

# **Thema: Verbesserung der Diagnose mit maschinellern Lernen**

*Masterarbeit*

eingereicht bei

Prof. Dr. Oliver Hinz

Professur für Wirtschaftsinformatik und Informationsmanagement

Fachbereich Wirtschaftswissenschaften

Goethe-Universität Frankfurt am Main

Betreuer: Benjamin Abdel Karim

Studienrichtung: Master Wirtschaftsinformatik  
7. Fachsemester

Abgabetermin: 11.06.2019

## Inhaltsverzeichnis

Abbildungsverzeichnis .....	4
Tabellenverzeichnis .....	5
Abkürzungsverzeichnis .....	6
Symbolverzeichnis .....	8
Vorwort .....	10
Kapitel 1 – Einleitung.....	11
1.1 Abstrakt .....	12
1.2 Problemstellung.....	13
1.3 Forschungsfrage .....	16
1.4 Thematische Abgrenzung .....	17
1.5 Aufbau der Arbeit.....	19
1.6 Notation .....	20
Kapitel 2 – Theoretischer Hintergrund .....	22
2.1 ML-Aufgaben.....	23
2.2 Aufgaben von überwachtem Lernen.....	25
2.2.1 Klassifikation und Regression .....	25
2.2.2 Erweiterung der binären Klassifikation zur Mehrfachklassifikation.....	26
2.3 Einschichtige Netzwerke.....	27
2.3.1 Neuronen als Vorbild für maschinelles Sehen .....	27
2.3.2 MPC-Neuron .....	27
2.3.3 Perzepron.....	28
2.3.4 Formale Definition des Perzeptrons .....	28
2.3.5 Perzeptron-Lernregel .....	29
2.3.6 ADALINE: Adaptive Lineare Neuronen .....	30
2.4 Mehrschichtige Netzwerke .....	33
2.4.1 Vom MLP zum (vanilla) Feed-Forward Netzwerk.....	33
2.4.2 Sequentielle Natur und Vorwärtspropagation.....	34
2.4.3 Anzahl von Parametern.....	35
2.4.4 Tiefe Neuronale Netzwerke.....	36
2.5 Lernmethoden.....	37
2.6 Bausteine von CNNs .....	40
2.6.1 Diskrete Faltung .....	41
2.6.2 Aktivierung .....	44
2.6.3 Pooling.....	46
2.6.4 Vollständig verknüpfte Schicht .....	47
2.6.5 Regularisierungsschicht .....	48
2.6.6 Ausgabeschicht .....	49
2.7 Modellaufbau nach KDD-Vorgehensmodell .....	50
Kapitel 3 – Aufbau eines produktiven ML-Modells .....	51
3.1 ML-Pipeline.....	52
3.2 CNN-Implementierung .....	65
3.3 System und Software-Anforderungen .....	78
Kapitel 4 – Ergebnis und Ausblick .....	79
4.1 Unterschiede moderner CNN-Implementierungen .....	80
4.2 Umgang mit unausgewogenen Datensätzen.....	88
4.3 CXR-Implementierung.....	89
4.4 Produktive ML-Einsatzfelder.....	91

4.5 Modularisierung der ML-Pipeline .....	92
Schlusswort .....	96
Literaturverzeichnis .....	98
Anhang .....	103
Endnoten .....	105

## Abbildungsverzeichnis

Abbildung 1: Von Regelbasierten Systemen bis deep learning (Quelle: Chartrand et al. 2017, S.2115) .....	13
Abbildung 2: Hardware-Übersicht (Eigene Darstellung) .....	14
Abbildung 3: Themenkomplex AI als Venn-Diagramm (Quelle: Goodfellow et al. 2016, S.9).....	17
Abbildung 4: ML-Lernalgorithmen (Eigene Darstellung) .....	23
Abbildung 5: multi-class versus multi-label (Eigene Darstellung) .....	26
Abbildung 6: Perzeptron (Quelle: Raschka/Mirjalili, 2016, S.57) .....	28
Abbildung 7: gewichtete Eingabe [Nettoeingabe] (Eigene Darstellung).....	28
Abbildung 8: optimale Lernrate (Quelle: Raschka/Mirjalili, 2016, S.63).....	30
Abbildung 9: Adaline-Einheit (Quelle: Raschka/Mirjalili, 2016, S.57).....	31
Abbildung 10: Gradientenabstiegsverfahren ( <i>engl. gradient descent</i> ) (Quelle: Raschka/Mirjalili, 2016, S.58) .....	31
Abbildung 11: Kompromiss zwischen Unter- und Überanpassung (Quelle: Raschka/Mirjalili, 2016, S.94).....	33
Abbildung 12: MLP mit zwei verdeckten Schichten (Eigene Darstellung).....	34
Abbildung 13: Vorwärtspropagation (Quelle: Raschka/Mirjalili, 2016, S.415).....	35
Abbildung 14: Merkmalshierarchie .....	36
Abbildung 15: Das Backpropagation-Verfahren (Quelle: Raschka/Mirjalili, 2016, S.418).....	38
Abbildung 16: Einfluss der Lernrate auf die Minimierung der Straffunktion .....	38
Abbildung 17: Einfluss der Merkmalsstandarisierung auf den Gradientenabstieg (Quelle: Raschka/Mirjalili, 2016, S.64) .....	39
Abbildung 18: sequentielle Natur des CNNs (Eigene Darstellung).....	40
Abbildung 19: Convolution zur Bildung von Merkmalskarten (Quelle: Raschka/Mirjalili, 2016, S.490) .....	40
Abbildung 20: Convolution-Block (Eigene Darstellung).....	41
Abbildung 21: eindimensionale diskrete Faltung (Quelle: Goodfellow et al. 2016, S.330).....	42
Abbildung 22: Padding-Strategie (Quelle: Raschka/Mirjalili, 2016, S.495).....	43
Abbildung 23: Pooling-Varianten (Quelle: Raschka/Mirjalili, 2016, S.500).....	46
Abbildung 24: Ein Convolution-Block (CP-Folge) (Quelle: Raschka/Mirjalili, 2016, S.503) .....	47
Abbildung 25: optimale Komplexität (Quelle: Müller/Guido, 2017, S.31) .....	48
Abbildung 26: Zwei Ausgabeschichten; binäre Klassifikation ( <i>links</i> ); multi-class ( <i>rechts</i> ) .....	49
Abbildung 27: KDD-Vorgehensmodell (Eigene Darstellung) .....	50
Abbildung 28: ML-Pipeline nach dem KDD-Vorgehensmodell (Eigene Darstellung).....	52
Abbildung 29: Bias-Varianz-Kompromiss (Vgl. Raschka/Mirjalili, 2016, S.212).....	53
Abbildung 30: Aufteilung des Datensatzes in drei Teilmengen (Quelle: Müller/Guido, 2017, S.246) .....	57
Abbildung 31: Aufteilung (Split) und zurückhalten der Testdaten (Vgl. Raschka/Mirjalili, 2016, S.206) .....	57
Abbildung 32: Daten aufteilen (Eigene Darstellung) .....	58
Abbildung 33: Kreuzvalidierung (Quelle: Müller/Guido, 2017, S.236) .....	59
Abbildung 34: ImageAugmentation visualisieren .....	69
Abbildung 35: Genauigkeit Lernkurve .....	70
Abbildung 36: Zielfunktion ( <i>loss</i> ) Lernkurve .....	70
Abbildung 37: Genauigkeit Lernkurve #2 .....	75
Abbildung 38: Zielfunktion ( <i>loss</i> ) Lernkurve #2 .....	75
Abbildung 39: Gewichtungen grafisch visualisieren .....	77
Abbildung 40: LeNet-Architektur (Quelle: LeCun et al. 1998, S.8).....	81
Abbildung 41: AlexNet-Architektur (Quelle: Krizhevsky et al. 2012, S.5).....	81
Abbildung 42: zunehmende Modellkomplexität .....	83
Abbildung 43: VGGNet Architektur.....	83
Abbildung 44: Das Inception-Modul .....	84
Abbildung 45: InceptionV3 Architektur (Quelle: Szegedy et al. 2014, S.4) .....	85
Abbildung 46: Der Residual-Block (Quelle: He et al. 2015, S.2).....	85
Abbildung 47: SqueezeNet-Kompression auf AlexNet (Quelle: Iandola et al. 2016, S.7).....	86
Abbildung 48: Energiekosten unterschiedlicher Rechenoperationen.....	87
Abbildung 49: CNN unter Einbezug von Text-Merkmalen (Quelle: Baltrushat et al. 2019, S.3).....	90
Abbildung 50: Vortraining und Transferlernen illustriert (Quelle: Chartrand et al. 2017, S.2127) .....	93

## T a b e l l e n v e r z e i c h n i s

Tabelle 1: Übersicht von Aktivierungsfunktionen (Eigene Darstellung) .....	46
Tabelle 2: Wahrheitsmatrix (Eigene Darstellung) .....	63
Tabelle 3: ImageNet ILSVRC (Eigene Darstellung) .....	80
Tabelle 4: CNN Überblick (Eigene Darstellung) .....	82
Tabelle 5: CNN-Architekturen (Eigene Darstellung) .....	85
Tabelle 6: ResNet-Trainings-Dauer .....	87
Tabelle 7: Deep-Learning für CXR (Quelle: Litjens et al. 2017, S.17) .....	89
Tabelle 8: Metriken des sortierten NIH-Datensatzes (Eigene Darstellung) .....	103
Tabelle 9: Hardware im Detailvergleich (Eigene Darstellung) .....	104

## Abkürzungsverzeichnis

**ASIC.** (*engl. application-specific integrated circuit, ASIC*).

**AUC.** (*engl. area under the curve, AUC*).

**batch-size.** Losgröße.

**CIFAR-10.** Bild-Datensatz mit zehn Zielklassen (Klassenbezeichnungen), ähnlich zu MNIST.

**CNN.** (*engl. convolutional neural network*).

**CNTK.** Bibliothek von Microsoft für maschinelles Lernen.

**compile.** Kompilieren (Zusammenbau des Modells bevor das Training beginnen kann).

**Conv.** Convolution-Operation (diskrete Faltung).

**CXR.** (*engl. Chest X-Ray*).

**DLSS.** (*engl. deep learning supersampling*).

**FC-Schicht.** vollständig verknüpfte Schicht (*engl. fully connected layer*).

**fit.** Trainieren des Modells (nach dem kompilieren).

**FLOPS.** Fließkommaoperationen pro Sekunde (*engl. floating point operations per second, FLOPS*).

**FPGA.** (*engl. field-programmable gate array, FPGA*).

**IDG.** Keras-Methode für Augmentation (Vorverarbeitung) von Bildern (ImageDataGenerator).

**ILSVRC.** (*engl. ImageNet Large Scale Visual Recognition Challenge*).

**Iris-Datensatz.** Schwertlilien-Datensatz (*engl. iris dataset*).

**KDD.** Vorbild der ML-Pipeline (*engl. knowledge discovery in databases*).

**Keras.** Eine High-Level-Bibliothek für ML-Bibliotheken wie TensorFlow, CNTK und Andere.

**kNN.** k-nächste-Nachbarn (*engl. k-Nearest-Neighbor*).

**LeNet.** Frühe CNN-Implementierung für den MNIST-Datensatz.

**loss.** Zielfunktion (Straffunktion).

**MCP.** McCulloch-Pittis-Neuron.

**metrics.** Metriken zur Beobachtung der Lernkurve.

**ML.** Maschinelles Lernen (*engl. machine learning, ML*).

**MLP.** Mehrschichtiges Perzeptron (*engl. multi layer perceptron, MLP*).

**MNIST.** Datensatz handgeschriebener Ziffern (*engl. Modified National Institute of Standards and Technology, MNIST*).

**MXU.** Matrixeinheit innerhalb des Google Cloud TPUs (*engl. Matrix Execution Unit, MXU*).

**NIH.** US-Gesundheitsinstitut (*engl. National Institutes of Health, NIH*).

**optimizer.** Lernalgorithmus (SGD, Adam, ...).

**ReLU.** Nichtlineare Aktivierungsfunktion (*engl. rectified linear unit*).

**ROC.** Grenzwertoptimierungskurve (*engl. Receiver-Operating-Characteristic, ROC*).

**SGD.** Stochastisches Gradientenabstiegsverfahren (*engl. stochastic gradient descent, SGD*).

**split.** Aufteilung (*engl. split*) des Datensatzes in disjunkte Teilmengen (ohne Schnittmenge).

**SVM.** Stützvektormethode (*engl. Support Vector Machines*).

**TDP.** Verlustleistung (*engl. thermal design power*).

**TensorFlow.** Open-Source-Bibliothek von Google für maschinelles Lernen.

**TPU.** Recheneinheit von Google mit ASIC-Logik (*engl. Tensor Processing Unit, TPU*).

**VGGNet.** CNN-Implementierung in zwei Ausprägungen: VGG16 und VGG19.

## Symbolverzeichnis

$f(x)$ :	Funktion mit Parameter $x$
$x$ :	Eingabe
$x^{(i)}$ :	einzelner Datenpunkt
$y$ :	Klassenbezeichnung
$n$ :	Anzahl Datenpunkte
$i$ :	Anzahl Klassenbezeichnungen / Zielklassen / Zielknoten
$\mathbb{R}^n$ :	Dimension der Eingabe
$y^{(i)}$ :	tatsächliche Klassenbezeichnung (analog zum einzelnen Datenpunkt $x^{(i)}$ )
$\hat{y}^{(i)}$ :	vorhergesagte Klassenbezeichnung
$\mathbb{R}$ :	die Menge der reellen Zahlen
$\theta$ :	Schwellenwert
$\phi(z)$ :	Aktivierungsfunktion
$z$ :	Nettoeingabe
$z^{(i)}$ :	einzelne Nettoeingabe $(z^{(i)} = w^T x^{(i)})$
$w$ :	Gewichtungsmatrix
$w_j$ :	einzelnes Gewicht
$w^T$ :	transponierte Gewichtungsmatrix
$w_0$ :	Nullgewicht (Bias-Einheit)
$\Delta w_j$ :	Änderung (Delta) eines einzelnen Gewichts
$\eta$ :	Lernrate
$J(w)$ :	Straffunktion mit Parameter $w$
$\nabla J(w)$ :	Gradienten-Schritt
$\frac{\partial J}{\partial w_j}$ :	Ableitung der Straffunktion nach einer einzelnen Gewichtung $w_j$
$l$ :	Schicht-Index (die Eingabeschicht besitzt den Index Null)
$h_i^{(l)}$ :	verdeckte Schicht Nummer $i$ , innerhalb des Schicht-Index $l$
$f$ :	Anzahl Filter ( <i>engl. number of filters</i> )
$m$ :	Filtergröße ( <i>engl. filter size</i> )
$s$ :	Schrittweite ( <i>engl. stride</i> )
$p$ :	Füllungsparameter ( <i>engl. padding</i> )
$o$ :	Ausgabeformat
$c$ :	Anzahl der Kanäle ( <i>engl. number of channels</i> )



$\lambda$ :	Regularisierungsparameter <i>lambda</i> (auch $\alpha$ <i>alpha</i> )
$\ w\ _1$ :	L1-Norm
$\ w\ _2^2$ :	L2-Norm
$k$ :	Anzahl von Aufteilungen bei der k-fachen Kreuzvalidierung
$\mu$ :	Mittelwert
$\sigma$ :	Standardabweichung ( <i>sigma</i> )
$TP$ :	Korrekt Positiv ( <i>engl. true positive</i> )
$FP$ :	Falsch-Positiv ( <i>engl. false positive</i> )
$TN$ :	Korrekt-Negativ ( <i>engl. false negative</i> )
$FN$ :	Falsch-Postiv ( <i>engl. false negative</i> )
$KKR$ :	Korrektklassifizierungsrate ( <i>engl. accuracy</i> )
$FQ$ :	Fehlerquote ( <i>engl. error rate</i> )
<i>recall</i> :	Trefferquote ( <i>engl. recall</i> )
<i>precision</i> :	Genauigkeit ( <i>engl. precision</i> )
$T$ :	Anzahl Korrekter Vorhersagen ( $T = FN + TP$ )
$C$ :	Convolution-Schicht
$C^4$ :	vier aneinandergereihte Convolution-Schichten
$C_{256}^{5;1}$ :	Convolution-Schicht mit Parametern $m = 5$ ; $s = 1$ ; $p = 0$ ; $f = 256$
$P$ :	Pooling-Schicht
$P_M$ :	Max-Pooling
$P_A$ :	Average-Pooling ( <i>Mean-Pooling</i> )
$P_G^{1 \times 1 \times 2048}$ :	Global-Average-Pooling mit Ausgabedimension $1 \times 1 \times 2048$
$(CP)^{m \times m \times f}$ :	Convolution-Pooling Folge ( <i>Convolution-Block</i> ) mit Parametern $m, p, f$
$FC$ :	vollständig verknüpfte Schicht ( <i>engl. fully connected layer</i> )
$FC_{4096}^2$ :	zwei aneinandergereihte FC-Schichten mit jeweils 4096 Einheiten
$I$ :	Inception-Modul (Sonderfall $I'$ mit anschließendem Pooling)
$D$ :	Dense-Modul

## Vorwort

Wäre das nicht genial, einen Computer mit der Erledigung von langweiligen und fehleranfälligen Aufgaben zu betrauen? Diese Möglichkeit gibt es. Doch der Prozess zur Entwicklung eines einsatzfähigen ML-Modells (am Beispiel eines Klassifikators) ist vor allem datengetrieben. Um diesen Anspruch zu erfüllen, muss es zunächst einmal genügend Datenpunkte geben, damit sich die zu lösende Aufgabe hinreichend parametrisieren lässt. Das wird am Gegenstand der vorliegenden Arbeit deutlich. Sie untersucht maschinelles Lernen zur Bild-Klassifikation anhand eines Datensatzes aus Röntgenbildern.

Diese Arbeit erhebt den Anspruch einer vollständigen Analyse der Analysemethoden für neuronale Netzwerke am Beispiel von CNNs, die unter Einhaltung einer Schrittfolge (ML-Pipeline) und Bewertungsmethoden geschieht. Die Schrittfolge enthält alle wichtigen Stationen auf dem Weg zu einem produktiven ML-Modell am Beispiel eines CXR-Klassifikators in Kapitel 4.4.. Beginnend beim Daten- und Problemverständnis und endend bei angemessenen Bewertungsmethoden werden in Kapitel 3 und 4 alle praktischen Werkzeuge bereitgestellt.

Ist der Datensatz zum Beispiel unausgewogen wie im Falle dieser Arbeit, dann sind besondere Schritte einzuleiten, dessen Unausgewogenheit bereits in der Datenanalyse im ersten Schritt der ML-Pipeline auffällt. Dies stellt ein großes Einsparpotenzial dar, bevor die ML-Pipeline unter falschen Annahmen weiter beschritten wird.

Der theoretische Hintergrund neuronaler Netze und deren Vorreiter werden in Kapitel 2 erklärt. CNN-Bausteine und die sequentielle Natur neuronaler Netze – basierend auf dem Perzeptron und dessen Weiterentwicklungen – sind eine wichtige Erkenntnis für eine möglichst vollständige Analyse von CNNs in Kapitel 4. Produktive Modelle zur Bild-Klassifikation proklamieren eine hohe Leistung und sind auf den ersten Blick eine schwer analysierbare Blackbox. Dieses Problem wird vor dem Hintergrund verschiedener CNN-Implementierungen untersucht.

Sämtliche CNN-Implementierungen und Codeschnipsel, die im Rahmen dieser Arbeit erstellt und trainiert wurden, sind online unter folgendem GitHub-Link abrufbar:

**<https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow>**

Die Installationsanweisungen für die Nachbildung des verwendeten Systems und Umgebung sind in der *Readme.md* (Vgl. GitHub-Link) und Kapitel 3.3 beschrieben (Vgl. 3.3 System und Software-Anforderungen).

## Kapitel 1 – Einleitung

Dieses Kapitel liefert eine abstrakte Beschreibung maschinellen Lernens und der allgemeinen Problemstellung. Daraus ergeben sich sowohl die Fragestellung als auch das Forschungsziel dieser Arbeit. Darüber hinaus wird der inhaltliche Gegenstand dieser Arbeit von anderen Themen aus dem Komplex des maschinellen Lernens und der künstlichen Intelligenz abgegrenzt. Zugleich finden sich in diesem Abschnitt ein thematischer Überblick über diese Arbeit sowie eine Beschreibung der Notation.

## 1.1 A b s t r a k t

Diese Arbeit beschäftigt sich mit der methodischen Analyse und praktischen Anwendung von überwachtem Lernen für die Klassifizierung eines Bild-Datensatzes mit der Maschine-Learning-Bibliothek TensorFlow (für die Programmiersprache Python). Der Keras Quellcode besitzt den Vorteil gegenüber einer reinen TensorFlow-Implementierung, verständlich und einfacher zu sein.

Zum Einsatz kommt der sortierte NIH-Datensatz<sup>1</sup> aus Brust-Röntgenbildern (*CXR*). Ziele sind die Entwicklung, Bewertung und Gegenüberstellung verschiedener CNN-Implementierungen sowie die Identifikation des besten Modells für die möglichst genaue Klassifikation des Datensatzes. Auf dem Weg zu einem trainierbaren Modell sind Entscheidungen hinsichtlich der Architektur des Modells und dessen Hyperparameter 3.1.8 zu treffen, die maßgeblichen Einfluss auf die Genauigkeit des trainierten Modells haben.<sup>1</sup>

Die wichtigsten Schritte, die für den Aufbau, das Training und die Bewertung eines Modells nötig sind, werden chronologisch den entscheidenden Etappen des Aufbaus der praktischen Implementierung zugeordnet (Vgl. 3.1 ML-Pipeline und 3.2 CNN-Implementierung). Zugleich werden deren Bausteine in Kapitel 2 methodisch analysiert: von der Eingabeschicht (Methoden zur Dimensionsreduktion Transformation und Normalisierung) über verdeckte Schichten zur Merkmalsextraktion bis hin zur finalen Klassifizierungsschicht (Vgl. 2.6 Bausteine von CNNs).

---

<sup>1</sup> (Vgl. <https://nihcc.app.box.com/v/ChestXray-NIHCC>)

## 1.2 Problemstellung

Im Hinblick auf Programmlogik und Ablaufregeln basiert der Einsatz von maschinellem Lernen (ML) im Vergleich zu klassischer Programmierung auf einem vollkommen neuen Programmierparadigma. In der klassischen Programmierung wird der Programmablauf durch Regeln und Zustände definiert (Vgl. Abbildung 1, *Rule-based systems*). Vergisst der Entwickler eine Regel oder wendet sie falsch an, dann ist das Programm fehlerhaft.

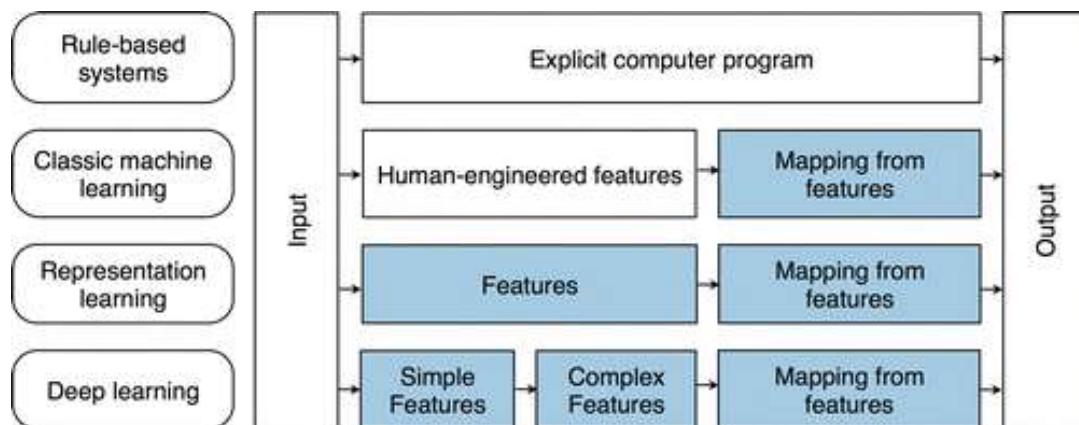


Abbildung 1: Von Regelbasierten Systemen bis deep learning (Quelle: Chartrand et al. 2017, S.2115)

In den Medien werden künstliche Intelligenz und ML häufig als Methoden beschrieben, für die man nur einen Knopf drücken müsse, damit sie funktionieren.<sup>ii</sup> Das trifft allerdings nur für die Anwendung von Prognosen und Beschreibungen zu. Denn für Vorhersagen wendet ein ML-Modell sein Wissen lediglich an.

Um dieses Wissen überhaupt zu generieren, stehen dem Lernverfahren die (unverarbeiteten Roh-)Daten als Vorlage der Zielmuster zur Verfügung. Das ML-Modell ist einerseits Repräsentation des Wissens aus den Eingabedaten und andererseits Endprodukt des Lernverfahrens. Seine internen Gewichtungen sind das Optimierungsziel verschiedener Lernalgorithmen (Vgl. 2.5 Lernmethoden). Dies geschieht durch die Anwendung stochastischer Methoden auf Grundlage der Eingabedaten – scheinbar wie von selbst. Die Form der Daten und das Entscheidungsziel, sei es Klassifikation, Regression oder Clustering, entscheiden über die passende Modellstruktur. Die Daten können hierbei strukturiert als Tabelle oder unstrukturiert in Form von Rohdaten wie Bildern, Text oder Sprache vorliegen. Die Umwandlung der Daten in ein maschinenlesbares Format ist eine wichtige Voraussetzung für den effizienten Einsatz von ML (Vgl. 3.1.5 Daten transformieren).

Regelmäßige Wettbewerbe (zum Beispiel ImageNet Large Scale Visual Recognition Challenge, ILSVRC)<sup>2</sup> und Ausschreibungen<sup>3</sup> verfolgen das Ziel, den kleinstmöglichen (Klassifikations-)Fehler für einen gegebenen Datensatz zu finden. Diese erreichen heute bereits niedrigere Fehlerraten als ein Mensch und fördern das Interesse an und Investitionen in KI-Projekte. Das Potenzial eines ML-Modells, weniger Fehler als ein Mensch zu begehen und dabei nicht müde zu werden, macht es außerordentlich reizvoll, monotone oder sich wiederholende Aufgaben einer Maschine zu überlassen. Dieses Potenzial ist der Treiber der jüngsten Fortschritte in Hardwareoptimierung, Modellarchitekturen und neuer Lernalgorithmen.

Die Weiterentwicklung spezialisierter Hardware wie Grafikkarten (*GPU*) und TPUs zählt ebenfalls zu den Meilensteinen für den effizienteren Einsatz paralleler Rechenoperationen, die für das Training von Modellen nötig sind. Eine Übersicht bietet Abbildung 2. Es ist zwar möglich, auf einem CPU zu trainieren. Allerdings hat ein GPU deutlich mehr Kerne als ein CPU. Und obwohl die GPU-Kerne deutlich niedriger getaktet sind als ein CPU, besitzt ein GPU deutlich mehr Kapazität für parallele Rechenoperationen (Vgl. Tabelle 9).

