. Es ähnelt der Funktionsweise von Neuronen im menschlichen Gehirn (McCulloch & Pitts 1943). Mehrere Eingabesignale laufen in einem Zellkern zusammen. Wird ein bestimmter Schwellenwert überschritten, erzeugt die Zelle ein Ausgabesignal.

Mit dem McCulloch-Pitts-Neuron (MCP-Neuron) ist es möglich die Klassenzugehörigkeit eines Objektes zwischen zwei Klassen vorherzusagen (binäre Klassifizierung). Wenn das Neuron feuert, gehört das Objekt zur ersten Klasse. Feuert das Neuron nicht, gehört das Objekt im Umkehrschluss zur Klasse Zwei.

Damit das Neuron aus mehreren Eingabesignalen die korrekte Klasse erkennt, müssen die verschiedenen Eingabesignale, die Merkmale der Objekte, nach ihrer Aussagekräftigkeit für das Klassifizierungsergebnis skaliert oder gewichtet werden. Dafür wird jedes Eingabesignal mit einem sogenannten Gewichtungskoeffizienten, Gewichtungen multipliziert. Danach werden alle Produkte summiert und die ergebende Summe mit dem Schwellenwert verglichen (Abbildung 2.1).

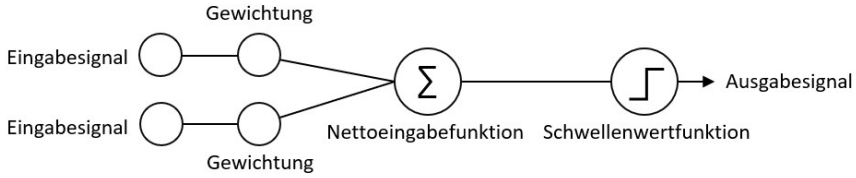
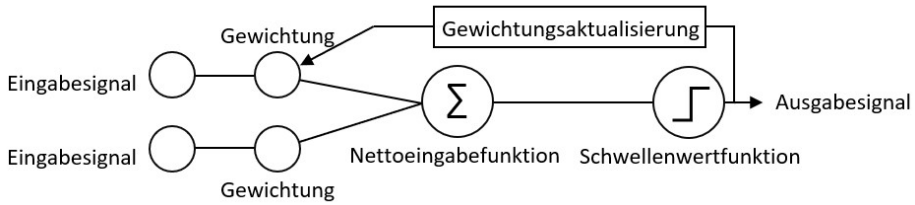


Abbildung 2.1: Schematische Darstellung eines MCP-Neurons.

Frank Rosenblatt veröffentlichte 1957 eine Perzeptron-Lernregel für das MCP-Neuronenmodell (Rosenblatt 1957). Diese Lernregel kann mit vorklassifizierten Trainingsdaten automatisch die besten Gewichtungskoeffizienten für die einzelnen Eingabesignale finden (Abbildung 2.2).



Angelehnt an: (Raschka & Mirjalili 2018: 46) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch02/images/02_04.png

Abbildung 2.2: Schematische Darstellung eines Perzeptron-Modells nach Rosenblatt.

Die mathematische Funktionsweise des Perceptrons

Da in Kapitel 3.2.4 das Perceptron im Tutorial programmiert werden soll, muss die mathematische Funktionsweise des Perceptrons verstanden sein. Daher wird nun auf die mathematischen Formeln eingegangen.

Jeder Eingabewert x besitzt einen Gewichtungskoeffizienten w. Die Summe aller Produkte der Eingabewerte mit ihren Gewichtungen ist die Nettoeingabe z (Funktion 2.1).

$$z = w_1 x_1 + ... + w_m x_m \tag{2.1}$$

Die Nettoeingabe z kann auch als Summe dargestellt werden (Funktion 2.2).

$$z = \sum_{j=1}^{m} x_j w_j \tag{2.2}$$

Ebenfalls kann die Nettoeingabe z als Skalarprodukt eines Eingabevektors z mit einem transponierten Gewichtungsvektor w dargestellt werden (Funktion 2.3).

$$\vec{w} = \begin{pmatrix} w_1 \\ ... \\ w_m \end{pmatrix}, \vec{x} = \begin{pmatrix} x_1 \\ ... \\ x_m \end{pmatrix}, z = \vec{w}^T \vec{x} \tag{2.3}$$

Nun wird die Nettoeingabe z einer Aktivierungsfunktion ϕ übergeben. Wenn z einen bestimmten Schwellenwert θ übersteigt, gibt sie eine Klassenbezeichnung aus. Es ist eine einfache Sprungfunktion (Funktion 2.4).

$$\phi(z) = \begin{cases} Klasse\ 1\ wenn\ z \geq \theta \\ Klasse\ 2\ wenn\ nicht \end{cases} \tag{2.4}$$

Diese Sprungfunktion setzt allerdings voraus, dass die beiden Klassen, zwischen denen unterschieden werden soll, grafisch linear trennbar sind (Abbildung 2.3). Linear nicht trennbare Klassen können nicht korrekt klassifiziert werden (Abbildung 2.4).

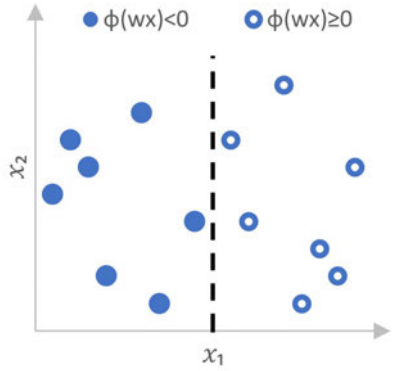


Abbildung 2.3: Linear trennbare Objekte

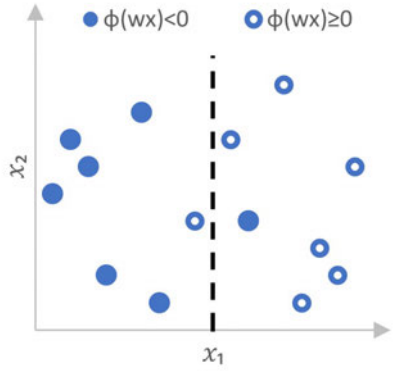


Abbildung 2.4: Linear nicht trennbare Objekte

Biaseinheit

Liegt eine Entscheidungsgrenze nicht bei dem Schwellenwert 0, kann eine weitere Einheit der Nettoeingabe hinzugerechnet werden, um die Nettoeingabe und die resultierende Entscheidungsgrenze so weit zu verschieben, dass die Entscheidungsgrenze wieder auf dem Schwellenwert 0 liegt. Diese Einheit wird Bias-Einheit genannt und ist der negativer Schwellenwert als Gewichtungskoeffizient w_0 multipliziert mit 1 als Eingabewert x_0. Damit ergibt sich die Nettoeingabe wie in Funktion 2.5.

$$z = w_0 1 + w_1 x_1 + ... + w_m x_m \tag{2.5}$$

Die Nettoeingabe z mit Biaseinheit w_0 und x_0 kann ebenfalls als Summe dargestellt werden (Funktion 2.6).

$$z = \sum_{j=0}^{m} x_j w_j \tag{2.6}$$

Bei der Darstellung als Skalarprodukt zweier Vektoren ändert sich nichts (Funktion 2.7).

$$\vec{w} = \begin{pmatrix} w_0 \\ ... \\ w_m \end{pmatrix}, \vec{x} = \begin{pmatrix} 1 \\ ... \\ x_m \end{pmatrix}, z = \vec{w}^T \vec{x} \tag{2.7}$$

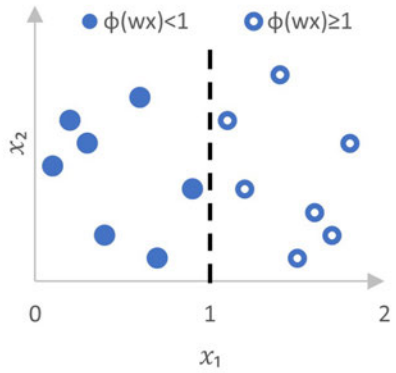


Abbildung 2.5: Modell mit Entscheidungsgrenze bei $\theta = 1$.

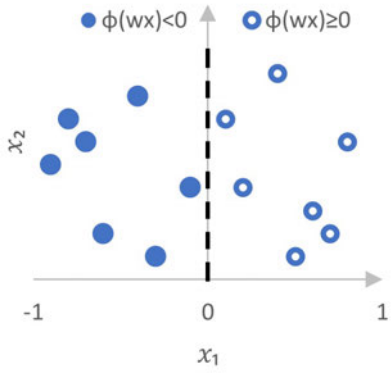


Abbildung 2.6: Modell mit Bias $w_0 = -1$ und neuer Entscheidungsgrenze bei $\theta = 0$.

Die Perzeptron-Lernregel

Die Funktionsweise der Perzeptron-Regel lässt sich wie folgt zusammenfassen:

1. Die Gewichtungen werden mit kleinen von 0 unterschiedlichen zufälligen Werten initialisiert.
2. Bei allen Elementen der Trainingsdaten werden die nächsten Schritte durchgeführt, bevor das nächste Element aus der Trainingsdatenmenge behandelt wird:
   a) Die Klasse des Elements wird vorhergesagt.
   b) Alle Gewichtungen werden anhand der vorhergesagten Klasse aktualisiert.
3. Schritt 2. wird wiederholt, bis die Vorhersage der Klassen genau genug ist.

Das Aktualisieren der Gewichtungen aufgrund der Vorhersagen der gesamten Trainingsdaten wird wiederholt. Ein Durchlauf durch die Trainingsdaten wird Epoche genannt. Die Gewichtungen werden mit einen bestimmten Wert aktualisiert. Dieser Wert wird mit der Perzeptron-Lernregel berechnet (Funktion 2.8). Dabei ist Δw_j der Wert der Gewichtungsänderung, η die Lernrate, $y^{(i)}$ die tatsächliche Klasse des i-ten Trainingsobjektes und $\hat{y}^{(i)}$ die vorhergesagte Klasse. Die Lernrate ist ein konstanter Wert zwischen 0 und 1 zur Regulierung der Anpassungsgeschwindigkeit. Je kleiner die Lernrate ist, desto langsamer nähern sich die Gewichtungen ihrer optimalen Werte.

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)} \tag{2.8}$$

Bei den Klassenbezeichnungen 1 und -1 ergibt der Klammerausdruck in Funktion 2.8 bei korrekter Vorhersage immer 0. Also werden die Gewichtungen bei einer richtigen Vorhersage mit 0 mal $x_j^{(i)}$ nicht verändert. Wird eine falsche Klasse vorhergesagt, ergibt der Klammerausdruck immer 2. Dann werden die Gewichtungen mit dem doppelten des Wertes von $x_j^{(i)}$ aktualisiert.

2.2.2 Die lineare Regression

Die lineare Regression kann als modifiziertes Perzeptron angesehen werden. Für die lineare Regression wird die Sprungfunktion des Perceptrons mit einer Linearfunktion ausgetauscht. Dies ist in Abbildung 2.7 dargestellt. Durch die lineare Funktion werden keine Klassen mehr vorhergesagt, sondern stetige Werte der linearen Funktion. Dies kann genutzt werden, um Daten zu verallgemeinern oder Vorhersagen zu treffen. Zum Beispiel kann die lineare Regression zur Vorhersage des Kaufverhaltens von Kunden genutzt werden oder zur einfachen Vorhersage von Aktienkursen. Grafisch kann die lineare Regression als Linie aufgefasst werden, die sich der Punktewolke der Testdatenmenge annähert (Abbildung 2.8). Mit den stetigen Werten gibt es keine Entscheidungsgrenze zwischen zwei Klassen sondern eine Trendlinie der Testdaten.

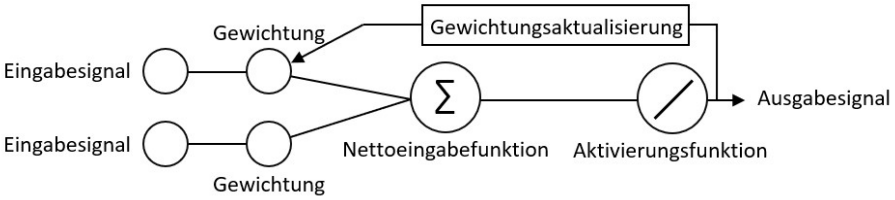


Abbildung 2.7: Schematische Darstellung einer multiplen linearen Regression.

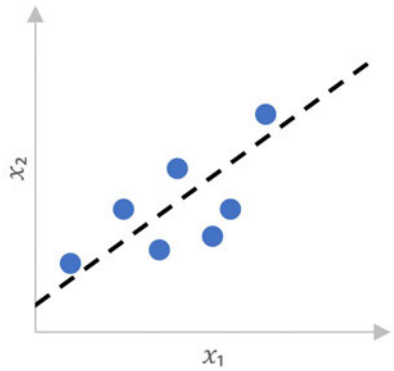


Abbildung 2.8: Lineare Annäherung an eine Punktewolke bei der linearen Regression.

Ebenso können auch durch die lineare Funktion die Ergebnisse der Nettoeingabefunktion selbst benutzt werden, um sie mit der tatsächlichen Klasse zu vergleichen. Dadurch wird der Wert der Gewichtungsaktualisierung abhängig von der Größe des Fehlers. Je größer der Fehler zwischen Nettoeingabeergebnis und tatsächlicher Klasse ist, desto größer wird auch der Wert der Gewichtungsaktualisierung und umgekehrt. Allerdings wird bei der einfachen linearen Regression nur ein Eingabesignal (Einflussgröße oder Regressor) benutzt. Sollen mehrere Eingabesignale benutzt werden, wird die multiple lineare Regression angewendet.

Straffunktion mit dem Verfahren der kleinsten Quadrate und Gradientenabstiegsverfahren

Die Berechnung der Gewichtungsaktualisierung ändert sich bei der linearen Regression. Im Gegensatz zum MCP-Neuron wird nicht mehr die vorhergesagte und tatsächliche Klassenbezeichnung verwendet. Stattdessen wird der Unterschied zwischen den berechneten stetigen Ergebnissen und der tatsächlichen Klassenbezeichnung verwendet. Diese Unterschiede oder auch Fehler aller Trainingselemente werden in einer Straffunktion J quadriert und summiert. Dieser Wert der Straffunktion soll minimiert werden. Die Funktion ist in Funktion 2.9 dargestellt.

$$\Delta J(w) = \frac{1}{2}\sum_{i=1}^{m}(y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{2.9}$$

Zur Aktualisierung der Gewichtung w_j wird der Gradient mit der partiellen Ableitung der Straffunktion ∂J nach den Gewichtungen ∂w_j berechnet (Funktion 2.10).

$$\frac{\partial J}{\partial w_j} = -\sum_{i=1}^{m}(y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{2.10}$$

Somit kann die Aktualisierung der Gewichtung w_j wie in Funktion 2.11 geschrieben werden.

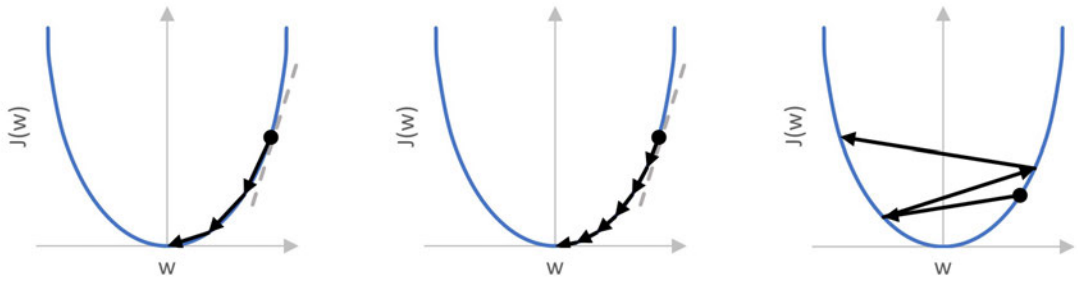
$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_{i=1}^{m}(y^{(i)} - \phi(z^{(i)}))x_j^{(i)} \tag{2.11}$$

Im Gegensatz zur Perzeptron-Lernregel werden nicht alle Gewichtungen nach jedem Trainingsobjekt inkrementell aktualisiert, sondern es werden alle Trainingsobjekte zur Berechnung der Gewichtungsaktualisierung benutzt. Deshalb wird es auch Stapelverarbeitung genannt. Ebenso ist $\phi(z^{(i)})$ eine reelle Zahl und keine ganzzahlige Klassenbezeichnung $\hat{y}^{(i)}$ wie bei der Perzeptron-Lernregel.

Die Lernrate η bei Straffunktionen mit Gradientenabstiegsverfahren

Mit der Lernrate wird die Schrittgröße bestimmt, mit der die Werte der Gewichtungen an ihre optimalen Werte angepasst werden. Der optimale Wert der Gewichtungen ist erreicht, wenn das globale Minimum der Straffunktion getroffen worden ist. Bei der Bestimmung der Lernrate kann es zu zwei Problemen kommen:

1. Die Lernrate ist zu klein: Der Algorithmus braucht zu lange, um zu konvergieren (Abbildung 2.9b).
2. Die Lernrate ist zu groß: Die Werte der Straffunktion werden größer, weil die Anpassungsschritte über das globale Minimum hinweggehen (Abbildung 2.9c).



(a) **(b)** **(c)**

Angelehnt an (Raschka & Mirjalili 2018: 58, 63) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch02/images/02_10.png https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch02/images/02_12.png

Abbildung 2.9: Anpassungsschritte bei passender Lernrate, die zum globalen Minimum der Straffunktion J konvergieren (a). Viele Anpassungsschritte bei zu kleiner Lernrate, die zum globalen Minimum der Straffunktion J konvergieren (b). Anpassungsschritte bei zu großer Lernrate, die über das globale Minimum hinausschießen und immer größer werden (c).

Polynomiale Regression

Wenn die zugrundeliegende Datenmenge nicht gut durch eine lineare Funktion angenähert werden kann, kann versucht werden die Datenmenge durch eine quadratische Funktion also einer Parabel oder einer Funktion mit höherem Polynom anzunähern (Abbildung 2.10). Bei dieser polynomialen Regression werden weitere Regressoren der Nettoeingabefunktion hinzugefügt. Diese Regressoren sind die Quadrate oder höheren Polynome der schon vorher vorhandenen Regressoren bzw. Eingabesignale (Abbildung 2.11).

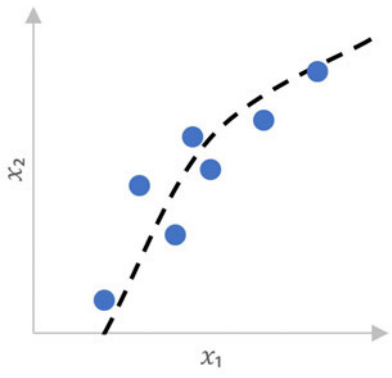


Abbildung 2.10: Polynomiale Annäherung an eine Datenmenge.

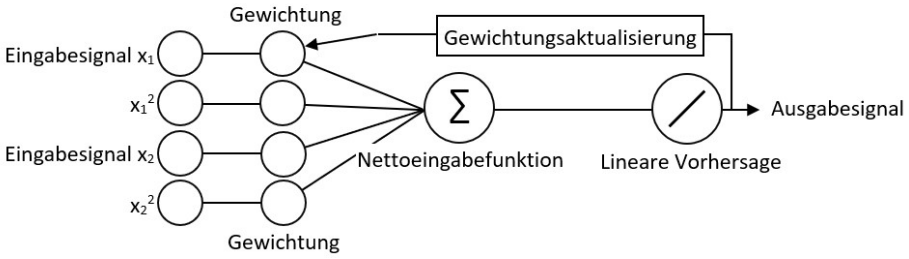
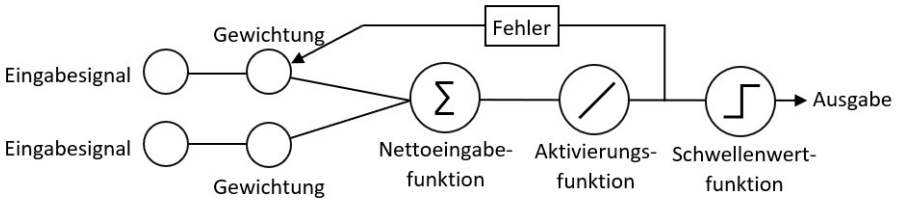


Abbildung 2.11: Schematische Darstellung einer polynomialen Regression.