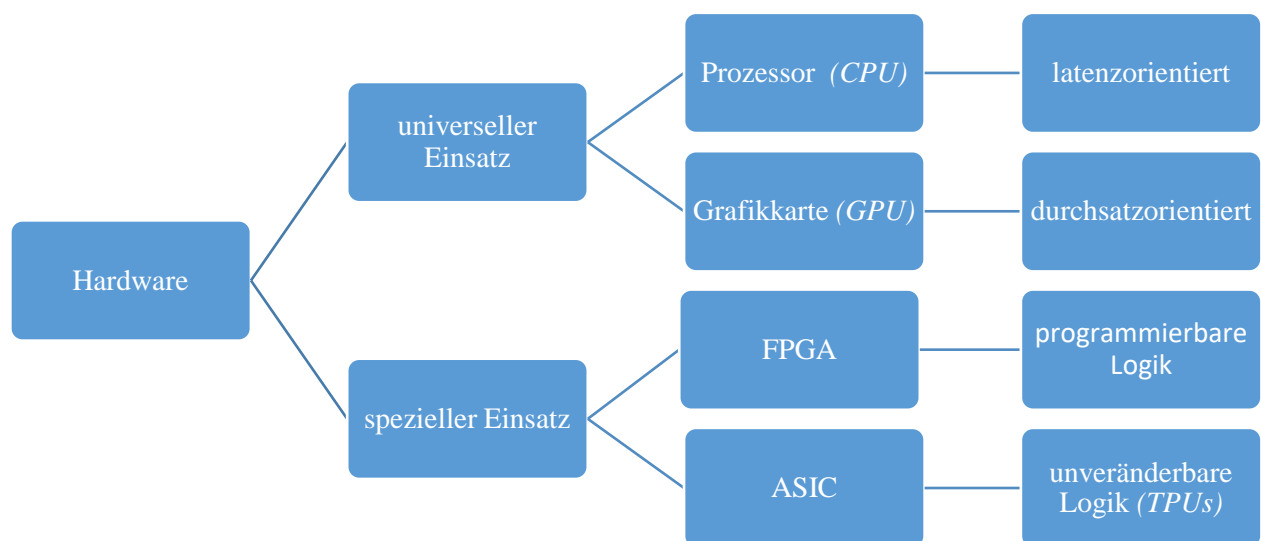


Abbildung 2: Hardware-Übersicht (Eigene Darstellung)<sup>4</sup>

Um die GPU-Auslastung auf einem hohen Niveau zu halten, sind häufige Speicherzugriffe auf den Datensatz notwendig. Daher nimmt die Speicherbandbreite (neben der Anzahl und dem Takt eines

<sup>2</sup> (Vgl. <http://www.image-net.org/index/>) und (Vgl. <https://www.kaggle.com/c/imagenet-object-localization-challenge/>, Abgerufen am 11.6.19)

<sup>3</sup> (Vgl. <https://www.kaggle.com/competitions/>, Abgerufen am 11.6.19)

<sup>4</sup> (Quelle: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.18, Abgerufen am 11.6.19)

Kerns; Vgl. Tabelle 9) erheblichen Einfluss auf die Geschwindigkeit des Trainings.<sup>5</sup> Vor dem Hintergrund wachsender Datenbanken und Anforderungen stellt sich das Training als sehr datenintensiver und „Hardware-hungriger“ Prozess dar. Daher werden die Hardware- und Speicheranforderungen zukünftiger Modelle voraussichtlich ebenso wachsen wie die Komplexität der Modelle selbst.<sup>iii</sup>

---

<sup>5</sup> (Vgl. <https://www.graphcore.ai/posts/why-is-so-much-memory-needed-for-deep-neural-networks>, Abgerufen am 11.6.19)

## 1.3 Forschungsfrage

Neuronale Netzwerke zur Klassifizierung von Bildern stammen aus der sequentiellen Modellfamilie (*engl. (vanilla) feedforward net*) und werden als convolutional neural network (CNNs) bezeichnet. CNNs erzielen seit dem Jahr 2012 bahnbrechend niedrige Fehlerquoten für die Klassifikation großer Bilddatensätze im Rahmen von Wettbewerben für maschinelles Sehen (*engl. computer vision*). Sie folgen den Prinzipien linearer Modelle und weisen gewisse Unterschiede zum Ursprung des mehrschichtigen Perzeptrons (*MLP*) auf. Ihre gemeinsamen Bausteine und Unterschiede sind Gegenstand von Kapitel 2.6 sowie 4.1.

Die für den Aufbau eines CNNs notwendigen Phasen (Vgl. 3.1 ML-Pipeline) werden vor dem theoretischen Hintergrund ein- (Vgl. 2.3 Einschichtige Netzwerke) sowie mehrschichtiger neuronaler Netzwerke (Vgl. 2.4 Mehrschichtige Netzwerke) analysiert und für die Implementierung eines CNNs (Vgl. 3.2 CNN-Implementierung) angewendet. Außerdem werden verschiedene Bewertungsmethoden für diverse Modelle (unter anderem selbst erstellte CNN-Implementierungen) angewendet, um deren Genauigkeit zu beurteilen (Vgl. 3.1.7 Bewertung des Modells). Im Anschluss wird in Kapitel 4.3 das beste Modell ausgewählt.

Im vierten und letzten Kapitel erfolgt eine Zusammenfassung der Ergebnisse. Dazu werden unterschiedliche CNN-Implementierungen anhand ihrer Architektur verglichen und wichtige Bausteine herausgearbeitet (Vgl. 4.1 Unterschiede moderner CNN-Implementierungen). Außerdem werden ML-Einsatzfelder in einem Krankenhaus diskutiert (Vgl. 4.4 Produktive ML-Einsatzfelder).



## 1.4 Thematische Abgrenzung

“A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ .” (Mitchell, 1997)<sup>6</sup>

Künstliche Intelligenz (*engl. artificial intelligence, AI*) beschreibt das Themengebiet rund um die Erzeugung von Wissen in Form von Mustern aus Daten. Maschinelles Lernen ist eine Teilmenge von KI, Repräsentationslernen<sup>iv</sup> (*engl. representation learning*) ist eine weitere Spezialisierung von ML, wie zum Beispiel tiefgründiges Lernen (*engl. deep learning*) am Beispiel des mehrschichtigen Perzeptrons (*MLP*) (Vgl. 2.4 Mehrschichtige Netzwerke).

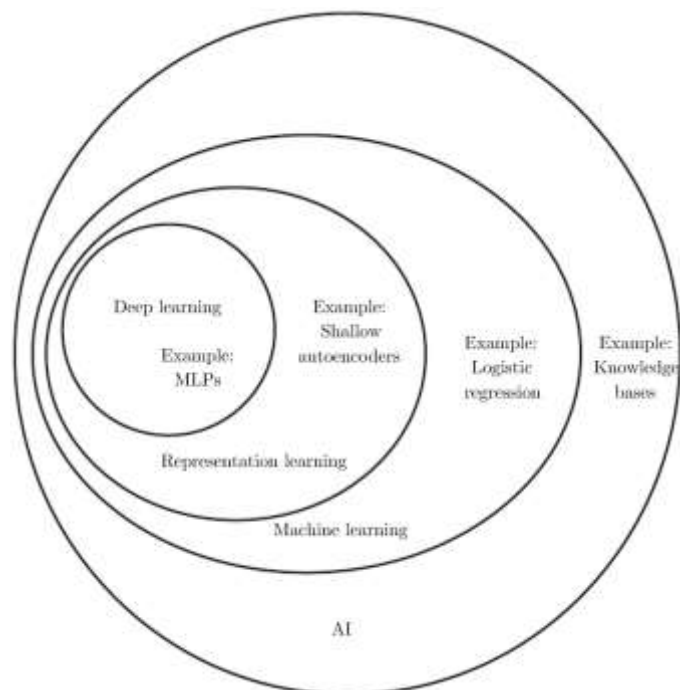


Abbildung 3: Themenkomplex AI als Venn-Diagramm (Quelle: Goodfellow et al. 2016, S.9)

Der Begriff maschinelles Lernen bezeichnet das Lernen aus Daten während der Anwendung stochastischer Methoden zur Minimierung der Zielfunktion (Aufgabe T)<sup>v</sup> wie zum Beispiel die logistische Regression (*engl. logistic regression*; Vgl. Abbildung 3). Diese gibt keine eindeutige Klassenbezeichnung aus, sondern ein Kontinuum (Vgl. 2.2 Aufgaben von überwachtem Lernen).

Die Aufgabe T eines trainierten Modells ist die Erstellung möglichst genauer Vorhersagen (*engl. prediction*) auf unbekannte Eingabedaten. Um überhaupt eine Vorhersage treffen zu können, nutzt ein

<sup>6</sup> (Vgl. <https://machinelearningmastery.com/what-is-machine-learning/>, Abgerufen am 11.6.19)

neuronales Netzwerk die gewichteten Eingabedaten. Beispiele zu den unterschiedlichen Aufgaben dieser drei Lernalgorithmen sind Bestandteil von Kapitel 2.1 (Vgl. 2.1 ML-Aufgaben).

Die Gewichtungen des Netzwerks werden vor Beginn des Trainings entweder zufällig initialisiert oder aus vergangenen Trainingssitzungen übernommen (*engl. transfer learning bzw. fine-tuning*) und während des Trainings optimiert. Die Anpassung und Aktualisierung (Optimierung) der Gewichte an die Zielklassen findet nur während des Trainings statt. Diese Phase wird als Gradientenabstieg unter dem Einsatz von Rückpropagation (*engl. backpropagation*) bezeichnet. Wobei die Rückpropagation streng genommen lediglich den Gradienten bestimmt, während das stochastische Gradientenabstiegsverfahren (SGD) ihn anwendet (Vgl. 2.5 Lernmethoden).<sup>vi</sup>

Rückpropagation wird oft fälschlicherweise mit dem gesamten Lernprozess gleichgesetzt oder als einziges Lernverfahren neuronaler Netzwerke bezeichnet. Dabei ist sie lediglich für die effiziente Berechnung des Gradientenabstiegs verantwortlich<sup>vii</sup>. Für die Prognose von Eingabedaten werden dagegen die Gewichtungen angewendet. Diese sequentielle Anwendung der Gewichtungen wird Vorwärtspropagation genannt, da die Information vorwärtsgerichtet, von der Eingabeschicht zur Ausgabeschicht, fließt.<sup>viii</sup> Zur Anpassung der Gewichte fließen die Informationen in die entgegengesetzte Richtung – zurück bis zur Eingabeschicht.<sup>ix</sup>

Ein trainiertes Modell ist das Resultat der abgeschlossenen Phasen – vom Verständnis der Daten über deren Vorverarbeitung bis hin zum Training der Eingabedaten (Vgl. 3.1 ML-Pipeline). Ein untrainiertes Modell unterscheidet sich davon (bei gleicher Aufgabe und Architektur) einzig im Zustand der Gewichtungen. Für die Bewertung der Leistung  $P$  eines Modells werden verschiedene Werkzeuge und Metriken vorgestellt (Vgl. 3.1.7 Bewertung des Modells), die vor, während und nach dem Training zum Einsatz kommen, um die Daten für ein Modell vorzubereiten, zu visualisieren (Vgl. 3.2.5.1 Transformation Visualisieren) oder Entscheidungen zu bewerten. Für das Training und die Vorhersage auf Basis eines nicht-ausgewogenen Datensatzes (*engl. imbalanced bzw. skewed*) ist besondere Vorsicht geboten (Vgl. 4.2 Umgang mit unausgewogenen Datensätzen).

## 1.5 Aufbau der Arbeit

Diese Arbeit ist in vier Kapitel gegliedert. Das erste Kapitel beschreibt das methodische Vorgehen, die Forschungsfrage sowie die Problemstellung. Des Weiteren wird das Thema dieser Arbeit von anderen ML-Themen abgegrenzt und die Notation erklärt.

Das zweite Kapitel umfasst den theoretischen Hintergrund dieser Arbeit. Beginnend bei den Aufgaben eines ML-Modells werden verschiedene Lernmethoden und -modelle erklärt – vom Prinzip des einschichtigen Perzeptrons (Vgl. 2.3.3 Perzepron) über die Erweiterung zum MLP bis hin zu neuronalen Netzwerken und CNNs. Im Anschluss erfolgen die Betrachtung des Aufbaus aus Schichten und CNN-Bausteinen sowie die Analyse des KDD-Vorgehensmodells zur Erstellung von Modellen als Vorlage der ML-Pipeline.

Kapitel drei beschäftigt sich mit der praktischen Schritt-für-Schritt Implementierung eines CNN-Modells in Python, analog zum KDD-Vorgehensmodell (Vgl. 2.7 Modellaufbau nach KDD-Vorgehensmodell). Der Quellcode wird wie beim KDD-Modell in mehrere Phasen unterteilt (Vgl. 3.1 ML-Pipeline).

Kapitel vier fasst die gewonnenen Erkenntnisse zu CNN-Architekturen (Vgl. 4.1 Unterschiede moderner CNN-Implementierungen), unausgewogenen Datensätzen (Vgl. 4.2 Umgang mit unausgewogenen Datensätzen), CXR-Referenzmodellen (Vgl. 4.3 CXR-Implementierung), ML-Modularisierung (Vgl. 4.5 Modularisierung der ML-Pipeline) und ML-Einsatzformen (Vgl. 4.4 Produktive ML-Einsatzfelder) zusammen. Dazu bietet das Kapitel einen thematischen Ausblick.

## 1.6 Notation

### 1.6.1 Mathematische Ausdrücke und Formel

Für mathematische Formeln, Ausdrücke und Variablen wird in dieser Arbeit die Schriftart *Cambria Math*, kursiv in der Schriftgröße 12 benutzt. Formeln werden außerdem durchnummeriert.

*Der Ausdruck  $\mathbb{N}$  wird als Kurzschreibweise für die Menge aller ganzen Zahlen benutzt; er beschreibt alle nicht-negativen und natürlichen Zahlen  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$  ohne die Zahl Null. Die exakte Notation lautet  $\mathbb{N}^* = \{1, 2, 3, \dots\} = \mathbb{N} \setminus \{0\}$ .<sup>7</sup>*

Beispiel:  $n \in \mathbb{N}$

Beispiel: Satz des Pythagoras: (Nummer)  $a^2 + b^2 = c^2$

### 1.6.2 Quellcode

Zur Hervorhebung von Ausdrücken, die aus dem Quellcode stammen, wird in dieser Arbeit die Schriftart *Consolas* in Schriftgröße 12 verwendet.

Beispiel: `model.fit`

### 1.6.3 Verweise auf das Literaturverzeichnis

Querverweise auf die Quellen im Literaturverzeichnis werden nach der Norm DIN ISO 690 vorgenommen. Genutzt wird folgendes Schema: (Nachname, Jahr, S.Seitenzahlen).

Beispiel: (Rosenblatt, 1957, S.10f)

Indirekte und sinngemäße Zitate werden darüber hinaus mit der Abkürzung Vgl. (Vergleiche) gekennzeichnet.

Beispiel (Vgl. Rosenblatt, 1957, S.10 ff.)

---

<sup>7</sup> Nach DIN-5473 (Vgl. <https://www.mathebibel.de/natuerliche-zahlen/>, Abgerufen am 28.4.19)

### 1.6.4 Querverweise auf Kapitel

Für die Orientierung innerhalb dieser Arbeit werden Querverweise auf Kapitel nach folgendem Schema gekennzeichnet: (Vgl. *Nummer des Abschnittes Name des Abschnittes*).

Beispiel (Vgl. 2.1 ML-Aufgaben).

### 1.6.5 Verweise auf Tabellen und Abbildungen

Querverweise auf Tabellen und Abbildungen werden nach einem anderen Schema hervorgehoben: (Vgl. *Typ + Laufende Nummerierung, (optional) Verweis auf Abbildungstext*)

Beispiel (Vgl. Tabelle 1, *erste Spalte*)

### 1.6.6 Beschriftung von Abbildungen und Tabellen

Abbildungen und Tabellen werden in ihrem Beschriftungsfeld durchnummeriert und im Abbildungs- und Tabellenverzeichnis organisiert. Die Quelle der Abbildung ist in der Beschriftung als Endnote enthalten, analog wenn die Eigendarstellung von einer anderen Abbildung inspiriert ist. Eigene Anfertigungen erhalten in der Bild- oder Tabellenbeschriftung eine zusätzliche Kennzeichnung mit „(Eigene Darstellung)“. Es wird folgendes Schema benutzt:

(Vgl. Abbildung *Nummer: Name* (Quelle: *Literaturverzeichnis/Internet-Fußnote*)), oder

(Vgl. Abbildung *Nummer: Name* (Eigene Darstellung))

Beispiel



Abbildung 1: LeNet-Architektur (Quelle: LeCun et al. 1998)

## Kapitel 2 – Theoretischer Hintergrund

Kapitel zwei gibt einen theoretischen Überblick zu den Grundkonzepten neuronaler Netzwerke sowie zu deren Ursprüngen in linearen Modellen. Beginnend beim MCP-Neuron, Perzeptron, Perzeptron-Lernregel und ihrer formaler Definition bis hin zur Adaline-Einheit (Vgl. 2.3 ) gelangt dieses Kapitel ab dem Abschnitt 2.4 zu dem Themenkomplex mehrschichtiger neuronaler Netzwerke (Vgl. 2.4 Mehrschichtige Netzwerke).

Die mehrschichtige Variante des Perzeptrons ist der ideale Ausgangspunkt für die Beschreibung neuronaler Netze hin zu einer spezialisierten Form neuronaler Netze für die Bilderkennung: die CNNs. Abschnitt 2.6 beschreibt CNNs im Detail. Der Aufbau und die Bausteine von CNNs werden aufgelistet und untersucht, wobei erst in Kapitel 4.1 methodische Unterschiede verschiedener CNN-Architekturen herausgearbeitet werden. Im Anschluss stehen die allgemeinen Lernmethoden für neuronale Netzwerke im Blickpunkt. Den Gradientenabstieg und seine Varianten zu verstehen ist eine wichtige Voraussetzung für die Optimierung neuronaler Netzwerke.

Zum Abschluss des Kapitels wird die KDD-Methode für den analytischen und produktiven Einsatz zur Wissensextraktion aus Daten vorgestellt, das KDD-Vorgehensmodell nach Fayyad. Es beschreibt den Prozess der Vorbereitung und Vorverarbeitung von Daten, um repräsentatives Wissen zu generieren (Vgl. 2.7 Modellaufbau nach KDD-Vorgehensmodell).

## 2.1 ML - Aufgaben

Aufgabe  $T$  eines trainierten Klassifizierungsmodells ist es, möglichst genaue Vorhersagen für unbekannte Eingabedaten zu treffen. Damit es überhaupt eine solche Vorhersage treffen kann, nutzt ein neuronales Netzwerk die gewichteten Eingabedaten als Entscheidungsgrundlage (Zielwert). Die Eingabedaten stellen die einzelnen Datenpunkte des Datensatzes dar, die wiederum eine Ansammlung aus Merkmalen (Attribute) sind.<sup>x</sup> Die Eingabedaten können in verschiedener Form vorliegen: Einerseits unvollständig, wenn gewisse Merkmale fehlen. Andererseits uneinheitlich belegt, wenn unterschiedliche Wertebereiche benutzt wurden.<sup>xi</sup>

Grundsätzlich sind alle ML-Aufgaben anhand des Vorhandenseins und Form der Eingabedaten  $E$  und ihres Ziels  $T$  kategorisierbar. Fehlen die Klassenbezeichnungen (Zielwert) im Vorhinein, spricht man in der Regel von unbewachtem Lernen. Wenn darüber hinaus der Datensatz fehlt, dann ist von selbstverstärkendem Lernen die Rede (Vgl. Abbildung 4):

überwachtes Lernen (engl. <i>supervised learning</i> )	<ul style="list-style-type: none"><li>• Eingabe <math>x</math> + Klassenbezeichnung <math>y</math></li><li>• Rückmeldung anhand des Zielwerts</li></ul>
unbewachtes Lernen (engl. <i>unsupervised learning</i> )	<ul style="list-style-type: none"><li>• Eingabe <math>x</math></li><li>• keine Klassenbezeichnung, kein Zielwert</li></ul>
selbstverstärkendes Lernen (engl. <i>reinforcement learning</i> )	<ul style="list-style-type: none"><li>• interagiert direkt mit der Umwelt</li><li>• Maximierung des Zielwerts (Score)</li></ul>

Abbildung 4: ML-Lernalgorithmen (Eigene Darstellung)<sup>xii</sup>

Über die eingesetzte Lernmethode entscheidet zunächst die Beschaffenheit des Datensatzes. Unbewachte oder überwachte Lernmethoden unterscheiden sich nur in der Gegebenheit der Klassenbezeichnung. Ein unbewachter Lernalgorithmus erhält keine Auswahl von Zielwerten, sondern ausschließlich den Datensatz zur Generierung seines Wissens. Die Ausgabe beim unbewachten Lernen ist meist eine Repräsentation der Trainingsdaten in Form einer Verteilung. Der unbewachte Lernalgorithmus lernt eine Schar von Funktionen, die den unterschiedlichen Kombinationen fehlender Eingaben entspricht.<sup>xiii</sup> Unbewachte Lernmethoden arbeiten nur mit der Eingabe, ohne Vorwissen des Resultats (Klassenbezeichnung) als zusätzliche Information.<sup>xiv</sup>

Selbstverstärkendes Lernen ist eine weitere Sonderform maschinellen Lernens. Da dem Lernalgorithmus überhaupt keine Vorab-(engl. *ex-ante*)-Information (in Form eines Datensatzes)

vorliegen, interagiert er direkt mit seiner Umwelt in einem Kreislauf (*engl. feedback loop*).<sup>xv</sup> Ein gutes Beispiel ist der Versuch einem Computer Atari-Spielen beizubringen, der Algorithmus versucht ohne Vorwissen und durch Ausprobieren die Höchstpunktzahl (*engl. highscore*) zu maximieren.<sup>xvi</sup>

Neben der Klassifikation gibt es viele weitere ML-Aufgaben, die an dieser Stelle nur aufgezählt werden: Regression, Clustering, Transkription wie Texterkennung (*engl. optical character recognition, OCR*) und Maschinenübersetzung.

Strukturierte Ausgaben bedeutet die Rückgabe eines Vektors. Sie umfassen die Tätigkeiten von Transkription und Übersetzung, zum Beispiel die Beschreibung des Bildinhalts in Textform (*engl. image caption*).<sup>xvii</sup>

Weitere Anwendungsbeispiele sind die Anomalie-Erkennung (Betrugserkennung) wie Spam-Filter (*engl. fraud- bzw. spam detection*), Synthese und Sampling wie DLSS<sup>8</sup>, die Zuteilung fehlender Werte sowie die Schätzung von Verteilungen.<sup>xviii</sup>

---

<sup>8</sup> (Vgl. <https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-your-questions-answered/>, Abgerufen am 11.6.19)



## 2.2 Aufgaben von überwachtem Lernen

Die zwei wesentlichen Aufgaben überwachtem Lernens sind Klassifikation und Regression. Ziel der Klassifikation ist die Vorhersage einer nominalen (unsortierten) Klassenbezeichnung (*engl. labels*). Ihre Ausgabe ist eine Funktion  $f(x) = y$ . Sie stellt eine Abbildung der Eingabe  $x$  auf die Zielklassen dar  $f : \mathbb{R}^n \rightarrow \{1, \dots, i\}^{\text{xxix}}$ . Die Funktion  $f$  bildet die Eingabedimension  $\mathbb{R}^n$  auf die Ausgabedimension  $i \in \mathbb{N}$  (Anzahl der Zielklassen) ab. Die Ausgabe entspricht der (höchsten) vorhergesagten Klassenwahrscheinlichkeit aus den vorgegebenen Möglichkeiten aller Klassenbezeichnungen.

Bei der Regression geht es hingegen um die Vorhersage kontinuierlicher Größen wie reeller Zahlen. Die Zielmenge entspricht einem Kontinuum.<sup>xx</sup> Solche Lernmethoden werden als überwacht bezeichnet, da die vorgegebene Klassenbezeichnung wie ein Lehrer agiert, der seinen Schülern (den ML-Modellen) Anweisungen gibt.<sup>xxi</sup>

Überwachte Lernalgorithmen arbeiten mit „Eingabe-Ausgabe-Paare[n]“<sup>xxii</sup> für Vorhersagen auf Basis ungesehener Daten. Ein Eingabe-Ausgabe-Paar setzt sich aus dem Datenpunkt  $x^{(i)} \in \mathbb{R}^n$  sowie den annotierten Klassenbezeichnungen  $y$  zusammen.<sup>xxiii</sup>

### 2.2.1 Klassifikation und Regression

Die Vorhersage einer Klassenbezeichnung kann entweder binär oder aus einer Liste von Möglichkeiten erfolgen. Binäre Klassifikation bedeutet, dass genau zwei Zielklassen vorliegen, zum Beispiel wahr oder falsch (Ja-/Nein-Frage). Zwei Zielklassen werden mit einem Ausgabeknoten abgebildet, der die Klassenwahrscheinlichkeit der positiven Klasse angibt.

Wenn es mehr als zwei Zielklassen der Anzahl  $i$  gibt, wird zwischen der Ausgabe genau einer Zielklasse (*engl. multi-class*) und mehreren Zielklassen (*engl. multi-label*) unterschieden (Vgl. Abbildung 5), wobei das Modell  $i \in \mathbb{N}$  viele Ausgabeknoten besitzt. Multi-label ist ein Spezialfall, da hierbei mehr als eine Klasse gleichzeitig auftreten darf – die Exklusivität der einzelnen Klasse ist aufgehoben. Im Rahmen der Datentransformation werden die Klassenbezeichnungen in Zahlen oder Vektoren umgewandelt (Vgl. 3.1.5 Daten transformieren).

Bei der Regression gilt es, eine kontinuierliche Größe vorherzusagen – im Sinne eines numerischen Wertes. Das bedeutet, dass die Liste von Möglichkeiten als Kontinuum im Rahmen einer numerischen und sortierten Zuordnung zu verstehen ist  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ .<sup>xxiv</sup> Die Kontinuität der Ausgabe ist das stärkste Unterscheidungsmerkmal der Regression im Vergleich zur Klassifikation.<sup>xxv</sup>










	Multi-Class	Multi-Label
$C = 3$	Datenpunkte	Datenpunkte
  	  	  
	Klassenbezeichnung	Klassenbezeichnung
	[0 0 1] [1 0 0] [0 1 0]	[1 0 1] [0 1 0] [1 1 1]

Abbildung 5: multi-class versus multi-label (Eigene Darstellung)<sup>9</sup>

### 2.2.2 Erweiterung der binären Klassifikation zur Mehrfachklassifikation

Der One-vs.-Rest-(OvR)- beziehungsweise One-vs.-All (OvA)-Ansatz ist die Erweiterung der binären Klassifikation für mehrere Zielklassen. Die OvR-Methode funktioniert wie  $i$ -viele binäre Klassifikatoren, wobei  $i \in \mathbb{N}$  die Anzahl der Klassenbezeichnungen ist.

Für die erste Klassenbezeichnung wandern alle anderen  $i - 1$  Klassenbezeichnungen in die Rest-Klasse für das Training des ersten Klassifikators. Die Rest-Klasse funktioniert für OvR analog zur negativen Klasse im binären Fall: Alle Datenpunkte und deren Klassenbezeichnung, die nicht zur positiven Klasse gehören, wandern in die Rest-Klasse.<sup>xxvi</sup>

Für die Anwendung der OvR-Methode wird die Vorhersage für einen Datenpunkt  $x^{(i)}$  auf alle  $i$  Klassifikatoren gleichzeitig angewendet. Derjenige Klassifikator, der den höchsten Ausgabewert besitzt, wird als vorhergesagte Klassenbezeichnung ausgegeben.

<sup>9</sup> (Quelle: [https://gombru.github.io/2018/05/23/cross\\_entropy\\_loss/](https://gombru.github.io/2018/05/23/cross_entropy_loss/), Abgerufen am 11.6.19)

## 2.3 Einsichtige Netzwerke

Dieser Abschnitt beleuchtet lineare Modelle als Ursprung neuronaler Netzwerke. Er erklärt die Konzepte der gewichteten Eingabe, Aktivierung und Sprungfunktion sowie die Perzeptron-Lernregel. Lineare Modelle bestehen aus einer einzelnen Ausgabeeinheit, die mit der gewichteten Eingabe verknüpft ist. Sie begründen das Verständnis des einzelnen Neurons auf Basis des MCP-Neurons und Perzeptron und besitzen daher nur eine Schicht. Ihre Geschichte begann 1943 beim MCP-Neuron und dem Perzeptron. Sie stellen die Grundbausteine von linearen Modellen mit Schwellenwert, deren Weiterentwicklung die Adaline-Einheit darstellt.<sup>xxvii</sup>

### 2.3.1 Neuronen als Vorbild für maschinelles Sehen

Frühe Implementierungen einfacher Neuronen (nach dem Vorbild einfacher Zellen des visuellen Kortex)<sup>xxviii</sup> lassen sich auf die Entwicklung des MPC-Neurons sowie dessen Weiterentwicklung zum Perzeptron und zur Adaline-Einheit<sup>xxix</sup> für die binäre Klassifikation zurückführen.

Moderne Implementierungen von neuronalen Netzen zur Bilderkennung nutzen bis heute die gleiche sequentielle Struktur wie das MPC-Neuron: Von der Eingabeschicht bis zur finalen Ausgabeschicht (*Klassifizierungsschicht*; Vgl. 2.6.6 Ausgabeschicht) wird die gewichtete Summe der Eingabe (*Nettoeingabe*) mit der Anwendung des Schwellenwertes vorwärts propagiert, um den Zielwert zu berechnen (Vorwärtspropagation). Die Perzeptron-Lernregel und das Gradientenabstiegsverfahren erleichtern das Verständnis darüber, wie neuronale Netze lernen – indem sie den Fehler eines falschen Zielwertes (falsche Vorhersage) zurück bis zur Eingabeschicht propagieren und ihre Gewichte aktualisieren (Vgl. 2.5 Lernmethoden).

### 2.3.2 MPC-Neuron

McCulloch und Pittis begründeten 1943 das Modell der vereinfachten Nervenzelle mit dem Ziel, die Funktion des Gehirns zu verstehen und eine künstliche Intelligenz nachzubilden. Daher ist das McCulloch-Pittis-Neuron nach dem Vorbild einer menschlichen Nervenzelle für die Signalverarbeitung und deren Weitergabe verantwortlich. Im Allgemeinen stellt das MPC-Neuron die Implementierung eines logischen Gatters mit binärer Ausgabe dar. Erreicht die gewichtete Eingabe (*Nettoeingabe*) einen gewissen Schwellenwert  $\theta$ , werden entweder die positive Klasse oder die negative Klasse ausgegeben (Vgl. 2.2.1 Klassifikation und Regression).<sup>xxx</sup>

### 2.3.3 Perzeptron

Die Perzeptron-Lernregel von Rosenblatt ist eine Weiterentwicklung des MCP-Neuronen-Modells. Die Perzeptron-Lernregel beschreibt einen Algorithmus zum automatischen Finden der optimalen Gewichtungparameter für die Gewichtung der Eingabe. Anhand der Eingabemerkmale  $(1, x_1, x_2, \dots, x_m)$  und ihrer Gewichtungen  $(w_0, w_1, w_2, \dots, w_m)$  wird entschieden, „[...] ob ein Neuron feuert oder nicht [...]“<sup>xxxix</sup> (Vgl. Abbildung 6). Die Tatsache des feuerns oder nicht-feuerns lässt sich auf die Entscheidung einer Klassenzugehörigkeit übertragen. Gehört das Element zu einer bestimmten Klasse oder nicht? Das bedeutet, dass die Nettoeingabe mit der Anwendung des Schwellenwertes zu einer binären Ausgabe transformiert wird, die eine Ja-/Nein-Entscheidung darstellt.<sup>xxxix</sup>

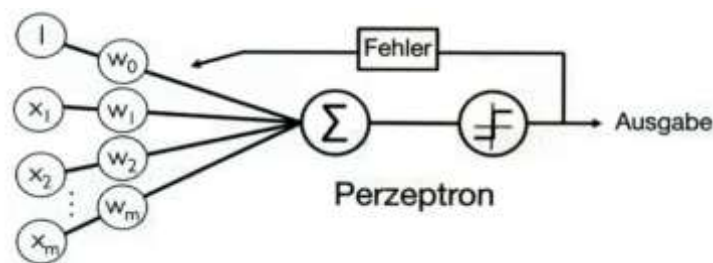


Abbildung 6: Perzeptron (Quelle: Raschka/Mirjalili, 2016, S.57)

### 2.3.4 Formale Definition des Perzeptrons

Die Aktivierungsfunktion  $\phi(z)$  stellt eine Linearkombination der Eingabewerte  $x$  und des Gewichtungsvektors  $w$  für die sogenannte Nettoeingabe  $z$  dar (Vgl. Abbildung 7, rechts):

$$(1) \quad z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{j=0}^m w_jx_j = w^T x$$

Wobei die Laufvariablen  $j$  bis  $m$  die Dimension der Eingabemerkmale  $x$  bezeichnen und die Bias-Einheit durch den Ausdruck  $w_0x_0$  repräsentiert wird. In der Literatur wird der negative Schwellenwert beziehungsweise das Nullgewicht  $w_0 = -\theta$  als Bias-Einheit bezeichnet. Das hochgestellte T des Ausdrucks  $w^T$  steht für die transponierte Gewichtungsmatrix.<sup>xxxix</sup>

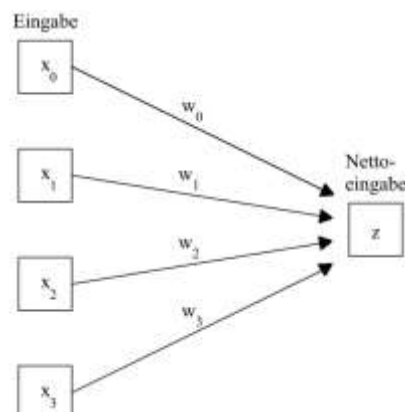


Abbildung 7: gewichtete Eingabe [Nettoeingabe] (Eigene Darstellung)

Wenn das Ergebnis der Aktivierungsfunktion den Schwellenwert  $\theta$  übersteigt, dann wird bei der binären Klassifikation die positive Klasse vorhergesagt, ansonsten die negative Klasse. Die Aktivierungsfunktion  $\phi(z)$  des Perzeptrons ist eine einfache Sprungfunktion (*engl. Heaviside*):<sup>xxxiv</sup>

$$(2) \quad \phi(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{else} \end{cases}$$

In der Regel wird ein Schwellenwert von  $\theta = 0.5$  genutzt ( $\theta \in \mathbb{R}$ ). Die wichtigste Voraussetzung für die Konvergenz des Perzeptrons ist die lineare Trennbarkeit (*des Entscheidungsraums*) der Zielklassen. Der Ausdruck der Konvergenz beschreibt dabei die Generalisierungsfähigkeit des Modells auf Basis der Trainingsdaten.<sup>xxxv</sup> Die Generalisierungsfähigkeit bezeichnet die Fähigkeit, gut mit ungesesehenen Daten zu funktionieren und den Generalisierungsfehler (*engl. generalization error*) möglichst gering zu halten.<sup>xxxvi</sup> Ohne eine lineare Entscheidungsgrenze reicht die Kapazität des Perzeptrons nicht aus, um ein Polynom höheren Grades als 1 abzubilden (zum Beispiel die quadratische Funktion).<sup>xxxvii</sup>

### 2.3.5 Perzeptron-Lernregel

Die Perzeptron-Lernregel beginnt mit der Initialisierung der Gewichte. Im zweiten Schritt werden alle Elemente der Trainingsmenge in einer Schleife durchlaufen. Zunächst geht es darum, den Zielwert anhand des Schwellenwertes der Nettoeingabe zu bestimmen und danach die Gewichte anhand der Richtigkeit des Zielwerts (*vorhergesagte Klassenbezeichnung*) zu aktualisieren.<sup>xxxviii</sup>

1. Initialisierung aller Gewichte  $w$  mit zufälligen Werten oder Null.
2. Für jedes Element  $x^{(i)}$  der Trainingsmenge:
  - a. Berechne den Zielwert  $\hat{y}$  (*Ausgabewert*) anhand des Schwellenwertes
  - b. Aktualisiere die Gewichte  $w_j$  im Gewichtsvektor  $w$  nach der Regel:

$$(3) \quad w_j = w_j + \Delta w_j$$

Die Veränderung  $\Delta w_j$  wird anhand der Perzeptron-Lernregel berechnet:

$$(4) \quad \Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

Der Parameter  $\eta$  ist die Lernrate (*engl. learning rate*) im Intervall  $[0,1]$ . Der Ausdruck  $y$  steht für die Klassenbezeichnung, wobei zwischen  $y^{(i)}$  (*tatsächliche Klassenbezeichnung*) und  $\hat{y}^{(i)}$  (*vorhergesagte Klassenbezeichnung*) des  $i$ -ten Elements unterschieden wird. Bei der korrekten Vorhersage der Klassenbezeichnung werden die Gewichtungen nicht verändert:

$$(5) \quad \text{korrekte Vorhersage: } \Delta w_j = \eta (1 - 1) x_j^{(i)} = 0.$$

Stellt sich die Vorhersage als falsch heraus, sind die Gewichtungen zur Zielklasse zu verschieben<sup>xxxix</sup>:

$$(6) \quad \text{falsche Vorhersage: } \Delta w_j = \eta (1 - -1) x_j^{(i)} = \eta (2) x_j^{(i)}.$$

Mit Blick auf das Verständnis der beiden multiplikativ verknüpften Faktoren  $\eta$  und  $x_j^{(i)}$  ist die gleichmäßige Aktualisierung der Gewichte hervorzuheben. Je höher die Ausprägung der Lernrate  $\eta$  (oder die Eingabe  $x$ ) ausfällt, desto stärker werden die Gewichte aktualisiert. Die Lernrate beeinflusst neben der Schrittweite auch die Konvergenzgeschwindigkeit während des Gradientenabstiegs (Vgl. Abbildung 6) – je nach Ausprägung werden die Gewichte in kleinen oder größeren Schritten angepasst. Der Einfluss der Lernrate wird in Kapitel 2.5 (Vgl. 2.5 Lernmethoden) genauer untersucht.<sup>xl</sup>

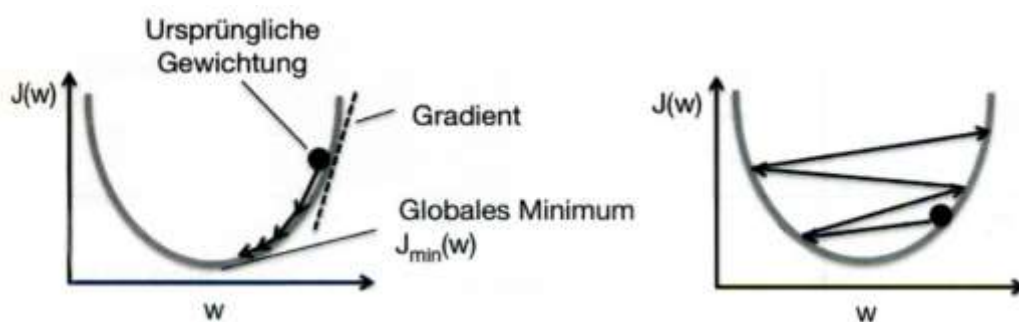


Abbildung 8: optimale Lernrate (Quelle: Raschka/Mirjalili, 2016, S.63)

### 2.3.6 ADALINE: Adaptive Lineare Neuronen

Die wesentliche Neuerung zwischen Adaline und dem Perzeptron ist, dass die Aktualisierung der Gewichte nicht allein von der Sprungfunktion abhängt (Anwendung des Schwellenwerts), sondern im Vorhinein durch eine lineare Aktivierungsfunktion beeinflusst wird (Vgl. Abbildung 9).<sup>xli</sup> Die Aktivierung bei Adaline ist mit verschiedenen Zuständen vergleichbar, da sie einer reellen Zahl entspricht.<sup>xlii</sup> Um die Gewichtungen zu aktualisieren, vergleicht Adaline die tatsächlichen Klassenbezeichnungen  $y^{(i)}$  mit der Aktivierungsfunktion  $\phi(z)$ , wohingegen das Perzeptron die tatsächlichen Klassenbezeichnungen  $y^{(i)}$  mit der vorhergesagten Klassenbezeichnung  $\hat{y}^{(i)}$  vergleicht.

Adaline vergleicht eine reelle Zahl, das Perzeptron die ganzzahlige Klassenbezeichnung mit der tatsächlichen Klassenbezeichnung.<sup>xliii</sup> Das Perzeptron aktualisiert inkrementell nach jedem einzelnen Trainingsobjekt nach dem Prinzip des Online-Learnings, Adaline gesammelt nach allen Objekten im Sinne der Stapelverarbeitung (Vgl. 2.5.3 Stochastisches Gradientenabstiegsverfahren).<sup>xliv</sup>

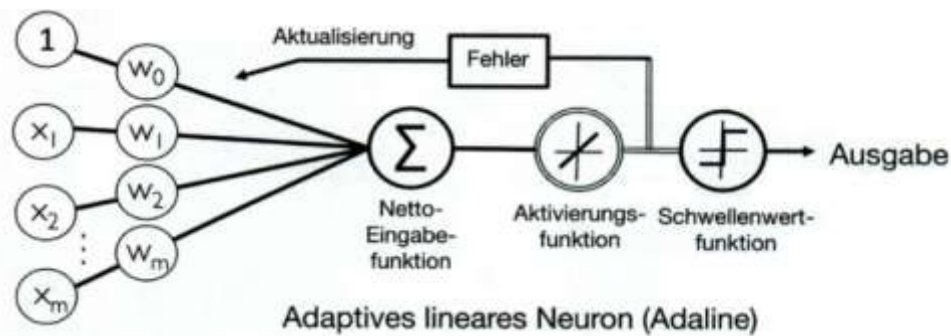


Abbildung 9: Adaline-Einheit (Quelle: Raschka/Mirjalili, 2016, S.57)

Die Aktivierungsfunktion gibt die Identität der Nettoeingabe unverändert weiter, da die Identität  $f(x) = x$  der Nettoeingabe nichts bewirkt:  $\phi(z) = z$ . In den vergangenen Jahrzehnten sind verschiedene Aktivierungsfunktionen entwickelt worden, die sich in ihrer Konvergenzgeschwindigkeit und Differenzierbarkeit unterscheiden (Vgl. 2.6.2 Aktivierung). Besondere Tragweite erlangt Adaline durch die Minimierung der Straffunktion<sup>xlv</sup> im Sinne einer zu optimierenden Zielfunktion. Die Straffunktion  $J(w)$  entspricht der Adaline-Zielfunktion:

$$(7) \quad J(w) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2,$$

wobei  $\phi(z^{(i)})$  die Aktivierung einer gewichteten initialen Eingabe  $x^{(i)}$  entspricht.  $J(w)$  ist konvex und differenzierbar – im Gegensatz zur Sprungfunktion. Die Gewichte werden optimiert, indem man einen Schritt des Gradientens  $\nabla J(w)$  entlang Straffunktion  $J(w)$  entlanggeht (Vgl. Abbildung 10):  $w = w + \Delta w$ , wobei zur Veränderung der Gewichtung (*delta*)  $\Delta w$  (Vgl. Formel 8) der negative Gradient mit der Lernrate  $\eta$  multipliziert wird:

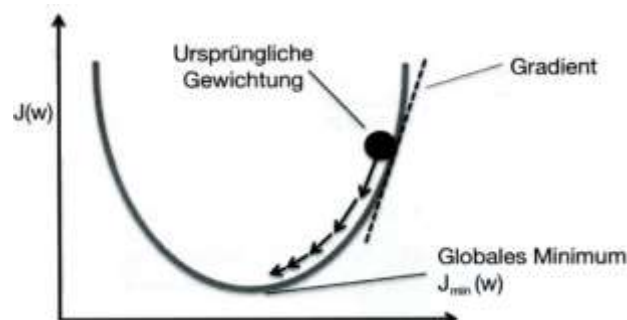
$$(8) \quad \Delta w = -\eta \nabla J(w).$$

Die Zielfunktion wird für das Optimierungsverfahren benutzt, indem die partiellen Ableitungen nach den Gewichtungen gebildet werden:

$$(9) \quad \frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)},$$

damit das einzelne Gewicht nach der Adaline-Lernregel aktualisiert werden kann.<sup>xlvi</sup>

$$(10) \quad \Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}.$$

Abbildung 10: Gradientenabstiegsverfahren (engl. *gradient descent*) (Quelle: Raschka/Mirjalili, 2016, S.58)

Die Definition und Minimierung der Straffunktion ist die Grundlage für fortgeschrittene Klassifizierungsalgorithmen wie die logistische Regression, SVM und kNN. Die logistische Regression unterscheidet sich von Adaline und Perzeptron lediglich darin, dass sie eine andere Aktivierungsfunktion verwendet: die Sigmoid-Aktivierung (Vgl. 2.6.2 Aktivierung).<sup>xlvii</sup>



## 2.4 Mehrschichtige Netzwerke

Ausgehend von der Erweiterung des einfachen Perzeptrons zu mehrschichtigen Strukturen des MLPs beginnt dieser Abschnitt mit der sequentiellen Natur des MLPs. Dazu werden seine Struktur und die Vorhersage von Zielwerten (beginnend bei der Nettoeingabe) erklärt, um das Verständnis zu tiefen neuronalen Netzwerken im nächsten Abschnitt zu erleichtern (Vgl. 2.5 Lernmethoden).

### 2.4.1 Vom MLP zum (vanilla) Feed-Forward Netzwerk

Wie in Kapitel 2.3 erläutert ist ein künstliches Neuron die Kombination aus Perzeptron-Einheit und der ADALINE-Lernregel. Das einzelne Neuron (Knoten) ist der Ursprung des Aufbaus komplexer Netzwerke aus hierarchisch-verketteten Schichten, die viele Knoten enthalten. Mit der Erweiterung des Perzeptrons zum mehrschichtigen Perzeptron (*engl. multi-layer perceptron, MLP*) als Verallgemeinerung des linearen Modells ist die Beschränkung des Problems und des Lösungsraumes auf lineare Trennbarkeit aufgehoben.<sup>xlvi</sup>

Für eine nichtlineare Entscheidungsgrenze werden einfach mehr Knoten und (oder) Schichten sowie eine nichtlineare Aktivierungsfunktion hinzugefügt, bis die nötige Modellkomplexität zur Konvergenz mit einem Hang zur Überanpassung erreicht wird.<sup>xlix</sup> Mit genügend Einheiten und Schichten kann letztlich jede Funktion gelernt werden – nicht nur lineare wie beim Perzeptron. Die nichtlineare Aktivierung wird hierbei als Trick angesehen, um kompliziertere Funktionen als lineare lernen zu können, wie zum Beispiel *ReLU* oder *Tanh* (Vgl. 2.6.2 Aktivierung).<sup>l</sup>

Wenn das Modell die nötige Komplexität zur Konvergenz erreicht und zu einer Überanpassung neigt (Vgl. Abbildung 11, *rechts*), wird Regularisierung angewandt (Vgl. 2.6.5 Regularisierungsschicht). Ziel ist eine absichtliche Einschränkung des Modells. Die Komplexität wird so lange gesenkt, bis die Überanpassung eliminiert ist (Vgl. Abbildung 11, *mitte*).<sup>li</sup>

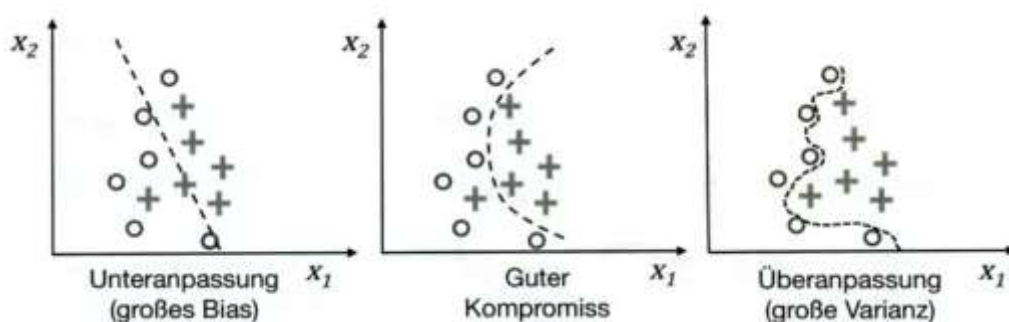


Abbildung 11: Kompromiss zwischen Unter- und Überanpassung (Quelle: Raschka/Mirjalili, 2016, S.94)

### 2.4.2 Sequentielle Natur und Vorwärtspropagation

Durch das Hinzufügen einer (oder mehrerer) verdeckter Schicht(en) wird die Aktivierung der Nettoeingabe zu einem mehrstufigen und sequentiellen Prozess. Die Berechnung einer Serie gewichteter Summen ist mathematisch äquivalent zur Berechnung exakt einer gewichteten Summe. Die Formel wird länger, doch die Anzahl der Eingabeparameter bleibt gleich. Damit die Aktivierung der Ausgabeschicht berechnet werden kann, ist die Aktivierung aller vorherigen Einheiten notwendig, welche wiederum auf die Aktivierung der vor-vorherigen Schicht bis zur Eingabeschicht angewiesen sind (Vgl. Abbildung 12).<sup>lii</sup>

Von der Eingabeschicht (Vgl. Abbildung 12, *links*) betrachtet wird die gewichtete Summe der Eingabemerkmale (die Nettoeingabe) auf die nachfolgenden Knoten abgebildet (Vgl. 2.3.4 Formale Definition des Perzeptrons). Hierbei wird zunächst die Aktivierung der ersten verdeckten Schicht  $h_i^{(l)}$  mit der gewichteten Eingabe bestimmt, wobei  $i$  den Index der verdeckten Einheit innerhalb der verdeckten Schicht  $l = 1$  beschreibt. Alle Aktivierungen der ersten Schicht  $h_{i=0}^{(l=1)}, h_1^{(1)}, h_2^{(1)}$  (Vgl. Abbildung 12, *Verdeckte Schicht 1*) sind die Eingabe der nächsten (verdeckten) Schicht  $h_i^{(2)}$ . Die Einheiten der ersten verdeckten Schicht bilden erneut die Nettoeingabe als Eingabe der nachfolgenden Schicht  $h_i^{(l+1)} = h_i^{(2)}$ . Dieses Vorgehen wird bis zur finalen Ausgabeschicht  $\hat{y}$  (Klassifizierungsschicht) wiederholt.<sup>liii</sup>

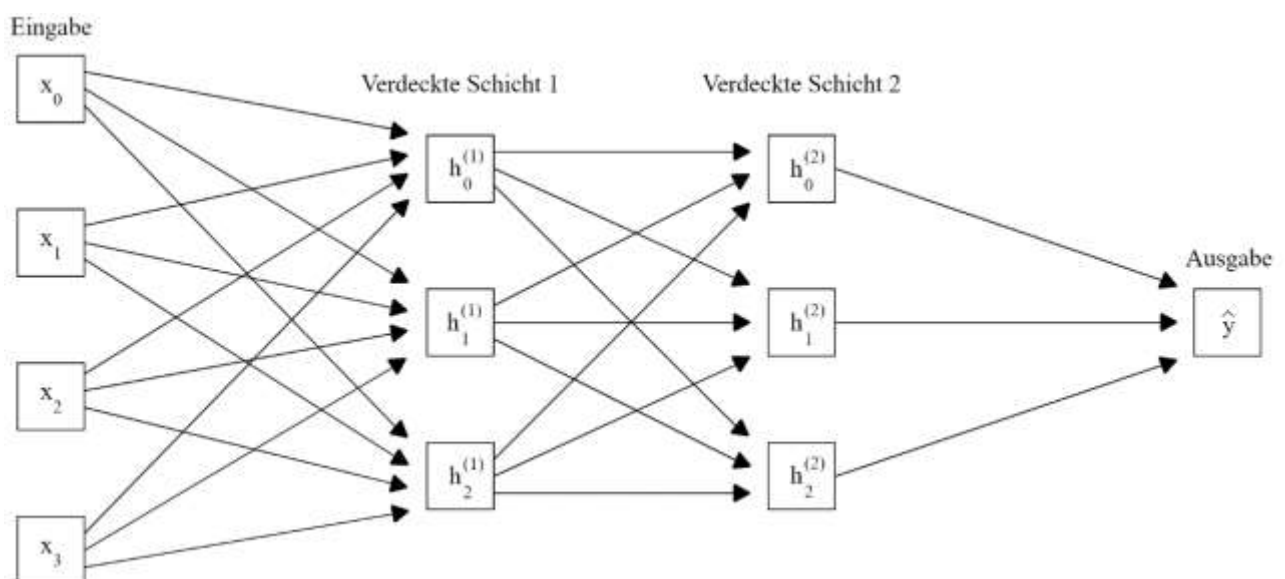


Abbildung 12: MLP mit zwei verdeckten Schichten (Eigene Darstellung)<sup>liv</sup>

Das Verfahren zur Vorhersage des Zielwerts nennt sich Vorwärtspropagation (*engl. Forward Propagation*). Für die Aktivierung der Ausgabeschicht  $\hat{y}$  werden die schichtweisen Gewichtungen des

Netzwerks sequentiell auf die initiale Eingabe  $x$  angewendet (Vgl. Abbildung 13). Dadurch lässt sich der finale Zielwert in der Klassifizierungsschicht vorhersagen (Vgl. 2.6.6 Ausgabeschicht).

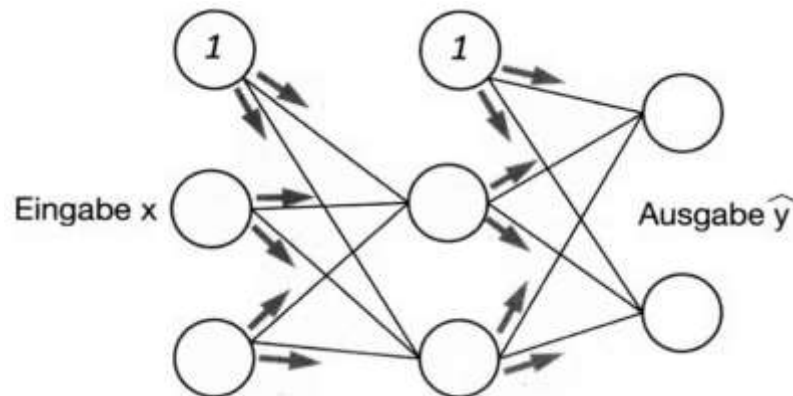
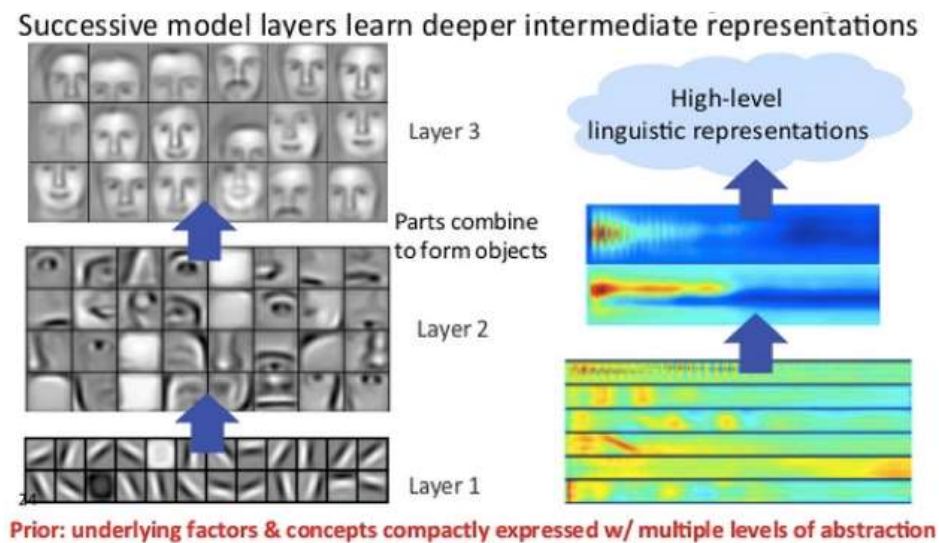


Abbildung 13: Vorwärtspropagation (Quelle: Raschka/Mirjalili, 2016, S.415)

### 2.4.3 Anzahl von Parametern

Neuronale Netzwerke zur Klassifikation von Bildern besitzen deutlich mehr Parameter in Form von Gewichtungen (Gewichtungskoeffizienten). Während eines der ersten CNNs (*LeNet*) zur Klassifikation von handschriebenen Ziffern (MNIST-Datensatz) mit 60.000 Parametern<sup>lv</sup> aus kam, nutzen heutige CNNs wie InceptionV3 für die Klassifikation des ImageNet-Datensatzes<sup>lvi</sup> bis zu mehrere zehn Millionen Parameter<sup>lvii</sup> (Vgl. Tabelle 4). LeNet war eines der ersten CNNs, die das Prinzip der diskreten Faltung anwendeten. Für die Erkennung von Mustern, zum Beispiel einer horizontalen Linie, wird eine Reihe von Filtern im Rahmen der Convolution-Operation auf die Eingabedaten angewendet, um Merkmalskarten zu erzeugen (Vgl. 2.6.1 Diskrete Faltung und 2.5 Lernmethoden). Ausgehend von der vorangegangenen Aktivierung erkennen nachfolgende Filter immer komplexere Strukturen, die eine Kombination der vorangehenden Filter darstellen (Vgl. Abbildung 14). Aus vertikalen und horizontalen Linien entstehen Kanten. Kantenkombinationen werden zu Objekten und Objekte zu Objektkombinationen.

Abbildung 14: Merkmalshierarchie<sup>10</sup>

InceptionV3 funktioniert ähnlich wie LeNet, ist dabei aber deutlich komplexer im Aufbau. Sein Eingabeformat (*engl. input shape*) beträgt mindestens  $32 \times 32$  und maximal  $299 \times 299$  Pixel. Damit die finale Klassifizierungsschicht die variablen Eingaben nutzen kann, müssen die Eingaben der Klassifizierungsschicht immer das gleiche Format haben. Manche CNN-Implementierungen verwenden einen variablen Versatz (*engl. offset*) im Sinne einer variierenden Schrittlänge zwischen den Pooling-Regionen, um das gleichförmige Eingabeformat der Klassifizierungsschicht sicherzustellen.<sup>lviii</sup> Die Architektur von InceptionV3 wird in Kapitel 4.1 genauer untersucht (Vgl. 4.1.4 InceptionV3).

## 2.4.4 Tiefe Neuronale Netzwerke

Die Kurzbezeichnung für tiefe neuronale Netzwerke lautet neuronale Netzwerke (*engl. neural networks bzw. deep learning*). CNNs sind sequentielle Modelle. Sie besitzen weniger Parameter (Gewichtungen) als das klassische MLP, da sie weniger vollständig verknüpfte Schichten beinhalten. Die Eingabe fließt in einem Informationsfluss durch das Netzwerk – zwischen Ein- und Ausgabeschicht immer weiter in tiefere (spätere) Schichten und enthaltene Knoten propagiert und wird dort verarbeitet (Vgl. Abbildung 13: Vorwärtspropagation). Abgesehen von der Iteration durch alle Trainings- und Validierungsdaten wenden CNNs keine Schleifen an. In Kapitel 4.1 findet eine Detailanalyse verschiedener CNN-Implementierungen statt (Vgl. 4.1 Unterschiede moderner CNN-Implementierungen).