2.2.3 Das adaptive lineare Neuron

Die lineare Regression sagt keine Klassenbezeichnung voraus, sondern erkennt einen Trend in den Daten. Damit ein neues Objekt klassifiziert wird, kann dem Modell der linearen Regression wieder eine Schwellenwertfunktion angefügt werden (Abbildung 2.12). Die Schwellenwertfunktion wandelt den stetigen Wert der linearen Funktion in eine Klassenbezeichnung um. Ist der Wert der linearen Funktion größer als ein festgelegter Schwellenwert, wird das Objekt als Klasse 1 klassifiziert. Ist der Wert kleiner als der Schwellenwert wird das Objekt als Klasse 0 klassifiziert.

Dieses Modell wird als adaptives lineares Neuron (Adaline) bezeichnet. Wie bei der linearen Regression wird für die Straffunktion das Ergebnis der Nettoeingabefunktion genutzt, um es mit der tatsächlichen Klasse zu vergleichen. Dadurch ist der Wert der Gewichtungsaktualisierung abhängig von der Größe des Fehlers. Je größer der Fehler zwischen Nettoeingabeergebnis und tatsächlicher Klasse ist, desto größer wird auch der Wert der Gewichtungsaktualisierung und umgekehrt. Adaline wurde von Bernard Widrow und Tedd Hoff entwickelt.



Angelehnt an: (Raschka & Mirjalili 2018: 57) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch02/images/02_09.png

Abbildung 2.12: Schematische Darstellung eines Adaline-Modells.

2.2.4 Die logistische Regression

Ausgehend von Adaline kann die lineare Aktivierungsfunktion durch eine logistische Funktion ersetzt werden. Dieses Modell wird logistische Regression genannt. Durch die logistische Funktion wird jedoch kein genauer Wert der Schwellenwertfunktion

übergeben. Sie gibt stattdessen die Wahrscheinlichkeit zurück, dass ein Objekt zu einer bestimmten Klasse gehört. Diese Wahrscheinlichkeit kann durch die Schwellenwertfunktion wieder zu einer konkreten Klassenbezeichnung umgewandelt werden. Doch zur vorhergesagten Klasse kann die logistische Regression noch zusätzlich die Wahrscheinlichkeit mitliefern, mit der das Objekt dieser Klasse angehört. Als Einstieg für die logistische Regression folgt eine kurze Einführung zu Wahrscheinlichkeiten und Chancen. Ein Verständnis von Wahrscheinlichkeiten ist nötig, um die Ausgabe der logistischen Aktivierungsfunktion interpretieren zu können.

Die Wahrscheinlichkeit

Die Wahrscheinlichkeit ist eine Zahl zwischen 0 und 1. Sie rechnet das positive Ereignis gegen alle möglichen Ereignisse auf. Sie gibt an, in wie viel Prozent aller Fälle es zu einem bestimmten Ereignis kommt.

Am Beispiel eines Würfelwurfes hat das Ereignis für das Würfelergebnis mit der Augenzahl 1 die Wahrscheinlichkeit von 1/6 (Funktion 2.12).

$$p(Augenzahl\ 1) = \frac{1}{6} \tag{2.12}$$

Bei einem Münzwurf liegt die Wahrscheinlichkeit für die eine oder die andere Seite der Münze bei 1/2 bzw. bei 0,5 (Funktion 2.13).

$$p(Kopfseite) = \frac{1}{2} \tag{2.13}$$

Die Chance

Die Chance R für das Ereignis $y = 1$ kann mit $arrayp(y = 1)/1 - p(y = 1)$ berechnet werden. Dabei ist p die Wahrscheinlichkeit für das Positivereignis y=1 und 1-p die Gegenwahrscheinlichkeit. Für das Beispiel des Würfelwurfes mit der Augenzahl 1 ergibt sich die Chance wie in Funktion 2.14.

$$P(Augenzahl\ 1)\frac{p(1)}{1-p(1)} = \frac{\frac{1}{6}}{1-\frac{1}{6}} = \frac{\frac{1}{6}}{\frac{5}{6}} = \frac{1}{5} = 0.2 \tag{2.14}$$

Bei einem Münzwurf liegt die Chance für die eine oder die andere Seite bei 1/1 bzw. 1. Dies ist gleichbedeutend mit einer Chance von 50/50 (Funktion 2.15).

$$P(Kopfseite) = \frac{\frac{1}{2}}{1-\frac{1}{2}} = \frac{\frac{1}{2}}{\frac{1}{2}} = \frac{0.5}{0.5} = \frac{50}{50} = \frac{1}{1} = 1 \tag{2.15}$$

Wenn bei einer Verlosung mit 3 Losen, 2 Lose Gewinne sind und 1 Los eine Niete ist, so liegt die Wahrscheinlichkeit für ein Gewinn bei 2/3 und die Chance bei 2 (Funktion 2.16).

$$P(Gewinn) = \frac{\frac{2}{3}}{1 - \frac{2}{3}} = \frac{\frac{2}{3}}{\frac{1}{3}} = \frac{2}{1} = 2 \tag{2.16}$$

Wenn also eine Chance unter 0 liegt, kommt es eher nicht zu diesem Ereignis. Ist die Chance für ein Ereignis 1, so sind die Chancen ausgeglichen. Ist die Chance für ein Ereignis höher als 1, so ist das Eintreten des Ereignisses sehr wahrscheinlich.

Das Logit

Die Logit-Funktion ist definiert als Logit $L(p) = ln(\frac{p(y=1)}{1-p(y=1)})$, der natürliche Logarithmus einer Chance. Die Logit-Funktion nimmt Werte zwischen 0 und 1 entgegen und bildet sie auf den gesamten Bereich der reellen Zahlen ab und kann daher Werte von minus bis plus unendlich annehmen. Chancen von über 1 werden von der Logit-Funktion bei der Übernahme auf den Wert 1 abgeschnitten. (Funktion 2.13)

$$L(p(y = 1|X = x)) = w_0x_0 + w_1x_1 + ... + w_mx_m = \sum_{j=1}^{m} x_jw_j = \vec{w}^T\vec{x} \tag{2.17}$$

$p(y = 1|X = x)$ steht für die bedingte Wahrscheinlichkeit, dass ein Objekt zur Klasse 1 gehört unter der Bedingung, dass das Merkmal X des Objekts den Wert x besitzt. Mit der Logit-Funktion kann der Einfluss eines Merkmals auf das Gesamtergebnis berechnet werden.

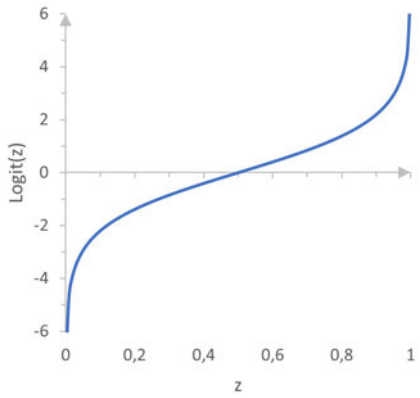


Abbildung 2.13: Werte der Logit-Funktion im Definitionsbereich von 0 bis 1.

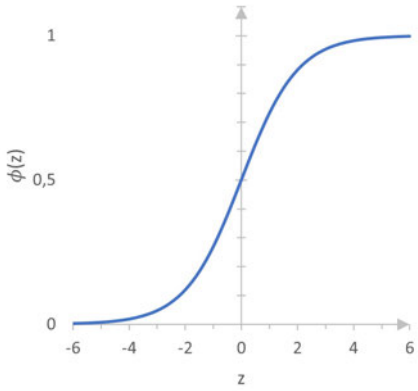
Die logistische Funktion

Es wird jedoch nicht nach dem Einfluss eines Merkmals auf das Ergebnis gesucht, sondern nach dem Ergebnis bei bestimmten Merkmalen. Dies kann mit dem Kehrwert der Logit-Funktion erreicht werden. Der Kehrwert der Logit-Funktion ist die

logistische Funktion (Funktion 2.18). Die logistische Funktion bildet den gesamten Bereich der reellen Zahlen auf Werte zwischen 0 und 1 ab (Abbildung 2.14).

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{2.18}$$

mit Nettoeingabe $z = w_0 x_0 + w_1 x_1 + ... + w_m x_m = \vec{w}^T \vec{x}$.



Angelehnt an: (Raschka & Mirjalili 2018: 83) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch03/images/03_02.png

Abbildung 2.14: Werte der Logistischen Funktion im Definitionsbereich von -6 bis 6.

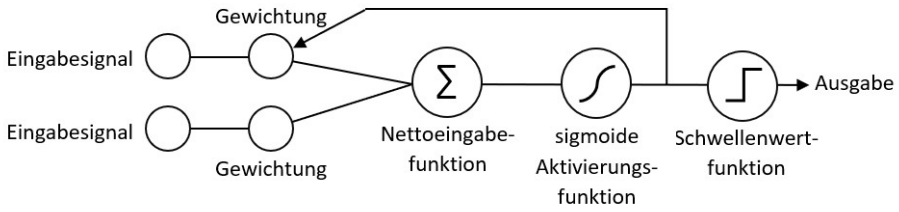
Die Werte der logistischen Funktion $\phi(z)$ gehen gegen 0, wenn z gegen minus unendlich geht. Geht z gegen positiv unendlich, geht $\phi(z)$ gegen 1. Bei $z = 0$ schneidet $\phi(z)$ die y-Achse bei 0,5. Durch das Aussehen der Kurve wird die logistische Funktion auch Sigmoid-Funktion genannt. Die Ausgabe von $\phi(z) = P(y = 1|x; w)$ wird interpretiert als Wahrscheinlichkeit, dass ein Objekt mit Merkmal x und seiner Gewichtung w zur Klasse 1 gehört. Im Umkehrschluss bedeutet das, dass das Objekt mit der Wahrscheinlichkeit $1 - \phi(z)$ zur Klasse 0 gehört (Funktion 2.19).

$$P(y = 0|x; w) = 1 - P(y = 1|x; w) \tag{2.19}$$

Wenn bei der Verlosung mit 3 Losen und 2 Gewinnen, Gewinn als 1 codiert, der Wert $\phi(z) = 0.9$ ergibt, bedeutet es, dass es sich bei dem derzeitigen Los mit einer Wahrscheinlichkeit von 90% um ein Gewinn-Los handelt. Andersherum kann auch gesagt werden, dass es sich mit der Wahrscheinlichkeit von 10% um eine Niete handelt (Funktion 2.20).

$$P(y = 0|x; w) = 1 - P(y = 1|x; w) = 1 - 0.9 = 0.1 \tag{2.20}$$

Die logistische Funktion ersetzt die lineare Aktivierungsfunktion von Adaline, dargestellt in Abbildung 2.15.



Angelehnt an: (Raschka & Mirjalili 2018: 83) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch03/images/03_03.png

Abbildung 2.15: Schematische Darstellung eines logistischen Regressions-Modells.

Die Straffunktion der logistischen Regression

Die Straffunktion der linearen Regression bzw. vom Adaline-Algorithmus muss für die logistische Regression verändert werden. Sie muss die logistische Funktion berücksichtigen. Zum Aktualisieren der Gewichtungen muss herausgefunden werden, wie gut das derzeitige Modell die Trainingsdaten klassifiziert hat. Davon soll der Wert der Aktualisierung abhängen. Dazu können die Wahrscheinlichkeit und die Gegenwahrscheinlichkeit in einer Funktion zusammengefasst werden (Funktion 2.21). Diese Funktion wird Bernoulli-Funktion genannt.

$$P(y^{(i)}|\vec{x}^{(i)};\vec{w}) = \phi(z^{(i)})^{y^{(i)}}(1-\phi(z^{(i)}))^{1-y^{(i)}} \tag{2.21}$$

Wird die Klasse $y = 1$ mit einer hohen Wahrscheinlichkeit $\phi(z) = 0,9$ korrekt vorhergesagt, ist der Wert der Bernoulli-Formel gleich dem Wert der korrekten Wahrscheinlichkeit (Funktion 2.22).

$$\begin{aligned} P(y=1|\vec{x};\vec{w}) &= 0,9^1(1-0,9)^(1-1) \\ &= 0,9 \cdot (0,1)^0 \\ &= 0,9 \cdot 1 \\ &= 0,9 \end{aligned} \tag{2.22}$$

Wird die Klasse $y = 1$ mit einer Wahrscheinlichkeit von $\phi(z) = 0,1$ falsch erkannt, ist der Wert der Bernoulli-Formel gleich dem Wert der falschen Vorhersage (Funktion 2.23).

$$\begin{aligned} P(y=1|\vec{x};\vec{w}) &= 0,1^1(1-0,1)^(1-1) \\ &= 0,1 \cdot (0,1)^0 \\ &= 0,1 \cdot 1 \\ &= 0,1 \end{aligned} \tag{2.23}$$

Wird jedoch die Klasse $y = 0$ korrekt vorhergesagt mit der Wahrscheinlichkeit $\phi(z) = 0,1$, beträgt der Wert der Bernoulli-Formel $1 - \phi(z)$ (Funktion 2.24).

$$\begin{aligned} P(y=0|\vec{x};\vec{w}) &= 0,1^0(1-0,1)^(1-0) \\ &= 1 \cdot (0,9)^1 \\ &= 1 \cdot 0,9 \\ &= 0,9 \end{aligned} \tag{2.24}$$

Somit kann der Wert der Bernoulli-Formel als Indikator für die Richtigkeit des gelernten Modells und seiner Gewichtungen für ein einziges Trainingsobjekt interpretiert werden. Wird die Bernoulli-Formel auf alle Trainingsobjekte angewendet, müssen die Werte der Bernoulli-Formel aller Trainingsobjekte multipliziert werden (Funktion 2.25). Diese Funktion wird Likelihood-Funktion genannt. Die Multiplikation der Werte bewirkt, dass das Ergebnis schlecht ist, auch wenn nur ein einziges Trainingsobjekt falsch vorhergesagt wurde.

$$\begin{aligned} L(\vec{w}) = P(\vec{y}|\vec{x};\vec{w}) &= \prod_{i=1}^{n} P(y^{(i)}|\vec{x}^{(i)};\vec{w}) \\ &= \prod_{i=1}^{n} (\phi(z^{(i)})^{y^{(i)}}(1-\phi(z^{(i)}))^{1-y^{(i)}} \end{aligned} \tag{2.25}$$

Durch das natürliche Logarithmieren der Likelihood-Funktion kann das Produkt als Summe der Faktoren geschrieben werden (Funktion 2.26). Die neue Funktion wird die Log-Likelihood-Funktion genannt.

$$l(\vec{w}) = \ln L(\vec{w} = \sum_{i=1}^{n} y^{(i)} \ln(\phi(z^{(i)})) + (1-y^{(i)}) \ln(1-\phi(z^{(i)})) \tag{2.26}$$

Entweder kann nun das Gradientenabstiegsverfahren als Optimierungsalgorithmus zum Maximieren der Log-Likelihood-Funktion genutzt werden oder sie kann zu einer Straffunktion umgeändert werden, die minimiert werden soll (Funktion 2.27).

$$J(\vec{w}) = \sum_{i=1}^{n} -y^{(i)} \ln(\phi(z^{(i)})) - (1-y^{(i)}) \ln(1-\phi(z^{(i)})) \tag{2.27}$$

Funktion 2.28 ist ein Beispiel anhand eines einzelnen Objektes mit korrekt vorhergesagter Klasse y=1. Der Wert der Straffunktion geht gegen 0.

$$\begin{aligned} J(\phi(z), y=1; \vec{w}) &= \sum_{i=1}^{n} -y \ln(\phi(z)) - (1-y)\ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -1 \ln(\phi(z)) - (1-1)\ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -\ln(\phi(z)) - 0\ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -\ln(\phi(z)) \end{aligned} \tag{2.28}$$

Wird hingegen die Klasse $y = 1$ fälschlicherweise als Klasse $y = 0$ hervorgesagt, steigt der Wert der Straffunktion ins Unendliche (Funktion 2.29).

$$\begin{aligned} J(\phi(z), y=0; \vec{w}) &= \sum_{i=1}^{n} -y \ln(\phi(z)) - (1-y)\ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -0 \ln(\phi(z)) - (1-0)\ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -1 \ln(1-\phi(z)) \\ &= \sum_{i=1}^{n} -\ln(1-\phi(z)) \end{aligned} \tag{2.29}$$

Also wird entweder der erste oder der zweite Term 0, wenn $y = 1$ oder $y = 0$ ist: Bei einer korrekten Klassifizierung wird der Wert der Straffunktion 0, bei einer falschen Vorhersage geht der Wert der Straffunktion gegen unendlich (Abbildung 2.16).

Wenn die Klasse $y = 1$ korrekt vorhergesagt worden ist, geht der Wert der Straffunktion gegen 0. Wird hingegen die Klasse $y = 1$ fälschlicher Weise als $y = 0$ vorhergesagt, geht der Wert der Straffunktion gegen unendlich. Ebenso bei korrekter oder falscher Vorhersage der Klasse $y = 0$ (Abbildung 2.16).

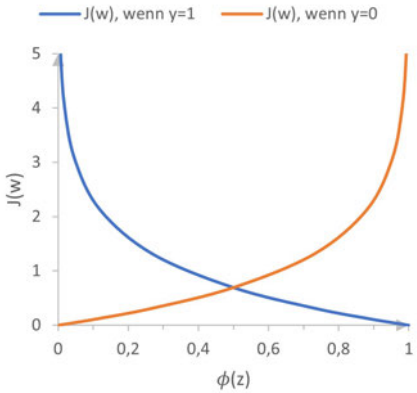
Wie oben geschrieben ist es nur eine kleine Veränderung des Adaline-Algorithmus zur logistischen Regression. Es muss nur die lineare Aktivierungsfunktion vom Adaline-Algorithmus durch die logistische Funktion und die Straffunktion durch die der logistischen Regression ersetzt werden: Die lineare Aktivierungsfunktion von Adaline wird zur logistischen Funktion (Funktion 2.30).

$$\phi(z) = z \Longrightarrow \phi(z) = \frac{1}{1 + e^{-z}} \tag{2.30}$$

Und die Summe der quadrierten Werte der Straffunktion von Adaline wird zur abgewandelten Log-Likelihood-Straffunktion (Funktion 2.31).

$$J(\vec{w} = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \phi(z^{(i)}))^2 \Longrightarrow \sum_{1=1}^{n} -y^{(i)}\ln(\phi(z^{(i)})) - (1-y^{(i)})\ln(1-\phi(z^{(i)})) \quad (2.31)$$

Das erwähnte Gradientenabstiegsverfahren der logistischen Regression ist dem des Adaline-Algorithmus gleich.



Angelehnt an: (Raschka & Mirjalili 2018: 86) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch03/images/03_04.png

Abbildung 2.16: Die Werte der Straffunktion bei der logistischen Regression.

2.2.5 Die Regularisierung

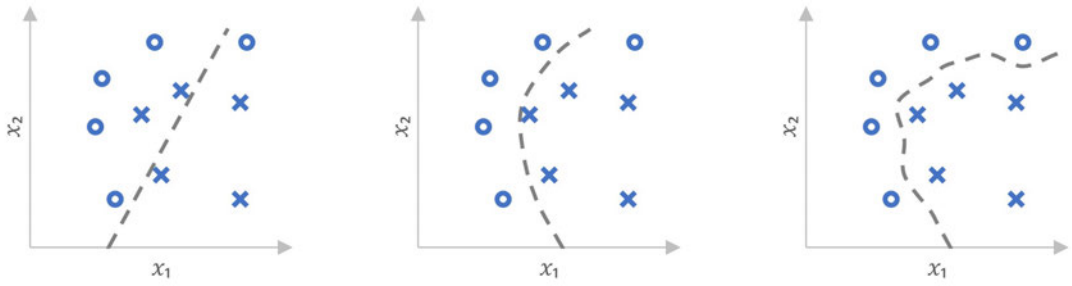
Werden zu viele Merkmale zum Trainieren eines Modells genutzt, kann es dazu kommen, dass das trainierte Modell die Trainingsobjekte zuverlässig erkennt aber keine neuen Daten. Dieses Problem wird Überanpassung genannt. Das trainierte Modell ist durch die vielen Merkmale zu komplex für die eigentliche Fragestellung geworden. Die Entscheidungsgrenze hat sich zu sehr den Trainingsobjekten angepasst, anstatt einen guten Durchschnitt abzubilden (Abbildung 2.17c). Umgekehrt kann es auch zu einer Unteranpassung kommen, wenn nur wenige Merkmale zur Verfügung standen und das trainierte Modell zu sehr verallgemeinert (Abbildung 2.17a). Es muss ein guter Kompromiss zwischen Unter- und Überanpassung gefunden werden (Abbildung 2.17b). Mit der Regularisierung kann die Komplexität des Modells an die Daten angepasst werden.