<sup>10</sup> (Quelle: <https://skymind.ai/wiki/neural-network>, Abgerufen am 11.6.19)

## 2.5 Lernmethoden

Durch den Vergleich des vorhergesagten Zielwerts im Zuge der Vorwärtspropagation (Vgl. Abbildung 13) und der anschließenden Anpassung der Gewichte im Zuge des Gradientenabstiegs wird ein mehrstufiges Netzwerk analog zur einschichtigen Adaline-Einheit mittels Minimierung der Zielfunktion optimiert (mehrstufige Perzeptron-Lernregel; Vgl. Abbildung 15). Die partiellen Ableitungen der Straffunktion (nach den einzelnen Gewichten) bestimmen den Optimierungsschritt zur Anpassung der Gewichte; sie zeigen immer entgegen der Richtung des Gradienten  $\nabla J(w)$  (Vgl. Abbildung 10).<sup>lix</sup>

### 2.5.1 Aktualisierung der Gewichte

Im Quellcode beginnt das Training mit dem Aufruf der *fit*-Methode auf ein kompiliertes Modell. Das Training ist in mehrere Epochen aufzuteilen, wobei die Zahl der Epochen von zehn bis in die Tausende reichen kann. Eine Epoche ist ein zweistufiger Prozess. Zunächst werden alle Trainingsdaten betrachtet, danach die Validierungsdaten. Eine Trainingsepoche umfasst die einmalige Betrachtung und Vorhersage des Zielwerts jedes einzelnen Datenpunktes aus der Menge der Trainingsdaten (Vgl. 2.4.2 Sequentielle Natur und Vorwärtspropagation). Nach jeder Vorhersage wird der Fehler des vorhergesagten Zielwerts im Vergleich zum tatsächlichen bestimmt. Dazu werden die Gewichtungen zum Vorteil der Zielklasse angepasst (Vgl. 2.3.5 Perzeptron-Lernregel). Für die effiziente Aktualisierung der Gewichte wird das Backpropagation-Verfahren (Vgl. Abbildung 15) nach dem Vorbild der Perzeptron Lernregel angewendet (Vgl. 2.5 Lernmethoden).

Wenn alle Datenpunkte des Trainings einmal betrachtet wurden, beginnt der Bewertungsschritt. Der Bewertungsschritt besitzt keinerlei Einfluss auf die Gewichtungen (Vgl. 3.1.7 Bewertung des Modells). Im Anschluss beginnt eine neue Epoche (Vgl. 3.2.6 Trainingsphase #1 [Pre-Training]).

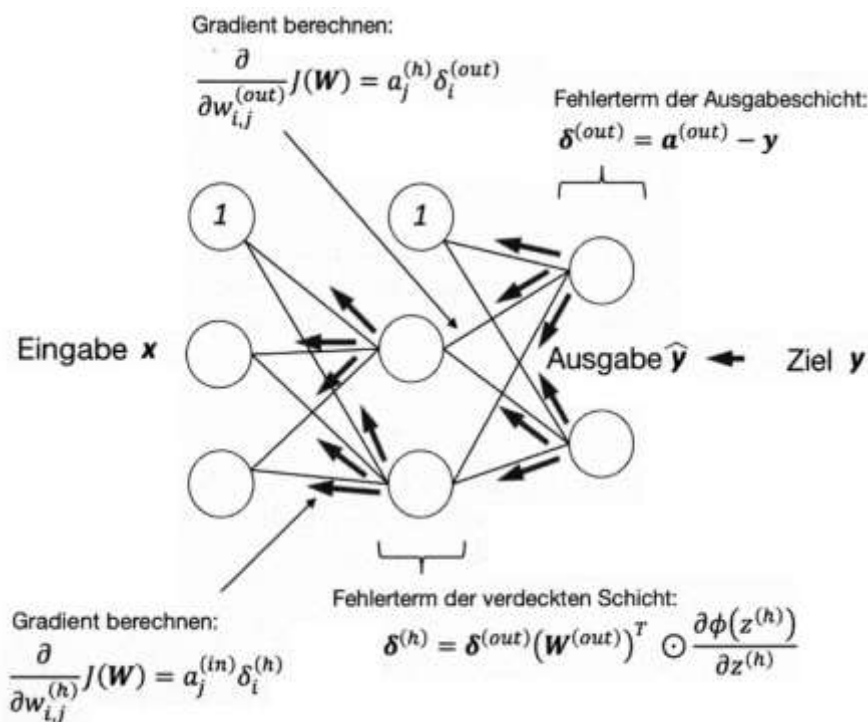
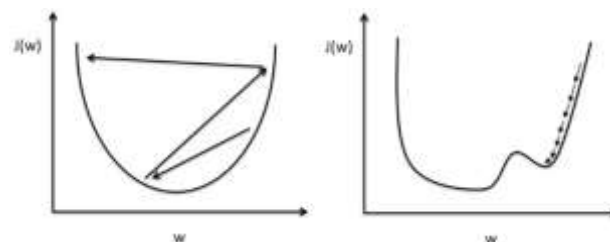


Abbildung 15: Das Backpropagation-Verfahren (Quelle: Raschka/Mirjalili, 2016, S.418)

## 2.5.2 Minimierung der Straffunktion

Wie in Kapitel 2.3.6 beschrieben, wird die Adaline-Straffunktion als Zielfunktion (optimizer) verwendet und zur Optimierung der Gewichtungen minimiert. Dabei sind die Lernrate  $\eta$  und die Anzahl der Epochen die zwei Ersten zu optimierenden Hyperparameter (Vgl. 3.1.3 Auswahl des Modells).

Abbildung 16: Einfluss der Lernrate auf die Minimierung der Straffunktion<sup>11</sup>

Wenn die Höhe der Lernrate nicht optimal ist, dann bewegt man sich entweder sehr langsam in Richtung des Minimums der Straffunktion, was einer sehr langsamen Konvergenz entspricht (Vgl. Abbildung 16, *rechts*). Oder man schießt über das Minimum hinaus (Vgl. Abbildung 16, *links*). Der Einsatz von Merkmalstandardisierung hilft dabei, das globale Minimum effizienter zu erreichen, da durch Daten-Standardisierung im Sinne der Standardnormalverteilung (Vgl. 3.1.5 Daten transformieren) weniger Optimierungsschritte erforderlich sind (Vgl. Abbildung 17, *rechts*).<sup>1x</sup>

<sup>11</sup> (Vgl. [https://sebastianraschka.com/Articles/2015\\_singlelayer\\_neurons.html](https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html), Abgerufen am 11.6.19)

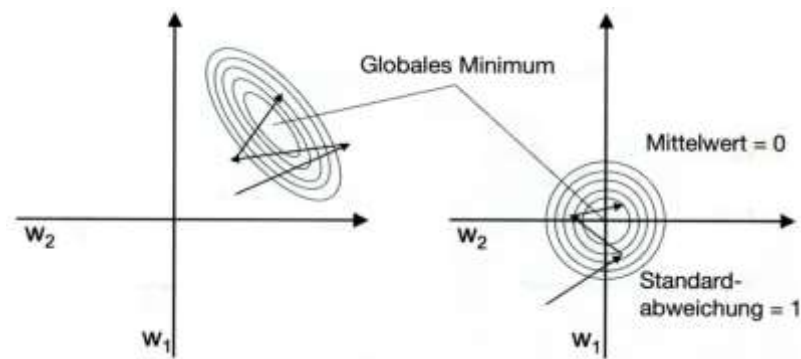


Abbildung 17: Einfluss der Merkmalsstandardisierung auf den Gradientenabstieg (Quelle: Raschka/Mirjalili, 2016, S.64)

### 2.5.3 Stochastisches Gradientenabstiegsverfahren

Die Trainings- und Validierungsdaten werden in Losen (*engl. batches*) organisiert. Ein einzelnes Los ist eine unabhängige Teilmenge des Datensatzes. Die Gewichte des Netzwerks werden nach der Betrachtung aller Objekte eines Loses aktualisiert. Es gibt zwei Extremformen und eine Mischform zur Auswahl der Losgröße (*engl. batch-size*): Online-Learning, Stapelverarbeitung (*engl. batch-learning*) und Mini-Stapelverarbeitung (*engl. mini-batch-learning*).<sup>lxi</sup>

Online-Learning bedeutet, die Gewichtungen nach jedem einzelnen Trainingsdatenpunkt  $x^{(i)}$  zu aktualisieren. Die Losgröße beträgt in diesem Fall 1. Wenn der Trainingsdatensatz  $n \in \mathbb{N}$  viele Datenpunkte enthält, dann werden die Gewichtungen des Netzwerks  $n$ -Mal pro Epoche aktualisiert. Häufige Aktualisierungen haben den Vorteil schneller zu konvergieren, obwohl das Verfahren mit einem erhöhten Rauschen verbunden ist. Diese erhöhte Varianz unterstützt die Überwindung lokaler Minima.<sup>lxii</sup>

Wenn die Losgröße der Anzahl der Datenpunkte im Trainingsdatensatz entspricht ( $\text{batch-size} = n$ ), dann wird das als Stapelverarbeitung bezeichnet und die Gewichtungen lediglich einmal pro Epoche aktualisiert.

In der Praxis ist die Stapelverarbeitung in Verbindung mit einer solch hohen Losgröße (je nach Datensatz und Hardware-Ausstattung) überhaupt nicht möglich, da der gesamte Trainingsdatensatz und das Netzwerk nicht in den Arbeitsspeicher passen. Wenn der Speicher nicht ausreicht, bricht das Training während der Laufzeit ab und wirft einen Out-Of-Memory-Fehler. Die Losgröße sollte reduziert werden. In der Regel wird die Losgröße als Mittelweg zwischen beiden Extrema  $1 \leq \text{batch-size} \leq n$  – aus dem Intervall  $[1, n]$  gewählt und Mini-Stapelverarbeitung genannt.

## 2.6 Bausteine von CNNs

Dieser Abschnitt befasst sich mit allen Bausteinen von CNNs. Von der diskreten Faltung (*engl. convolution*), Pooling (*engl. subsampling*) bis hin zu Regularisierungsschichten über vollständig verknüpfte Schichten (*engl. fully connected layer*) bis zur finalen Ausgabe in der Klassifizierungsschicht.

Aufgrund ihrer sequentiellen Natur (ohne Schleifen) sind CNN-Implementierungen sehr gut anhand ihrer Architektur vergleichbar.<sup>lxiii</sup> Der Aufruf von `model.summary()` auf ein kompiliertes Modell liefert eine Zusammenfassung der Schichten sowie der dort angewandten Operationen, des Ein-Ausgabeformats und der Anzahl der Parameter je Schicht – beginnend bei der Eingabeschicht bis zur Ausgabe. Im Allgemeinen besitzt ein CNN sieben Bausteine; die Schichten zwischen Ein- und Ausgabe können sich wiederholen (Vgl. Abbildung 18 und Vgl. 3.2.3 Modellauswahl, *letzte Zeile*):

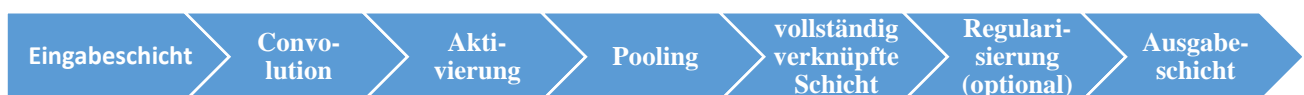


Abbildung 18: sequentielle Natur des CNNs (Eigene Darstellung)

Alle Schichten zwischen Ein- und Ausgabeschicht bezeichnet man als verdeckte Schicht (*engl. hidden layer*).<sup>lxiv</sup> Frühe Schichten (näher zur Eingabeschicht) aus Convolution-Blöcken liefern der nächsten Schicht Ansammlungen aus Merkmalskarten (*engl. feature map*) (Vgl. Abbildung 19). Diese sind das Resultat der Filteranwendung (*engl. kernel*) innerhalb der Convolution-Schicht. Merkmalskarten werden als rezeptive Felder bezeichnet und nach der Aktivierung (Vgl. 2.6.2 Aktivierung) an die Pooling-Schicht weitergegeben.<sup>lxv</sup>

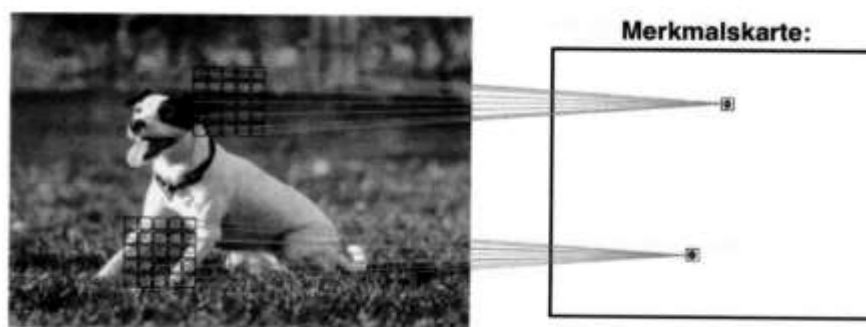


Abbildung 19: Convolution zur Bildung von Merkmalskarten (Quelle: Raschka/Mirjalili, 2016, S.490)

Nach dem Pooling (Vgl. 2.6.3 Pooling) werden die Daten an eine vollständig verknüpfte Schicht übergeben (Vgl. 2.6.4 Vollständig verknüpfte Schicht). Auf eine vollständig verknüpfte Schicht folgt in der Regel entweder eine Regularisierungsschicht oder eine weitere vollständig verknüpfte Schicht



(wie bei AlexNet Vgl. 4.1.2 AlexNet). Im letzten Schritt erreichen die Daten die Ausgabeschicht (Vgl. 2.6.6 Ausgabeschicht).

Ein wichtiger Baustein von CNNs ist der Convolution-Block (*engl. convolution block*) im Sinne einer komplexen Zelle.<sup>lxvi</sup> Er setzt sich aus drei einfachen Zellen (sequentiellen Operationen) zusammen:<sup>lxvii</sup>

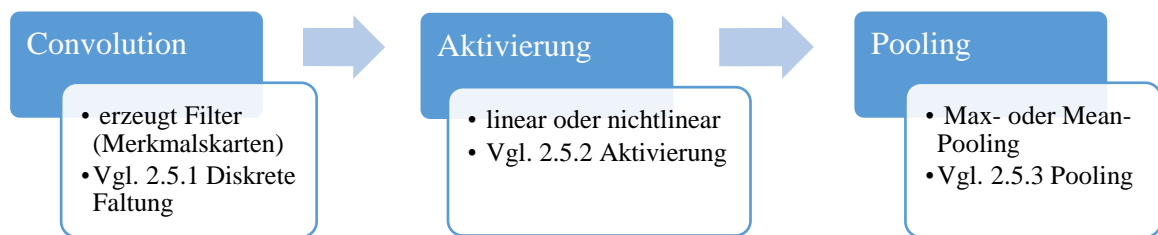


Abbildung 20: Convolution-Block (Eigene Darstellung)<sup>lxviii</sup>

Im Convolution-Block (Vgl. Abbildung 20) eines CNN sinkt im Allgemeinen die Auflösung relativ zur Eingabe. Zugleich steigt die Zahl der Kanäle – je nach Ausprägung der Convolution-Hyperparameter, wobei die Pooling-Operation (unter Einhaltung der lokalen Invarianz, Vgl. 2.6.3 Pooling) die Größe der Merkmalskarten in der Regel halbiert (Vgl. Abbildung 50, *roter Kasten oben*).<sup>lxix</sup>

## 2.6.1 Diskrete Faltung

Die diskrete Faltung (*engl. convolution*) arbeitet unter dem Erhalt intrinsischer Informationen, denn sie erhält die Struktur der Eingabe: die drei Achsen Höhe, Breite und Farbkanäle.<sup>lxx</sup> Dabei bietet die diskrete Faltung drei Vorteile gegenüber der Matrixmultiplikation eines vollständig verknüpften MLPs.<sup>lxxi</sup>

1. Spärliche Wechselwirkung (*engl. sparse interaction*), da nur wenige Einheiten zur Erzeugung der Ausgabe beitragen.<sup>lxxii</sup>
2. Gemeinsame Nutzung von Parametern (*engl. parameter sharing, tied weights*), weil die gleichen Gewichte an vielen Stellen der Eingabe genutzt werden.<sup>lxxiii</sup>
3. Äquivalente Repräsentationen (*engl. equivariant representations*), bedeuten eine gleichförmige Änderung der Ausgabe, wenn sich die Eingabe ändert.<sup>lxxiv</sup>

### 2.6.1.1 eindimensionale diskrete Faltung

Die Formel der diskreten Faltung zweier eindimensionaler Vektoren, Eingabe  $x$  und Filter  $w$ :

$$(11) \quad y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i - k] w[k]$$

Wobei der Index  $i$  in den eckigen Klammern alle Elemente der Ausgabe  $y = x * w$  durchläuft und den Index  $k$  aller Merkmale (Knoten) iteriert. Eine einzelne Convolution-Schicht wird in der Regel durch die Angabe von fünf Parametern initialisiert:<sup>12</sup>

1. Die Anzahl der Filter  $f$  (*engl. filters*) legt die Dimension der Ausgabe fest.
2. Die Filtergröße  $m$  (*engl. kernel size*) bestimmt die Größe des rezeptiven Feldes.
3. Mit der Schrittweite  $s$  (*engl. stride*) wird die Position der nächsten Filteranwendung beeinflusst.
4. Der Füllungsparameter  $p$  (*engl. padding*) ermöglicht es, die Eingabe künstlich zu vergrößern (seitlich mit Nullen auffüllen), um das Ausgabeformat zu beeinflussen.
5. Aktivierungsfunktion  $\phi(z)$  (Vgl. 2.6.2 Aktivierung)

Die diskrete eindimensionale Faltung ist eine Anwendung eines Filters auf die Eingabe. Sie führt zur Ausgabe einer Merkmalskarte je Filter (Vgl. Abbildung 21; Vgl. Abbildung 19):

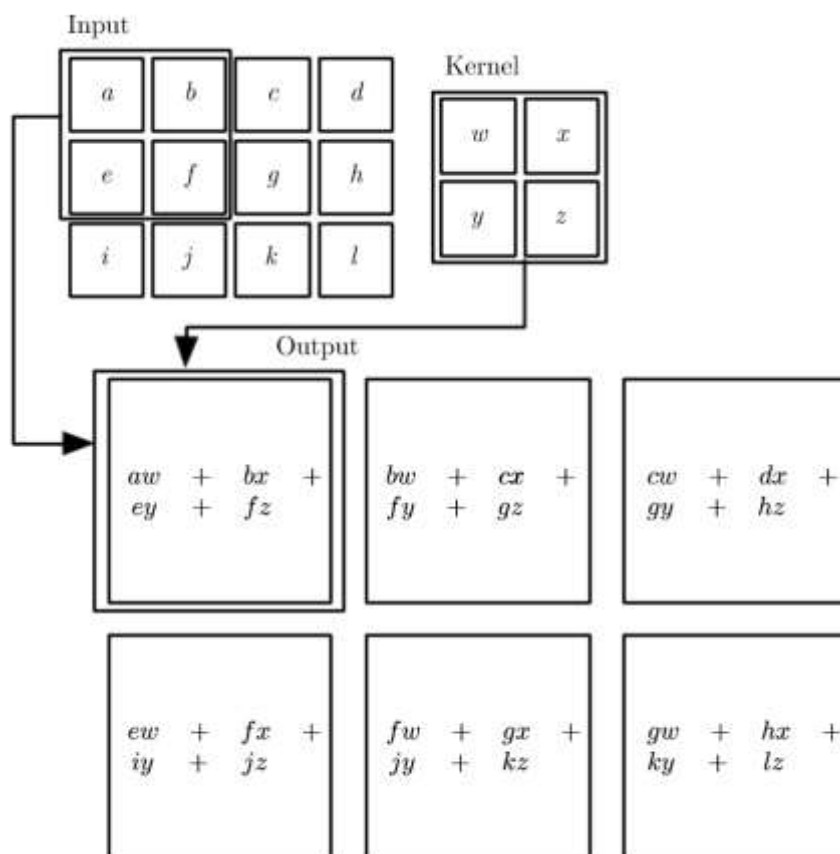


Abbildung 21: eindimensionale diskrete Faltung (Quelle: Goodfellow et al. 2016, S.330)

Abbildung 21 zeigt den einfachsten Fall  $[m = 2; s = 1; p = 0]$ <sup>lxxv</sup>. Der Filter (Vgl. Abbildung 21, *Kernel*) wird nacheinander über alle Positionen der Eingabe (Vgl. Abbildung 21, *Input*) geschoben und erzeugt schrittweise die sechs einzelnen Positionen der Merkmalskarten (*Output*), eine pro Filter  $f$ .

<sup>12</sup> (Vgl. <https://keras.io/layers/core/>, Abgerufen am 11.6.19)

Die vier Filterelemente des Kernels  $[w, x, y, z]$  werden zunächst auf die ersten vier Elemente der Eingabe angewendet  $[a, b, c, f]$  und anschließend für das erste Element der Ausgabe summiert  $[aw + bx + cy + fz]$ .

Anschließend wird der Kernel um eine Position nach rechts verschoben und erneut angewendet (Eingabe  $[b, c, f, g]$ ) – so lange bis der Filter nicht mehr vollständig in eine Eingabespalte passt. Daher beträgt im dritten Schritt die (letzte) Eingabe der ersten Spalte  $[c, d, g, h]$ . Im vierten Schritt muss darum eine Spalte nach unten gewechselt werden (Eingabe  $[e, f, i, j]$ ). Dieses Vorgehen wird wiederholt, bis die gesamte Eingabe betrachtet wurde. Die Ausgabe hat das Format:

$$(12) \quad o = \left\lfloor \frac{n+2p-m}{s} \right\rfloor + 1.$$

Der Parameter  $n$  beschreibt die Größe des Eingabevektors mit Padding  $p$ , Filtergröße  $m$  und Schrittweite  $s$ . Bei einer größeren Schrittweite ( $s = 2$ ) wird die Eingabe  $[b, c, f, g]$  hingegen im zweiten Schritt übersprungen. Die Eingabe lautet stattdessen  $[c, d, g, h]$ . Das Ausgabeformat sinkt in diesem Beispiel um eine Dimension, von  $2 \times 3$  auf  $2 \times 2$ , da der Filter einer größeren Schrittweite nur noch einmal in jede Ecke passt.

Die Padding-Strategie beeinflusst einerseits die Art und Weise, wie Randpixel gewichtet werden, und andererseits das Ausgabeformat der Filteranwendung. Eine gültige Padding-Strategie wählt den Padding-Parameter  $p \geq 0$  nach folgenden Kriterien (Vgl. Abbildung 22):<sup>lxxvi</sup>

- *Full-Padding* ( $p = m - 1$ ) erhöht die Dimension der Ausgabe.
- *Same-Padding* erzeugt das gleiche Ausgabeformat wie die Eingabe.
- *Valid-Padding* mit  $p = 0$  bedeutet kein Padding.

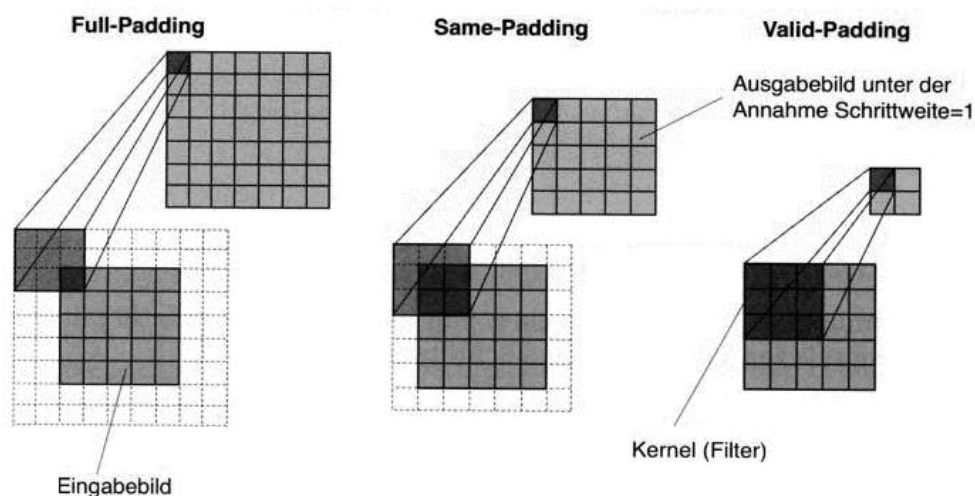


Abbildung 22: Padding-Strategie (Quelle: Raschka/Mirjalili, 2016, S.495)

### 2.6.1.2 mehrdimensionale diskrete Faltung

Der mehrdimensionale Raum arbeitet mit den Matrizen der Gewichtungen  $W$  und der Eingabe  $X$ , wobei  $n_1 * n_2$  die Dimension der Eingabe und  $m_1 * m_2$  die Dimension des Filters beschreibt und der Filter allgemein kleiner als die Eingabe ist ( $m \leq n$ ).

$$(13) \quad Y = X_{n_1 * n_2} * W_{m_1 * m_2}$$

$$(14) \quad Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

Im mehrdimensionalen Fall wird die Formel der diskreten Faltung um einen weiteren Index modifiziert, da der Filter nun auf mehrere Dimensionen gleichzeitig angewandt wird. Für den dreidimensionalen Eingang eines Farbbildes wird jeder einzelne Filter  $f$  nun in Form eines dreidimensionalen Würfels auf alle Dimensionen (Farbkanäle) gleichzeitig angewendet. Er erzeugt pro Filterwürfel  $f$  eine Merkmalskarte.<sup>lxxvii</sup>

### 2.6.2 Aktivierung

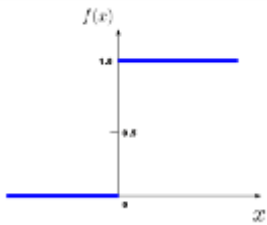
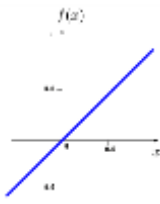
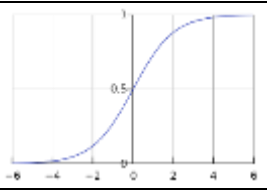
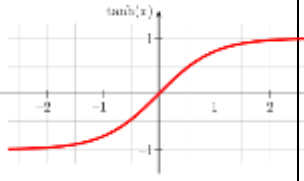
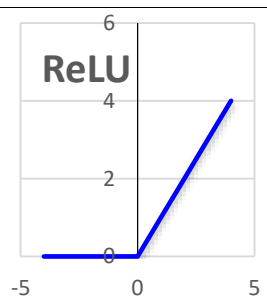
Die historische Entwicklung der Aktivierungsfunktion, beginnend beim Schwellenwert des Perzeptrons bis hin zur nichtlinearen ReLU-Einheit, zeigt die Herausforderungen, mit denen Forscher und Entwickler auf dem Weg zu heute eingesetzten Aktivierungen konfrontiert waren (Vgl. Tabelle 1). Dieser Abschnitt erläutert nun die Ursachen, die zur Veränderung der Aktivierungsfunktion geführt haben.<sup>lxxviii</sup>

Die Aktivierung des einschichtigen Perzeptrons nutzt den Schwellenwert der gewichteten Eingabe (Vgl. Abbildung 7). Da der Schwellenwert jedoch nicht differenzierbar ist und nur lineare Entscheidungsgrenzen abbilden kann, fand das Perzeptron in der Vergangenheit nur wenig Beachtung. Minsky kritisierte beispielsweise<sup>lxxix</sup>, das einzelne Perzeptron sei unfähig, die nichtlineare Exklusiv-Oder-Funktion (XOR-Gatter) abzubilden – das mehrschichtige Perzeptron (*MLP*) hingegen schon.<sup>lxxx</sup>

Nach den Problemen des Perzeptrons gelangte man zu den differenzierbaren Sigmoid-und-Tanh-Aktivierungen, die wiederum neue Schwierigkeiten mit sich brachten, da die Ableitung an den Rändern bei ihnen null beträgt. Sigmoid-und-Tanh-Aktivierungen weisen verschwindende Gradienten auf. Für sehr große und sehr kleine Aktivierungen (in den Randbereichen von Tanh und Sigmoid) lernt das neuronale Netzwerk sehr langsam, da die Gewichte überhaupt nicht oder nur in sehr kleinen Schritten aktualisiert werden.<sup>13</sup>

<sup>13</sup> (Vgl. <https://sefiks.com/2018/05/31/an-overview-to-gradient-vanishing-problem/>, Abgerufen am 11.6.19)

Zur Beschleunigung der Konvergenz sollte die Aktivierungsfunktion so gestaltet werden, dass sie keine Sattelpunkte oder Regionen enthält, an denen die Steigung Null beträgt. Somit kann die Ableitung der Aktivierungsfunktion niemals null werden.