Die Regularisierung kann die Komplexität des Modells steuern. Sie filtert zudem Rauschen aus den Daten und hilft bei der Handhabung von stark korrelierenden Merkmalen. Bei der Regularisierung werden extreme Werte von Merkmalen abgeschwächt bzw. bestraft. Ein häufig genutztes Regularisierungsverfahren ist die L2-Regularisierung mit λ als Regularisierungsparameter. Der L2-Regularisierungsterm ist in der Funktion 2.32 dargestellt.

$$\frac{\lambda}{2}\|\vec{w}\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2 \tag{2.32}$$

Dabei wird die Regularisierung stärker je höher λ gewählt wird. Dieser Term wird der Straffunktion angehängt werden (Funktion 2.33).

$$J(\vec{w}) = [\sum_{i=1}^{n} -y^{(i)} \ln(\phi(z^{(i)})) - (1 - y^{(i)}) \ln(1 - \phi(z^{(i)}))] \frac{\lambda}{2}\sum_{j=1}^{m} w_j^2 \tag{2.33}$$



(a) **(b)** **(c)**

Angelehnt an: (Raschka & Mirjalili 2018: 94) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch03/images/03_07.png

Abbildung 2.17: Modell mit Unteranpassung (a). Guter Kompromiss zwischen einer Unter- und Überanpassung (b). Modell mit Überanpassung (c).

2.2.6 Optimierungsalgorithmen

Zum Minimieren einer Straffunktion können unterschiedliche Verfahren genutzt werden. Je größer die Datenmenge wird, desto länger dauert das Gradientenabstiegsverfahren (Gradient Descent, GD) als Stapelverarbeitung aus 2.2.2. Ebenso werden in neuronalen Netzen häufig andere Optimierungsalgorithmen angewendet. Da diese bei der Programmierung des MLP im Kapitel 3.2.9 angewendet werden, werden an dieser Stelle Alternativen zum GD als Stapelverarbeitung aufgeführt.

Stochastisches Gradientenabstiegsverfahren

Bei dem Gradientenabstiegsverfahren zur Straffunktionsminimierung als Stapelverarbeitung wird bei jedem Schritt die gesamte Trainingsdatenmenge neu ausgewertet. Bei sehr großen Datensätzen kann das zu einem sehr zeitaufwändigen Training führen. Mit dem stochastischen Gradientenabstiegsverfahren (SGD, Stochastic Gradient Descent) kann das verhindert werden. Es werden nicht mehr alle Gewichtungen auf

einmal aktualisiert mit der Summe aller Abweichungen aller Trainingsobjekte. Stattdessen wird bei dem stochastischen Gradientenabstiegsverfahren für jedes einzelne Trainingsobjekt die Gewichtungen inkrementell aktualisiert (Funktion 2.34). Daher wird es auch iteratives Gradientenabstiegsverfahren genannt.

$$\delta\vec{w}^{(i)} = \eta(y^{(i)} - \phi(z^{(i)}))\vec{x}^{(i)} \tag{2.34}$$

Das inkrementelle Aktualisieren der Gewichtungen ermöglicht es, das schon trainierte Modell schnell an neue Daten anzupassen, ohne es noch einmal mit den gesamten Trainingsdaten zu trainieren. Damit kann das SGD für das online Learning, bei dem neue Daten in Echtzeit integriert werden können, verwendet werden. Da das Modell für das Anpassen an neue Daten nicht die alten Trainingsdaten benötigt, müssen diese alten Trainingsdaten nicht aufbewahrt werden und können gelöscht werden.

Mini-Batch Learning

Das Mini-Batch Learning stellt einen Kompromiss zwischen Gradientenabstiegsverfahren als Stapelverarbeitung und SGD dar. Dabei wird das Gradientenabstiegsverfahren auf kleine Teile der Trainingsdatenmenge angewandt. Durch die häufigere Aktualisierung der Gewichtungen konvergiert der Algorithmus in den kleinen Teilmengen schneller.

Merkmalstandardisierung für das Gradientenabstiegsverfahren

Das Gradientenabstiegsverfahren kann verbessert werden, indem die benutzten Daten standardisiert werden. Das heißt, dass der Mittelwert einer Merkmalsspalte bei 0 liegt und die Standardabweichung der Elemente in der Merkmalsspalte 1 beträgt. Zum Standardisieren der Merkmalsspalte j muss der Mittelwert μ der Merkmalsspalte von den Elementen der Merkmalsspalte abgezogen werden und durch die Standardabweichung σ geteilt werden (Funktion 2.35). Dabei ist x_j ein Vektor mit den j-ten Merkmalswerten der gesamten Trainingsdatenmenge.

$$x_j\prime = \frac{x_j - \mu_j}{\sigma_j} \tag{2.35}$$

Durch das Standardisieren kommt es nicht mehr zu Verzerrungen der Werte der Straffunktion über mehrere Merkmale.

2.2.7 Klassifizieren mehrerer Klassen

Alle bisher vorgestellten Algorithmen sind binäre Modelle. Wenn mehrere Klassen klassifiziert werden sollen, kann die One-versus-All-Methode (OvA, oder One-versus-Rest, OvR) angewendet werden. OvA benötigt allerdings die Klassenzugehörigkeiten der Objekte in besonderer Form. Diese Form der Klassenzugehörigkeiten kann mit der One-Hot-Codierung erreicht werden.

One-Hot-Codierung

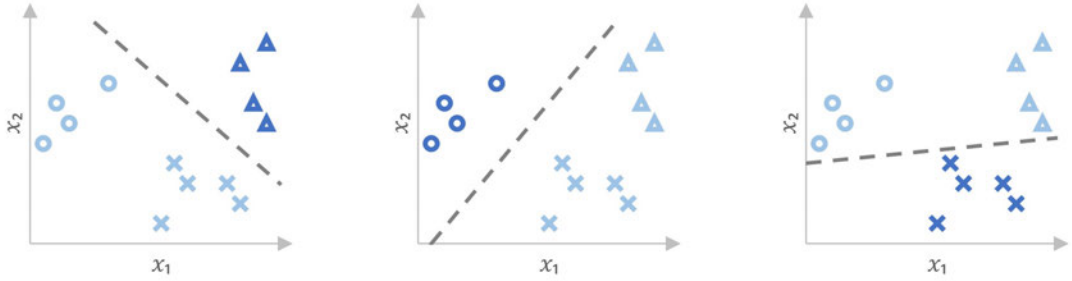
Bei der One-Hot-Codierung wird ein kategorielles Merkmal zu mehreren Merkmalen aufgespalten. Im Fall der Mehrfachklassifizierung wird nicht ein Merkmal der Objekte, sondern die Objektklassen aufgespalten. Wenn in 3 Klassen unterschieden werden soll, so muss für jede Klassenbezeichnung eine eigene Spalte erstellt werden. In dieser Spalte wird dann nur noch die Zugehörigkeit eines Objektes zu dieser Klasse gekennzeichnet. In den Spalten der anderen Klassen wird gekennzeichnet, dass das Objekt nicht zu diesen Klassen gehört. So können die Klassenbezeichnungen 1, 2 und 3 in drei Spalten unterteilt werden. Bei allen Objekten der Klasse 1 wird in der Spalte von Klasse 1 True bzw. 1 eingetragen. Bei allen anderen Objekten, die nicht zur Klasse 1 gehören wird in die Spalte von Klasse 1 False bzw. 0 eingetragen. So wird auch mit den Klassenspalten 2 und 3 verfahren (Tabelle 2.1).

Klasse		Klasse 1	Klasse 2	Klasse 3
1	$\Longrightarrow$	1	0	0
2	$\Longrightarrow$	0	1	0
3	$\Longrightarrow$	0	0	1

Tabelle 2.1: Aufspaltung von drei Klassen auf drei Spalten mittels One-Hot-Codierung

One-versus-All-Methode

Liegen die Klassenzugehörigkeiten der Objekte in der One-Hot-Codierung vor, so kann die One-versus-All-Methode angewendet werden, um mehrere Klassen zu klassifizieren. Bei OvA wird für jede Klasse ein eigener Klassifizierer, ein eigenes Modell trainiert. Bei jedem Klassifizierer ist die gesuchte Klasse die positive Klasse und alle anderen Klassen werden zur negativen Klasse gezählt. Bei dem Klassifizieren eines Objekts wird das Objekt von allen Klassifizierern klassifiziert. Danach wird überprüft, welcher Klassifizierer das überprüfte Objekt am wahrscheinlichsten zu seiner eigenen Klasse zählt. Die Klasse dieses Klassifizierers wird dem Objekt zugewiesen.



(a) **(b)** **(c)**

Abbildung 2.18: Klassifizierer 1: Klasse Dreiecke werden gegenüber den anderen erkannt (a). Klassifizierer 2: Klasse Kreise werden gegenüber den anderen erkannt (b). Klassifizierer 3: Klasse Kreuze werden gegenüber den anderen erkannt(c).

Abbildung 2.18 ist ein Beispiel mit drei Klassen (Dreiecke, Kreise, Kreuze). Für jede der drei Klassen werden eigene Klassifizierer benötigt, um sie voneinander zu unterscheiden. Die gestrichelten Linien stellen die Entscheidungsgrenzen dar.

2.3 Künstliche neuronale Netze

Für komplexere Aufgaben reichen einfache Neuronen nicht mehr aus. Es werden ganze Strukturen und Ensembles aus Neuronen benötigt. Daher besteht das neuronale Netz aus mehreren neuronalen Einheiten. Das Wissen über die Funktionsweise neuronaler Einheiten ist Voraussetzung, um neuronale Netze zu verstehen. Ebenso ist es notwendig die Vorgänge in einem vollverknüpften Netz zu kennen, um die Vorteile der konvolutionalen neuronalen Netze (convolution neural network, CNN) zu erkennen. Daher wird in diesem Kapitel erst auf die Funktionsweise des vollverknüpften mehrschichtigen Perzeptron (Multi-Layer Perceptron, MLP) eingegangen, bevor die Vorteile eines CNN erläutert werden. Im Anschluss folgt noch eine kurze Auflistung von einigen CNN-Architekturen. Die Abschnitte des Kapitels sind:

2.3.1 Das mehrschichtige Perzeptron

2.3.3 Das konvolutionale neuronale Netz

2.3.4 CNN-Architekturen

2.3.1 Das mehrschichtige Perzeptron

Bei einem mehrschichtigem Perzeptron (Multi-Layer Perceptron, MLP) sind in Schichten mehrere neuronale Elemente nebeneinander und in Schichten hintereinander geschaltet. Diese neuronalen Elemente sind keine ganzen Perzeptronen im Sinne des

McCulloch-Pitts-Neurons, sondern nur die Nettoeingabefunktionen mit den Aktivierungsfunktionen. Dabei sind alle Eingabesignale mit allen neuronalen Elementen der ersten Schicht verbunden. Alle neuronalen Elemente einer Schicht sind mit allen Elementen der nächsten Schicht verbunden. Bei einer Klassifizierungs-Aufgabe mit trainiertem Modell werden so die Signale von vorne nach hinten durchgereicht und die Ausgabe ist die Klassenbezeichnung. Daher werden Netze mit dieser Art der Signalverarbeitung auch Feedforward-Netze genannt. Bei einer binären Unterscheidung reicht wie bei einfachen Neuronen ein neuronales Element als Ausgabeschicht. Soll in mehrere Klassen unterschieden werden, muss ähnlich der OvA-Methode mit One-Hot-Codierung die Ausgabeschicht aus mehreren neuronalen Elementen bestehen.

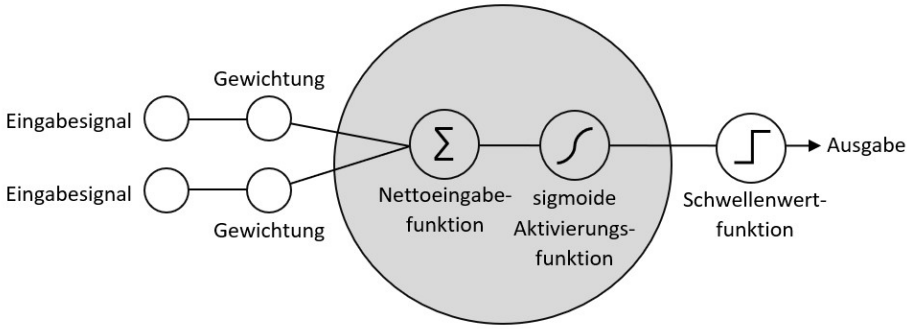


Abbildung 2.19: Schematische Darstellung des Modells der sigmoiden Regression.

In Abbildung 2.19 sind die Elemente Grau hinterlegt, die in der Abbildung 2.20 der schematischen Darstellung eines MLP zusammengefasst wurden.

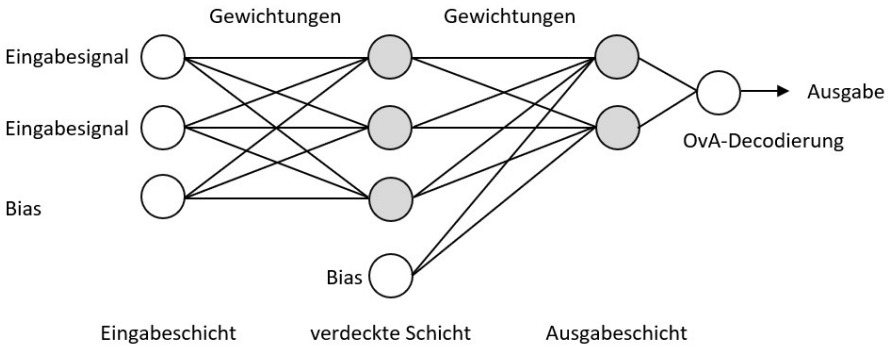


Abbildung 2.20: Schematische Darstellung eines MLP.

Bei Abbildung 2.20 bestehen die verdeckten und Ausgabe-Elemente jeweils aus einer Nettoeingabefunktion und einer Aktivierungsfunktion. Ebenso besitzen alle Verbindungen zu neuronalen Elementen Gewichtungen.

Vorwärtspropagation

Die Signale werden von der Eingabeschicht nach hinten über verdeckte Schichten zur Ausgabeschicht weiterverarbeitet (Abbildung 2.21). Durch die vollständige Verknüpfung der Eingabesignale mit der ersten verdeckten Schicht hat ein einzelnes Eingabesignal auf alle neuronalen Elemente der verdeckten Schicht Auswirkung. Durch die Gewichtungen, mit denen das einzelne Signal verrechnet wird, sind die Auswirkungen auf die folgenden neuronalen Elemente unterschiedlich. Ebenso hat ein einzelnes neuronales Element aus einer ersten verdeckten Schicht Auswirkung auf alle neuronalen Elemente der folgenden Schicht. Diese Auswirkungen sind ebenfalls durch weitere Gewichtungen unterschiedlich stark.

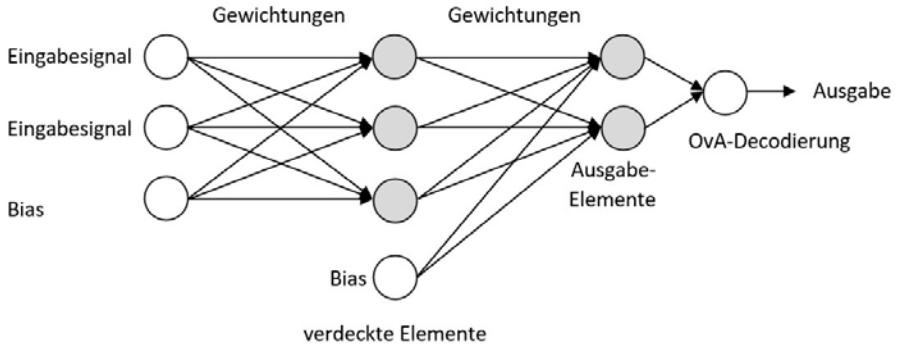


Abbildung 2.21: Schematische Darstellung der Merkmalsweitergabe bei der Forwardpropagation.

Die Anzahl der Schichten und die Anzahl der neuronalen Einheiten können als Hyperparameter eines neuronalen Netzes angesehen werden. Um die Auswirkungen dieser Parameter auf das Netz und sein Training verstehen zu können, müssen die mathematische Funktionen betrachtet werden. In einem MLP sind alle Eingabemerkmale mit allen neuronalen Einheiten der nächsten verdeckten Schicht verbunden. Die Nettoeingabe der verdeckten Schicht $\vec{z}^{(h)}$ summiert alle Eingabemerkmale $\vec{x}$ plus Bias x^0 zusammen (Funktion 2.36).

$$\vec{z}^{(h)} = x^0 w1, 0^{(h)} + x^1 w_{2,0}^{(h)} + x^m w_{m+1,0}^{(h)} = \vec{a}^{(in)} \vec{W}^{(h)} \tag{2.36}$$

In Funktion 2.36 steht $a^{(h)}$ für die logistische Aktivierungsfunktion 2.37.

$$\begin{aligned} \vec{a}^{(h)} &= \phi(z^{(h)}) \\ \phi(\vec{z}) &= \frac{1}{1 + e^{-z}} \end{aligned} \tag{2.37}$$

Die Nettoeingabe und Aktivierung kann für alle n Trainingsobjekte in der verdickten Schicht verallgemeinert werden (Funktion 2.38).

$$\begin{aligned} \vec{Z}^{(h)} &= \vec{A}^{(in)}\vec{W}^{(h)} \\ \vec{A}^{(h)} &= \phi(\vec{Z}^{(h)}) \end{aligned} \tag{2.38}$$

Auf gleiche Weise kann die Nettoeingabe und Aktivierung der anderen verdeckten Schichten und der Ausgabeschicht dargestellt werden (Funktion 2.39).

$$\begin{aligned} \vec{Z}^{(out)} &= \vec{A}^{(h)}\vec{W}^{(out)} \\ \vec{A}^{(out)} &= \phi(\vec{Z}^{(out)}) \end{aligned} \tag{2.39}$$

Die Anzahl der neuronalen Einheiten und die Anzahl der Schichten hat somit signifikante Auswirkungen auf die Anzahl der Berechnungen, die in einem Netz berechnet werden müssen.

Tabellarische Darstellung der Rechenoperationen

Zur Übersicht über die Rechenoperationen in einem kleinen neuronalen Netz sind in der Tabelle 2.2 die Berechnungen dargestellt. Das abgebildete Netz besitzt einen Eingabevektor x mit 2 Merkmalen, 3 neuronalen Einheiten in der verdeckten Schicht und 2 Ausgabeeinheiten. Farblich hinterlegt sind jeweils ein Merkmal, seine Gewichtung und ihre Verrechnung in der folgenden Nettoeingabe- bzw. Aktivierungsfunktion.

Eingabe $\vec{a}^{(in)}$: $1 \times m$-Matrix

Bias x^0	Merkmal x^1	Merkmal x^2

m: Anzahl der Merkmale + Bias

m = 2 Merkmale + Bias = 3

Gewichtungen $\vec{W}^{(h)}$: $m \times d$-Matrix

$w_{1,0}^{(h)}$	$w_{2,0}^{(h)}$	$w_{3,0}^{(h)}$
$w_{1,1}^{(h)}$	$w_{2,1}^{(h)}$	$w_{3,1}^{(h)}$
$w_{1,2}^{(h)}$	$w_{2,2}^{(h)}$	$w_{3,2}^{(h)}$

m: Anzahl der Merkmale + Bias

d: Anzahl der neuronalen Einheiten

m = 2 + Bias = 3

d = 3

Nettoeingabe $\vec{z}^{(h)} = \vec{a}^{(in)}\vec{W}^{(h)}$: $1 \times d$-Matrix

$z_1^{(h)} =$	$z_2^{(h)} =$	$w_3^{(h)} =$
$x^0 w_{1,0}^{(h)}$	$x^0 w_{1,1}^{(h)}$	$x^0 w_{1,1}^{(h)}$
$+x^1 w_{2,0}^{(h)}$	$+x^1 w_{2,2}^{(h)}$	$+x^1 w_{2,2}^{(h)}$
$+x^2 w_{3,0}^{(h)}$	$+x^2 w_{3,2}^{(h)}$	$+x^2 w_{3,2}^{(h)}$

d: Anzahl der neuronalen Einheiten

d = 3

Aktivierung $\vec{a}^{(h)} = \phi(z^{(h)}$: $1 \times g$-Matrix

Bias $a_0^{(h)}$	$a_1^{(h)} = \phi(z_1^{(h)})$	$a_2^{(h)} = \phi(z_2^{(h)})$	$a_3^{(h)} = \phi(z_3^{(h)})$

g: Anzahl der neuronalen Einheiten + Bias

g = 3 + Bias = 4

Gewichtungen $\vec{W}^{(h)}$: $g \times h$-Matrix

$w_{1,0}^{(out)}$	$w_{2,0}^{(out)}$
$w_{1,1}^{(out)}$	$w_{2,1}^{(out)}$
$w_{1,2}^{(out)}$	$w_{2,2}^{(out)}$
$w_{1,3}^{(out)}$	$w_{2,3}^{(out)}$

g: Anzahl der neuronalen Einheiten + Bias

h: Anzahl der Klassen

g = 2 + Bias = 4

h = 2

Nettoeingabe $\vec{z}^{(out)} = \vec{a}^{(h)}\vec{W}^{(out)}$: $1 \times h$-Matrix

$z_1^{(out)} =$	$z_2^{(out)} =$
$a_0^{(h)} w_{1,0}^{(out)}$	$a_0^{(h)} w_{2,0}^{(out)}$
$+a_1^{(h)} w_{1,1}^{(out)}$	$+a_1^{(h)} w_{2,1}^{(h)}$
$+a_2^{(h)} w_{1,2}^{(out)}$	$+a_2^{(h)} w_{2,2}^{(h)}$
$+a_3^{(h)} w_{1,3}^{(out)}$	$+a_3^{(h)} w_{2,3}^{(h)}$

h: Anzahl der Klassen

h = 2

Aktivierung $\vec{a}^{(out)} = \phi(\vec{z}^{(out)})$: $1 \times h$-Matrix

$a_1^{(out)} = \phi(z_1^{(out)})$	$a_2^{(out)} = \phi(z_2^{(out)})$

h: Anzahl der Klassen

h = 2

Tabelle 2.2: Nettoeingabe und Aktivierung der verdeckten Schicht und der Ausgabeschicht.