Name	Formel	Abbildung	Vor- und Nachteile
Schwelle n-wert	(15) $\phi(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ 0, & \text{else} \end{cases}$		<ul style="list-style-type: none"> <li>✓ Variabler Schwellenwert</li> <li>✗ Nur für lineare Entscheidungsgrenzen</li> <li>✗ Nicht differenzierbar (Sprungstelle <math>x = 0</math>)</li> </ul>
Identität <sup>14</sup>	(16) $\phi(z) = z$		<ul style="list-style-type: none"> <li>✓ Differenzierbar</li> <li>✓ Zentriert <math>[-x, x]</math></li> <li>✗ Nicht-Nullzentriert <math>[0, 1]</math></li> <li>✗ Konvergenz</li> </ul>
Sigmoid <sup>15</sup>	(17) $\phi(z) = \frac{1}{1 + e^{-z}}$		<ul style="list-style-type: none"> <li>✓ Differenzierbar</li> <li>✗ Nicht-Nullzentriert <math>[0, 1]</math></li> <li>✗ Verschwindender Gradient</li> </ul>
Tanh <sup>16</sup>	(18) $\phi(z) = \frac{1 - e^{-2z}}{1 + e^{-2z}}$		<ul style="list-style-type: none"> <li>✓ Differenzierbar</li> <li>✓ Nullzentriert <math>[-1, 1]</math></li> <li>✓ Erhöhte Konvergenz</li> <li>✗ Verschwindender Gradient</li> </ul>
Rectified Linear Unit (ReLU)	(19) $\phi(z) = \max(0, z)$		<ul style="list-style-type: none"> <li>✓ 6x effizienter als Tanh</li> <li>✓ Differenzierbar (Sprungstelle <math>x = 0</math>)</li> <li>✓ Vermeidung des verschwindenden Gradientens</li> <li>✗ Wegfall von Einheiten (engl. dying ReLU)<sup>17</sup></li> </ul>

<sup>14</sup> (Vgl. <http://www.informatics4kids.de/index.php/softcomputing-kuenstliche-intelligenz-komplexe-systeme/neuronale-netze/was-sind-neuronale-netze/107-alles-oder-nichts-die-aktivierungsfunktion>, Abgerufen am 11.6.19)

<sup>15</sup> (Vgl. [https://gombru.github.io/2018/05/23/cross\\_entropy\\_loss/](https://gombru.github.io/2018/05/23/cross_entropy_loss/), Abgerufen am 11.6.19)

<sup>16</sup> (Vgl. [https://commons.wikimedia.org/wiki/File:Hyperbolic\\_Tangent.svg](https://commons.wikimedia.org/wiki/File:Hyperbolic_Tangent.svg), Abgerufen am 11.6.19)

<sup>17</sup> (Vgl. <https://sefiks.com/2018/05/31/an-overview-to-gradient-vanishing-problem/>, Abgerufen am 11.6.19)

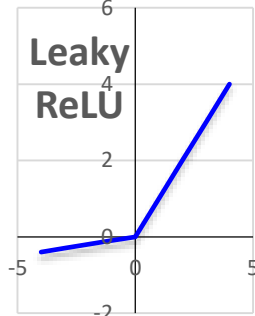
Leaky ReLU	$(20) \phi(z) = \begin{cases} \alpha * z, & \text{if } z < 0 \\ z, & \text{else} \end{cases}$		<ul style="list-style-type: none"> <li>✓ Löst das Problem <i>sterbender ReLUs</i></li> <li>✓ Differenzierbar</li> <li>✗ Weiterer Hyperparameter <math>\alpha</math> für negative Eingaben</li> </ul>
------------	---	--	--

Tabelle 1: Übersicht von Aktivierungsfunktionen (Eigene Darstellung)<sup>lxxx</sup>

### 2.6.3 Pooling

Pooling (auch als Subsampling bezeichnet) bietet drei Vorteile: Der erste ist die lokale Invarianz, sodass „[...] *kleine Veränderungen in der lokalen Nachbarschaft nicht zu einem anderen Ergebnis [...] führen.*“<sup>lxxxii</sup>. Zweitens reduzieren sie die Anzahl von Merkmalen, welche zu einer erhöhten Effizienz führt. Drittens beinhalten Pooling-Schichten keine Parameter, weil sie immer die gleiche Operation (ohne Gewichtung) durchführen. Daher werden Convolution-Pooling-Folgen als Convolution-Block (Parameter-Block oder komplexe Zelle) zusammengefasst (Vgl. Abbildung 24).

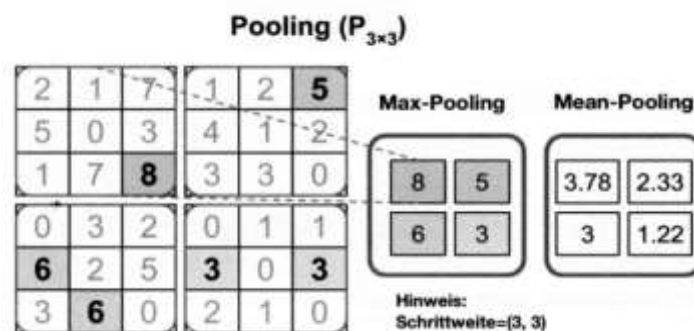


Abbildung 23: Pooling-Varianten (Quelle: Raschka/Mirjalili, 2016, S.500)

Für CNN-Implementierungen unterscheidet man zwischen *Max*- und *Mean*-Pooling (Vgl. Abbildung 23, *rechts*). Eine Pooling-Schicht wird mit dem Ausdruck  $P_{m_1 \times m_2}$  beschrieben, wobei der Parameter  $m_i$  das Format des sogenannten Pooling-Fensters (*engl. pooling window*) beschreibt. In der Regel wird Pooling so definiert, dass die Schrittweite dem Pooling-Fenster entspricht,  $s = (m_1, m_2)$ . Damit wird verhindert, dass es zu Überschneidungen kommt.

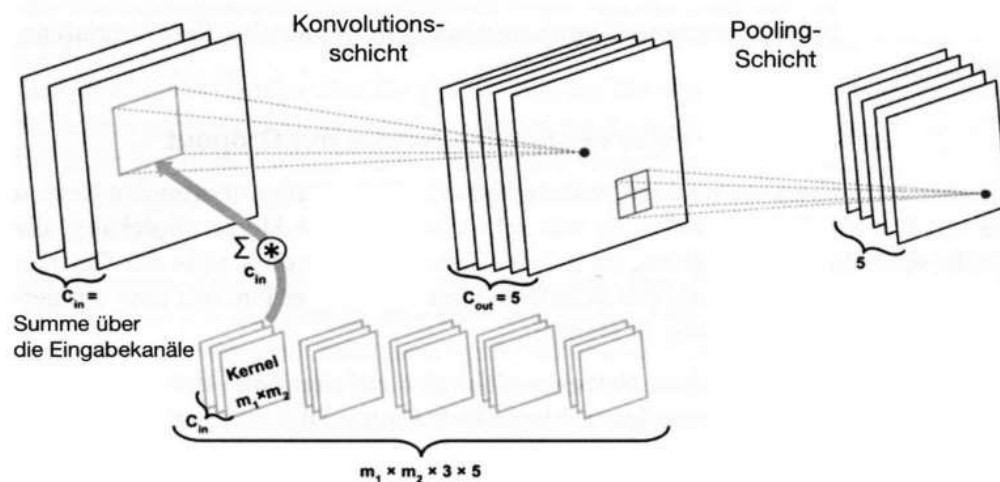


Abbildung 24: Ein Convolution-Block (CP-Folge) (Quelle: Raschka/Mirjalili, 2016, S.503)

## 2.6.4 Vollständig verknüpfte Schicht

Vollständig verknüpfte Schichten funktionieren analog zum vollständig-verknüpften MLP (Vgl. Abbildung 12, *Verdeckte Schicht*). FC-Schichten beinhalten die meisten Parameter, weil jeder einzelne Knoten der FC-Schicht  $l$  mit allen Knoten der vorhergehenden Schicht  $l - 1$  verknüpft ist.<sup>lxxxiii</sup> Wenn beide Schichten jeweils  $n$  Einheiten (Knoten) beinhalten, besitzt jeder einzelne  $n$ -viele gewichtete Eingänge. Die Anzahl der Parameter in der FC-Schicht  $l$  (oder  $l - 1$ ) ist die Anzahl der Eingabe-Einheiten zum Quadrat:  $n * n = n^2$  Gewichtungspare (Vgl. 2.4.3 Anzahl von Parametern). Bevor die Ausgabe einer Pooling-Schicht (die nach der Convolution-Block Konvention auf eine Aktivierungsschicht folgt) an eine FC-Schicht übergeben werden kann, findet in der Regel das abflachen (*engl. flatten*) der Ausgangsschicht mit der `Flatten()`<sup>18</sup>-Operation statt (Vgl. 3.2.3 Modellauswahl, Zeile 13f). Die Operation liefert eine Kette von Parametern: die eindimensionale Matrix  $(x_1 * x_2 * c)$ , wobei  $x_1, x_2$  der Eingabedimension und  $c$  der Anzahl von Kanälen (*Merkmalskarten*) entspricht<sup>lxxxiv</sup>.

Auf eine FC-Schicht folgt entweder eine optionale Regularisierungsschicht (Vgl. nächster Abschnitt), eine weitere FC-Schicht (Vgl. 4.1.2 AlexNet), oder die Ausgabe (*Klassifizierungsschicht*) (Vgl. 2.6.6 Ausgangsschicht).

<sup>18</sup> (Vgl. <https://keras.io/layers/core/>, Abgerufen am 11.6.19)

## 2.6.5 Regularisierungsschicht

Ziel der Regularisierungsschicht(en) ist es, ein Modell vorsätzlich einzuschränken, um eine Überanpassung zu vermeiden. In fünf verschiedenen Ansätzen kommen unterschiedliche Methoden für die Einschränkung der Modellkomplexität (Vgl. Abbildung 25) zum Einsatz, die im folgenden Abschnitt untersucht werden:<sup>lxxxv</sup>

1. L1-Regularisierung<sup>lxxxvi</sup>
2. L2-Regularisierung  
(L2-Norm, L2-Schrumpfung)<sup>lxxxvii</sup>
3. Dropout<sup>lxxxviii</sup>
4. Image Augmentation<sup>lxxxix</sup>
5. Batch Normalisierung<sup>xc</sup>

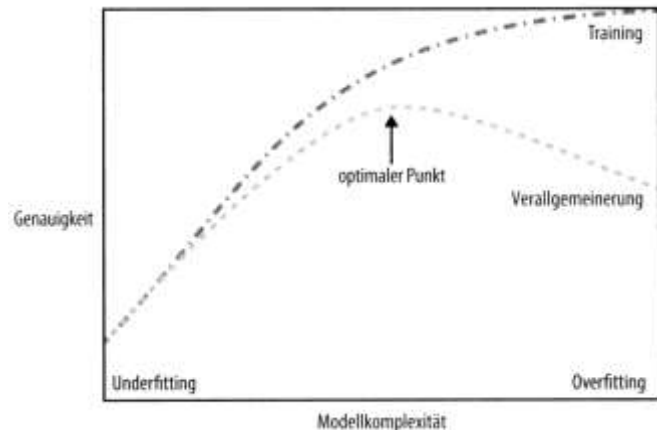


Abbildung 25: optimale Komplexität (Quelle: Müller/Guido, 2017, S.31)

L1 reduziert den Beitrag der irrelevanten Gewichtungsparameter auf exakt null. Folglich eliminiert sie den Beitrag unwichtiger Gewichtungsparameter vollständig. L1-Regularisierung kommt zum Einsatz, wenn nur wenige Merkmale zum Ergebnis beitragen. Sie ist daher leichter zu interpretieren.<sup>xcii</sup>

$$(21) \quad L1: \|w\|_1 = \sum_{j=1}^m |w_j|. \text{xcii}$$

L2 wird als Gewichtszerfall (*engl. weight decay*) oder L2-Schrumpfung bezeichnet. Sie ist weniger streng als L1 und zwingt die unwichtigen Gewichtungsparameter auf einen möglichst kleinen Merkmalsraum nahe null.<sup>xciii</sup>

$$(22) \quad L2 - \text{Norm: } \|w\|_2^2 = \sum_{j=1}^m w_j^2. \text{xciv}$$

Die zu bestrafenden Gewichtungsparameter werden anhand des Regularisierungsparameters  $\lambda$  (*lambda*), dem sogenannten Strafterm, welcher als Zusatzterm in der Straffunktion zum Einsatz kommt, schrittweise bestraft. Somit werden kleine Gewichtungen begünstigt und große Gewichtungen bestraft.<sup>xcv</sup>

Dropout ist eine weitere Technik, die ähnlich zur L2-Regularisierung funktioniert, da gewisse Gewichtungsparameter ebenfalls auf exakt null gesetzt werden. Im Gegensatz zu L2 trifft sie die Auswahl der wegzufallenden Parameter allerdings zufällig nach dem Anteil der Wegfallrate (*engl. keep probability bzw. dropout ratio*). Die Auswahl der betroffenen Einheiten erfolgt nach jeder Epoche erneut. Somit trainiert Dropout verschiedene Teilmengen eines Netzwerks. Die Muster selbst können redundant sein, um robuste Repräsentationen zu erhalten.<sup>xcvi</sup>



Image-Augmentation als Regularisierungsmethode beschränkt sich auf Resampling-Methoden zur künstlichen Erhöhung oder Verminderung des Anteils einer Klassenbezeichnung (*engl. oversampling bzw. undersampling*), indem vorhandene Datenpunkte mittels Image-Augmentation manipuliert werden. Größter Vorteil des Resampling ist, dass die manipulierten Datenpunkte weiterhin die gleiche Klassenbezeichnung wie das Original haben. Somit können viele Datenpunkte als Variation des Originals erstellt werden, um die Variation eines Datensatzes besser abzubilden.<sup>xcvii</sup>

### 2.6.6 Ausgabeschicht

Die Ausgabeschicht (*Klassifizierungsschicht*) besitzt genauso viele Knoten wie es Zielwerte gibt – entweder genau ein Knoten (Vgl. Abbildung 26, *links*) bei der binären Klassifikation oder  $i \in \mathbb{N}^*$  viele Knoten, wobei  $i$  der Anzahl von Zielklassen entspricht (Vgl. Abbildung 26, *rechts*). Ein Ausgabeknoten gibt die Trefferwahrscheinlichkeit<sup>xcviii</sup> an. Für die binäre Klassifikation ist das die Wahrscheinlichkeit der positiven Klasse oder die Trefferwahrscheinlichkeit der einzelnen Klassen im mehrklassigen Fall.<sup>xcix</sup>

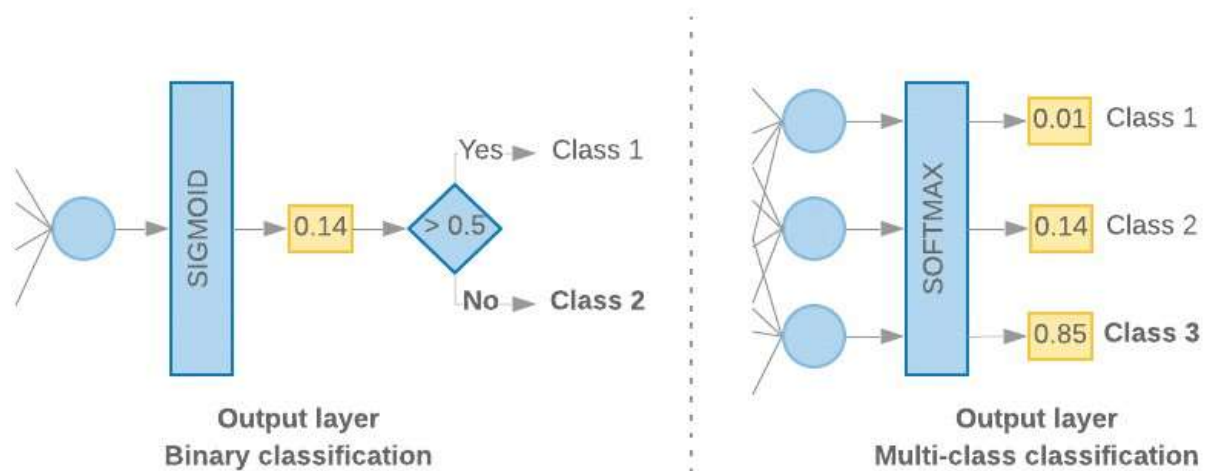


Abbildung 26: Zwei Ausgabeschichten; binäre Klassifikation (*links*); multi-class (*rechts*)<sup>19</sup>

Neben der Anzahl der Ausgabeknoten (Vgl. Abbildung 26, *Anzahl gelbe Knoten*) gibt es je nach Klassifizierungsproblem einen weiteren Unterschied in der finalen Aktivierungsfunktion (Vgl. Abbildung 26, *blauer Kasten*), sie definiert die Straffunktion (*loss*)<sup>20</sup>. Für binäre Klassifikation sind das der binäre Kreuzentropieverlust (*engl. binary cross-entropy loss*) und im mehrklassigen Fall der kategorische Kreuzentropieverlust (*engl. categorical cross-entropy loss*) und andere.<sup>c</sup>

<sup>19</sup> (Quelle: <https://developers.google.com/machine-learning/guides/text-classification/step-4>, Abgerufen am 11.6.19)

<sup>20</sup> (Vgl. <https://keras.io/losses/>, Abgerufen am 11.6.19)

## 2.7 Modellaufbau nach KDD-Vorgehensmodell

Das KDD-Vorgehensmodell (*engl. knowledge discovery in databases, KDD*) ist für die Gewinnung kompakter Repräsentation aus Daten (in Form von Wissen) zuständig, in dessen Kern der Data-Mining-Schritt stattfindet.<sup>ci</sup> Der Data-Mining-Schritt liefert dem Anwender Ansammlungen von Mustern beziehungsweise Modelle über Daten.<sup>cii</sup> Das Endprodukt des KDD-Vorgehensmodells stellt (abstraktes) Wissen als Repräsentation der Daten in Form von Mustern (*engl. pattern*) dar. Das KDD-Vorgehensmodell ist datengetrieben, das erzeugte Wissen sehr domänenspezifisch. Es wird von der Benutzerauswahl beeinflusst. Dabei werden Daten als Ansammlung von Wissen und Muster als Beschreibung von Teilmengen unterschieden.<sup>ciii</sup>

Neben der Repräsentation des Wissens kommt der Beurteilung von (Un-)Sicherheit im KDD-Vorgehensmodell besondere Bedeutung zu. Quantitative Methoden wie die Korrektklassifizierungsrate oder monetärer Nutzen dienen dem objektiven Verständnis. Daneben beschreibt es subjektive Bewertungsmethoden (Neuheit, Verständlichkeit).<sup>civ</sup>

Das KDD-Vorgehensmodell ist in neun Schritte unterteilt. Zu Beginn fließen die unverarbeiteten (Roh-)Daten ein, und zum Schluss als Wissen (extrahierte Muster) zu verlassen.

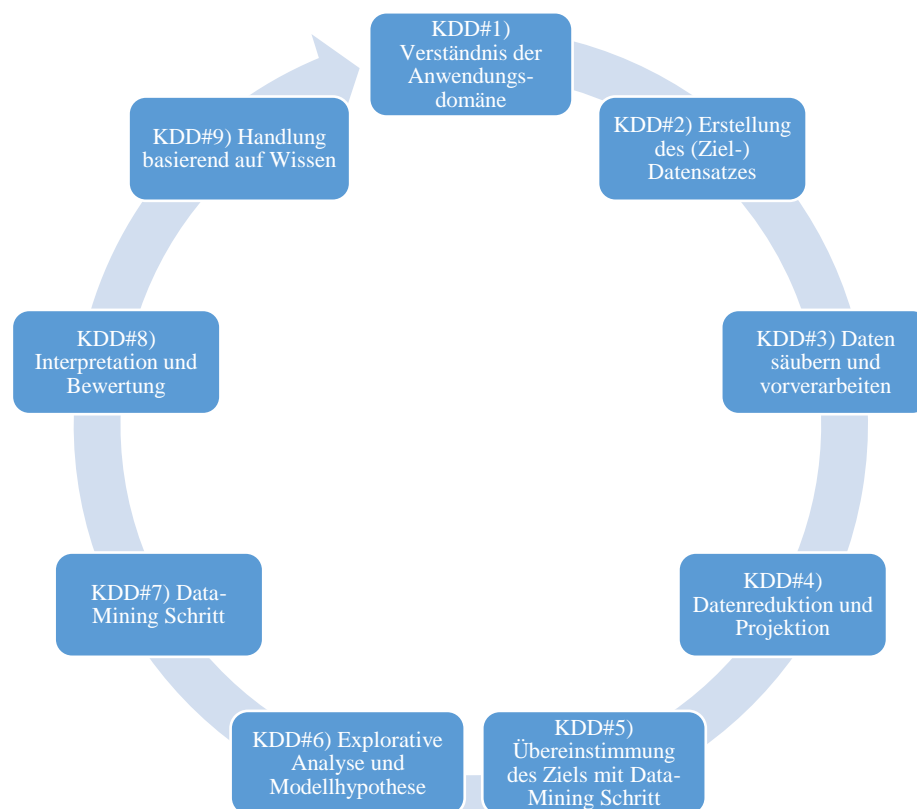


Abbildung 27: KDD-Vorgehensmodell (Eigene Darstellung)<sup>cv</sup>

## Kapitel 3 – Aufbau eines produktiven ML-Modells

Dieses Kapitel beschreibt alle praktischen Schritte, die zum Aufbau eines ML-Modells nötig sind. Im ersten Abschnitt (Vgl. 3.1 ML-Pipeline) werden die Schritte aufgelistet und im Detail beschrieben. Im darauffolgenden Abschnitt (Vgl. 3.2 CNN-Implementierung) wird der Quellcode analog zu den Schritten der ML-Pipeline präsentiert. Abschließend steht die System-Konfiguration, die für das Training benutzt wurde, im Mittelpunkt der Betrachtung.

Unabhängig von der zu lösenden Aufgabe und des Datensatzes sind die wichtigsten Schritte zum Aufbau eines (robusten) ML-Modells den neun Schritten des KDD-Vorgehensmodells zuzuordnen. Der Aufbau und das Training von Modellen ist ein iterativer Prozess, der mit Entscheidungen zur Modellarchitektur, Datentransformation und der Überprüfung des Gelernten einhergeht. Die Schritte umfassen das Verständnis der Eingabedaten sowie deren Transformation und Aufteilung für die Bewertung, wobei vor dem Training (und der Bewertung des Modells) unsicher ist, was die optimale Methode zur Transformation und Aufteilung ist (Vgl. Schritt 3.1.1 bis 3.1.6).

Metriken, die jeden Trainingsabschnitt (*Epoche*) während des Trainings in Form einer Lernkurve dokumentieren, sind ein gutes Hilfsmittel, um zu erkennen, wenn ein Modell nicht gut funktioniert und das Training frühzeitig abgebrochen werden muss. Mit dem Abschluss des Trainings und der damit einhergehenden Erkenntnis der Bewertung (Vgl. 3.1.7 Bewertung des Modells) werden die Hyperparameter des Modells bei Bedarf angepasst und die Pipeline zum Teil erneut durchlaufen (zurück zu Schritt 3.1.3).

### 3.1 ML-Pipeline

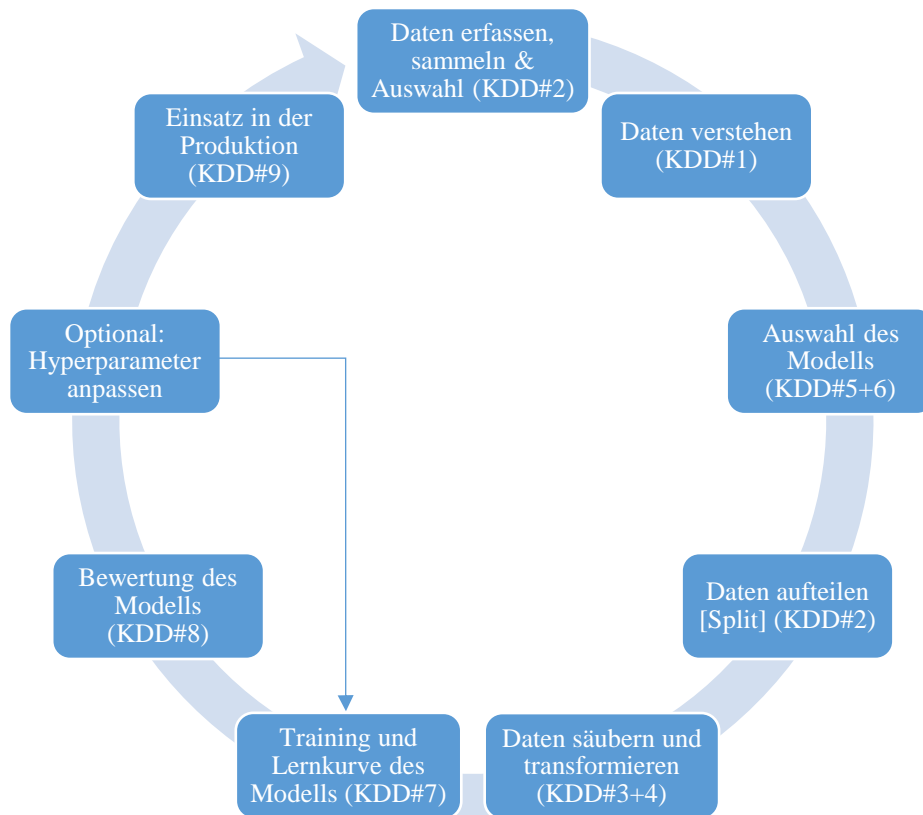


Abbildung 28: ML-Pipeline nach dem KDD-Vorgehensmodell (Eigene Darstellung)<sup>21</sup>

#### 3.1.1 Daten erfassen, sammeln und auswählen

Das Sammeln von Daten im Sinne von Erhebung und Auswahl befasst sich mit allen Tätigkeiten rund um die Erstellung eines Datensatzes als Wissensgrundlage des Modells. Es gibt viele öffentlich zugängliche Datensammlungen und Ausschreibungen auf Datensätze.<sup>22</sup> Außerdem stehen APIs für die Datenabfrage von Webseiten zur Verfügung, zum Beispiel die Twitter-Bibliothek<sup>23</sup>.

Damit ein Modell gute Vorhersagen treffen kann, muss es in der Lage sein, die Daten zu generalisieren. Dafür ist es sehr wichtig, eine ausreichende Menge an Datenpunkten zu haben. Die Erhebungsdaten sollten einer Verteilung folgen, die der späteren Zielverteilung entspricht. So lassen sich systematische Unterschiede zwischen den Datenpunkten verhindern. Der Datensatz beziehungsweise die Verteilung seiner Datenpunkte sollte den Raum aller möglichen Eingaben abdecken. Hierfür gibt es zwei unterschiedliche Ansätze: die *natürliche Verteilung* und die *ausgewogene Verteilung*. Eine natürliche

<sup>21</sup> Nach dem Vorbild des Workflows von Google (Vgl. <https://developers.google.com/machine-learning/guides/text-classification/>)

<sup>22</sup>(Vgl. <https://www.kaggle.com/competitions>, Abgerufen am 11.6.19)

<sup>23</sup> (Vgl. <https://developer.twitter.com/#analyzehome>, Abgerufen am 11.6.19)

Verteilung repräsentiert Merkmalsausprägungen in ihrer natürlichen Häufigkeit. Das bedeutet, dass seltene Merkmale weniger häufig vertreten sind als übliche – mit der steigenden Gefahr, dass gewisse Merkmalsausprägungen überhaupt nicht vertreten sind und folglich die Daten nur einer Teilmenge der Verteilung folgen. Die ausgewogene Verteilung beinhaltet dagegen je Klassenbezeichnung gleich viele Datenpunkte. Ausgewogene Verteilungen können mit Hilfe von Resampling-Methoden erstellt werden. Für die Erkennung seltener Ereignisse gibt es in der Regel weniger Datenpunkte. Solche Phänomene lassen sich unter dem Begriff nicht-balancierte Datensätze wie im Falle der Betrugserkennung zusammenfassen (Vgl. 4.2 Umgang mit unausgewogenen Datensätzen).