Die Gewichtungsmatrix

Bei einem neuronalen Netz ist zu beachten, dass die Gewichtungsmatrix keine gleichmäßige Matrix ist. Die Anzahl der Elemente in einer Dimension sind nicht unbedingt gleich. Die Gewichtungsmatrix $\vec{W}$ kann als Sammlung aus den unterschiedlich großen Gewichtungsmatrizen der Schichten $\vec{W}^{(n)}$ angesehen werden. Diese Matrizen besitzen nicht alle die gleiche Spalten- und Zeilenanzahl. In Abbildung 2.22 ist eine solche ungleichmäßige Gewichtungsmatrix dargestellt. Die Gewichtungsmatrix wird bei der Analyse eines Farbbildes sehr groß. Schon der Eingangs-Merkmalsraum ist bei einem Farbbild eine $3 \times h \times b$-Matrix mit der Anzahl von Pixeln in der Höhe als h und der Anzahl von Pixeln in der Breite als b. Bei einer Höhe und Breite eines Bildes von 28 Pixeln ergeben sich schon $3 \cdot 28 \cdot 28 = 2.352$ Elemente in der Eingangsmatrix. Jedes Element der ersten verdeckten Schicht erhält zu jedem dieser 2.352 Eingangsmerkmalen eine Gewichtung. Dabei kann eine Schicht aus hunderten neuronalen Einheiten bestehen. Die Einheiten einer zweite verdeckten Schicht besitzen zu allen Einheiten der vorangegangenen Schicht eine Gewichtung.

Gewichtungsmarix $\boldsymbol{W}$

Gewichtungen $\boldsymbol{W}^{(h)}$: $m \times d$-Matrix

$w_{1,0}^{(h)}$	$w_{2,0}^{(h)}$	$w_{3,0}^{(h)}$
$w_{1,1}^{(h)}$	$w_{2,1}^{(h)}$	$w_{3,1}^{(h)}$
$w_{1,2}^{(h)}$	$w_{2,2}^{(h)}$	$w_{3,2}^{(h)}$

m: *Anzahl von Merkmalen + Bias*
d: *Anzahl von neuronalen Einheiten*
$m = 2 + Bias = 3$
$d = 3$

Gewichtungen $\boldsymbol{W}^{(out)}$: $g \times h$-Matrix

$w_{1,0}^{(out)}$	$w_{2,0}^{(out)}$
$w_{1,1}^{(out)}$	$w_{2,1}^{(out)}$
$w_{1,2}^{(out)}$	$w_{2,2}^{(out)}$
$w_{1,3}^{(out)}$	$w_{2,3}^{(out)}$

g: *Anzahl von neuronalen Einheiten + Bias*
h: *Anzahl von Klassen*
$g = 3 + Bias = 4$
$h = 2$

Abbildung 2.22: Tabellarische Darstellung der Gewichtungsmatrix bei einer verdeckten Schicht und der Ausgabeschicht.

Die logistische Straffunktion

Bei der Straffunktion eines neuronalen Netzes ist zu beachten, dass das Netz aus mehreren Schichten mit mehreren Einheiten besteht. Der Wert der Straffunktion berechnet sich aus den Fehlern aller Einheiten in allen Schichten. Zudem neigen neuronale Netze zur Überanpassung an die Trainingsdaten. In diesen Fällen kann das trainierte Modell neue Daten nicht mehr so gut klassifizieren wie die Trainingsdaten. Eine Lösung für dieses Problem ist die Regularisierung der Straffunktion. In diesem Beispiel wird der L2-Regularisierungsterm verwendet. Eine weitere Methode die Überanpassung zu minimieren nennt sich Dropout und wird in Kapitel 2.3.3 behandelt. Die logistische Straffunktion der logistischen Regression aus dem Kapitel 2.2.4 ist in Funktion 2.40 abgebildet.

$$J(\vec{w}) = \sum_{i=1}^{n} -y^{(i)} \ln(a^{(i)}) - (1 - y^{(i)}) \ln(1 - a^{(i)}) \tag{2.40}$$

In Funktion 2.40 ist der Ausdruck $\phi(z^{(i)})$ als a dargestellt. Zur Regularisierung der Straffunktion wird ihr der L2-Regularisierungsterm $\frac{\lambda}{2}\|\vec{w}\|^2$ angehängt (Funktion 2.41).

$$J(\vec{w}) = -[\sum_{i=1}^{n} -y^{(i)} \ln(a^{(i)}) - (1 - y^{(i)}) \ln(1 - a^{(i)})] + \frac{\lambda}{2}\|\vec{w}\|^2 \tag{2.41}$$

Da es im Netz mehr als ein neuronales Element gibt muss die logistische Straffunktion für alle neuronalen Elemente j verallgemeinert werden. In Funktion 2.42 ist die Verallgemeinerung der logistischen Straffunktion ohne L2-Term abgebildet.

$$J(\vec{w}) = -\sum_{i=1}^{m}\sum_{j=1}^{n} -y_j^{(i)} \ln(a_j^{(i)}) - (1 - y_j^{(i)}) \ln(1 - a_j^{(i)}) \tag{2.42}$$

Der L2-Regularisierungsterm muss ebenfalls verallgemeinert werden, indem alle Gewichtungen aller neuronaler Elemente aus einer Schicht zu ihrer nächsten zusammengezählt werden (Funktion 2.43).

$$L2 = \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u}\sum_{j=1}^{u+1}(w_{j,i}^{(l)})^2 \tag{2.43}$$

In 2.43 steht L für die Anzahl der Schichten und u bezieht sich auf die Anzahl der Einheiten in der Schicht. Dieser verallgemeinerte L2-Regularisierungsterm muss der verallgemeinerten Straffunktion angefügt werden (2.44).

$$J(\vec{w}) = -[\sum_{i=1}^{m}\sum_{j=1}^{n} -y_j^{(i)} \ln(a_j^{(i)}) - (1 - y_j^{(i)}) \ln(1 - a_j^{(i)})] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u}\sum_{j=1}^{u+1}(w_{j,i}^{(l)})^2 \tag{2.44}$$

Die in Funktion 2.44 dargestellte Straffunktion kann nun zum Berechnen der Kosten pro Epoche genutzt werden. Für die Fehlerrückführung des Fehlers der Ausgabeschicht rückwärts durch das Netz wird diese Funktion allerdings nicht benötigt.

Backpropagation

Bei der Backpropagation wird der Fehler der Ausgabeschicht von hinten nach vorne propagiert. Der Fehler muss für jede Schicht einzeln berechnet werden. Zu beachten ist, dass die Aktivierungsfunktionen in den Schichten bei der Backpropagation ebenfalls zurückgerechnet werden muss. Da mit einem neuronalen Netz mehrere Klassen auf einmal klassifiziert werden können, muss als erstes der ganze Fehlervektor der Ausgabeschicht $\delta^{(out)}$ berechnet werden (Funktion 2.45). Dabei ist $a^{(out)}$ der vorhergesagte und y der tatsächliche Fehlervektor.

$$\vec{\delta}^{(out)} = \vec{a}^{(out)} - \vec{y} \tag{2.45}$$

Bei der Berechnung der Fehlervektoren der verdeckten Schichten kann nicht einfach der Fehler der Ausgabeschicht mit den Gewichtungen der verdeckten Schichten multipliziert werden. Es muss beachtet werden, dass die Aktivierungsfunktionen der vorherigen Schichten zurückgerechnet werden müssen. So müssen die Ableitungen der sigmoiden Aktivierungsfunktion mit den Fehlern der verdeckten Schicht multipliziert werden (Funktion 2.46).

$$\begin{aligned} \vec{\delta}^{(h)} &= \vec{\delta}^{(out)}(\vec{W}^{(out)})^T * \frac{\partial \phi(z^{(h)})}{\partial z^{(h)}} \\ \frac{\partial \phi(z^{(h)})}{\partial z^{(h)}} &= a^{(h)} * (1 - a^{(h)}) \\ \vec{\delta}^{(h)} &= \vec{\delta}^{(out)}(\vec{W}^{(out)})^T * (a^{(h)} * (1 - a^{(h)})) \end{aligned} \tag{2.46}$$

Mit den Termen der Fehlermatrix $\vec{\delta}^{(h)}$ können die Ableitungen der Straffunktion erstellt werden (Funktion 2.47).

$$\begin{aligned} \frac{\partial J}{\partial w_{i,j}^{(out)}} &= a_j^{(h)} \delta_i^{(out)} \\ \frac{\partial J}{\partial w_{i,j}^{(h)}} &= a_j^{(in)} \delta_i^{(h)} \end{aligned} \tag{2.47}$$

Nun müssen die partiellen Ableitungen aller Knoten von allen Schichten l und der Fehler des Knotens in der nächsten Schicht kumuliert werden. Dabei ist zu beachten, dass $\Delta_{i,j}^{(l)}$ für alle Trainingselemente berechnet werden muss. Δ kann auch für alle Schichten l vektorisiert dargestellt werden (Funktion 2.48).

$$\begin{aligned} \Delta_{i,j}^{(l)} &:= \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \\ \vec{\Delta}^{(out)} &= \Delta^{(out)} + (\vec{A}^{(h)})^T \vec{\delta}^{(out)} \\ \vec{\Delta}^{(h)} &= \Delta^{(h)} + (\vec{A}^{(in)})^T \vec{\delta}^{(h)} \end{aligned} \tag{2.48}$$

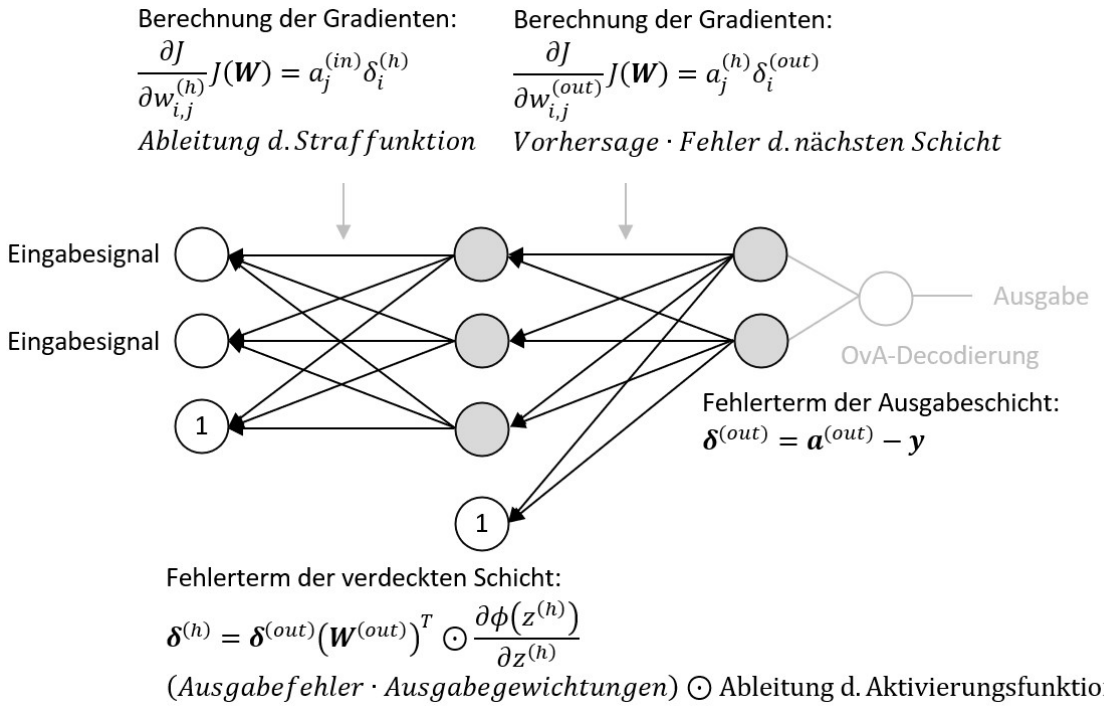
Nach dem Kumulieren der partiellen Ableitungen kann der L2-Regularisierungsterm addiert werden (Funktion 2.49).

$$\vec{\Delta}^{(l)} := \vec{\Delta}^{(l)} + \lambda^{(l)} \tag{2.49}$$

Nach der Berechnung der Gradienten können die Gewichtungen zum Aktualisieren wieder einen Schritt in die entgegengesetzte Richtung der Gradienten versetzt werden (2.50).

$$\vec{W}^{(l)} := \vec{W}^{(l)} - \eta\vec{\Delta}^{(l)} \tag{2.50}$$

Nach der Aktualisierung der Gewichtungen wird in der nächsten Trainingsepoche mit der Feedforward-Propagation der neue Fehler der Ausgabeschicht erzeugt. Mit diesem neuen Fehler kann das Backpropagation-Verfahren wiederholt und die Gewichtungen erneut aktualisiert werden.



Angelehnt an: (Raschka & Mirjalili 2018: 418) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch12/images/12_12.png

Abbildung 2.23: Schematische Darstellung der Fehlerrückführung bei der Backpropagation.

2.3.2 Verknüpfung von Layern

Werden dem neuronalen Feedforward-Netz mehrere verdeckte Schichten hinzugefügt, werden im Backpropagation-Verfahren zur Berechnung der Fehlergradienten immer öfter die Lernrate mit dem Fehlerterm multipliziert. So wird der Gradient immer kleiner, je weiter nach vorne der Fehler propagiert wird. Das führt dazu, dass die Gewichtungen der vorderen Schichten nicht so stark berichtigt werden, wie die der hinteren Schichten. Dieses Problem wird verschwindender Gradient genannt.

Um dieses Problem zu vermeiden, können die vorderen neuronalen Einheiten nicht nur mit der folgenden Schicht verknüpft sein, sondern sie können auch mit darauf

folgenden Schichten verbunden sein. Diese Verbindungen werden Shortcuts genannt. Dadurch werden die Gewichtungen der vorderen Schichten zusätzlich schon früher im Backpropagation-Verfahren berichtigt.

Shortcuts werden in bestimmten Netzwerk-Architekturen verwendet. Diese Netzwerk-Architekturen werden Residual neural Networks genannt. Auf Residual neural Networks wird im Abschnitt Architekturen von konvolutionalen neuronalen Netzen Abschnitt 2.3.4 eingegangen.

Rekurrente neuronale Netze

Neuronale Elemente können nicht nur mit folgenden Schichten verbunden sein, sondern auch mit sich selbst, mit Einheiten der gleichen Schicht oder mit Einheiten vorheriger Schichten. Das ermöglicht es, Informationen der Ausgabeschichten in einem weiteren Durchlauf in der Eingabeschicht wiederzuverwenden. Es werden die Ergebnisse des vorherigen Durchlaufs im aktuellen Durchlauf verrechnet. Dadurch kann das Netz eine ganze zusammenhängende Kolonne von Informationen auswerten und die verarbeiteten Informationen sogar in einem Gedächtnis speichern. Dies ermöglicht die Klassifikation von fortlaufenden Audiosignalen und Videos. Da rekurrente Vernetzung jedoch nicht in einem CNN genutzt wird, kann das RNN an dieser Stelle außen vor gelassen werden.

2.3.3 Das konvolutionale neuronale Netz

Soll ein neuronales Netz zur Bildklassifizierung genutzt werden, würde sich bei der Verwendung eines vollverknüpften MLPs eine Gewichtungsmatrix mit Höhe $\times$ Breite $\times$ Anzahl der Farbkanäle $\times$ Anzahl der Layer ergeben, vorausgesetzt die verdeckten Ebenen sind so groß wie das Eingabebild. Hinzu kommen noch die Gewichtungen in der letzten Schicht mit einer Anzahl an neuronalen Einheiten wie es Klassen gibt. Bei einem 100x100 Pixel großen Farbbild, 2 Schichten und 2 Klassen ergeben sich somit $100 \cdot 100 \cdot 3 \cdot 2 + 100 \cdot 100 \cdot 2 = 62.000$ Gewichtungen ohne Bias. Um die Gewichtungsmatrix zu verkleinern, können diskrete Faltungen verwendet werden. Dabei wird eine kleinere Gewichtungsmatrix als Fensterausschnitt über das Bild geschoben und die Werte der verrechneten Ausschnitte miteinander summiert. Diese Verrechnung wird diskrete Faltung genannt, daher auch der Name des Netzes (convolution: enlg. Faltung).

Diskretes Falten

Vorausgesetzt, dass die direkte Umgebung eines Pixels für den Informationsgehalt des Pixels relevant ist, kann die Information für jeden Pixel mit seiner Umgebung mit den gleichen Gewichtungen berechnet werden. Der Vorgang der Verrechnung des Pixels mit seiner Umgebung wird diskrete Faltung genannt. Die Operation der diskreten Faltung ist wie in Funktion 2.51 festgelegt.

$$\begin{aligned} &\vec{Y} = \vec{X} * \vec{W} \longrightarrow = \vec{Y}[i,j] = \sum_{k=-\infty}^{+\infty} \sum_{l=-\infty}^{+\infty} \vec{X}[i-k, j-l]\vec{W}[k,l] \\ &mit \vec{X}_{n \times m}, \vec{W}_{o \times p} \; und \; o \geq n, p \geq m \end{aligned} \tag{2.51}$$

In Funktion 2.51 ist $\vec{Y}$ das Ergebnis der Faltungsoperation des Eingabesignals $\vec{X}$ mit der Gewichtungsmatrix $\vec{W}$. Die Gewichtungsmatrix wird Kernel bzw. Filter genannt. $[i, j]$ gibt den Index des Elements in der Matrix an. Die Kernelmatrix muss kleiner oder höchstens gleich groß der Eingabematrix sein. Bei der Berechnung eines Wertes der Ausgabematrix wird das Skalarprodukt von dem $o \times p$-großen Bereich der Eingabematrix an der Stelle i, j und der Kernelmatrix berechnet. Zu beachten ist, dass bei der diskreten Faltung $\vec{X}$ und $\vec{W}$ gegensätzlich indiziert werden. In der Praxis werden keine unendlich großen Kernelmatrizen berechnet. So kann die Faltungsoperation auch wie in Funktion 2.52 geschrieben werden. Es werden üblicherweise 3×3 oder 5×5 große Matrizen als Kernel genutzt.

$$\vec{Y}[i,j] = \sum_{k=0}^{o-1} \sum_{l=0}^{p-1} \vec{X}[i-k, j-l]\vec{W}[k,l] \tag{2.52}$$

Im folgenden Beispiel ist zu sehen, dass eine 0-Indexierung der Ausgabematrix einen Rand für die Eingabematrix benötigt. Dieser Rand wird mit Nullen aufgefüllt. Damit auch die hohen Indizes der Eingabematrix mit Nullwerten verrechnet werden, muss die Ergebnismatrix jedoch vergrößert werden. Das Auffüllen der Eingabematrix mit Nullwerten wird Zero-Padding oder einfach Padding bezeichnet.

Für das Beispiel seien $\vec{X}_{4\times4}$ als Eingabebild und $\vec{W}_{2\times2}$ als Kernel gegeben (Tabelle 2.3).