Bei der Auswahl der passenden Klassenverteilung ist ein Kompromiss zwischen Varianz (Sensitivität) und Bias (Spezifität) zu finden.<sup>24</sup> Die Varianz beschreibt die Veränderung der Vorhersagen auf unterschiedliche Trainingsdatensätze (Vgl. 3.1.7 Bewertung des Modells, *Kreuzvalidierung*), Trainings- und Validierungs-KKR weisen bei erhöhter Varianz einen Abstand auf (Vgl. Abbildung 29, *oben rechts*). Der Bias bestimmt, wie stark der Klassifikator von der tatsächlichen Vorhersage abweicht, und misst den systematischen Fehler des Modells (Vgl. Abbildung 29, *oben links*).<sup>cvi</sup>

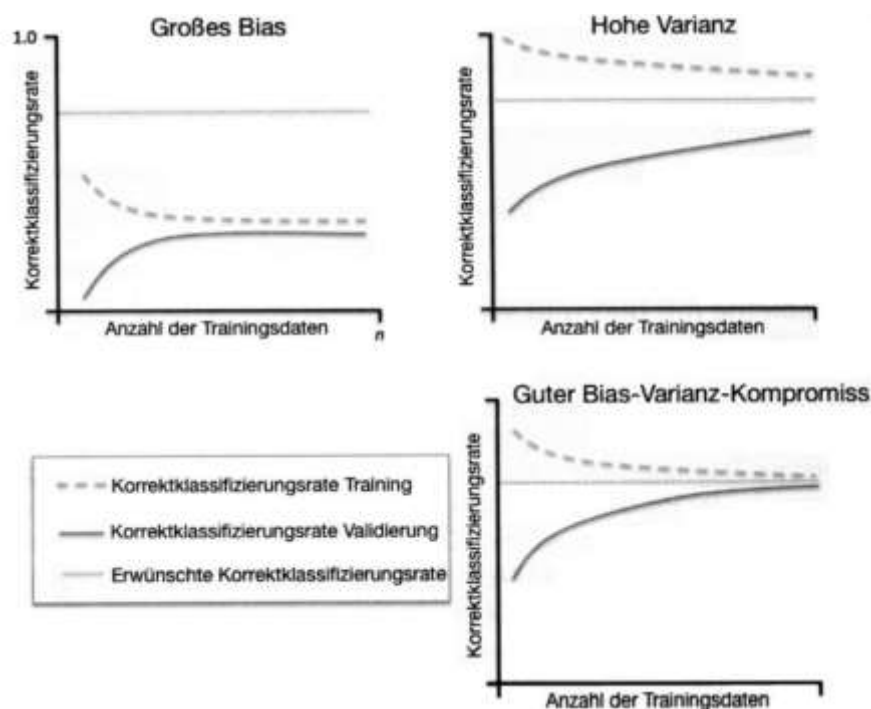


Abbildung 29. Bias-Varianz-Kompromiss (Vgl. Raschka/Mirjalili, 2016, S.212)

### 3.1.2 Daten verstehen

Ein gutes Verständnis des Datensatzes kann die Genauigkeit des Modells verbessern; zum Beispiel, wenn eine ungleiche Klassenverteilung vor dem Training Rücksicht findet. Das Verständnis der Daten

<sup>24</sup> (Vgl. <https://elitedatascience.com/bias-variance-tradeoff>, Abgerufen am 11.6.19)

erfordert Exploration und Visualisierung der (Roh-)Daten (und der Metadaten). Damit kann sichergestellt werden, dass diese Daten den Erwartungen entsprechen. Daten zu sichten und die Erkenntnisse grafisch aufzubereiten sind die Hauptaufgaben im zweiten Schritt des ML-Workflows.

Daten können generell strukturiert oder unstrukturiert vorliegen. Bei strukturierten Daten in Form einer Datenbank besteht die Exploration im Betrachten der Datenbankspalten. Dadurch entsteht ein Gefühl für die Merkmale (Attribute) und die Betrachtung von Ausprägungen in den Datenbankreihen, insbesondere die Erkenntnis von Wertebereichen und Ausreißern. Anschließend werden ausgewählte Merkmale (Spalten) in Streudiagrammen oder Korrelationsmatrizen visualisiert, um Minima, Maxima und Ausreißer grafisch darzustellen.<sup>cvii</sup>

Unstrukturierte Daten sind Bilder, Text und Sprache. Sie werden ebenfalls gesichtet und strukturiert, um Besonderheiten zu identifizieren. Bilder sind Sequenzen aus Zahlen. Sie bestehen aus Bildpunkten und deren Koordinate, die in Matrizen aus Pixeln abgebildet werden. Farbige Bilder sind Listen aus Zahlen (mehrdimensionale Matrizen), eine Dimension für jeden Farbkanal aus der Menge Rot, Grün und Blau. Ein Kanal liefert verschiedene Sichten auf die Daten, so wie der linke und rechte Audiokanal eines Videos oder Audiodatei. Höhe und Breite des Bildes können als ordnende Achse verstanden werden, ähnlich der Zeitachse in einem Video. Mit Hilfe der diskreten Faltung (Vgl. 2.6.1 Diskrete Faltung) bleibt diese Information erhalten.<sup>cviii</sup>

Generell sind Bilddaten einer hohen Variation ausgesetzt. Zum einen kann ein Bild eines Objekts aus verschiedenen Blickwinkeln aufgenommen sein, unterschiedliche Belichtung und Schattierungen aufweisen – je nach Tageszeit und Beleuchtung – es kann verdeckt, deformiert oder nur teilweise sichtbar sein und sich im Hintergrund stark unterscheiden.<sup>ci<sup>x</sup></sup> Zum anderen ist es möglich, innerhalb einer Objektklasse weitere Unterklassen zu finden. Die Interklassenvariabilität (*engl. inter-class variability*) beschreibt die Varianz zwischen Objektklassen. Analog beschreibt die klasseninterne Variabilität (*engl. intra-class variability*) die Varianz innerhalb einer Objektklasse.<sup>25</sup>

Neben der Exploration gibt es Metriken zur Strukturierung eines Datensatzes.<sup>26</sup>

1. Anzahl der Datenpunkte (*engl. samples*).
2. Anzahl der Klassen (die Anzahl der Kategorien oder Themen (Zielwerte) im Datensatz).

---

<sup>25</sup> (Vgl. <https://elitedatascience.com/bias-variance-tradeoff>, Abgerufen am 11.6.19)

<sup>26</sup> (Vgl. <https://developers.google.com/machine-learning/guides/text-classification/step-2>, Abgerufen am 11.6.19)

3. Anzahl der Datenpunkte pro Klasse.
4. Anzahl der Merkmale je Datenpunkt.
5. Häufigkeitsverteilung der Datenpunkte.
6. Verteilung der Probenlänge.

Die Metriken 1 bis 4 sind auf den sortierten NIH-Datensatz anwendbar (Vgl. Tabelle 8).

Die gewonnenen Erkenntnisse aus Exploration und Visualisierung des Datensatzes sowie die Metriken zur Verteilung kommen im folgenden Abschnitt Modellauswahl (Vgl. 3.1.3 Auswahl des Modells) und Datentransformation im Rahmen der Vorverarbeitung (*engl. preprocessing*) zum Einsatz (Vgl. 3.1.5 Daten transformieren).

### 3.1.3 Auswahl des Modells

Mit dem Bewusstsein darüber, welche Eigenschaften der Datensatz beinhaltet und welches Problem zu lösen ist, beginnt die Suche nach einer ähnlichen Modellvorlage. Für viele Problemstellungen und Datensätze gibt es öffentlich abrufbare Modellarchitekturen und vortrainierte Modelle.<sup>27</sup> Besonders beliebte Datensätze für Einsteiger sind *Iris*, *Boston-Housing*, *MNIST* und *CIFAR-10*, da sie wegen ihrer relativ niedrigen Komplexität strukturell überschaubar sind.<sup>28</sup> Folgende Punkte sind zu beachten und im Quellcode größtenteils in der Aufbauphase zu finden (Vgl. 3.2.3 Modellauswahl):

1. Architektur (Modellkomplexität) und Initialisierung der Gewichte
  - a. Ganz von vorn (*engl. from scratch*)
  - b. Transferlernen mit Feinanpassung (*engl. transfer-learning, fine-tuning*)
2. Regularisierungsmethode zur Beschleunigung der Konvergenz und zur Senkung der Modellkomplexität (Vgl. 2.6.5 Regularisierungsschicht) und darüber hinaus:
  - a. Batch-Normalisierung: Standardisierung von Merkmalskarten
  - b. Optional: Image Augmentation.
    - i. wenn der Datensatz zu klein ist (*engl. resampling*)
    - ii. zur Verminderung der Varianz (Regularisierung)
3. Aufbau und Anzahl von Trainingsschritten innerhalb von `model.compile`: Zielfunktion (`loss`) und Metriken der Lernkurve (`metrics`), Epoche, Batch, Lernrate und Lernalgorithmus (`optimizer`) zur Optimierung der Gewichtungen (Vgl. 2.5 Lernmethoden)

---

<sup>27</sup> (Vgl. <https://keras.io/applications/>, Abgerufen am 11.6.19)

<sup>28</sup> (Vgl. <https://keras.io/datasets/>, Abgerufen am 11.6.19)

Neben dem architekturellen Neuaufbau eines Modells können alternativ komplett fertig trainierte Modelle (*engl. pre-trained*) oder nur deren Architektur mit zufälligen Gewichtungen initialisiert werden. Vorgefertigte Modelle lassen sich sehr einfach und schnell innerhalb weniger Programmzeilen auf das eigene Problem anwenden – solange der Datensatz des ursprünglichen Trainings hinreichend mit dem Zieldatensatz verwandt ist. Dieser Ansatz heißt Transferlernen (*engl. transfer-learning*). Transferlernen ermöglicht eine Zeitersparnis gegenüber der Neuentwicklung und ermöglicht die Übernahme von vorgelernten Strukturen und Filtern des ursprünglichen Datensatzes, ohne sich mit der Modellarchitektur im Detail beschäftigen zu müssen.

Für neuronale Netzwerke empfiehlt es sich, zunächst ein Modell auszuwählen, das zur Überanpassung neigt. Anschließend gilt es, die Parameter durch Regularisierungstechniken schrittweise so lange zu reduzieren, bis die Überanpassung eliminiert ist (Vgl. 2.4.1 Vom MLP zum (vanilla) Feed-Forward Netzwerk).<sup>CX</sup>

### 3.1.4 Daten aufteilen

Damit die Präzision für die Generalisierungsfähigkeit der Eingabedaten beurteilt werden kann – genauer gesagt die gelernte Erkenntnis aus den Trainingsdaten – ist es wichtig, den Datensatz in der Explorationsphase so aufzuteilen (*engl. split*), dass nicht alle Datenpunkte für das Training verwendet werden. Stattdessen bildet man mindestens eine unabhängige (disjunkte) Teilmenge aus dem gesamten Datensatz, um die Generalisierungsfähigkeit des Modells auf ungesehene Daten zu beurteilen – den sogenannten Validierungs-Datensatz. Erst, wenn die Explorationsphase abgeschlossen ist, empfiehlt es sich, den gesamten Datensatz (ohne Aufteilung) für das Training zu verwenden, um eine tiefere Erkenntnis zu erreichen. Während der Aufteilung sollten hinreichend viele Daten zur Überprüfung von Hypothesen vorhanden sein.

Der Validierungs-Datensatz wird in der Literatur oft als Test-Datensatz bezeichnet, obwohl diese Bezeichnung eigentlich erst bei der mehrfachen Aufteilung (Kreuzvalidierung) des Datensatzes verwendet wird (Vgl. Abbildung 30). Der Validierungs-Datensatz beurteilt die wichtigste Aufgabe des Modells – die Genauigkeit der Vorhersagen als Ausdruck der Generalisierungsfähigkeit (Korrektklassifizierungsrate) auf neue und ungesehene Daten. Er nimmt jedoch keinen direkten Einfluss auf die Gewichtungen, sondern wirkt sich lediglich indirekt, über das Endergebnis, auf die beteiligten Entwickler aus. Bei einem schlechten Ergebnis werden am Ende der Trainingsphase in der Regel die Modellparameter angepasst und erneut trainiert – bis ein brauchbares Modell entsteht. Die



Validierungsdaten werden daher zur Optimierung der Modellparameter benutzt. (Vgl. Abbildung 30, *mitte*)

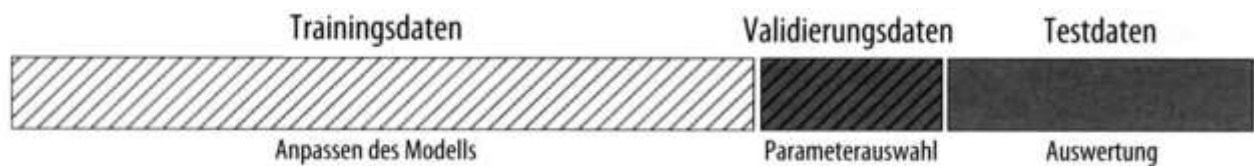


Abbildung 30: Aufteilung des Datensatzes in drei Teilmengen (Quelle: Müller/Guido, 2017, S.246)

Der Test-Datensatz ist eine weitere disjunkte Teilmenge der Gesamtdaten und wird nur für die Modellauswahl (Vgl. Abbildung 30, *rechts*) benutzt. Nach dem Training mehrerer Modelle (mit dem Trainingsdatensatz) und der Überprüfung auf den Validierungsdatensatz wird der Test-Datensatz dazu verwendet, die Genauigkeit der unterschiedlichen Modelle (Hyperparameter-Ausprägung) auf den gleichen Test-Datensatz zu beurteilen. In der Regel wird der Test-Datensatz bis zur Modellauswahlphase zurückgehalten (*engl. Holdout-Method*), damit diese Datenpunkte nicht verbraucht werden (Vgl. Abbildung 31, *Endgültige Leistungseinschätzung*).<sup>cxi</sup>

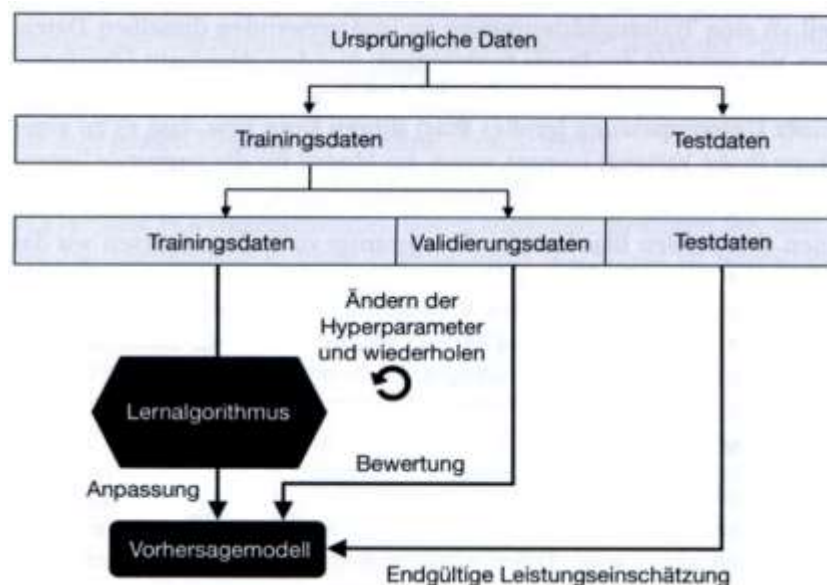


Abbildung 31: Aufteilung (Split) und zurückhalten der Testdaten (Vgl. Raschka/Mirjalili, 2016, S.206)

Da das Training außerdem von der Struktur der Eingabedaten abhängt, ist sicherzustellen, dass der Datensatz vor der Bildung jedes Splits und nach jeder Epoche/Batch gemischt wird, um eine Sortierung der Daten zu verhindern. Eine Vorsortierung oder ungleichmäßige Sortierung kann den Lernprozess beeinflussen: Wird zum Beispiel nach Dateiname oder Klasse sortiert, dann besteht bei der Aufteilung die Gefahr, dass ein nicht-repräsentativer Datensatz entsteht. Der *seed*-Parameter<sup>29</sup> ist eine praktische

<sup>29</sup> (Vgl. <https://keras.io/preprocessing/image/>)

Hilfe zur Bildung von abbildbaren Aufteilungen; durch ihn lassen sich Mischungen nachbilden. Zur Aufteilung und Bewertung gibt es drei grundsätzliche Ansätze:

1. Vorgegebene Aufteilungen (*engl. official split*) sind speziell auf Datensätze zugeschnitten und gewährleisten dadurch die Vergleichbarkeit. Für den unsortierten Datensatz dieser Arbeit kann die vorgegebene Aufteilung aus den Dateien `train_val_list.txt` sowie `test_list.txt` benutzt werden.<sup>30</sup>
2. Training, Validierung (*Development*) und (optionaler) Test Split: Aufteilung des Datensatzes nach einem festen Anteil der Gesamtdaten, so wie in Abbildung 32 dargestellt. Bei kleinen Datensätzen wird das Öfteren auf den Testdatensatz verzichtet, um mehr Daten für Training und Validierung nutzen zu können. Ist der Datensatz wiederum sehr groß, kann die Aufteilung zum Vorteil des Trainingsdatensatzes erhöht werden.



Abbildung 32: Daten aufteilen (Eigene Darstellung)

3. Kreuzvalidierung (*engl. cross validation*)<sup>cxii</sup> ist eine Aufteilung in  $k \in \mathbb{N}$  Teilmengen und des Trainierens mit  $k - 1$  Teilmengen mit der  $k$ -ten (letzten) Teilmenge als Validierungsmenge. Im nächsten Durchlauf rotiert die Teilmenge des Validierungsdatensatzes über den Datensatz (Vgl. Abbildung 33, *schwarze Fläche*) und vermindert dadurch die Gefahr eines zu groben Train-/Validation-Splits.<sup>31</sup> Während der Kreuzvalidierung werden  $k$ -viele Modelle trainiert und am Ende der Durchschnitt aller Ergebnisse bestimmt. Der Durchschnitt der Kreuzvalidierung hilft dabei, die Empfindlichkeit des Modells auf unterschiedliche Aufteilungen zu beurteilen, und gibt Hinweise, ob mehr Daten gebraucht werden oder nicht.<sup>cxiii</sup> Durch Rotation erhält jede Teilmenge die Chance, im Trainings- oder Validierungsdatensatz zu landen. Darüber hinaus gibt es erweiterte Formen der Kreuzvalidierung:
  - a. Stratifizierte Kreuzvalidierung<sup>cxiv</sup> (*engl. stratified cross validation*) unter Erhalt der Klassenanteile,
  - b.  $k$ -fache Kreuzvalidierung (*engl. k-fold cross validation*)<sup>cxv</sup>, sowie

<sup>30</sup> (Vgl. <https://nihcc.app.box.com/v/ChestXray-NIHCC>, Abgerufen am 11.6.19)

<sup>31</sup> (Vgl. <https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6>, Abgerufen am 11.6.19)

- c. Leave-One-Out-Kreuzvalidierung<sup>cxvi</sup> als Extremform: ein Datenpunkt in jedem Validierungsdatensatz.

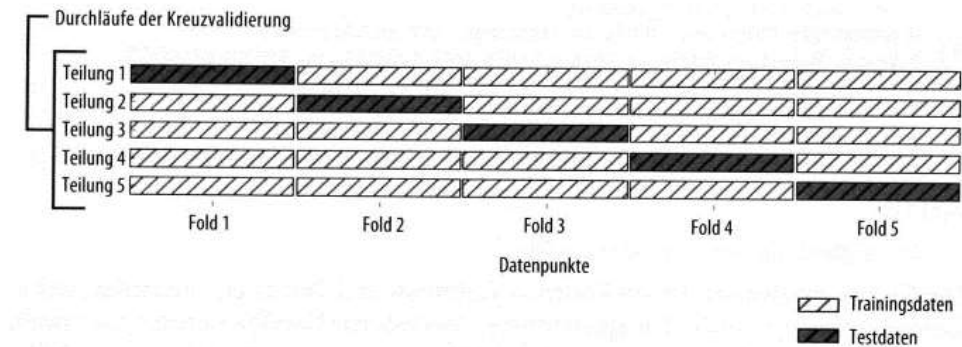


Abbildung 33: Kreuzvalidierung (Quelle: Müller/Guido, 2017, S.236)

Da einzig und allein der Trainingsdatensatz Einfluss auf den Lernprozess (die Gewichtungen) nimmt, ist eine faire und repräsentative Aufteilung des Datensatzes oberstes Gebot. Durch den Einsatz der  $k$ -fachen Kreuzvalidierung können die Daten effizienter genutzt werden, da statt eines klassischen 80/20-Split bei der zehnfachen Kreuzvalidierung zehn unterschiedliche 80/20-Splits erstellt werden und zehn separate Modelle trainiert werden.<sup>cxvii</sup>

Eine glückliche Aufteilung im schlechtesten Fall, die alle schwierigen Datenpunkte im Training betrachtet und alle einfachen während der Validierung, kann zu falschen Schlüssen führen, obwohl man lediglich Glück bei der Aufteilung hatte. Daher ist die  $k$ -fache Kreuzvalidierung eine robuste Methode zur präziseren Bewertung verschiedener Aufteilungen. Der Nachteil der Kreuzvalidierung ist der zusätzliche Rechenaufwand, da  $n$ -viele Modelle zu trainieren sind.<sup>cxviii</sup>

### 3.1.5 Daten transformieren

Eine robuste Transformation der Daten ist wichtig, um die Attribute (Merkmale) so zu codieren, dass sie unabhängig von ihrer Einheit und ihrem Wertebereich sind und die Eingabedaten gerecht repräsentieren. Ohne das vorausgesetzte Verständnis der Daten erhält man selten ein Modell, das in der Lage ist, die Trainingsdaten zu generalisieren.

Viele Klassifikation-Algorithmen aus der Familie der linearen Modelle sind auf eine Standardisierung der Eingabedaten angewiesen – sonst konvergieren sie nicht, wie zum Beispiel SVM-Algorithmen und neuronale Netzwerke.<sup>cxix</sup> Zur Merkmalstandardisierung mit Mittelwert  $\mu = 0$  und Standardabweichung  $\sigma = 1$  wird die Formel zur Standardnormalverteilung benutzt:

$$(23) \quad x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

$j$  beschreibt den Index des zu standardisierenden Merkmals  $x$ . Von der Merkmalsausprägung  $x_j$  wird ihr Mittelwert  $\mu_j$  subtrahiert und im Anschluss durch die Standardabweichung  $\sigma_j$  des Merkmals dividiert. Für andere lineare Modelle wie kNN, logistische Regression oder Entscheidungsbäume ist eine Merkmalstandardisierung nicht notwendig. Die wichtigsten Operationen der Datentransformation lauten:

1. Dimensionsreduktion (*engl. feature selection*): aussagekräftige Merkmale auswählen und unwichtige entfernen
2. Normalisierung begrenzt den Wertebereich auf ein festes Intervall, zum Beispiel  $[0, 1]$  oder  $[-1, +1]$ .
3. Umwandlung kategorischer Daten (Vgl. 2.2.1 Klassifikation und Regression)
  - a. Vektorisieren: One-Hot-Kodierung<sup>cxx</sup> (*engl. one hot encoding*) für eine Zielklasse (*multi-class*) oder Multi-Hot-Kodierung (*engl. multi hot encoding*) bei mehreren Zielklassen (*multi-label*),
  - b. Repräsentation durch Zahlen
4. Dekomposition (Token bilden): Aufteilung in Teilbilder (*engl. patches*).
5. Merkmalsgenerierung: Attribute erzeugen (*engl. feature engineering*), Bildung von Relationen aus Merkmalen unter einem neuen Merkmalsraum.<sup>cxxi</sup>
  - a. Merkmalsextraktion (*engl. feature cross, feature extraction*)
6. Bild-Manipulationen (*engl. image augmentation*): Anwendung von Bild-Operationen, die das Bild durch Rotation, Spiegelung oder die Extraktion zufälliger Ausschnitte verändern.

Im Gegensatz zur Merkmalsextraktion benutzt der Deep-Learning-Ansatz keine ausgewählten Merkmale als Eingabe. Stattdessen verarbeitet er einfach die gesamten Eingabedaten und sucht die optimale Gewichtung der Eingabe-Merkmale mit dem Gradientenabstiegsverfahren (Vgl. 2.5 Lernmethoden). Eine Auswahl der oben aufgezählten Möglichkeiten transformiert die Eingabedaten mittels einer Kodierung der Zielwerte und Image Augmentation.

Neben der üblichen Normalisierung stehen für Bilddaten weitere Methoden zur Merkmalsgenerierung und Manipulation von Bildern zur Verfügung. Sie beginnen bei der Extraktion zufälliger Ausschnitte, Drehungen und Spiegelungen und reichen bis hin zu Resampling-Methoden (Vgl. 3.2.5 Datentransformation, Zeile 11f). Die Keras-Bibliothek ImageDataGenerator (IDG)

vereinfacht die Anwendung der Transformationen (und Aufteilung des Datensatzes) auf wenige Programmzeilen und ist daher ein mächtiges Werkzeug zur Manipulation von Bildern.<sup>32</sup>

Die Klassenbezeichnungen liegen zu Beginn als Zeichenkette vor. Der IDG erzeugt kodierte Klassenbezeichnungen, um den Speicherbedarf zu vermindern. Außerdem werden die Pixel-Merkmale im Intervall [0,1] mittels des `rescale` Attributs normalisiert. Des Weiteren wird jedes Bild sowohl mit dem Durchschnitt (`samplewise_center`) als auch mit der Standardabweichung der Klasse (`samplewise_std_normalization`) normalisiert, um die Varianz der Eingabe zu vermindern (Vgl. 3.2.5 Datentransformation, Zeile 9 ff.).

### 3.1.6 Training und Lernkurve des Modells

Nach Abschluss des Modellaufbaus und einer robusten Datentransformation geht es mit der Trainingsphase weiter. Vor Beginn des Trainings muss ein kompiliertes Modell mit dem Aufruf der `compile`-Methode erstellt werden.

Um die Lernkurve während des Trainings überhaupt beobachten zu können, müssen die gewünschten Metriken zum Kompilierzeitpunkt genannt sein. Dazu wird das `metrics`-Attribut innerhalb des Aufrufs von `compile` benutzt. Neben den Metriken werden hier außerdem der Lernalgorithmus (`optimizer`), Lernrate und Zielfunktion (`loss`) festgelegt. Der anschließende Aufruf der `fit`-Methode nutzt das kompilierte Modell und seine Daten, um das Training zu starten. In dieser Arbeit wird `model.fit_generator` statt `model.fit` benutzt, da die Daten mit dem IDG organisiert werden (Vgl. 3.2.5 Datentransformation sowie 3.2.6 Trainingsphase #1 [Pre-Training], Zeile 1). Es ist zu beachten, dass nicht alle Metriken mit jedem Lernalgorithmus funktionieren, zum Beispiel die erweiterten Metriken von `keras_metrics` haben für die tiefe Feinanpassung nicht funktioniert und wurden aus diesem Anlass nur für das Vortraining benutzt (Vgl. 3.2.8 Tiefere Feinanpassung *versus* 3.2.3 Modellauswahl).<sup>cxxii</sup>

Die `fit`-Methode erwartet Parameter für die Anzahl der Epochen, die Anzahl der Schritte pro Epoche, Trainings- und Validierungsmenge sowie die optionale Gewichtung der Klassenbezeichnungen nach dem `class_weight`-Parameter. Während des Trainings kann der verantwortliche Entwickler des Modells keine Änderungen an der Modellarchitektur oder den Hyperparametern vornehmen. Es sei denn, diese sind im Vorhinein wie die schrittweise Absenkung der Lernrate definiert.<sup>33</sup>

---

<sup>32</sup> (Vgl. <https://developers.google.com/machine-learning/guides/text-classification/step-3>, Abgerufen am 11.6.19)

<sup>33</sup> (Vgl. <https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>, Abgerufen am 11.6.19)

Die Beobachtung der Lernkurve ist ein wichtiger Bestandteil des Trainings. Anhand ausgewählter Metriken kann eine Über- oder Unteranpassung frühzeitig erkannt werden, um entsprechende Gegenmaßnahmen zu treffen. Denkbar ist beispielsweise, das Training frühzeitig zu beenden (*engl. early stop*), und stattdessen eine Anpassung der Modellarchitektur oder deren Hyperparameter vorzunehmen (Vgl. 3.2.10.1 Lernkurve betrachten; 3.2.3 Modellauswahl).<sup>cxxiii</sup>

Die Gittersuche (*engl. grid search*) ist eine Besonderheit des Trainings. Mit ihr werden verschiedene Modellausprägungen unter Variation der Hyperparameter trainiert, um das beste Modell auszuwählen. Sie wird außerdem als Rastersuche bezeichnet (Vgl. 3.1.3 Auswahl des Modells).<sup>cxxiv</sup>

### 3.1.7 Bewertung des Modells

*„Es ist wichtig zu betonen, dass die Komplexität des Modells eng mit der Variabilität der Eingabedaten im Trainingsdatensatz zusammenhängt: je mehr Varianten die Datenpunkte im Datensatz enthalten, umso komplexer darf das Modell sein [...]. Allerdings hilft es nicht, die gleichen Daten einfach zu duplizieren oder sehr ähnliche Daten zu sammeln.“ (Vgl. Müller/Guido, 2017, S.31).*

Methoden zur Beurteilung der Generalisierungsfähigkeit<sup>cxxv</sup>

1. Betrachtung der Lernkurve<sup>cxxvi</sup>
2. Korrektklassifizierungsrate (*engl. accuracy*)<sup>cxxvii</sup> und Zielfunktion (*engl. loss*)
3. Wahrheitsmatrix (*engl. confusion matrix*)<sup>cxxviii</sup>
4. Trefferquote (*engl. recall*), Genauigkeit (*engl. precision*) und F1-Maß<sup>cxxix</sup>
5. Grenzwertoptimierungskurven (*engl. Receiver-Operating-Characteristic, ROC*)<sup>cxxx</sup>
6. Fehler-Arten (*TP, TN, FP, FN*) und deren Gewichtung
7. Visualisierung (Vgl. 3.2.10.3 Gewichtungen visualisieren)

Bei Betrachtung der Lernkurve können die Korrektklassifizierungsraten des Trainings- und Validierungsdatensatzes (*engl. training-accuracy bzw. validation-accuracy*) einen Unterschied aufweisen. Es wird zwischen Über- oder Unteranpassung (*engl. overfit bzw. underfit*) anhand verschiedener Metriken unterschieden.<sup>cxxxi</sup> Der Ausdruck *overfit* beziehungsweise *underfit* adressiert eine zu hohe oder niedrige Generalisierungsfähigkeit eines Modells, die durch den Vergleich von Trainings- und Validierungsergebnissen deutlich wird. Zur Anpassung werden Modellkapazität, Anzahl der Knoten und Schichten sowie das Ausmaß der eingesetzten Regularisierung verändert.<sup>cxxxii</sup>

Eine Überanpassung wird in der Literatur mit hoher Varianz verglichen. Hohe Varianz bedeutet, dass das Modell mit den Trainingsdaten gut funktioniert und eine hohe Korrektklassifizierungsrate aufweist, aber nicht für neue und ungesehene Daten des Testdatensatzes (*Holdout*). Ein überangepasstes Modell ist nicht in der Lage, die Daten aus dem Validierungsdatensatz zu verallgemeinern, da es zu komplex ist. Es konvergiert auf die Trainingsdaten, aber nicht auf die Validierungsdaten. Daher lernt es die Trainingsdaten förmlich auswendig – ohne eine Erkenntnis, die es in die Lage versetzt, den Validierungsdatensatz zu generalisieren.<sup>cxxxiii</sup>

Im Gegensatz dazu bedeutet Unteranpassung, dass ein Modell zu einfach ist und weder die Trainingsdaten noch die Validierungsdaten verallgemeinern kann. Ein unterangepasstes Modell konvergiert nicht und weist eine geringe Genauigkeit auf. Es kann die Variabilität des Datensatzes nicht abbilden. In der Literatur wird die Unteranpassung gern mit einem hohen Bias verglichen.<sup>cxxxiv</sup> Die Korrektklassifizierungsrate (*KKR*) bezeichnet den Anteil der korrekten Vorhersagen, *KKR* ist der Kehrwert der Fehlerquote:

$$(24) \quad KKR = \frac{TP+TN}{FP+FN+TP+TN}.$$

$$(25) \quad KKR = 1 - FQ, \text{ mit } FQ = \frac{FP+FN}{FP+FN+TP+TN}.$$

Zur Beurteilung der Generalisierungsfähigkeit eines Modells sollten stets aktuelle Vorhersagen erstellt und mit dem Zielwert verglichen werden. In der Wahrheitmatrix (Vgl. Tabelle 2) sind korrekte und falsche Vorhersagen der einzelnen Klassen gegenübergestellt, aufgeteilt nach der Art des Fehlers: falsch positiv (*engl. false positive, FP*), richtig positiv (*engl. true positive, TP*), falsch negativ (*engl. false negative, FN*) und richtig negativ (*engl. true negative, TN*). Die Gewichtung unterschiedlicher Klassifikationsfehler werden in (Vgl. 4.2 Umgang mit unausgewogenen Datensätzen) untersucht.

	Tatsächlich wahr	Tatsächlich falsch
Wahre Vorhersage	richtig positiv (TP)	falsch positiv (FP)
Falsche Vorhersage	falsch negativ (FN)	richtig negativ (TN)

Tabelle 2: Wahrheitmatrix (Eigene Darstellung)<sup>cxxxv</sup>

Das Verhältnis richtig-positiver-Vorhersagen dividiert durch die Gesamtmenge aller positiven Vorhersagen ist die Genauigkeit (*engl. precision*).<sup>cxxxvi</sup>

$$(26) \quad precision = \frac{TP}{FP+TP}.$$

Der Anteil der richtig-positiven Objekte an der Gesamtmenge aller positiven Vorhersagen  $T$  beschreibt die Trefferquote (*engl. recall*). Sie wird außerdem als Richtig-Positiv-Rate (*RPR*) bezeichnet:<sup>cxxxvii</sup>

$$(27) \quad recall = RPR = TP/T = \frac{TP}{FN+TP}, \text{ mit } T = FN + TP.$$

Die Kombination aus *recall* und *precision* beschreibt das F1-Maß.<sup>cxxxviii</sup>

$$(28) \quad \text{F1-Maß} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

### 3.1.8 Hyperparameter anpassen

Je nach Bewertungsergebnis sind die Parameter des Modells anzupassen. Für die Parameterauswahl gibt es keine allgemeingültige Vorgehensweise. Am besten eignet sich eine Parametersuche mittels der Gittersuche oder einer zufälligen Suche.<sup>cxxxix</sup>

Es lassen sich drei Arten von Parametern unterscheiden.<sup>34</sup>

1. Lernparameter
  - a. Metriken (*metrics*) zur Bewertung und Betrachtung der Lernkurve
  - b. Lernalgorithmus (*loss*) (Vgl. 3.2.3 Modellauswahl, Zeile 28)
  - c. Optimierungsalgorithmus (*optimizer*)
2. Trainingsparameter
  - a. Lernrate  $\eta$  (Vgl. 2.5.2 Minimierung der Straffunktion)
  - b. Losgröße (*batch-size*)
  - c. Epochen (*epochs*)
  - d. Frühzeitiges Stoppen (*early\_stopping*)
3. Modellparameter (und Komplexität)
  - a. Anzahl der Knoten (Neuronen) und Schichten
  - b. Regularisierungs-Methode und -Rate (Vgl. 2.6.5 Regularisierungsschicht)
  - c. Convolution Parameter  $f, m, s, p$  (Vgl. 2.6.1.1 eindimensionale diskrete Faltung)

### 3.1.9 Einsatz in der Produktion

1. Vorhersagen treffen.
2. Einbindung des Klassifikators in den produktiven Alltag seines Einsatzfeldes (Krankenhausprozess von der Bestellung des Röntgenbildes über die Aufnahme und Bildgebung bis hin zur Klassifikation des Radiologen – oder alternativ eine Webseite zum Upload für die entfernte Diagnose).
3. Regelmäßige Neubewertung, indem neue Produktionsdaten regelmäßig mit dem Klassifikator abgeglichen werden. Ab einer bestimmten Abweichung: Neubeginn des Trainings, damit das Modell aktuell bleibt.

---

<sup>34</sup> (Vgl. <https://developers.google.com/machine-learning/guides/text-classification/step-5>, Abgerufen am 11.6.19)



## 3.2 CNN-Implementierung

Der hier präsentierte Quellcode ist die strukturierte Variante des Codes und unter GitHub einsehbar. Der zunächst unstrukturierte Code wurde in die ML-Pipeline überführt und strukturiert. Im Wurzelverzeichnis des GitHub-Links sind beide Versionen einsehbar (die Abbildungen stammen aus den Auswertungen im Quellcode):<sup>35</sup>

1. Unstrukturiert: *CNN.ipynb*
2. Strukturiert: *CNN\_to\_Pipeline.ipynb*

### 3.2.1 Datenauswahl

```

1. base_dir = '/media/ente/M2/2018 - 11 - sorted data'
2. labels = [ 'Atelectasis', 'Cardiomegaly', 'Consolidation', 'Edema', 'Effusion', 'Emphysema',
  'Fibrosis', 'Hernia', 'Infiltration', 'Mass', 'No Finding', 'Nodule', 'Pleural_Thickenin
  g', 'Pneumonia', 'Pneumothorax' ]
3. import os
4. ate_dir = os.path.join(base_dir, labels[0])
5. car_dir = os.path.join(base_dir, labels[1])
6. con_dir = os.path.join(base_dir, labels[2])
7. ede_dir = os.path.join(base_dir, labels[3])
8. eff_dir = os.path.join(base_dir, labels[4])
9. emp_dir = os.path.join(base_dir, labels[5])
10. fib_dir = os.path.join(base_dir, labels[6])
11. her_dir = os.path.join(base_dir, labels[7])
12. inf_dir = os.path.join(base_dir, labels[8])
13. mas_dir = os.path.join(base_dir, labels[9])
14. nof_dir = os.path.join(base_dir, labels[10])
15. nod_dir = os.path.join(base_dir, labels[11])
16. ple_dir = os.path.join(base_dir, labels[12])
17. pne_dir = os.path.join(base_dir, labels[13])
18. pn2_dir = os.path.join(base_dir, labels[14])
19. ate_fnames = os.listdir(ate_dir)
20. car_fnames = os.listdir(car_dir)
21. con_fnames = os.listdir(con_dir)
22. ede_fnames = os.listdir(ede_dir)
23. eff_fnames = os.listdir(eff_dir)
24. emp_fnames = os.listdir(emp_dir)
25. fib_fnames = os.listdir(fib_dir)
26. her_fnames = os.listdir(her_dir)
27. inf_fnames = os.listdir(inf_dir)
28. mas_fnames = os.listdir(mas_dir)
29. nof_fnames = os.listdir(nof_dir)
30. nod_fnames = os.listdir(nod_dir)
31. ple_fnames = os.listdir(ple_dir)
32. pne_fnames = os.listdir(pne_dir)
33. pn2_fnames = os.listdir(pn2_dir)

```

### 3.2.2 Datenverständnis

#### 3.2.2.1 Klassenverteilung

```

1. print ('total Atelectasis images:      ', len(os.listdir(ate_dir)))
2. print ('total Cardiomegaly images:      ', len(os.listdir(car_dir)))

```

<sup>35</sup> (Vgl. <https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow>, Abgerufen am 11.6.19)

```

3. print ('total Consolidation images: ', len(os.listdir(con_dir)))
4. print ('total Edema images: ', len(os.listdir(edema_dir)))
5. print ('total Effusion images: ', len(os.listdir(eff_dir)))
6. print ('total Emphysema images: ', len(os.listdir(emp_dir)))
7. print ('total Fibrosis images: ', len(os.listdir(fib_dir)))
8. print ('total Hernia images: ', len(os.listdir(her_dir)))
9. print ('total Infiltration images: ', len(os.listdir(inf_dir)))
10. print ('total Mass images: ', len(os.listdir(mas_dir)))
11. print ('total No_Finding images: ', len(os.listdir(nof_dir)))
12. print ('total Nodule images: ', len(os.listdir(nod_dir)))
13. print ('total Pleural_Thickening images: ', len(os.listdir(ple_dir)))
14. print ('total Pneumonia images: ', len(os.listdir(pne_dir)))
15. print ('total Pneumothorax images: ', len(os.listdir(pn2_dir)))

```

```

total Atelectasis images:      2015
total Cardiomegaly images:     464
total Consolidation images:    633
total Edema images:            310
total Effusion images:         1804
total Emphysema images:        461
total Fibrosis images:         366
total Hernia images:           53
total Infiltration images:     4640
total Mass images:             1043
total No_Finding images:       28000
total Nodule images:           1273
total Pleural_Thickening images: 525
total Pneumonia images:        161
total Pneumothorax images:     1099

```

### 3.2.3 Modellauswahl

```

1. from keras.applications.vgg16 import VGG16
2. from keras.preprocessing import image
3. from keras.models import Model, Sequential
4. from keras.layers import Dense, Dropout, Flatten
5. from keras import backend as K
6. from keras import applications
7. from keras.optimizers import SGD
8. import keras_metrics as km
9.
10. # build the VGG16 network
11. base_model = applications.VGG16 (weights='imagenet', include_top=False,
    input_shape = (224,224,3))
12. x = base_model.output
13. x = Flatten(input_shape=base_model.output_shape[1:])(x)
14. x = Dense(1024, activation='relu')(x)
15. x = Dropout(0.5)(x)
16. # and a classification layer -- we have 15 classes
17. predictions = Dense(len(labels), activation = 'softmax') (x)
18.
19. # this is the model we will train
20. model = Model(inputs=base_model.input, outputs=predictions)
21.
22. # set the first 19 layers (up to the last conv block)
23. # to non-trainable (weights will not be updated)
24. for layer in model.layers[:19]:
25.     layer.trainable = False
26.
27. # compile the model with a SGD/momentum optimizer and a very slow learning rate.
28. model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=1e-4, momentum=0.9), metric
    s = ['categorical_accuracy', km.precision(), km.recall()])
29. model.summary()

```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_1 (Dense)	(None, 1024)	25691136
dropout_1 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 15)	15375
Total params: 40,421,199		
Trainable params: 25,706,511		
Non-trainable params: 14,714,688		

### 3.2.4 Datenverteilung [Split] +

```
1. validation_split = 0.2 #setze den Anteil des Validierungsdatensatzes
```

### 3.2.5 Datentransformation

```

2. batch_size = 32
3. rescale = 224
4. target_size = (rescale, rescale)
5. #input_shape = (rescale, rescale, 3)
6.
7. from tensorflow.keras.preprocessing.image import ImageDataGenerator
8. from tensorflow.keras.preprocessing.image import array_to_img, img_to_array, load_img
9. datagen = ImageDataGenerator(
10.     rescale=1./255,
11.     samplewise_center=True,
12.     samplewise_std_normalization=True,
13.     validation_split= validation_split) #setze validation-split Anteil
14.
15. train_generator = datagen.flow_from_directory(
16.     base_dir,
17.     target_size=target_size,
18.     batch_size=batch_size,
19.     class_mode='categorical',
20.     color_mode='rgb',
21.     subset='training') #setze Trainingsdatensatz (training data)
22.
23. validation_generator = datagen.flow_from_directory(
24.     base_dir, # gleicher Ordner wie Trainingsdaten
25.     target_size=target_size,
26.     batch_size=batch_size,
27.     #early_stopping = keras.callbacks.EarlyStopping(monitor='val_acc', min_delta=0, patience=3, verbose=1, mode='auto'),
28.     class_mode='categorical',
29.     color_mode='rgb',
30.     subset='validation') #setze Validierungsdatensatz (validation data)
31. print(validation_generator.class_indices) #kodierte Klassenbezeichnungen
32. print(validation_generator.classes)

```

Found 34283 images belonging to 15 classes.

Found 8564 images belonging to 15 classes.

```
{'Atelectasis': 0, 'Cardiomegaly': 1, 'Consolidation': 2, 'Edema': 3, 'Effusion': 4,
'Emphysema': 5, 'Fibrosis': 6, 'Hernia': 7, 'Infiltration': 8, 'Mass': 9, 'No Finding':
10, 'Nodule': 11, 'Pleural_Thickening': 12, 'Pneumonia': 13, 'Pneumothorax': 14}
[ 0  0  0 ... 14 14 14]
```

#### 3.2.5.1 Transformation Visualisieren

```

1. img_path = os.path.join(ate_dir, ate_fnames[316])
2. img = load_img(img_path, target_size=target_size) # this is a PIL image
3. x = img_to_array(img) # Numpy array with shape (150, 150, 3)
4. x = x.reshape((1,) + x.shape) # Numpy array with shape (1, 150, 150, 3)
5.
6. # The .flow() command below generates batches of randomly transformed images
7. # It will loop indefinitely, so we need to `break` the loop at some point!
8. i = 0
9. for batch in datagen.flow(x, batch_size=1):
10.     plt.figure(i)
11.     imgplot = plt.imshow(array_to_img(batch[0]))
12.     i += 1
13.     if i % 1 == 0:
14.         break

```

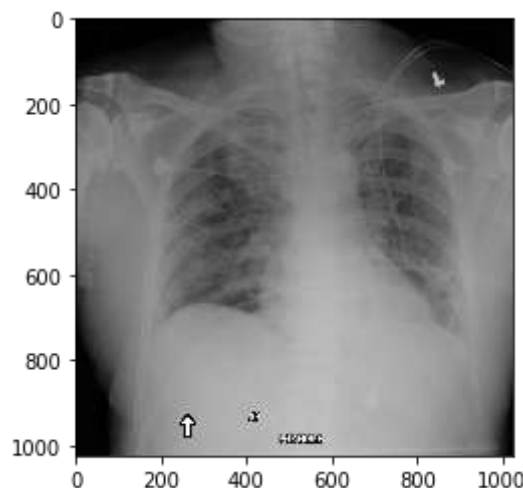


Abbildung 34: ImageAugmentation visualisieren

### 3.2.6 Trainingsphase #1 [Pre-Training]

```

1. history1=model.fit_generator(
2.     train_generator,
3.     steps_per_epoch = train_generator.samples // batch_size,
4.     validation_data = validation_generator,
5.     validation_steps = validation_generator.samples // batch_size,
6.     class_weight = 'balanced', #setze Strafterm für nicht-balancierten Datensatz
7.     epochs = 10) #setze die Anzahl der Epochen für das Vortraining

```

Epoch 1/10

```

1071/1071 [=====] - 618s 577ms/step - loss: 1.4269 -
categorical_accuracy: 0.6494 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3545 - val_categorical_accuracy: 0.6536 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 2/10

```

1071/1071 [=====] - 609s 569ms/step - loss: 1.3661 -
categorical_accuracy: 0.6532 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3356 - val_categorical_accuracy: 0.6542 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 3/10

```

1071/1071 [=====] - 609s 569ms/step - loss: 1.3447 -
categorical_accuracy: 0.6535 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3286 - val_categorical_accuracy: 0.6528 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 4/10

```

1071/1071 [=====] - 617s 576ms/step - loss: 1.3285 -
categorical_accuracy: 0.6531 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3290 - val_categorical_accuracy: 0.6520 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 5/10

```

1071/1071 [=====] - 623s 582ms/step - loss: 1.3169 -
categorical_accuracy: 0.6525 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3321 - val_categorical_accuracy: 0.6545 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 6/10

```

1071/1071 [=====] - 614s 573ms/step - loss: 1.3050 -
categorical_accuracy: 0.6530 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3092 - val_categorical_accuracy: 0.6530 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

Epoch 7/10

```

1071/1071 [=====] - 602s 562ms/step - loss: 1.2955 -
categorical_accuracy: 0.6531 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3136 - val_categorical_accuracy: 0.6540 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00
Epoch 8/10
1071/1071 [=====] - 602s 562ms/step - loss: 1.2847 -
categorical_accuracy: 0.6535 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3081 - val_categorical_accuracy: 0.6517 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00
Epoch 9/10
1071/1071 [=====] - 602s 562ms/step - loss: 1.2775 -
categorical_accuracy: 0.6533 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3025 - val_categorical_accuracy: 0.6532 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00
Epoch 10/10
1071/1071 [=====] - 601s 561ms/step - loss: 1.2649 -
categorical_accuracy: 0.6537 - precision: 0.0000e+00 - recall: 0.0000e+00 - val_loss:
1.3008 - val_categorical_accuracy: 0.6524 - val_precision: 0.0000e+00 - val_recall:
0.0000e+00

```

### 3.2.7 Modellbewertung

#### 3.2.7.1 Lernkurve [Pre-Training]

```

1. from matplotlib import pyplot
2. pyplot.plot(history1.history['categorical_accuracy'])
3. pyplot.plot(history1.history['val_categorical_accuracy'])
4. pyplot.title('Training and validation accuracy')
5. pyplot.show()
6.
7. pyplot.plot(history1.history['loss'])
8. pyplot.plot(history1.history['val_loss'])
9. pyplot.title('Training and validation loss')
10. pyplot.show()

```



Abbildung 35: Genauigkeit Lernkurve



Abbildung 36: Zielfunktion (loss) Lernkurve

#### 3.2.7.2 Wahrheitsmatrix und Klassifizierungsbericht [Pre-Training]

```

1. import numpy as np
2. from sklearn.metrics import classification_report, confusion_matrix
3. Y_pred = model.predict_generator(validation_generator, validation_generator.samples // batch_size+1)
4. y_pred = np.argmax(Y_pred, axis=1)
5. print('Confusion Matrix')
6. print(confusion_matrix(validation_generator.classes, y_pred))
7. print('Classification Report')
8. print(classification_report(validation_generator.classes, y_pred, target_names=labels))

```

Confusion Matrix

```
[[ 0  0  0  0  4  0  0  0  6  0 393  0  0  0  0]
```

```
[ 0  0  0  0  1  0  0  0  0  0  91  0  0  0  0]
[ 0  0  0  0  1  0  0  0  2  0 123  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  62  0  0  0  0]
[ 0  0  0  0  1  0  0  0  1  0 358  0  0  0  0]
[ 0  0  0  0  1  0  0  0  2  0  89  0  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  72  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  10  0  0  0  0]
[ 0  0  0  0  5  0  0  0  5  0 918  0  0  0  0]
[ 0  0  0  0  1  0  0  0  1  0 206  0  0  0  0]
[ 0  0  0  0 20  0  0  0 41  0 5538 0  0  0  1]
[ 0  0  0  0  2  0  0  0  3  0  249 0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  105 0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  32  0  0  0  0]
[ 0  0  0  0  1  0  0  0  3  0 215  0  0  0  0]]
```

## Classification Report

	precision	recall	f1-score	support
Atelectasis	0.00	0.00	0.00	403
Cardiomegaly	0.00	0.00	0.00	92
Consolidation	0.00	0.00	0.00	126
Edema	0.00	0.00	0.00	62
Effusion	0.03	0.00	0.01	360
Emphysema	0.00	0.00	0.00	92
Fibrosis	0.00	0.00	0.00	73
Hernia	0.00	0.00	0.00	10
Infiltration	0.08	0.01	0.01	928
Mass	0.00	0.00	0.00	208
No Finding	0.65	0.99	0.79	5600
Nodule	0.00	0.00	0.00	254
Pleural_Thickening	0.00	0.00	0.00	105
Pneumonia	0.00	0.00	0.00	32
Pneumothorax	0.00	0.00	0.00	219
micro avg	0.65	0.65	0.65	8564
macro avg	0.05	0.07	0.05	8564
weighted avg	0.44	0.65	0.52	8564

## 3.2.8 Tiefere Feinanpassung

```
1. # at this point, the top layers are well trained and we can start fine-tuning
2. # convolutional layers from inception V3. We will freeze the bottom N layers
3. # and train the remaining top layers.
4.
5. # let's visualize layer names and layer indices to see how many layers
6. # we should freeze:
7. for i, layer in enumerate(base_model.layers):
8.     print(i, layer.name)
9.
10. # we chose to train the first convolution blocks, i.e. we will freeze
11. # the first 24 layers and unfreeze the rest:
12. for layer in model.layers[:15]:
13.     layer.trainable = False
14. for layer in model.layers[15:]:
15.     layer.trainable = True
16.
17. # we need to recompile the model for these modifications to take effect
18. # we use SGD with a low learning rate
19. from keras.optimizers import SGD
20. #metrics= ['categorical_accuracy']
21. model.compile(optimizer=SGD(lr=0.0001, momentum=0.9), loss='categorical_crossentropy',
    metrics=['categorical_accuracy'])
```

```
22. model.summary()
```

```
0 input_1
1 block1_conv1
2 block1_conv2
3 block1_pool
4 block2_conv1
5 block2_conv2
6 block2_pool
7 block3_conv1
8 block3_conv2
9 block3_conv3
10 block3_pool
11 block4_conv1
12 block4_conv2
13 block4_conv3
14 block4_pool
15 block5_conv1
16 block5_conv2
17 block5_conv3
18 block5_pool
```

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808



block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_1 (Dense)	(None, 1024)	25691136
dropout_1 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 15)	15375
=====		
Total params: 40,421,199		
Trainable params: 32,785,935		
Non-trainable params: 7,635,264		

### 3.2.9 Trainingsphase #2 [Feinanpassung]

```

1. # we train our model again (this time fine-tuning the deeper Conv layers)
2. # class_weight = {0: 3.7, 1: 50.1, 2: 2.2, ..., 14: 0.4}
3. history2 = model.fit_generator(
4.     train_generator,
5.     steps_per_epoch = train_generator.samples // batch_size,
6.     validation_data = validation_generator,
7.     validation_steps = validation_generator.samples // batch_size,
8.     class_weight = 'balanced',
9.     epochs = 35)

```

Epoch 1/35

1071/1071 [=====] - 615s 574ms/step - loss: 1.2614 - categorical\_accuracy: 0.6535 - val\_loss: 1.2925 - val\_categorical\_accuracy: 0.6518

Epoch 2/35

1071/1071 [=====] - 600s 560ms/step - loss: 1.2326 - categorical\_accuracy: 0.6556 - val\_loss: 1.2936 - val\_categorical\_accuracy: 0.6465

Epoch 3/35

1071/1071 [=====] - 610s 570ms/step - loss: 1.2099 - categorical\_accuracy: 0.6568 - val\_loss: 1.3085 - val\_categorical\_accuracy: 0.6484

Epoch 4/35

1071/1071 [=====] - 604s 564ms/step - loss: 1.1842 - categorical\_accuracy: 0.6596 - val\_loss: 1.2828 - val\_categorical\_accuracy: 0.6542

Epoch 5/35

1071/1071 [=====] - 602s 562ms/step - loss: 1.1615 - categorical\_accuracy: 0.6618 - val\_loss: 1.2892 - val\_categorical\_accuracy: 0.6479

Epoch 6/35

1071/1071 [=====] - 602s 562ms/step - loss: 1.1408 - categorical\_accuracy: 0.6648 - val\_loss: 1.2730 - val\_categorical\_accuracy: 0.6497

Epoch 7/35

1071/1071 [=====] - 600s 560ms/step - loss: 1.1147 - categorical\_accuracy: 0.6678 - val\_loss: 1.2896 - val\_categorical\_accuracy: 0.6492

Epoch 8/35

1071/1071 [=====] - 599s 559ms/step - loss: 1.0877 - categorical\_accuracy: 0.6722 - val\_loss: 1.3072 - val\_categorical\_accuracy: 0.6477

Epoch 9/35

1071/1071 [=====] - 600s 560ms/step - loss: 1.0630 - categorical\_accuracy: 0.6754 - val\_loss: 1.2822 - val\_categorical\_accuracy: 0.6472

Epoch 10/35

1071/1071 [=====] - 599s 560ms/step - loss: 1.0345 - categorical\_accuracy: 0.6823 - val\_loss: 1.3174 - val\_categorical\_accuracy: 0.6474