$\vec{X}_{4\times4}:$	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$			
	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$\vec{W}_{2\times2}:$	$w_{0,0}$	$w_{0,1}$
	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$		$w_{1,0}$	$w_{1,1}$
	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$			

Tabelle 2.3: Beispiel diskrete Faltung: Eingabebild und Kernelmatrix

Eine Y-Indexierung bei 0 benötigt negative Indexierungen der Eingabematrix. Die fehlenden Werte werden mit Nullen aufgefüllt (Tabelle 2.4).

$x_{-1,-1}$	$x_{-1,0}$			
$x_{0,-1}$	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$
	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$
	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$
	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$

Tabelle 2.4: Beispiel diskrete Faltung: Negative Indices an Position 0x0.

$$\begin{aligned}\vec{Y}[0,0] &= \sum_{k=0}^{2-1}\sum_{l=0}^{2-1}\vec{X}[0-k,0-l]\vec{W}[k,l] = \sum_{k=0}^{1}\sum_{l=0}^{1}\vec{X}[-k,-l]\vec{W}[k,l] \\ &= (x_{0,0}w_{0,0} + x_{0,-1}w_{0,1}) + (x_{-1,0}w_{1,0} + x_{-1,-1}w_{1,1})\end{aligned} \tag{2.53}$$

$x_{-1,-1}$	$x_{-1,0}$	$x_{-1,1}$		
$x_{0,-1}$	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$
	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$
	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$
	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$

Tabelle 2.5: Beispiel diskrete Faltung: Negative Indices an Position 0x1.

$$\begin{aligned}\vec{Y}[0,1] &= \sum_{k=0}^{2-1}\sum_{l=0}^{2-1}\vec{X}[0-k,1-l]\vec{W}[k,l] = \sum_{k=0}^{1}\sum_{l=0}^{1}\vec{X}[-k,1-l]\vec{W}[k,l] \\ &= (x_{0,1}w_{0,0} + x_{0,0}w_{0,1}) + (x_{-1,1}w_{1,0} + x_{-1,0}w_{1,1})\end{aligned} \tag{2.54}$$

$x_{-1,-1}$	$x_{-1,0}$	$x_{-1,1}$	$x_{-1,2}$	$x_{-1,3}$
$x_{0,-1}$	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$
$x_{1,-1}$	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$
$x_{2,-1}$	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$
$x_{3,-1}$	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$

Tabelle 2.6: Beispiel diskrete Faltung: Eingabematrix mit negativen Indices aufgefüllt.

$$\begin{aligned}\vec{Y}[3,3] &= \sum_{k=0}^{2-1}\sum_{l=0}^{2-1}\vec{X}[3-k,3-l]\vec{W}[k,l] = \sum_{k=0}^{1}\sum_{l=0}^{1}\vec{X}[3-k,3-l]\vec{W}[k,l] \\ &= (x_{3,3}w_{0,0} + x_{3,2}w_{0,1}) + (x_{2,3}w_{1,0} + x_{2,2}w_{1,1})\end{aligned} \tag{2.55}$$

Damit auch die hohen Indices der Eingabematrix mit Nullwerten verrechnet werden, muss die Ergebnismatrix größer werden als die Eingabematrix (2.7).

$x_{-1,-1}$	$x_{-1,0}$	$x_{-1,1}$	$x_{-1,2}$	$x_{-1,3}$	$x_{-1,4}$
$x_{0,-1}$	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{0,4}$
$x_{1,-1}$	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$
$x_{2,-1}$	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$
$x_{3,-1}$	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$
$x_{4,-1}$	$x_{4,0}$	$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$

Tabelle 2.7: Beispiel diskrete Faltung: Die Eingabematrix mit Rand.

$$\begin{aligned}\vec{Y}[4,4] &= \sum_{k=0}^{2-1}\sum_{l=0}^{2-1}\vec{X}[4-k,4-l]\vec{W}[k,l] = \sum_{k=0}^{1}\sum_{l=0}^{1}\vec{X}[4-k,4-l]\vec{W}[k,l] \\ &= (x_{4,4}w_{0,0} + x_{4,3}w_{0,1}) + (x_{3,4}w_{1,0} + x_{3,3}w_{1,1})\end{aligned} \tag{2.56}$$

$\vec{X}^p_{6\times 6}$:

0	0	0	0	0	0
0	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	0
0	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	0
0	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	0
0	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	0
0	0	0	0	0	0

$\vec{Y}_{5\times 5}$:

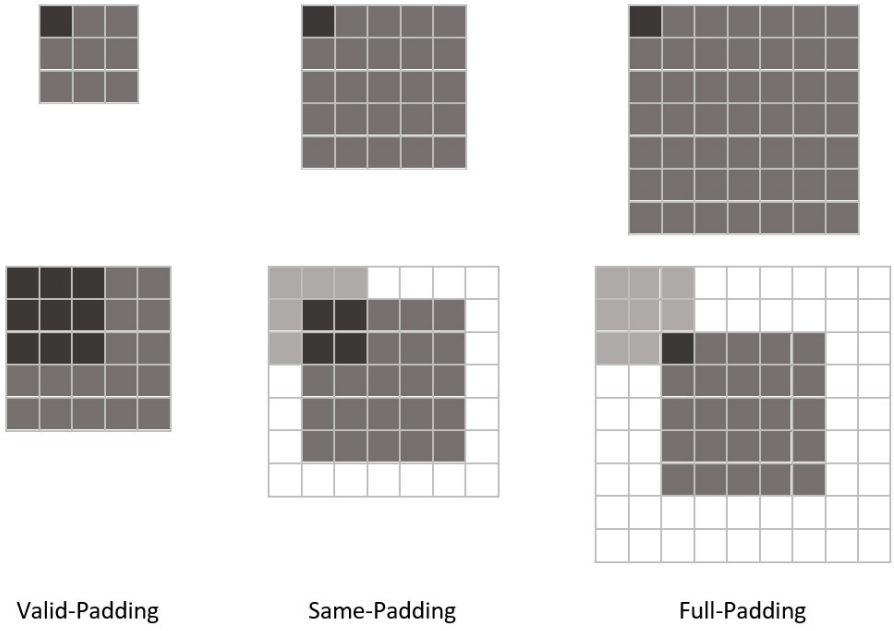
$y_{0,0}$	$y_{0,1}$	$y_{0,2}$	$y_{0,3}$	$y_{0,4}$
$y_{1,0}$	$y_{1,1}$	$y_{1,2}$	$y_{1,3}$	$y_{1,4}$
$y_{2,0}$	$y_{2,1}$	$y_{2,2}$	$y_{2,3}$	$y_{2,4}$
$y_{3,0}$	$y_{3,1}$	$y_{3,2}$	$y_{3,3}$	$y_{3,4}$
$y_{4,0}$	$y_{4,1}$	$y_{4,2}$	$y_{4,3}$	$y_{4,4}$

Tabelle 2.8: Beispiel diskrete Faltung: Eingabematrix und Ergebnismatrix

Padding

Abhängig von der Größe der Kernelmatrix können auch mehrere Reihen und Spalten mit Nullwerten angehängt werden, also ein anderes Padding gewählt werden. Ebenso kann auch die Eingabematrix nicht erweitert werden, was jedoch eine Verschiebung der Indexierung der Ergebnismatrix zur Folge hat. Wird die Eingabematrix nicht erweitert, wird von Valid-Padding gesprochen. Besitzt die Eingabematrix einen Rand, der es ermöglicht, dass die Ergebnismatrix gleich groß ist, wird von Same-Padding gesprochen. Sollen die Kanten der Eingabematrix einzeln mit der Kernelmatrix verrechnet werden, wird die Ausgabematrix größer, es wird wieder eine Indexverschiebung benötigt, und wird Full-Padding genannt. Auf Abbildung 2.24 stellt die untere Reihe die Eingabebilder mit Padding und die obere Reihe die Ergebnismatrizen dar.

Soll die Ergebnismatrix gleich groß der Eingabematrix sein, also Same-Padding benutzt werden, ist die Breite des Randes abhängig von der Größe der Filtermatrix. Wächst die Kernelmatrix, muss auch das Padding wachsen. Abbildung 2.25 stellt die Ergebnismatrix mit der erforderlichen Eingabematrix bei einer 5×5 Filtermatrix dar.



Angelehnt an: (Raschka & Mirjalili 2018: 495) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch15/images/15_11.png

Abbildung 2.24: Unterschiedliche Padding-Methoden.

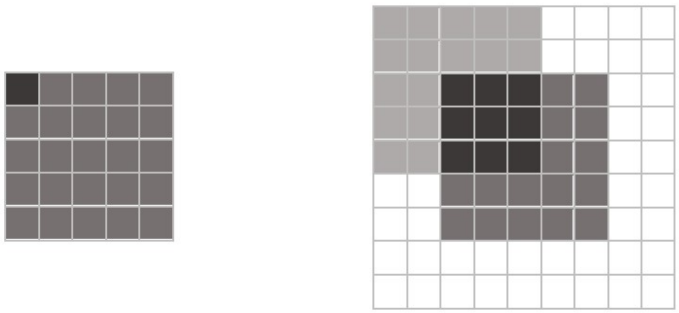


Abbildung 2.25: Padding von 2 für Same-Padding bei einer 5×5 Filtermatrix.

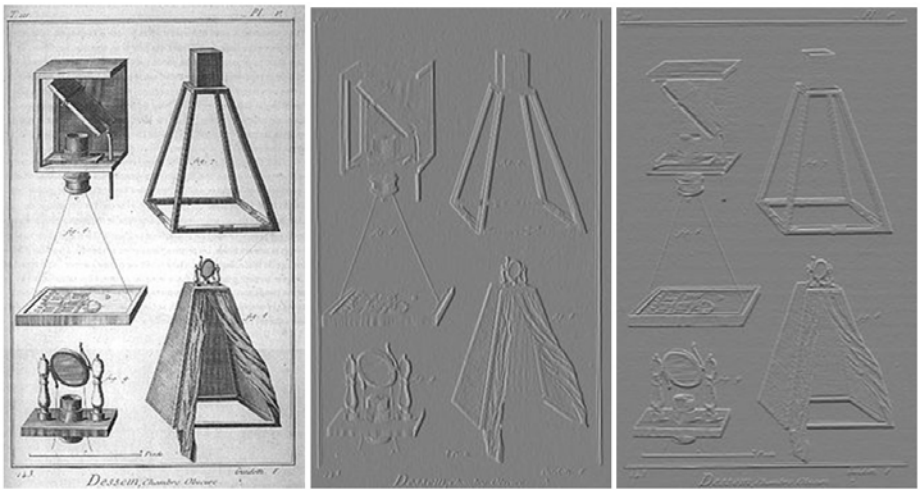
Filter

Die Kernelmatrizen können bei der Faltung als eine Art Filter benutzt werden. Durch Filter können Bildelemente hervorgehoben werden. So können Kernelmatrizen zur Erkennung von Kanten verwendet werden, zum Beispiel mittels des Sobel-Operators (Funktion 2.57). Bei dem Sobel-Operator werden die senkrechten und horizontalen Kanten separat mit den Sobel-Operatoren $\vec{S}_x$ und $\vec{S}_y$ erkannt. Dabei werden die Sobel-Operatoren als eine kleine Gewichtungsmatrizen genutzt. Die entstehenden Matrizen sind 3×3 große Matrizen (Funktion 2.58).

$$\vec{S}_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}, \; \vec{S}_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{2.57}$$

$$\vec{S}_x = \begin{bmatrix} 1 & 0 & 1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \; \vec{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.58}$$

Mit S_x und S_y werden schon zwei Kernelmatrizen verwendet, die zwei unterschiedliche Ergebnismatrizen erzeugen. Die Sobel-Operatoren markieren in der Ergebnismatrix einen Pixel, der keine Kante darstellt, mit dem Wert Null. Eine Kante von dunkel zu hell wird mit positiven Werten und eine Kante von hell zu dunkel wird mit negativen Werten dargestellt. Bei der Abbildung 2.26 wurde der Nullwert als mittleres Grau dargestellt.



Quellen: https://commons.wikimedia.org/wiki/File:Camera_obscura.jpg
https://commons.wikimedia.org/wiki/File:Camera_obscura-Sobel-Horizontal.jpg
https://commons.wikimedia.org/wiki/File:Camera_obscura-Sobel-Vertikal.jpg

Abbildung 2.26: Originalbild, gefaltet mit vertikalem und horizontalem Sobel.

Die Verwendung von kleinen, speicherschonenden Kernelmatrizen ermöglicht es mehrere Kernelmatrizen auf eine Eingabe anzuwenden. Dadurch können mehrere wichtige Bildelemente auf einmal herausgestellt werden. Diese Kernelmatrizen eines Layers besitzen die gleiche Größe. Die Kernelmatrix wird so bei einem einkanaligem Eingabebild dreidimensional. Die Ergebnismatrizen werden als sogenannte Merkmalskarten in einer einzigen Ergebnismatrix pro Layer gespeichert.

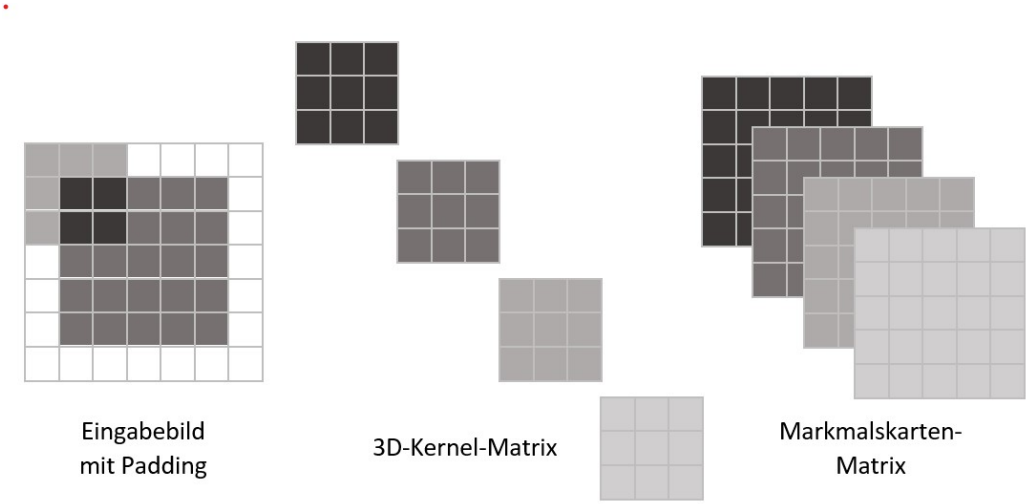


Abbildung 2.27: Dimensionalität der Matrizen bei einem Bild mit einem Farbkanal.

Filter bei Bildern mit mehreren Farbkanälen

Ist das Eingabebild ein Farbbild, besitzt es drei Farbkanäle. Für jeden Filter erhalten die drei Farbkanäle unterschiedliche Kernelmatrizen. Die resultierenden Matrizen der Farbkanäle eines Filters werden summiert (Matrixsummierung) und bilden eine einzelne Merkmalskarte. Diese Merkmalskarten können wiederum als Kanal in der Merkmalskarten-Matrix angesehen werden. Ebenso wird die Kernelmatrix vierdimensional.

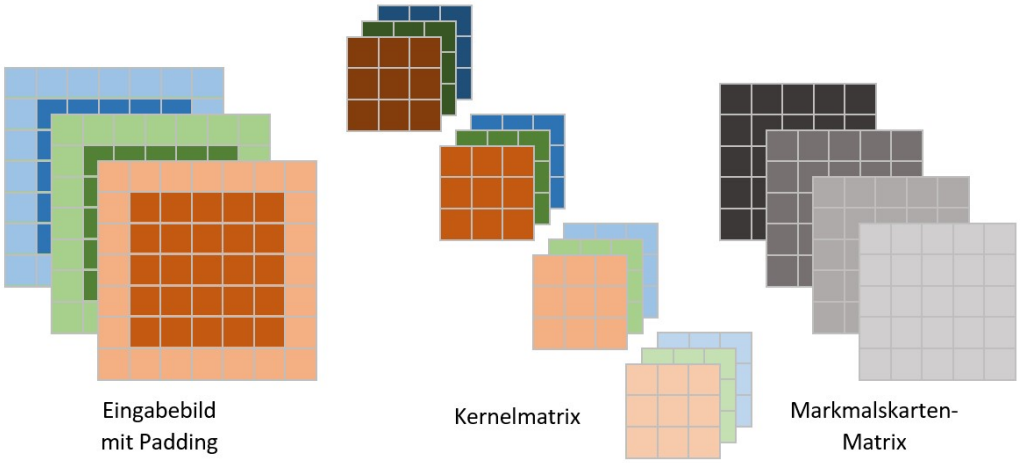


Abbildung 2.28: Dimensionalität der Matrizen bei einem Bild mit einem Farbkanal.

Anzahl an Kanäle mit 1x1-Faltungen minimieren

In den verdeckten Schichten ist die Tiefe, also die Anzahl an Kanälen der Eingabematrizen abhängig von der Anzahl der genutzten Filter im vorangegangenen Schritt. So kann die Tiefe der Eingabematrix der verdeckten Schicht sehr groß werden. In der aktuellen verdeckten Schicht werden wiederum alle Filter auf alle Kanäle der Eingabematrix angewendet und die Ergebnismatrix der verdeckten Schicht wächst weiter. Ebenso wird die vierdimensionale Kernelmatrix immer größer. Um diesem Wachstum entgegenzuwirken, können 1x1-Faltungen genutzt werden. Eine 1×1-Faltung ist eine Kernelmatrix mit Abmessung dargestellt in Funktion 2.59.

$$\begin{aligned} &1 \times 1 \times m \times n \\ &m = \textit{Anzahl der Eingabekanäle} \\ &n = \textit{Anzahl der Ausgabekanäle} \end{aligned} \tag{2.59}$$

Die 1×1-Faltung kann dazu benutzt werden, um eine tiefe Merkmalskarten-Matrix auszudünnen. Mit der Faltungsoperation werden die Kanäle zu einer oder wenige Merkmalskarten zusammengerechnet. Jeder Pixel eines Kanals wird mit den Pixeln der anderen Kanäle an gleicher Stelle gefaltet. In Abbildung 2.29 sind die Minimierungen zweier Merkmalskarte mit drei Dimensionen dargestellt. Die Merkmalskarten werden durch die Faltungsoperation zu einer eindimensionalen Merkmalskarte.

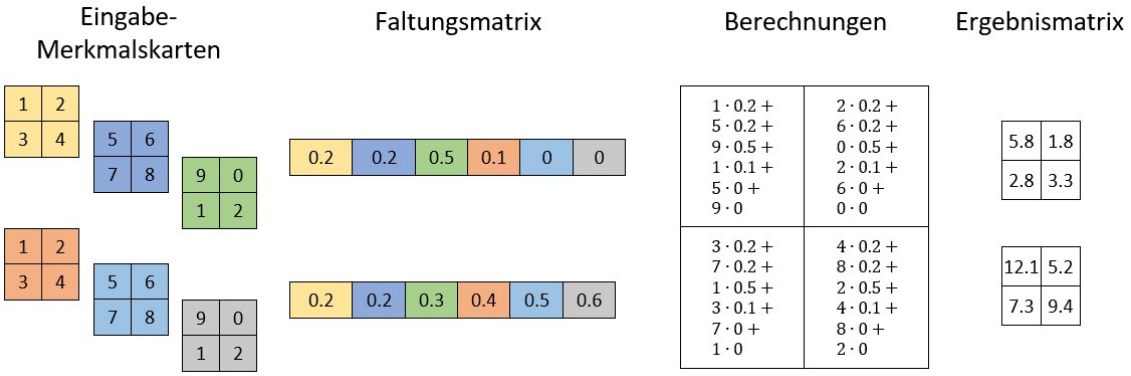


Abbildung 2.29: Merkmalskarten nach einer 1×1 Faltung.

Durch 1×1-Faltungen als Zwischenschritt zwischen einer Eingabematrix und einer größeren Faltungsmatrix, wie zum Beispiel einer 5x5-Faltung, kann die Anzahl von Gewichtungen stark minimiert werden. Im Beispiel der Tabelle 2.9 sind die Matrizengrößen einer einfachen Faltung dargestellt. Die angegebenen Matrizen benötigen insgesamt 1.536.000.000 Gewichtungen (2.60).

Eingabematrix	Filtermatrix	Ergebnismatrix
$100 \times 100 \times 192$	$5 \times 5 \times 192 \times 32$	$100 \times 100 \times 32$

Tabelle 2.9: Matrizengrößen ohne 1x1-Faltung

$$(100 \cdot 100 \cdot 32) \cdot 192 \cdot (5 \cdot 5) = 1.536.000.000 \tag{2.60}$$

Wird dem Beispiel eine 1×1-Faltung wie in Tabelle 2.10 zwischengeschaltet, werden nur noch etwa 1/10 an Gewichtungen benötigt (Funktion 2.61).

Eingabematrix	1×1-Faltung	Faltungsergebnis
$100 \times 100 \times 192$	$1 \times 1 \times 192 \times 16$	$100 \times 100 \times 16$
Filtermatrix	Ergebnismatrix	
$5 \times 5 \times 192 \times 32$	$100 \times 100 \times 32$	

Tabelle 2.10: Matrizengrößen mit 1x1-Faltung

$$\begin{aligned} &((100 \cdot 100 \cdot 16) \cdot 192 \cdot (1 \cdot 1) + ((100 \cdot 100 \cdot 32) \cdot 16 \cdot (5 \cdot 5) \\ &= 30.720.000 + 128.000.000 \\ &= 158.720.000 \end{aligned} \tag{2.61}$$

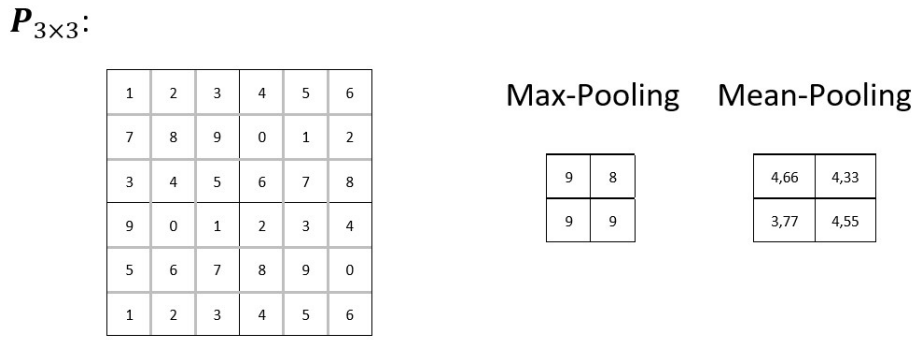
Die Gewichtungsmatrix

Die Nutzung von Filtermatrizen, sogenannten Kerneln, als Gewichtungen reduziert die Gewichtungsmatrix gegenüber einem vollverknüpften Netz. Selbst bei mehreren Filtern bleibt die Anzahl der Gewichtungen geringer als bei einem vollverknüpften Netz. Für das oben erwähnte 100x100 Pixel große Farbbild und 2 Schichten ergaben sich 62.000 Gewichtungen bei Vollvernetzung. Mit der Verwendung von üblichen 200 3x3-Kerneln in jeweils 2 Layern und mit 2 Klasse ergeben sich $3 \cdot 3 \cdot 200 \cdot 2 + 100 \cdot 100 \cdot 2 = 23.600$ Gewichtungen ohne Bias. Die Werte der Filter sind die Gewichtungen der Gewichtungsmatrix. Daher werden die Filter bei dem Training von dem CNN selbstständig erlernt. Somit werden vor dem Training die relevanten Merkmale von Objekten nicht mehr vorgegeben, sondern von dem Netz eigenständig erlernt.

Subsampling / Pooling

Die Verwendung von Filtern führt jedoch dazu, dass das Modell sehr anfällig für Rauschen ist und schon leicht verrauschte Bilder nicht mehr korrekt klassifiziert werden. Dieses Problem wird behoben, indem die Umgebung von Pixeln nach der Faltung in

einer Pooling-Schicht nochmal miteinander verrechnet wird. Gängig sind das Max-Pooling und das Mean-Pooling. Durch das Verrechnen von Pixeln in der Pooling-Schicht wird die Ergebnismatrix kleiner, dargestellt in Abbildung 2.30. Die Größe der verrechneten Umgebung muss vor dem Training bestimmt werden und kann als Hyperparameter angesehen werden.



Angelehnt an: (Raschka & Mirjalili 2018: 500) https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch15/images/15_16.png

Abbildung 2.30: Verrechnung der Werte in einer Pooling-Schicht.

Regularisierung mit Dropout

Ebenso kommt es bei der Verwendung von vielen Filtern schnell zu Overfitting. Das Modell verlässt sich dann zu sehr auf das Ergebnis von wenigen Filtern, die zu sehr auf die Trainingsbilder zugeschnitten sind. Um zu vermeiden, dass das Modell sich zu sehr auf wenige Filter verlässt, können im Training die zu den Filtern gehörenden neuronalen Einheiten ausgeschaltet werden. Die Ausschaltung von neuronalen Einheiten in jedem Trainingsschritt passiert zufällig. Somit kann das Modell sich nicht auf bestimmte Filter festlegen, wenn diese im Training abgeschaltet worden sind. Es ist dann gezwungen weitere nicht so spezialisierte Filter zu trainieren. Ist das Training abgeschlossen, werden zur Klassifikation wieder alle Filter genutzt.

Transfer learning

Das Trainieren eines CNN benötigt wie bei jedem Neuronenmodell viele Trainingsdaten. Diese Trainingsdaten sind im Fall des CNN Bilder. Je nach Anwendung können es auch großformatige Bilder sein. Diese ganzen Trainingsbilder zu speichern, benötigt viel Speicherplatz. Ebenso wird selbst bei guter Hardware viel Zeit benötigt, um das Modell zu trainieren und an die Trainingsbilder anzupassen. Eine Möglichkeit, die Trainingsbilder und die für das Training benötigte Zeit zu minimieren, ist es vortrainierte Modelle zu benutzen und diese nur noch an die eigenen Daten in einem letzten Layer anzupassen.

2.3.4 CNN-Architekturen

Mit der Zeit wurden unterschiedliche CNN-Architekturen entwickelt und mit unterschiedlichen Datensätzen getestet. Einige dieser Architekturen finden sich schon als Funktion in den gängigen ML-Bibliotheken wie TensorFlow. Jede Architektur widmet sich jeweils einem Problem der CNNs. Probleme sind unter anderem lange Trainingszeiten, keine hohe Genauigkeit der Klassifikation oder das Problem des verschwindenden Gradienten.

ResNet

Residual neural networks besitzen Shortcuts über 2 bis 3 Layer. Dies löst das Problem des verschwindenden Gradienten, indem zur Aktualisierung der Gewichtungen der ersten Schichten nicht nur die Schicht genau danach genutzt wird, sondern auch noch die Schichten davor, dargestellt in Abbildung 2.31c. (He & Zhang & Ren & Sun 2022)

Inception

Große Kernelmatrizen werden bei der Inception-Architektur durch mehrere kleine Kernelmatrizen ersetzt. Die kleinen Kernelmatrizen werden parallel gefaltet und danach zu einer einzigen Merkmalskartenmatrix zusammengesetzt, dargestellt in Abbildung 2.32b. Dies verkürzt die Zeit zum Berechnen der Matrizen. (Szegedy & Liu & Jia & Sermanet & Reed & Anguelov & Erhan & Vanhoucke & Rabinovich 2014)

Inception-ResNet

Bei Inception-ResNet werden die Konzepte der Inception- und der ResNet-Architektur miteinander verbunden. In die Inception-Architektur werden wiederholt über mehrere Faltungsoperationen Shortcuts ermöglicht, dargestellt in Abbildung 2.32c. (Szegedy & Ioffe & Vanhoucke & Alemi 2016)

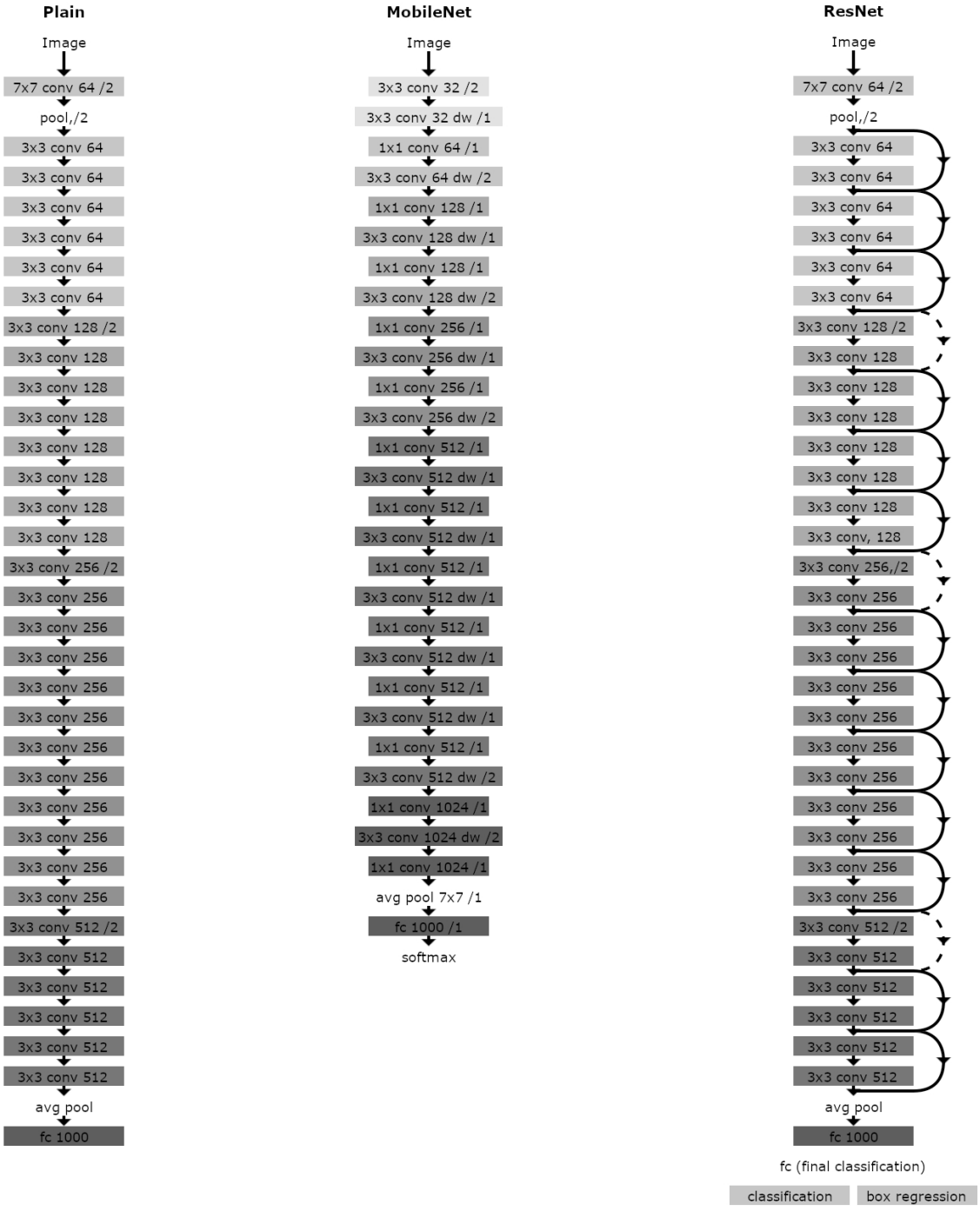
MobileNet

Im MobileNet wurden größere Kernelmatrizen mit der Inception-Methode zu 3×3-Kerneln aufgespalten. Die Merkmalskarten werden jedoch nicht zu einer einzigen Merkmalskartenmatrix zusammengesetzt. Stattdessen werden die Merkmalskarten mit einer 1×1-Faltung zu einer einzigen Merkmalskarte minimiert, dargestellt in Abbildung 2.31b. (Howard & Zhu & Chen & Kalenichenko & Wang & Weyand & Andreetto & Adam 2017)

DenseNet

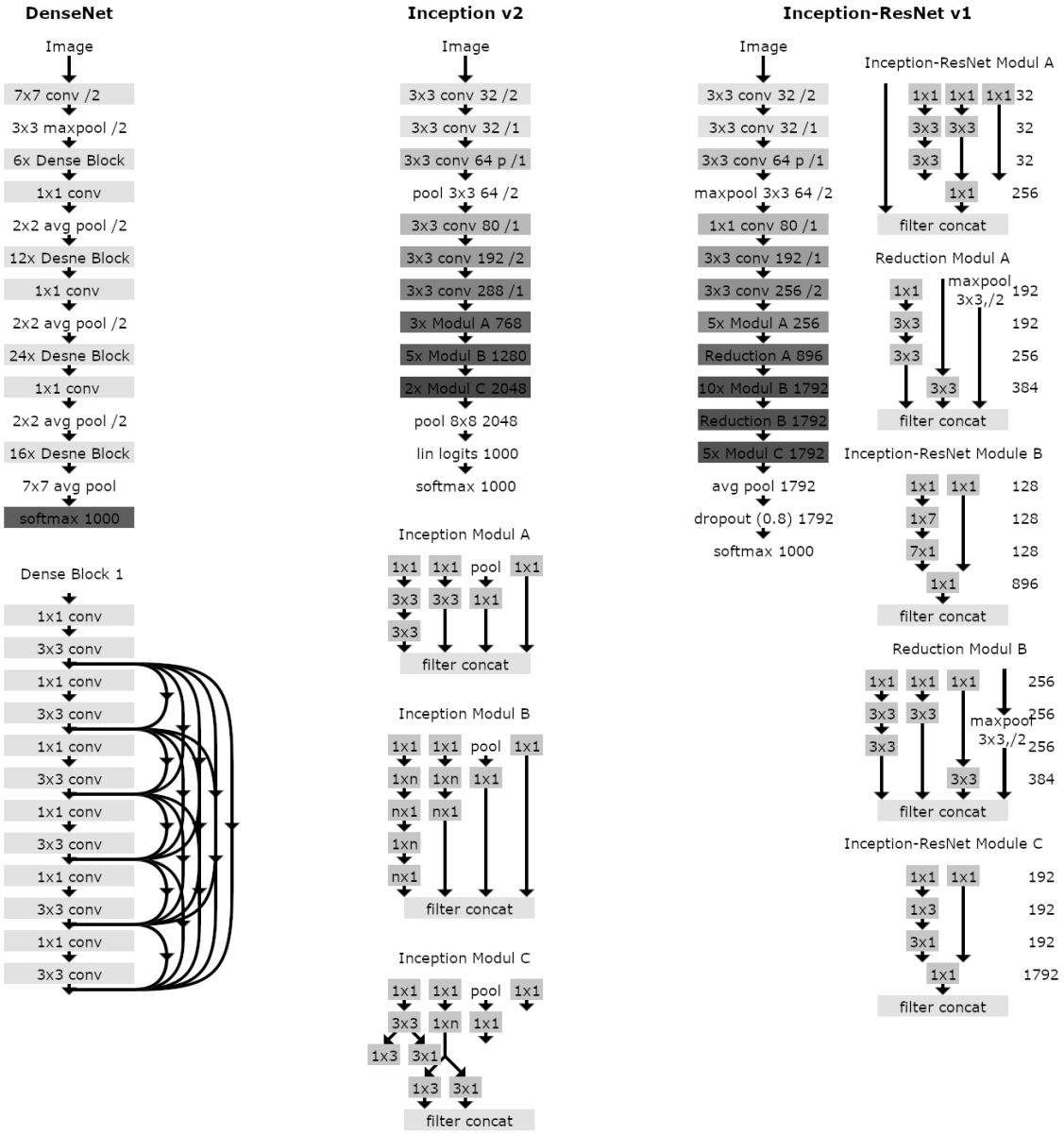
Bei DenseNet werden Shortcuts nicht nur über eine oder zwei Schichten genutzt. Jede Schicht ist mit allen folgenden Schichten verknüpft. Die Merkmalsmatrizen der vorangegangenen Schichten werden in einer Schicht wie bei der Inception-Architektur

zu einer einzigen Merkmalskartenmatrix zusammengefasst. Es werden dabei weniger Filter pro Schicht verwendet, um das Modell dünner und kompakter zu halten, dargestellt in Abbildung 2.32a. Huang & Zhuang & van der Maaten & Weinberger (2018)



Angelehnt an: (He & Zhang & Ren & Sun 2022: 4) (a),(c), (Howard & Zhu & Chen & Kalenichenko & Wang & Weyand & Andreetto & Adam 2017: 4) (b)

Abbildung 2.31: Schema einer einfachen CNN-Architektur (a). Schema der MobileNet-Architektur (b). Schema der ResNet-Architektur (c).



(a) **(b)** **(c)**

Angelehnt an: (Huang & Zhuang & van der Maaten & Weinberger 2018: 4) (a), (Szegedy & Ioffe & Vanhoucke & Alemi 2016) (b), (c)

Abbildung 2.32: Schema der DenseNet-Architektur (a). Schema der Inception v2-Architektur (b). Schema der Inception-ResNet v1-Architektur (c).

3 Tutorials mit unterstützenden Programmierbeispielen

Es gibt viele Tutorials zum Thema ML. Unterschiedliche Anbieter nutzen unterschiedliche Medien und Formen für ihre Tutorials. Es gibt Bücher, Blogs, Online-Tutorials und Videos. Jedes Medium besitzt seine eigenen Vorteile. In Büchern wird stark auf die theoretischen Grundlagen eingegangen, in Online-Tutorials auf die Programmierung der Algorithmen, in Videos auf die Benutzung von Bibliotheken. In diesem Kapitel sollen Live-Tutorials erstellt werden, die die Stärken der einzelnen Medien zusammenbringen. Sie sollen die mathematischen Grundlagen vermitteln und gleichzeitig die Programmierung der Algorithmen beinhalten. Dabei sollen sie durch den Vortrag die Vorteile von Videos nutzen, indem die schriftlichen Inhalte verbal erklärt werden. Die auf die theoretischen Inhalte folgende Programmierung soll es ermöglichen die Algorithmen auf eine andere, nichtmathematische Weise darzustellen. Dies kann die Funktionsweise der Algorithmen verdeutlichen. In den hier erarbeiteten Live-Tutorials werden also nicht nur Bibliotheken genutzt, sondern die Algorithmen erklärt und programmiert.

Zuerst müssen die Themen der Tutorials bestimmt werden. Das Ziel ist ein Tutorial über CNNs. Es müssen also Algorithmen in den Tutorials erläutert werden, auf denen das CNN aufbaut. Anfangen sollen die Tutorials mit einem möglichst einfachen Algorithmus. Dieser kann in den folgenden Tutorials schrittweise verbessert werden. Die Reihenfolge der Algorithmen wird im Kapitel 3.1 erläutert. Die Reihe der Algorithmen hat sich wie folgt ergeben:

3.1.1 Das McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

3.1.2 Die lineare Regression

3.1.3 Das adaptive lineare Neuron

3.1.4 Die logistische Regression

3.1.5 Das mehrschichtige Perzeptron

3.1.6 Das konvolutionale neuronale Netz

Die Theorie- und Praxisteile der Tutorials sollen ausgewogen sein. Die Tutorials zu einem Thema sollen möglichst nicht länger als eine Stunde werden. So sollen der Theorieteil und der Praxisteil jeweils nicht länger als 30 Minuten lang werden. Nach 30 Minuten soll die Präsentationsart vom schriftlichen Erarbeiten der theoretischen

Inhalte zur Programmierung mit Python wechseln. Dies bringt Abwechslung in den Vortrag der Tutorials und hält die Aufmerksamkeit der Zuhörer hoch (ProLehre: 27).

Bevor in den Tutorials die Programmierbeispiele genutzt werden können, müssen die theoretischen Grundlagen der Algorithmen vermittelt werden. Für die Vorträge wurden die theoretischen Inhalte gesammelt und aufbereitet. Die Inhalte wurden in eine präsentable Form gebracht. Zur visuellen Unterstützung wurde zusätzlich ein Zeichentablet und die Notiz-Software OneNote von Microsoft eingesetzt. Die Vorträge wurden eingesprochen, geübt und als Video aufgenommen. Auf die Theorie der Algorithmen wurde in Kapitel 2 eingegangen.

Anschließend wurde zu jedem Algorithmus ein Programmierbeispiel erarbeitet. Der Code der Programmierbeispiele muss in einem Tutorial vorgeführt werden können. Die programmierten Modelle müssen deshalb innerhalb des Live-Tutorials trainierbar und ausführbar sein. Diese Voraussetzungen schränken die Auswahl der Programmierbeispiele stark ein. Die Programmierbeispeiele wurden ebenfalls eingesprochen, geübt und als Video aufgenommen.

Die in dieser Arbeit erstellten Vorträge und Programmierbeispiele wurden im Rahmen von Tutorials in dem Kurs „Adaptive Systems and Artificial Intelligence“(ASaAI) des Studienganges Media Systems an das HAW Hamburg vorgeführt. Dadurch konnten die Tutorials interessierten Personen präsentiert werden. Videoaufnahmen der Tutorials liegen der Arbeit auf einem Speichermedium bei. Die Videos sind für die Öffentlichkeit zusätzlich online unter (Braun 2022) verfügbar.

Die Erstellung der Tutorials wird aufgeteilt in die theoretischen und die praktischen Abschnitte der Tutorials:

3.1 Das Erstellen der Tutorials

Die Tutorials sollen aus zwei Teilen bestehen: den theoretischen Grundlagen der Algorithmen und jeweils einem verdeutlichenden Programmierbeispiel. Vor der Programmierung soll nicht nur oberflächlich auf den Algorithmus eingegangen werden. Es soll auch auf die mathematischen Funktionen eingegangen werden, damit diese nachprogrammiert werden können. Eine schematische Darstellung des Algorithmus kann die grobe Struktur des Codes vorgeben. Die tatsächlichen Rechenschritte müssen jedoch der mathematischen Funktion entnommen werden.

Die Tutorials starten mit einer groben Übersicht über den vorgestellten Algorithmus. Dazu soll möglichst eine grafische Darstellungen der Algorithmen genutzt werden. Diese schematische Darstellung muss recherchiert und erstellt werden. Nach der Übersicht über den Algorithmus wird auf die mathematischen Funktionen eingegangen. Dazu müssen die mathematischen Funktionen gesammelt werden. Zudem müssen

die Algorithmen in einen logischen Zusammenhang gebracht werden, damit sie in den Tutorials aufeinander aufbauen können.

Es folgt die Reihe der logisch aufeinander aufbauenden Algorithmen. Diese Reihenfolge der Algorithmen wurde schon in Kapitel 2 genutzt, um die theoretischen Inhalte der Tutorials in dieser Arbeit zu vermitteln. Auf eine Unterscheidung zwischen Neuronen und Netze wird hier verzichtet.

3.1.1 Das McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

3.1.2 Die lineare Regression

3.1.3 Das adaptive lineare Neuron

3.1.4 Die logistische Regression

3.1.5 Das mehrschichtige Perzeptron

3.1.6 Das konvolutionale neuronale Netz

Zu den theoretischen Grundlagen der Algorithmen wurden Videos erstellt. Dadurch wurde die Vortragszeit ermittelt. Die Vortragszeiten dienen im Kapitel 4 als Grundlage zur Analyse und Diskussion. Die Videos sind unter (Braun 2022) abrufbar.

3.1.1 Das McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

Das Perzeptron stellt den einfachsten Algorithmus dar. Er demonstriert die Funktion der Gewichtungen von Eingabesignalen. Eine Nettoeingabefunktion gefolgt von einer Schwellenwertfunktion führt zur Vorhersage einer Klasse. Mit der Rosenblatt-Lernregel wird die epochenweise Aktualisierung der Gewichtungen eingeführt. Die einzelnen Komponenten des Algorithmus können sehr gut in einem Schema dargestellt werden. Die grundlegende Theorie zum Perzeptron ist in einem Tutorial innerhalb von 20 Minuten vortragbar.

3.1.2 Die lineare Regression

Alternativ kann auch die lineare Regression als Anfangsalgorithmus gewählt werden. Bei der lineare Regression wird die Schwellenwertfunktion durch eine lineare Funktion ersetzt. Zusätzlich kann mit den Werten der linearen Funktion eine Straffunktion erstellt werden. Diese Straffunktion kann mit dem Gradientenabstiegsverfahren als Optimierungsalgorithmus minimiert werden, um die Gewichtungen zu aktualisieren. Dadurch eignet sich die lineare Regression als zweiter Algorithmus. Das Schema des Perzeptrons kann an den Algorithmus der linearen Regression angepasst werden. Die grundlegende Theorie zur lineare Regression kann innerhalb von 16 Minuten vorgetragen werden.

3.1.3 Das adaptive lineare Neuron

Als Verbindung der beiden vorangegangenen Algorithmen kann der Adaline-Algorithmus angesehen werden. Bei Adaline werden die Werte der lineare Funktion zum Aktualisieren der Gewichtungen genutzt. Als Optimierung kann wieder das Gradientenabstiegsverfahren angewendet werden. Zusätzlich zur linearen Regression wird allerdings noch die Schwellenwertfunktion des Perzeptrons genutzt, um die stetigen Werte in Klassenbezeichnungen umzuwandeln. Die Schemas des Perzeptrons und der linearen Regression lassen sich leicht zu einem Schema zusammenführen. Die grundlegende Theorie zum Adaline-Algorithmus ist in einem Tutorial innerhalb von sieben Minuten vortragbar.

3.1.4 Die logistische Regression

Adaline kann zur logistischen Regression verbessert werden, indem die lineare Aktivierungsfunktion durch die logistische Funktion ersetzt wird. Allerdings muss die Funktionsweise der logistischen Funktion erläutert werden. Dazu muss auf Wahrscheinlichkeiten und Chance eingegangen werden. Die logistische Straffunktion muss ebenfalls erläutert werden. Das Adaline-Schema kann leicht an die logistische Regression angepasst werden. Da die grundlegende Theorie zur logistischen Regression eine Wiederholung der Wahrscheinlichkeiten und einen logischen Aufbau der Funktionen beinhaltet, steigt die Vortragszeit auf etwa 45 Minuten.

3.1.5 Das mehrschichtige Perzeptron

Mit der logistischen Regression sind alle Bestandteile von neuronalen Einheiten vorgestellt worden. Darauf aufbauend kann nun das erste Netz aus neuronalen Einheiten, das Multi-Layer Perzeptron, eingeführt werden. Allerdings muss nun die Vorwärtspropagation über mehrere Ebenen mit mehreren Einheiten pro Schicht erläutert werden. Durch die Vervielfachung der Rechenschritte wird der Rechenaufwand bei einem Rechenbeispiel groß. Dazu kommt noch die Aktualisierung der Gewichtungen der einzelnen Neuronen der einzelnen Schichten bei der Backpropagation. Vorher mussten nur die Gewichtungen der Eingangssignale für ein einzelnes Neuron durchlaufen werden. Bei einem einfachen Rechenbeispiel mit einer versteckten Schicht zwischen Eingabe- und Ausgabeschicht mit 2 Eingabesignalen, 2 versteckten und einer Ausgabeeinheit plus Biaseinheiten müssen bei der Vorwärtspropagation schon zwei Neuronen durchlaufen werden. Bei der Backpropagation kann bei Zuhilfenahme des Computers die Berechnungen für die Schichten vektorisiert berechnet werden. Dennoch muss auch bei der Backpropagation die Aktualisierung für die Ausgabeschicht und die verdeckte Schicht berechnet werden. Erst dann kann bei einer wiederholten Vorwärtspropagation ein besseres Vorhersageergebnis erzielt werden. Zur übersichtlicheren Darstellung des Schemas des MLP müssen die Komponenten zusammengefasst werden. Die Nettoeingabefunktion und die folgende Aktivierungsfunktion können zu einem Modul

zusammengefügt werden. Die Gewichtungen der Neuronen können garnicht mehr als eigene Komponenten dargestellt werden. Das Vortragen der MLP-Grundlagen dauert 62 Minuten.

3.1.6 Das konvolutionale neuronale Netz

Im Gegensatz zur Backpropagation des MLP ist die diskrete Faltung des CNN grafisch einfach darzustellen. Das Padding kann tabellarisch und grafisch dargestellt werden. Ebenso die Auswirkungen von mehreren Filtern auf die Merkmalskarten und die Gewichtungsmatrix. Die 1×1-Faltung und das Pooling kann ebenfalls in tabellarischer Form erläutert werden. Somit sind alle Hauptbestandteile des CNN eingeführt. Auf ein Rechenbeispiel durch die Schichten der Ebenen wird verzichtet, da es dem Beispiel des MLP ähnelt. Die Einführung der CNN-Bestandteile kann innherhalb von 33 Minuten vorgetragen werden.

3.1.7 Zusammenfassung der Themen

Algorithmus	Inhalte
Perzeptron	Funktionsweise des Neurons Nettoeingabe Schwellenwertfunktion Lernregel
lineare Regression	stetige Werte Straffunktion Optimierungsalgorithmus
Adaline	Aktivierungsfunktion
logistische Regression	Wahrscheinlichkeiten log. Aktivierungsfunktion log. Straffunktion
MLP	Layer Feedforward Backpropagation
CNN	diskrete Faltung Padding Filter 1x1-Faltung Pooling

Tabelle 3.1: Themen der Tutorials

3.2 Programmierbeispiele

Zum besseren Verständnis der Algorithmen sollen sie in einem Beispiel programmiert werden. Damit die Algorithmen leichter programmiert werden können, wurden im vorangegangenen Kapitel 3.1 Tutorials zu den mathematischen Grundlagen der Algorithmen erstellt. In diesem Kapitel werden Programmierbeispiele gesucht, ausgeführt, getestet und optimiert.

3.2.1 Voraussetzungen der Programmierbeispiele

Die Programmiersprache der Code-Beispiele soll leicht verständlich sein, um keine zusätzliche Barriere auf dem Weg zum Verständnis der Algorithmen zu sein. Die Performance der Programmiersprache steht im Rahmen der Arbeit und ihrer Tutorials nicht im Vordergrund. Eine kostenlose Verfügbarkeit ist ebenso Voraussetzung. Naheliegend ist die Verwendung von Python: Python ist leicht zu lernen und zu lesen. Die Programmiersprache besitzt effiziente Bibliotheken (Raschka & Patterson & Nolet 2020). Speziell für ML besitzt Python GPU-basierte Bibliotheken wie PyTorch und CUDA (Meta 2021) (Nvidia developer 2022). Daher werden die Beispiele in Python mit zusätzlichen Bibliotheken programmiert. Python ist eine leicht verständliche Sprache. Durch Python-Notebooks können bei Vorträgen Codefragmente einzeln ausgeführt werden.

In diesem Kapitel werden Codebeispiele anderer Autoren genutzt und auf ihre Eignung für ein Live-Tutorial überprüft und gegebenenfalls angepasst. Allerdings müssen die Codes zum Vortragen in einem Tutorial geeignet sein. Die Programmierbeispiele sollen nicht nur Funktionen von ML-Bibliotheken nutzen. In den Beispielen sollen die Algorithmen nachprogrammiert werden. Die Algorithmen werden dadurch auf eine weitere Art dargestellt. Dies bringt die Funktionalität der Algorithmen hervor und vereinfacht das Verständnis der Algorithmen.

Kriterien zur Auswahl der Codes sind folgende:

1. Programmiersprache Python
2. Einfacher Code
3. Programmierung der Algorithmen
4. Kurze Trainingsdauer der Modelle

Zur Verfügung stehende Codes können aus Online-Tutorials, Video-Tutorials oder aus Büchern stammen. Online-Tutorials sind auf den Webseiten der Bibliotheken zu finden (Meta 2022), (PyTorch 2021), (TensorFlow 2021). Ebenfalls bieten viele Blogs Tutorials an (Brownlee 2021). Manche Anbieter haben sich auf Tutorials spezialisiert (Tutorials Point 2021). Video-Tutorials bieten sie kostenpflichtig an. Über YouTube ist eine riesige Anzahl an Video-Tutorials verfügbar (Stand: 01.03.2022). Amazon listet über 2000 Bücher zum Thema Machine Learning mit Python auf (Stand:

01.03.2022). Viele Blog-Tutorials und Videos thematisieren allerdings nur die Anwendung der Bibliotheken. Die Algorithmen werden nur oberflächlich beleuchtet. Die Programmierung der Algorithmen wird von Tutorial-Anbietern und in Büchern dargestellt, beispielsweise die logistische Regression (Tutorials Point). Diese Codes können als Vorlage für die Tutorials dieser Arbeit dienen. Sie müssen noch auf ihre Eignung für ein Live-Tutorial überprüft werden. Das heißt, sie müssen nachprogrammiert und ausgeführt werden. Bei der Ausführung der Codes werden die programmierten Modelle trainiert. Die Trainingsdauer muss überprüft werden. Ebenso muss die Programmierung nachvollzogen und gut erklärbar sein. Das Vorführen der Programmierung sollte den zeitlichen Rahmen eines Tutorials nicht übersteigen. Bei einer Vorführung in einer Unterrichtseinheit sollte die Programmierung im Tutorial nicht länger als eine halbe Stunde dauern. Fallen schon bei der Programmierung Unstimmigkeiten auf, muss der Code an sich umprogrammiert werden. Wenn die Trainingsdauer zu lange ist, muss der Code optimiert werden. Die Trainingsdauer der Modelle wird gemessen, indem die Zeit zwischen Start und Ende des Algorithmus berechnet wird. Es wurde der Durchschnittswert aus zehn Trainings gebildet. Es können Hyperparameter geändert werden oder die Trainingsdaten gestutzt werden, um die Trainingszeit zu verkürzen. Eventuell stellt sich ein Code auch als unzureichend heraus und es muss nach einem alternativen Beispielcode für die Tutorials gesucht werden.

3.2.2 Entwicklungsumgebung und genutzte Hardware

Das Trainieren der programmierten Modelle ist neben der Menge der Trainingsdaten und der Auswahl der Hyperparameter ebenfalls sehr von der verwendeten Hardware abhängig. Wird langsamere Hardware als in dieser Arbeit verwendet, kann es auch zu längeren Trainingszeiten führen. Daher wird vom Autor empfohlen, gleichwertige oder bessere Hardware zu verwenden. Zum Testen und Vorführen der Programmierbeispiele wurde ein Computer mit einem Intel Core i9-9900KF Prozessor benutzt. Der Prozessor besitzt eine Grundtaktfrequenz von 3,60 GHz. Zur Programmierung eines CNN wird die TensorFlow-Bibliothek für Python benutzt. TensorFlow kann mit Hilfe von Nvidia CUDA auf der Grafikkarte betrieben werden. Durch die Parallelisierung der Programmabläufe kann die Grafikkarte schneller als die CPU arbeiten (Nvidia developer 2022). Die verwendete Grafikkarte ist eine Nvidia Geforce RTX 2080 Ti mit 1,35 GHz Grundtaktfrequenz. Wird eine AMD-Grafikkarte betrieben, kann nicht Tensorflow mit CUDA verwendet werden. Eine AMD-Grafikkarte kann unter Linux mit der Python-Bibliothek PyTorch verwendet werden. Die Arbeit wurde hingegen auf einem Windows-Betriebssystem erstellt. Die benutzte Python Version ist 3.8.8.

3.2.3 Künstliche Neuronen

Künstliche Neuronen gehören noch zu den einfachen Algorithmen des maschinellen Lernens. Die Programmierung lässt sich leicht umsetzen, insofern nicht auf Performance bei großen Datenmengen geachtet wird. Die programmierten Beispiele sollen

aufeinander aufbauen. Der Code eines vorangegangen Algorithmus soll möglichst bei der Programmierung des folgenden Algorithmus genutzt und umprogrammiert werden.

Die verwendeten Codes müssen im Rahmen eines Tutorials vorgetragen werden können. Dazu müssen die den oben genannten Herausforderungen erfüllen. Als erstes werden die Codebeispiele aus dem Buch „Machine Learning mit Python und Scikit-Learn und TensorFlow“(Raschka & Mirjalili 2018) auf ihre Eignung überprüft. Die Beispiele sind leicht verständlich. Die Algorithmen sind als Klasse geschrieben, sodass sie handzuhaben sind wie Klassen aus importierten Bibliotheken. Damit der Code mit der neuesten Python-Version 3 verwendet werden konnte, musste er aktualisiert werden.

Die Beispiele folgen der bisherigen Anordnung der Algorithmen:

3.2.4 McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

3.2.5 Lineare Regression

3.2.6 Das adaptive lineare Neuron

3.2.7 Die logistische Regression

Zu den Programmierbeispielen wurden ebenfalls Videos erstellt. Dadurch wurde die Zeit ermittelt, die benötigt wird, um die Codes vorzustellen. Die Vortragszeiten dienen im Kapitel 4 als Grundlage zur Analyse und Diskussion. Die Videos sind unter (Braun 2022) abrufbar. Die verwendeten Codes sind im Anhang dieser Arbeit enthalten.

3.2.4 McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel

Für das Programmierbeispiel des Perzeptrons wurde der Code aus (Raschka & Mirjalili 2018: 47) genutzt. Im Beispiel für das McCulloch-Pitts-Neuron mit Rosenblatt-Lernregel soll der Algorithmus als Klasse programmiert werden und die Klasse getestet werden. Zur Programmierung des Algorithmus gehört die Berechnung der Nettoeingabe, sowie die Berechnung der Vorhersage einer Klassenzugehörtigkeit eines Objektes. Ebenso muss die Klasse eine Methode zum Anpassen des Modells an Trainingsdaten besitzen. Dies ist mit der Initialisierung, drei Funktionen und zwei Schleifen in 23 Zeilen Code umgesetzt.

Bei dem Testen der programmierten Klasse muss das Modell mit Trainingsdaten trainiert werden. Nach dem Trainieren kann das Training und das trainierte Modell ausgewertet werden. Trainiert wird das Modell mit dem Iris-Datensatz, einem für Vorschau- und Lehrzwecke oft genutztem Datensatz. Der Datensatz enthält jeweils 50 Messungen zu 3 Arten der Schwertlilie. Gemessen wurde die Breite und die Länge des Kelchblatts und des Blütenblatts. Zur einfacheren zweidimensionalen Visualisierung des Ergebnisses wird das Modell jedoch nur mit zwei Merkmalen trainiert. Die Messergebnisse der Kelchblattlänge und Blütenblattlänge der Iris Setosa und der Iris

Versicolor sind linear trennbar. Daher werden diese beiden Merkmale der beiden Arten genutzt, um das Finden einer Entscheidungsgrenze zu demonstrieren. Nach dem Importieren der Daten und dem Trainieren des Modells wird der Fortschritt des Lernens anhand der Anzahl der Fehlklassifizierungen pro Epoche als Grafik ausgegeben. Ebenso kann nun die Entscheidungsgrenze zwischen den genutzten Blumenarten in ein Koordinatensystem eingezeichnet werden.

Die durchschnittlich benötigte Zeit zum Trainieren des Perzeptrons mit den zwei Features des Iris-Datensatzes beträgt 0.016 Sekunden. Somit ist die Vorstellung des Codes innerhalb eines Tutorials möglich. Zum Vortrag der theoretischen Grundlagen des Perzeptrons kommt das Vortragen des Programmcodes. Das Vortragen des Codes benötigt weitere 20 Minuten. Somit steigt die für das Perzeptron benötigte Zeit auf 40 Minuten.

3.2.5 Lineare Regression

Für das Programmierbeispiel der linearen Regression wurde der Code aus (Raschka & Mirjalili 2018: 327) genutzt. Bei der Umsetzung der linearen Regression kann der Code des Perzeptrons umprogrammiert werden. Neben dem Klassennamen muss die Berechnung der Vorhersage von einer Klassifizierung zur Wiedergabe des Ergebnisses der Netzeingabefunktion geändert werden. Ebenso muss in der fit-Methode eine Schleife wegfallen und die Aktualisierung der Gewichtungen und die Kostenfunktion neu berechnet werden.

Da die logistische Regression kein Klassifizierer ist, kann sie nicht mehr mit dem Iris-Datensatz trainiert werden. Raschka nutzt in seinem Buch einen Datensatz zum Wohnungswesen von Boston aus dem Jahr 1978 (Raschka & Mirjalili 2018: 321). Mit den Daten zur Raumanzahl und der Wohnungspreise lässt sich mit Hilfe der linearen Regression eine Verbindung zwischen Raumanzahl und Wohnungspreis herstellen. Nach der grafischen Ausgabe der Kosten beim Training lässt sich dieser Zusammenhang im geplotteten Analyseergebnis der logistischen Regression erkennen.

Ohne zweite Schleife in der fit-Methode benötigt das Trainieren des Modells der linearen Regression mit den zwei Features des Datensatz zum Wohnungswesen von Boston aus dem Jahr 1978 nur noch Nanosekunden. Somit ist die Ausführung des Codes innerhalb eines Tutorials möglich. Zu der grundlegenden Theorie der linearen Regression kommt die Vorstellung des praktischen Beispiels. Da bei der Programmierung auf den vorherigen Code zurückgegriffen werden kann, wird nicht mehr so viel Zeit für die Programmierung des Algorithmus benötigt. Somit kann die Programmierung innerhalb von 14 Minuten vorgestellt werden. Die für die lineare Regression benötigte Zeit steigt von 16 Minuten auf 30 Minuten an.

3.2.6 Das adaptive lineare Neuron

Für das Programmierbeispiel des adaptiven linearen Neurons wurde der Code aus (Raschka & Mirjalili 2018: 59) genutzt. Wieder kann auf den Code des vorherigen

Algorithmus zurückgegriffen werden. Änderungen im Code sind der Klassenname, eine neue Aktivierungsfunktion, die Rückkehr zur Vorhersagemethode des Perzeptrons und eine Neuberechnung des Fehlers mit eingefügter Aktivierungsfunktion. Für das Training des Modells kann wieder der Iris-Datensatz genutzt werden.

Das Trainieren des Adaline-Modells mit den zwei Features des Iris-Datensatzes benötigt nur Nanosekunden. Somit ist die Ausführung des Codes innerhalb eines Tutorials möglich. Der Code zur Programmierung und Ausführung des Codes benötigt sieben Minuten. In ihnen wird der Algorithmus umgesetzt und anschließend das Modell trainiert, ebenso die Ausgabe der Kosten pro Epoche und die Ausgabe der Grafik zu der Entscheidungsgrenze zwischen den beiden Blumenarten. Somit verdoppelt sich die Vortragszeit des Perzeptrons auf 14 Minuten.

3.2.7 Die logistische Regression

Für das Programmierbeispiel der logistische Regression wurde der Code aus (Raschka & Mirjalili 2018: 87) genutzt. Änderungen zum vorherigen Code des Adaline-Algorithmus sind der Klassenname, die logistische Aktivierungsfunktion und die logistische Straffunktion zur Berechnung der Kosten. Damit die logistische Funktion nicht öfters berechnet werden muss, wird die Berechnung des Fehlers aufgeteilt. Somit kann das Ergebnis der logistischen Funktion auch zum Berechnen der Kosten genutzt werden. Es darf bei der Klassenvorhersage nicht vergessen werden die Klassen -1 auf 0 und den Schwellenwert von 0 auf 0,5 zu setzen. Wird dies nicht gemacht, werden sämtliche Objekte als Klasse 1 vorhergesagt. Bei der Umprogrammierung des vorherigen Adaline-Codes kann schnell etwas vergessen werden. Eine Code-Vorlage mit deutlicher Hervorhebung der Änderungen ist bei dem Vortragen des Codes nützlich gewesen.

Die zum Trainieren des Modells der logistischen Regression mit den zwei Features des Iris-Datensatzes benötigt nur Nanosekunden. Somit ist die Ausführung des Codes innerhalb eines Tutorials möglich. Insofern der Code für die logistische Regression ohne Fehler programmiert worden ist, werden nur neun Minuten zum Präsentieren des Codes benötigt. Die für die logistische Regression benötigte Zeit steigt von 45 Minuten nur noch auf 54 Minuten an.

3.2.8 Künstliche neuronale Netze

Netze aus künstlichen Neuronen sind komplexer als einfache neuronale Klassifizierer. Es müssen nicht mehr nur die Gewichtungen einer einzelnen neuronalen Einheit aktualisiert werden, sondern die Gewichtungen von mehreren Einheiten in mehreren Schichten zueinander. Auch die Programmierung von diesen Berechnungen ist nicht mehr in so wenigen Zeilen Code umzusetzen. Doch soll zum besseren Verständnis der Algorithmen möglichst viel selbst programmiert werden. Daher wird das erste neuronale Netz, das MLP, noch mit Python programmiert. Da anzunehmen ist, dass

das Trainieren der Modelle viel Zeit benötigt, muss das Training bezüglich der Hyperparameter und Trainingsdaten optimiert werden. Spätestens bei dem CNN wird TensorFlow für Python benutzt. Dadurch ist es möglich das Training auf der Grafikkarte berechnen zu lassen.

Es folgen die beiden letzten Programmierbeispiele:

3.2.9 Das mehrschichtige Perzeptron

Der Code zur Programmierung des MLP ist wieder aus dem Buch Machine Learning mit Python und Scikit-Learn und TensorFlow (Raschka & Mirjalili 2018: 399) entnommen. Der Code musste auf die neueste Python-Version aktualisiert werden. Er besteht aus 74 Zeilen Code ohne Kommentare. In ihm wird nicht nur die Aktualisierung der Gewichtungen über eine Schicht, sondern auch die One-Hot-Codierung und das Minibatch-Verfahren erläutert. Weiterhin wird eine sigmoide Aktivierungsfunktion genutzt. Der Code ist für die Verwendung des MNIST-Datensatzes geschrieben. Daher basiert die One-Hot-Codierung nur auf Zahlen. In der fit-Methode wird in einer ersten Schleife die Epochen und in einer zweiten Schleife die Trainingsobjekte der Minibatches durchlaufen. In der Schleife werden die Eingabesignale durch das Netz vorwärtspropagiert. Die Vorwärtspropagation wird auch zur Evaluation der Epochen benötigt und zur letztendlichen Klassifikation der Objekte. Daher ist die Vorwärtspropagation in eine eigene Funktion ausgelagert. In ihr wird die Nettoeingabe und die Aktivierungsfunktion der verdeckten Schicht und die der Ausgabeschicht berechnet, ähnlich der Schleife der einfachen neuronalen Klassifizierer. In der zweiten Schleife des MLP erfolgt nur noch die Aktualisierung der Gewichtungen. Zur Evaluation der Epochen wird nach der Gewichtungsaktualisierung die Korrektklassifizierungsrate vor und nach der Aktualisierung berechnet und ausgegeben.

Der MNIST-Datensatz besteht aus 60.000 Bildern von handgeschriebenen Zahlen. Die Bilder sind 28 x 28 Pixel groß. Für das Training werden im Buch 55.000 Bilder und zur Validierung 5000 Bilder genutzt (Raschka & Mirjalili 2018: 406). Die für das Training benötigte Zeit für 200 Epochen beträgt 4 Minuten und 18 Sekunden. Damit ist der Originalcode in einem Live-Tutorial nicht vorführbar. Der Code muss verändert werden, sodass das Training des MLP etwa eine halbe Minute dauert. Mit der Minimierung der Epochen von 200 auf 100 Durchläufe und dem Beschneiden der Trainingsdaten auf 20.000 Bilder lässt sich das Training auf 45,815 Sekunden verkürzen. Damit ist das Trainieren des MLP in einem Live-Tutorial möglich. Das Vorstellen des langen Codes benötigt allerdings viel Zeit. Zu den 60 Minuten Theorie kommen weitere 60 Minuten zur Programmierung des MLP. Damit steigt die für das MLP benötigte Zeit auf 120 Minuten.

3.2.10 Das konvolutionale neuronale Netz

Da die Programmierung einer Klasse für das MLP sehr zeitaufwändig war, wird bei dem CNN darauf verzichtet. Bei der Programmierung des CNN soll nun eine Bibliothek genutzt werden. Zur Verfügung stehen TensorFlow, Keras und PyTorch. Als erstes wurde versucht mit TensorFlow zu arbeiten. In im Internet verfügbaren Tutorials werden jedoch oft veraltete TensorFlow-Versionen genutzt. Eine Aktualisierung der Codes hat sich als schwierig herausgestellt. Der ergebende Code eignet sich durch die Nutzung des compat.v1-Moduls nicht zur Vorführung. Viele Methoden werden durch unverständliche Kompatibilitäts-Methoden ersetzt. Dadurch wird der Code und seine Funktion unübersichtlich. Für ein Einführungstutorial ist das compat.v1-Modul nicht geeignet. Nach langer Recherche wurde jedoch noch ein Code der Tensorflow-Version 2 gefunden. Der Code ist übersichtlich und leicht verständlich. Es muss nicht auf eine andere ML-Bibliothek ausgewichen werden.

Im Programmierbeispiel wird nun TensorFlow verwendet. Dadurch wird die Programmierung der einzelnen Schicht vereinfacht und die Anordnung der verschiedenen Layer kann in den Vordergrund treten. Der Code basiert auf dem TensorFlow Tutorial Image classification der TensorFlow-Website (TensorFlow 2022). Die Reihenfolge der Codefragmente wurde für das Tutorial allerdings geändert, damit zuerst das Modell erstellt wird. Danach erst wird der Datensatz importiert. Zum Erstellen des Modells werden nur noch die einzelnen Layer des Sequential-Model von TensorFlow arrangiert. Allerdings wird zum Erstellen des Modells die Anzahl der Klassen benötigt. Daher folgt der Import des Datensatzes. Der Datensatz enthält 3670 Bilder unterschiedlicher Größe. Bei dem Import werden sie auf 180x180 Pixel zugeschnitten. Nach dem Import werden die Daten standardisiert und vervielfältigt, um eine größere Trainingsdatenmenge zu erhalten. Mit der nun bekannten Anzahl der Klassen kann das Modell kompiliert und trainiert werden.

Das Training über 15 Epochen mit TensorFlow benötigt im Durchschnitt 21,32 Sekunden. Damit ist das Trainieren des CNN in einem Live-Tutorial möglich. Durch die Nutzung der Bibliotheken schrumpft der programmiererische Aufwand und das Programmierbeispiel kann in 10 Minuten vorgetragen werden. Damit steigt die fr das CNN-Tutorial benötigte Zeit auf 43 Minuten an.

4 Evaluation

Zur Evaluation der Tutorials und der Programmierbeispiele werden in diesem Kapitel die Erfahrungen aus dem Erstellen der Tutorials zusammengefasst. Anschließend folg die Evaluation der erstellten Tutorials.

4.0.1 Zusammenfassung der Tutorials

Bei einfachen neuronalen Algorithmen ist die Länge der Tutorials noch kurz. Die zum Vortragen benötigte Zeit liegt um die 20 Minuten. Die theoretischen Grundlagen zur logistischen Regression können allerdings nicht durch Code unterstützt werden. Mit Erstellung der Aktivierungsfunktion und der Straffunktion steigt die benötigte Zeit schon auf etwa 45 Minuten. Die Programmierung hingegen ist in wenigen Minuten umzusetzen. Die für die Theorie des MLP benötigte Zeit steigt weiter auf etwa eine Stunde. Die Programmierung des MLP als Klasse ohne Nutzung von Bibliotheken beträgt eine weitere Stunde. Die MLP-Tutorials sind damit ebenfalls zu lang und müssen geändert werden. Die Theorie des CNN mit der Faltungsoperation übersteigt nur minimal den gewünschten Zeitrahmen. Durch die Verwendung von TensorFlow und Keras bei der Programmierung des CNN kann die Vortagszeit gegenüber dem MLP stark verkürzt werden. Die Vortragszeiten der einzelnen Tutorialbeiträge sind in der Tabelle 4.1 zusammengestellt.

Algorithmus	**durchschnittl. Training (sec)**	**Theorie (min)**	**Programmierung (min)**
Perzeptron	0.016	19.5	22.3
Lin. Regression	0.0	16.0	14.0
Adaline	0.0	7.0	7.0
Log. Regression	0.0	44.2	8.3
MLP	45.815	62.3	60.3
CNN	21.32	33.3	10.0

Tabelle 4.1: Übersicht über die Vortragsdauer der Tutorials

4.0.2 Evaluation der Tutorials

Die einfachen neuronalen Algorithmen lassen sich innerhalb von kurzen Tutorials erläutern. Programmierbeispiele können zur Unterstützung der theoretischen Inhalte eingesetzt werden.

Das Tutorial zu den theoretischen Grundlagen der logistischen Regression ist zu lang. Es übersteigt die gewünschten 30 Minuten für diese Vortragsmethode. Es muss sinnvoll gekürzt werden. Eine Möglichkeit der Kürzung ist die Entnahme der Wiederholung der Wahrscheinlichkeiten. Dadurch werden etwa fünf Minuten Vortragszeit gespart. Die Wiederholung kann separat und optional angeboten werden. Als weitere Kürzung kommt das Gradientenabstiegsverfahren für die logistische Straffunktion in Frage. Das Gradientenabstiegsverfahren der logistischen Regression ist zu dem Gradientenabstiegsverfahren der linearen Regression gleich. Daher kann eine andere Herleitung der Gewichtungsaktualisierung gewählt werden. Es könnte darauf hingewiesen werden, dass bei der Aktualisierung der Gewichtungen nur der Wert der Aktivierungsfunktion wichtig ist. Die Art der Aktivierungsfunktion ist hingegen irrelevant. Dadurch könnten weitere acht Minuten gespart werden. Damit wäre das Tutorial zu den theoretischen Grundlagen der logistischen Regression nur noch 32 Minuten lang. Diese Vortragszeit ist wieder akzeptabel.

Die Tutorials zu den theoretischen Grundlagen des MLP und die Programmierung des MLP sind zu lang. Beide übersteigen die 30 Minuten pro Vortragsmethode. Bei den theoretischen Grundlagen wird ein Beispielnetz durchgerechnet. Die Vorstellung kann nicht einfach verkürzt werden, ohne Teile des Algorithmus auszulassen. Nur durch das Auslassen des Rechenbeispiels könnten die Inhalte kompakter wiedergegeben werden. Doch werden durch das Rechenbeispiel die Schritte des Algorithmus sehr gut deutlich. Für das Rechenbeispiel spricht auch die parallele Berechnung von Python. Dadurch wird die Vortragsart abwechslungsreicher. Es kann überlegt werden, ob dieses Tutorial nicht so gelassen wird.

Die Programmierung des MLP kann gekürzt werden, da die einzelnen Schritte des Algorithmus schon bei dem Rechenbeispiel vorgeführt worden sind. Wichtig sind die Einführung des Mini-Batch-Verfahrens und das Anwendungsbeispiel. Damit stehen zwölf Minuten fest. Wie sehr der Vortrag der MLP-Klasse dazwischen komprimiert werden kann, muss in einem weiteren Schritt ausgetestet werden. Anstatt einer Komprimierung der Inhalte könnte wie bei dem CNN für die Programmierung des MLP eine Bibliothek genutzt werden. Dies hat jedoch den Nachteil, dass die Programmierung den Algorithmus nicht mehr verdeutlichen würde. Damit würde ein Ziel dieser Tutorials nicht mehr erreicht werden.

Über die technische Anwendbarkeit von Programmierbeispielen in Tutorials hinaus sollte zudem der Nutzen der Programmierbeispiele analysiert und evaluiert werden. In dieser Arbeit wurde davon ausgegangen, dass Anwendungsbeispiele positive Effekte auf den Lernerfolg haben. Diese Behauptung müsste bestätigt oder widerlegt werden. Dazu müssten die Tutorials mit Programmierbeispielen einem größeren Publikum vorgeführt werden. Die Tutorials sollten an einem größeren Publikum getestet werden. Feedback muss besser gesammelt werden.

4.0.3 Ausblick

Die Kürzungen der Tutorials müssen umgesetzt, getestet und analysiert werden. Zur Bestätigung oder Widerlegung der Behauptung, dass Programmierbeispiele einen positiven Effekt auf den Lernerfolg haben, müssen die Tutorials an einem größeren Publikum getestet werden. Anhand anschließender Tests könnte der Lernerfolg gemessen werden. Feedback kann mit Online-Befragungen gesammelt werden.

Um die Grundlagen von ML-Algorithmen zu disseminieren, sollten die aufgezeichneten Tutorials über weitere Kanäle verbreitet werden, damit mehr interessierte Personen darauf Zugriff erhalten. Weitere Kanäle können eine eigene Website sein oder ein YouTube-Kanal sein.

Zudem könnten Tutorials zu weiteren Algorithmen erstellt werden. Rekurrentes Lernen mit Shortcuts zu vorherigen Layern kann an das MLP-Tutorial angeknüpft werden. Abseits von neuronalen Algorithmen kann auf andere Algorithmen zur Klassifikation, Regression und zum Clustering eingegangen werden. So können die in Kapitel 2.1.1 vorgestellten Algorithmen in neuen Tutorials erläutert und mit Programmierbeispielen unterstützt werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Alpaydin: „Maschinelles Lernen“, *De Gruyter Oldenbourg*, 2. Aufl., 2019

Aunkofer: „Maschinelles Lernen: Parametrisierte und nicht-parametrisierte Verfahren“, *data-science-blog*, https://data-science-blog.com/blog/2018/02/11/maschinelles-lernen-parametrisierte-und-nicht-parametrisierte-verfahren/, 2018, letzter Zugriff: 16.02.2022

Braun: Progammierbeispiele https://cloud.haw-hamburg.de/index.php/s/oFBp9PN6M6wQxj4, 2022, letzter Zugriff: 25.1.2022, Passwort: 2022

Brownlee: *Machine Learning Mastery*, https://machinelearningmastery.com/perceptron-algorithm-for-classification-in-python/, letzter Zugriff: 03.03.2022

Brownlee: *Perceptron Algorithm for Classification in Python*, https://machinelearningmastery.com/category/python-machine-learning/, letzter Zugriff: 01.03.2022

He & Zhang & Ren & Sun: „Deep Residual Learning for Image Recognition“, *arxiv*, https://arxiv.org/pdf/1512.03385.pdf, 2015, letzter Zugriff: 7.2.2022

Howard & Zhu & Chen & Kalenichenko & Wang & Weyand & Andreetto & Adam: „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications“, *arxiv*, https://arxiv.org/pdf/1704.04861.pdf, 2017, letzter Zugriff: 7.2.2022

Huang & Zhuang & van der Maaten & Weinberger: „Densely Connected Convolutional Networks“, *arxiv*, https://arxiv.org/pdf/1608.06993.pdf, 2015, letzter Zugriff: 10.2.2022

Loviscach: *Mensch-Maschine-Interaktion: Themen und Termine*, https://j3l7h.de/lectures/2122ws/Mensch-Maschine-Interaktion/ThemenUndTermine.html, letzter Zugriff: 03.03.2022

McCulloch & Pitts: „A Logical Calculus of the Ideas Immanent in Nervous Activity“, *The bulletin of mathematical biophysics* 5(4):115-113, 1943, https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf, 2014, letzter Zugriff: 18.2.2022

Meta: *Tools: PyTorch*, https://ai.facebook.com/tools/pytorch/, letzter Zugriff: 28.02.2022

scikit-learn developers: *scikit-learn Tutorials*, https://scikit-learn.org/stable/tutorial/index.html, letzter Zugriff: 01.03.2022

Nvidia: *Technische Daten der 20er-Serie vergleichen*, https://www.nvidia.com/de-de/geforce/graphics-cards/compare/?section=compare-20, 2022, letzter Zugriff: 07.02.2022

Nvidia developer: *CUDA Zone*, https://developer.nvidia.com/cuda-zone, letzter Zugriff: 20.02.2022

ProLehre: „Grundprinzipien und Erfolgsfaktoren guter Lehre“, *Technische Universität München*, https://www.prolehre.tum.de/fileadmin/w00btq/www/Angebote_Broschueren_Handreichungen/prolehre_erfolgsfaktoren.pdf, 2015, letzter Zugriff: 03.3.2022

PyTorch: *PyTorch Tutorials*, https://pytorch.org/tutorials/, letzter Zugriff: 01.03.2022

Raschka & Mirjalili: „Machine Learning mit Python und Scikit-Learn und TensorFlow“, *mitp*, 2. Aufl., 2018

Raschka & Patterson & Nolet: „Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence“, *arXiv*, https://arxiv.org/pdf/2002.04803.pdf, 2020, letzter Zugriff: 28.02.2022

Rosenblatt: „The Perceptron, a Perceiving and Recognizing Automaton“, *Cornell Aeronautical Laboratory*, 1957, https://blogs.umass.edu/brain-wars/files/2016/03/rosenblatt-1957.pdf letzter Zugriff: 18.02.2022

Szegedy & Ioffe & Vanhoucke & Alemi: „Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning“, *arxiv*, https://arxiv.org/pdf/1602.07261v2.pdf, 2016, letzter Zugriff: 07.02.2022

Szegedy & Liu & Jia & Sermanet & Reed & Anguelov & Erhan & Vanhoucke & Rabinovich: „Going deeper with convolutions“, *arxiv*, https://arxiv.org/pdf/1409.4842v1.pdf, 2014, letzter Zugriff: 07.02.2022

TensorFlow, *Image classification*, 2022, https://www.TensorFlow.org/tutorials/images/classification letzter Zugriff: 20.02.2022

TensorFlow: *TensorFlow-Tutorials*, 2021, https://www.tensorflow.org/tutorials letzter Zugriff: 01.03.2022

Tutorials Point: *Classification Algorithms - Logistic Regression*, https://www.tutorialspoint.com/machine_learning_with_python/classification_algorithms_logistic_regression.htm, letzter Zugriff: 01.03.2022

Tutorials Point: *Machine Learning Courses*, https://www.tutorialspoint.com/search/machinelearning, letzter Zugriff: 03.03.2022

Tutorials Point: *Machine Learning Tutorials*, https://www.tutorialspoint.com/machine_learning_tutorials.htm, letzter Zugriff: 01.03.2022

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Tobias Braun