Epoch 11/35

```
1071/1071 [=====] - 599s 559ms/step - loss: 1.0071 -  
categorical_accuracy: 0.6865 - val_loss: 1.3069 - val_categorical_accuracy: 0.6498  
Epoch 12/35  
1071/1071 [=====] - 600s 560ms/step - loss: 0.9747 -  
categorical_accuracy: 0.6936 - val_loss: 1.3169 - val_categorical_accuracy: 0.6455  
Epoch 13/35  
1071/1071 [=====] - 605s 565ms/step - loss: 0.9451 -  
categorical_accuracy: 0.6999 - val_loss: 1.3094 - val_categorical_accuracy: 0.6467  
Epoch 14/35  
1071/1071 [=====] - 607s 567ms/step - loss: 0.9122 -  
categorical_accuracy: 0.7073 - val_loss: 1.3409 - val_categorical_accuracy: 0.6457  
Epoch 15/35  
1071/1071 [=====] - 605s 565ms/step - loss: 0.8802 -  
categorical_accuracy: 0.7187 - val_loss: 1.3107 - val_categorical_accuracy: 0.6410  
Epoch 16/35  
1071/1071 [=====] - 599s 559ms/step - loss: 0.8436 -  
categorical_accuracy: 0.7272 - val_loss: 1.3710 - val_categorical_accuracy: 0.6432  
Epoch 17/35  
1071/1071 [=====] - 602s 562ms/step - loss: 0.8056 -  
categorical_accuracy: 0.7388 - val_loss: 1.3515 - val_categorical_accuracy: 0.6425  
Epoch 18/35  
1071/1071 [=====] - 604s 564ms/step - loss: 0.7701 -  
categorical_accuracy: 0.7476 - val_loss: 1.3757 - val_categorical_accuracy: 0.6380  
Epoch 19/35  
1071/1071 [=====] - 601s 562ms/step - loss: 0.7351 -  
categorical_accuracy: 0.7595 - val_loss: 1.3943 - val_categorical_accuracy: 0.6401  
Epoch 20/35  
1071/1071 [=====] - 602s 562ms/step - loss: 0.6932 -  
categorical_accuracy: 0.7710 - val_loss: 1.4007 - val_categorical_accuracy: 0.6344  
Epoch 21/35  
1071/1071 [=====] - 600s 560ms/step - loss: 0.6585 -  
categorical_accuracy: 0.7822 - val_loss: 1.4265 - val_categorical_accuracy: 0.6363  
Epoch 22/35  
1071/1071 [=====] - 601s 561ms/step - loss: 0.6204 -  
categorical_accuracy: 0.7972 - val_loss: 1.4612 - val_categorical_accuracy: 0.6447  
Epoch 23/35  
1071/1071 [=====] - 603s 563ms/step - loss: 0.5800 -  
categorical_accuracy: 0.8085 - val_loss: 1.4784 - val_categorical_accuracy: 0.6219  
Epoch 24/35  
1071/1071 [=====] - 600s 560ms/step - loss: 0.5481 -  
categorical_accuracy: 0.8216 - val_loss: 1.5415 - val_categorical_accuracy: 0.6349  
Epoch 25/35  
1071/1071 [=====] - 601s 561ms/step - loss: 0.5102 -  
categorical_accuracy: 0.8324 - val_loss: 1.5511 - val_categorical_accuracy: 0.6418  
Epoch 26/35  
1071/1071 [=====] - 602s 562ms/step - loss: 0.4786 -  
categorical_accuracy: 0.8426 - val_loss: 1.5750 - val_categorical_accuracy: 0.6362  
Epoch 27/35  
1071/1071 [=====] - 601s 561ms/step - loss: 0.4466 -  
categorical_accuracy: 0.8574 - val_loss: 1.6134 - val_categorical_accuracy: 0.6328  
Epoch 28/35  
1071/1071 [=====] - 600s 560ms/step - loss: 0.4156 -  
categorical_accuracy: 0.8671 - val_loss: 1.6777 - val_categorical_accuracy: 0.6360  
Epoch 29/35  
1071/1071 [=====] - 603s 563ms/step - loss: 0.3791 -  
categorical_accuracy: 0.8785 - val_loss: 1.6189 - val_categorical_accuracy: 0.6234  
Epoch 30/35  
1071/1071 [=====] - 601s 561ms/step - loss: 0.3539 -  
categorical_accuracy: 0.8863 - val_loss: 1.6450 - val_categorical_accuracy: 0.6258
```

```

Epoch 31/35
1071/1071 [=====] - 602s 562ms/step - loss: 0.3219 -
categorical_accuracy: 0.9000 - val_loss: 1.6273 - val_categorical_accuracy: 0.6078
Epoch 32/35
1071/1071 [=====] - 601s 561ms/step - loss: 0.2944 -
categorical_accuracy: 0.9071 - val_loss: 1.6412 - val_categorical_accuracy: 0.6077
Epoch 33/35
1071/1071 [=====] - 601s 561ms/step - loss: 0.2753 -
categorical_accuracy: 0.9154 - val_loss: 1.7529 - val_categorical_accuracy: 0.6084
Epoch 34/35
1071/1071 [=====] - 601s 561ms/step - loss: 0.2510 -
categorical_accuracy: 0.9237 - val_loss: 1.7092 - val_categorical_accuracy: 0.6163
Epoch 35/35
1071/1071 [=====] - 604s 564ms/step - loss: 0.2333 -
categorical_accuracy: 0.9307 - val_loss: 1.9145 - val_categorical_accuracy: 0.6214

```

### 3.2.10 Feinanpassung bewerten

#### 3.2.10.1 Lernkurve betrachten

```

1. from matplotlib import pyplot
2. pyplot.plot(history2.history['categorical_accuracy'])
3. pyplot.plot(history2.history['val_categorical_accuracy'])
4. pyplot.title('Training and validation accuracy')
5. pyplot.show()
6.
7. pyplot.plot(history2.history['loss'])
8. pyplot.plot(history2.history['val_loss'])
9. pyplot.title('Training and validation loss')
10. pyplot.show()

```

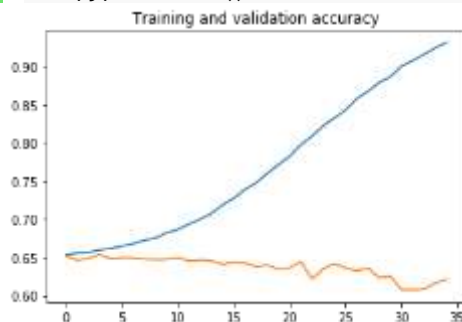


Abbildung 37: Genauigkeit Lernkurve #2

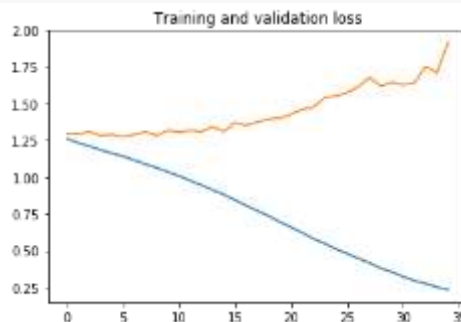


Abbildung 38: Zielfunktion (loss) Lernkurve #2

#### 3.2.10.2 Wahrheitsmatrix und Klassifizierungsbericht

```

1. #Confusion Matrix and Classification Report
2. import numpy as np
3. from sklearn.metrics import classification_report, confusion_matrix
4. Y_pred = model.predict_generator(validation_generator, validation_generator.samples // bat
ch_size+1)
5. y_pred = np.argmax(Y_pred, axis=1)
6. print('Confusion Matrix')
7. print(confusion_matrix(validation_generator.classes, y_pred))
8. print('Classification Report')
9. print(classification_report(validation_generator.classes, y_pred, target_names=labels))

```

Confusion Matrix

```

[[ 17  2  0  9  1  2  0  8  1 361  2  0  0  0]
 [ 2  1  0  0  4  0  1  0  3  0 80  0  1  0]
 [ 4  1  0  1  5  0  0  0  6  0 108  0  0  1]
 [ 1  1  0  0  0  0  1  0  2  0 56  0  1  0]
 [ 12  3  0  1 13  0  1  0 23  0 303  1  0  3]

```

```
[ 3  1  1  0  2  0  0  0  5  0  80  0  0  0  0]
[ 1  0  0  0  3  0  0  0  5  0  64  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  10  0  0  0  0]
[ 29  1  0  1  35  0  2  0  40  2  810  6  1  0  1]
[ 13  1  0  0  6  1  0  0  9  2  174  1  0  0  1]
[ 198  27  7  4  141  3  18  0  215  11  4935  17  12  0  12]
[ 12  3  0  0  5  0  1  0  17  0  213  1  0  0  2]
[ 2  0  0  0  3  0  1  0  4  0  93  0  1  0  1]
[ 1  0  0  0  0  0  0  0  0  0  31  0  0  0  0]
[ 8  0  0  0  5  0  1  0  11  0  194  0  0  0  0]]
```

## Classification Report

	precision	recall	f1-score	support
Atelectasis	0.06	0.04	0.05	403
Cardiomegaly	0.02	0.01	0.02	92
Consolidation	0.00	0.00	0.00	126
Edema	0.00	0.00	0.00	62
Effusion	0.06	0.04	0.04	360
Emphysema	0.00	0.00	0.00	92
Fibrosis	0.00	0.00	0.00	73
Hernia	0.00	0.00	0.00	10
Infiltration	0.11	0.04	0.06	928
Mass	0.12	0.01	0.02	208
No Finding	0.66	0.88	0.75	5600
Nodule	0.04	0.00	0.01	254
Pleural_Thickening	0.06	0.01	0.02	105
Pneumonia	0.00	0.00	0.00	32
Pneumothorax	0.00	0.00	0.00	219
micro avg	0.59	0.59	0.59	8564
macro avg	0.08	0.07	0.06	8564
weighted avg	0.45	0.59	0.50	8564

## 3.2.10.3 Gewichtungen visualisieren

```
1. import random
2. from tensorflow.keras.preprocessing.image import img_to_array, load_img
3. # Let's define a new Model that will take an image as input, and will output
4. # intermediate representations for all layers in the previous model after
5. # the first.
6. successive_outputs = [layer.output for layer in model.layers[1:]]
7. visualization_model = Model(base_model.input, successive_outputs)
8.
9. # Let's prepare a random input image of a cat or dog from the training set.
10. cat_img_files = [os.path.join(ate_dir, f) for f in ate_fnames]
11. dog_img_files = [os.path.join(car_dir, f) for f in car_fnames]
12. img_path = random.choice(cat_img_files + dog_img_files)
13.
14. img = load_img(img_path, target_size=target_size) # this is a PIL image
15. x = img_to_array(img) # Numpy array with shape (150, 150, 3)
16. x = x.reshape((1,) + x.shape) # Numpy array with shape (1, 150, 150, 3)
17.
18. # Rescale by 1/255
19. x /= 255
20.
21. # Let's run our image through our network, thus obtaining all
22. # intermediate representations for this image.
23. successive_feature_maps = visualization_model.predict(x)
24.
25. # These are the names of the layers, so can have them as part of our plot
26. layer_names = [layer.name for layer in model.layers]
27.
```

```

28. # Now let's display our representations
29. for layer_name, feature_map in zip(layer_names, successive_feature_maps):
30.     if len(feature_map.shape) == 4:
31.         # Just do this for the conv / maxpool layers, not the fully-connected layers
32.         n_features = feature_map.shape[-1] #number of features in feature map
33.         # The feature map has shape (1, size, size, n_features)
34.         size = feature_map.shape[1]
35.         # We will tile our images in this matrix
36.         display_grid = np.zeros((size, size * n_features))
37.         for i in range(n_features):
38.             # Postprocess the feature to make it visually palatable
39.             x = feature_map[0, :, :, i]
40.             x -= x.mean()
41.             x /= x.std()
42.             x *= 64
43.             x += 128
44.             x = np.clip(x, 0, 255).astype('uint8')
45.             # We'll tile each filter into this big horizontal grid
46.             display_grid[:, i * size : (i + 1) * size] = x
47.         # Display the grid
48.         scale = 20. / n_features
49.         pyplot.figure(figsize=(scale * n_features, scale))
50.         pyplot.title(layer_name)
51.         pyplot.grid(False)
52.         pyplot.imshow(display_grid, aspect='auto', cmap='viridis')

```

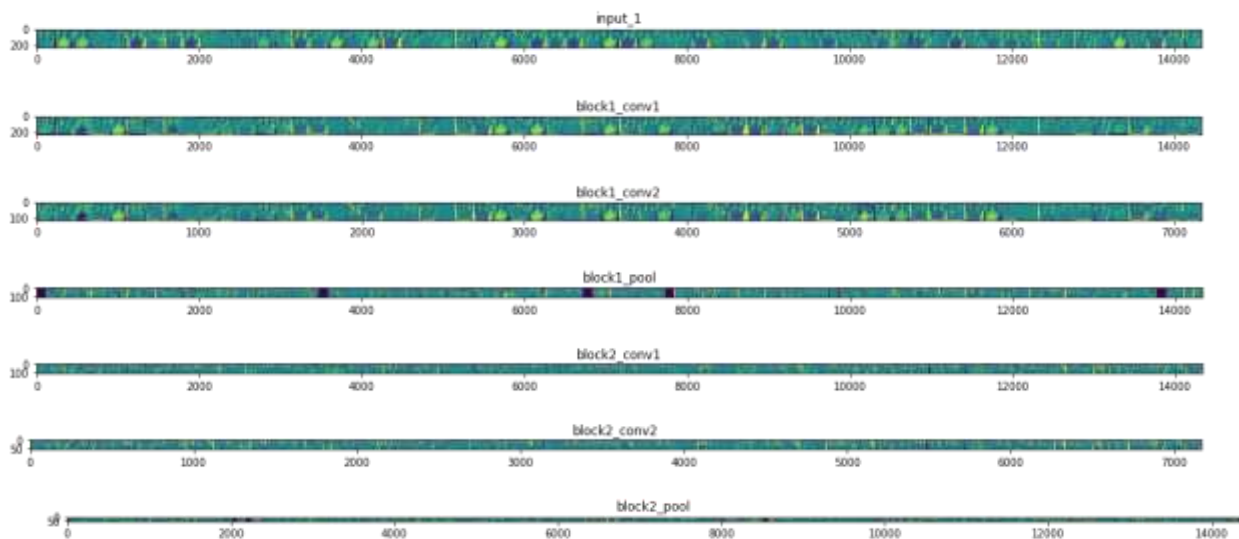


Abbildung 39: Gewichtungen grafisch visualisieren

### 3.2.11 Gewichte exportieren

```

1. import h5py
2. export_dir= '/home/ente/Dropbox/THESIS/Code/Transfer_InceptionV3/Export'
3. model.save_weights(export_dir + 'my_model_weights.h5')
4.
5. ##### save as JSON
6. json_string = model.to_json()
7.
8. ##### save as YAML
9. yaml_string = model.to_yaml()

```

### 3.3 System und Software-Anforderungen

Modellaufbau, Training und die abschließende Bewertung fanden anfangs auf zwei Laptops statt: Einem MacBook Pro (i7 CPU, 16GB RAM, GPU NVIDIA GeForce 650GT 1GB/250GB HDD) und einem Lenovo ThinkPad X1 (i7 CPU, 8GB RAM, 120GB SSD). Anhand der Trainingszeiten kristallisierte sich relativ früh heraus, dass diese Konfiguration unter reinem CPU-Training zu langsam ist. Schließlich nahm eine Epoche bis zu zwei Tage Zeit in Anspruch. Obwohl das MacBook eine dedizierte Grafikkarte besitzt, eigneten sich diese GPU-Ressourcen nicht für das Training, da es keinen aktuellen CUDA-Treiber für MacOS gibt bzw. NVIDIA wegen eines Rechtsstreites keine neuen Treiber für MacOS bereitstellt.<sup>36</sup>

Daher wurde für diese Arbeit ein PC mit folgender Konfiguration benutzt:

- Prozessor: AMD Ryzen 5 2600
- Grafikkarte: Palit NVIDIA GeForce RTX 2080 Ti, 11GB
- Arbeitsspeicher: 16GB (2x8GB) Hynix HyperX DDR4, 2666Mhz, CL16
- Mainboard: Gigabyte B450M-S2H
- SSD: Patriot Burst 240GB
- M2-SSD: Transcend NVME, 256GB

Software-Anforderungen:

- Betriebssystem Ubuntu 18.04.02 LTS<sup>37</sup>
- Anaconda 2019.03<sup>38</sup> für Python 3.7
- CUDA Toolkit Update 1 10.1<sup>39</sup>
- cudNN v7.5.1<sup>40</sup>
- NVIDIA Treiber 430.14
- Jupyter Notebook<sup>41</sup>

---

<sup>36</sup> (Vgl. <https://www.pcbuildersclub.com/2019/01/zwischen-apple-und-nvidia-herrscht-angeblich-eine-stille-feindschaft/>, Abgerufen am 11.6.19)

<sup>37</sup> (Vgl. <https://www.ubuntu.com/download/desktop>, Abgerufen am 11.6.19)

<sup>38</sup> (Vgl. <https://www.anaconda.com/distribution/#download-section>, Abgerufen am 11.6.19)

<sup>39</sup> (Vgl. <https://developer.nvidia.com/cuda-downloads>, Abgerufen am 11.6.19)

<sup>40</sup> (Vgl. <https://developer.nvidia.com/cudnn>, Abgerufen am 11.6.19)

<sup>41</sup> (Vgl. <https://jupyter.org/>, Abgerufen am 11.6.19)

## Kapitel 4 – Ergebnis und Ausblick

Dieses Kapitel zeigt die Ergebnisse im Umgang mit verschiedenen CNN-Implementierungen, produktiven Einsatzfeldern und ein ausgewähltes CXR-Referenzmodell für den NIH-Datensatz.

Zunächst werden CNN-Unterschiede anhand der CNN-Bausteine am Vorbild vergangener ImageNet-Referenzmodelle herausgearbeitet. Anschließend wird der Umgang mit unausgewogenen Datensätzen erläutert.

Im dritten Abschnitt wird die beste Modellarchitektur zur CXR-Klassifikation anhand des Modells von Baltrushat et al. erklärt, wobei auf andere Implementierungen (die weniger Attribute nutzen) lediglich verwiesen wird. Im vierten Abschnitt werden Einsatzmethoden von ML im Kontext eines Krankenhauses aus Sicht von Radiologen beleuchtet.

Im letzten Abschnitt wird einerseits das Vorgehen zur Aufsetzung (und zum Training) eines Transferlernen-Ansatzes erklärt, andererseits dessen Möglichkeiten und Grenzen.

## 4.1 Unterschiede moderner CNN - Implementierungen

AlexNet, der Gewinner des ImageNet-Wettbewerbs aus dem Jahre 2012, führte zu bahnbrechenden Erfolgen, da es die Fehlerrate des Vorjahressiegers (XRCE<sup>42</sup>) fast halbierte (Vgl. Tabelle 3: ImageNet ILSVRC). AlexNet benutzt eine ähnliche Netzwerkarchitektur wie LeNet<sup>cxl</sup> (Vgl. Tabelle 5). Spätere Gewinner des Wettbewerbs haben die Architektur von AlexNet in den darauffolgenden Jahren immer weiter verfeinert, um die Fehlerrate zu senken. AlexNet, InceptionV3, VGGNet und ResNet stellen vier Meilensteine dar und werden im Folgenden näher untersucht.

Jahr	ILSVRC - Klassifikation	Fehlerrate
2011	XRCE	25,7 % <sup>43</sup>
2012	AlexNet ( <i>SuperVision</i> )	15,3 % <sup>44</sup>
2013	ZFNet ( <i>ZF</i> )	13,5 % <sup>45</sup>
2014	InceptionV3 ( <i>GoogLeNet</i> )	6,66 % <sup>46</sup>
2014	VGGNet ( <i>VGG</i> )	7,4 % <sup>47</sup>
2015	ResNet ( <i>MSRA</i> )	3,57 % <sup>48</sup>
2016	InceptionV4 ( <i>Trimps-Soushen</i> )	2,99 % <sup>49</sup>
2017	( <i>Trimps-Soushen</i> )	2,48 % <sup>50</sup>

Tabelle 3: ImageNet ILSVRC (Eigene Darstellung)

Tabelle 3 zeigt die Gewinner des ImageNet-Wettbewerbs aus den Jahren 2011 bis 2017 und ihre Fehlerrate im Klassifikations-Teilwettbewerb. Die mittlere Spalte (*ILSVRC - Klassifikation*) umfasst den Namen des Netzwerks sowie (in Klammern) den Namen des Teams.<sup>51</sup>

Der ImageNet-Wettbewerb besteht seit 2014 aus drei Teilwettbewerben:

1. Objekterkennung (*engl. object detection, DET*),
2. Objektlokalisierung (*engl. object localization, LOC*) und
3. Objekterkennung aus Videos (*engl. object detection from video, VID*),

<sup>42</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2011/ilsvrc11.pdf>, Abgerufen am 11.6.19)

<sup>43</sup> (Vgl. <http://image-net.org/challenges/LSVRC/2011/>, Abgerufen am 11.6.19)

<sup>44</sup> (Vgl. <http://image-net.org/challenges/LSVRC/2012/results.html#t1>, Abgerufen am 11.6.19)

<sup>45</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2013/results.php#cls>, Abgerufen am 11.6.19)

<sup>46</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2014/results#clsloc>, Abgerufen am 11.6.19)

<sup>47</sup> (Vgl. Ebenda)

<sup>48</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2015/results#loc>, Abgerufen am 11.6.19)

<sup>49</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2016/results#loc>, Abgerufen am 11.6.19)

<sup>50</sup> (Vgl. <http://www.image-net.org/challenges/LSVRC/2017/results#loc>, Abgerufen am 11.6.19)

<sup>51</sup> (Vgl. <http://image-net.org/challenges/LSVRC/2015/results#team>, Abgerufen am 11.6.19)



In dieser Arbeit werden ausschließlich die Ergebnisse des Klassifikationswettbewerbs (aus dem Teilwettbewerb 2: Objektlokalisierung) verwendet. Seit 2018 wird der Wettbewerb über die Plattform kaggle ausgetragen.<sup>52</sup>

#### 4.1.1 LeNet als Vorbild von AlexNet

Die Architektur von LeNet beinhaltet acht Schichten (inklusive Ein- und Ausgabeschicht): Zwei Convolution-Blöcke (Convolution-Sigmoid-AvgPooling-Folgen), bevor die fünfte Convolution-Schicht (C5) in zwei FC-Schichten übergeht (Vgl. Abbildung 40). Die Pooling-Strategie lautet 2x2 Average-Pooling ( $s = 2$ ). LeNet unterscheidet zehn Zielklassen  $i$  (wobei  $i \in \{0, 1, 2, \dots, 9\}$ ) – wie an der Dimension der Ausgabeschicht zu erkennen ist (Vgl. Abbildung 40, *OUTPUT*).

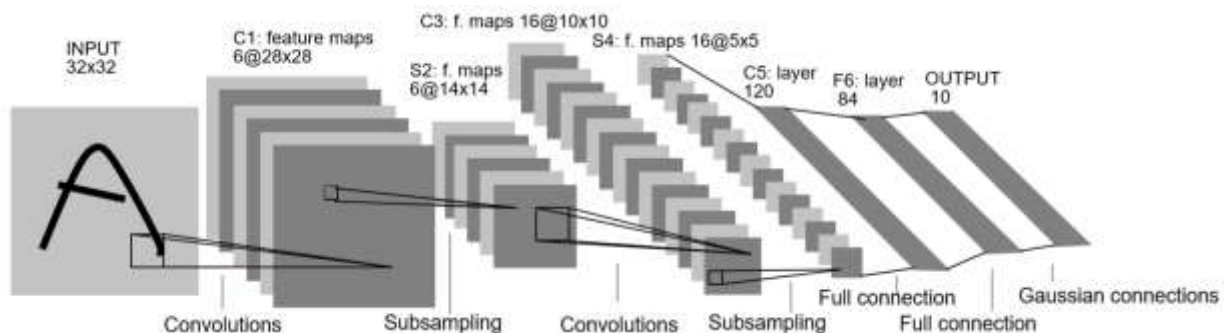


Abbildung 40: LeNet-Architektur (Quelle: LeCun et al. 1998, S.8)

#### 4.1.2 AlexNet

Die Architektur von AlexNet ist durch drei Convolution-ReLU-MaxPooling-Folgen (*Convolution-Blöcke*) und die anschließende Überführung in drei vollständig verknüpfte Schichten geprägt. Die letzte FC-Schicht beschreibt die Ausgabeschicht mit ihren 1000 Zielklassen (Vgl. Tabelle 5, *erste Zeile*  $FC_{1000}$ ).

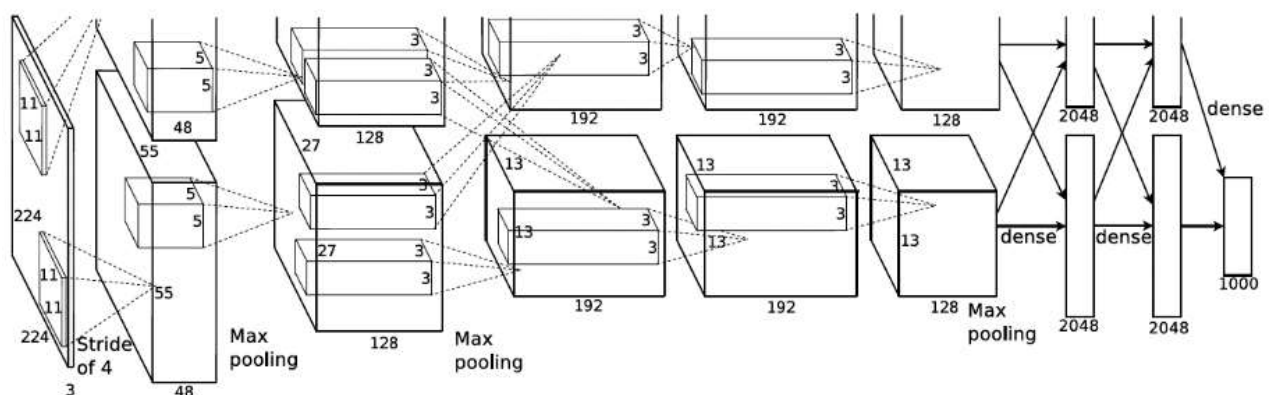


Abbildung 41: AlexNet-Architektur (Quelle: Krizhevsky et al. 2012, S.5)

<sup>52</sup> (Vgl. <https://www.kaggle.com/c/imagenet-object-localization-challenge/>, Abgerufen am 11.6.19)

Inklusive Ein- und Ausgabeschicht besteht AlexNet aus zehn Schichten (Vgl. Tabelle 4 und Abbildung 41). Darüber hinaus nutzt AlexNet Image-Augmentation und Dropout zur Verminderung einer Überanpassung (*overfit*) (Vgl. 2.6.5 Regularisierungsschicht).<sup>cxli</sup> Die Architektur ist aufgrund von Speicherbeschränkungen zweigeteilt; das Training fand auf zwei Grafikkarten statt. Im Folgenden wird die verbundene (schichtenweise aufsummierte) Architektur analysiert (ohne Teilung).

Name	#Parameter	Max	Post Conv.	C-Blöcke (#S)	#Schichten
AlexNet <sup>cxlii</sup>	60.000.000	224	6x6x256	3 (5)	10
DenseNet121 <sup>cxliii</sup>	8.062.504	224	7x7x1024	5 (8)	429 (121)
DenseNet169	14.307.880	224	7x7x1664		597 (169)
DenseNet201	20.242.984	224	7x7x1920		709 (201)
InceptionV3 <sup>cxliv</sup>	23.851.784	299	8x8x2048	12	313 (159)
InceptionV4 <sup>cxlv</sup>	55.873.736	299	8x8x1536		782 (572)
LeNet <sup>cxlvi</sup>	59.900	32	5x5x16 <sup>cxlvii</sup>	2 (3)	8 (7)
MobileNet <sup>cxlviii</sup>	4.253.864	224	7x7x1024		93 (88)
MobileNetV2 <sup>cxlix</sup>	3.538.984	224	7x7x1280		157 (88)
NASNetMobile	5.326.716	224	7x7x1056		771 (-)
NASNetLarge <sup>cl</sup>	88.949.818	331	11x11x4032		1041 (-)
ResNet50 <sup>cli</sup>	25.636.712	224	7x7x2048	2 (46) <sup>clii</sup>	177 (-)
ResNet50V2 <sup>cliii</sup>	25.613.800	224			(-)
ResNeXt50 <sup>cliv</sup>	25.097.128	224			(-)
SqueezeNet <sup>clv</sup>	1.248.424	224	13x13x1000		
VGG16 <sup>clvi</sup>	138.357.544	224	7x7x512	5 (13) <sup>clvii</sup>	23 (23)
VGG19	143.667.240	224	7x7x512	5 (16) <sup>clviii</sup>	26 (26)
Xception <sup>clix</sup>	22.910.480	299	10x10x2048	4 (36) <sup>clx</sup>	133 (126)

Tabelle 4: CNN Überblick (Eigene Darstellung)<sup>cxli</sup>

Tabelle 4 bietet einen Überblick über CNN-Architekturen, die aus dem ImageNet-Wettbewerb zwischen 2012 und 2017 als Referenzmodell oder Sieger hervorgegangen sind. Einzige Ausnahmen sind ZFNet (2013) und LeNet (1998). Die Spalten von Tabelle 4 beinhalten den Namen des Netzwerkes (*Name*), die Anzahl der Parameter (*#Parameter*), das maximale Eingabeformat (*Max*), das Ausgabeformat vor der ersten FC-Schicht (*Post Conv*) sowie die Anzahl der Convolution-Blöcke und (in Klammern) die Anzahl der Convolution-Schichten (*Conv-Blöcke (#S)*). Die letzte Spalte zählt die Anzahl der Schichten (inklusive Ein- und Ausgabeschicht), während die Module-Architektur von InceptionV3 und historischen Nachfolgern die Anzahl von Schichten explodieren lässt. Der Wert in Klammern beschreibt die Angabe von Keras (*#Schichten*).<sup>53</sup> Die Referenzmodelle *ResNet101*<sup>54</sup>, sowie

<sup>53</sup> (Vgl. <https://keras.io/applications/>, Abgerufen am 11.6.19)

<sup>54</sup> (Vgl. [https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow/blob/master/cnn%20basic%20architecture%20\(vgl.%20%5B4.1%5D%20Tabelle%20%205%2B6\)/ResNet101.ipynb](https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow/blob/master/cnn%20basic%20architecture%20(vgl.%20%5B4.1%5D%20Tabelle%20%205%2B6)/ResNet101.ipynb), Abgerufen am 11.6.19)

dessen Nachfolger *ResNeXt* und ähnliche, konnten im Rahmen der Modellanalyse aufgrund einer nicht funktionierenden Dokumentation nicht nachgebaut werden und daher grau hinterlegt.

#### 4.1.3 Architekturelle Meilensteine

“We need to go deeper”<sup>55</sup>,

lautete die Begründung für den Aufbau tiefer Netzwerke wie InceptionV3. Moderne CNNs gewinnen zunehmend an Breite und Tiefe. Breit im Sinne der Anzahl von Knoten, tief im Sinne der Anzahl von Schichten.<sup>clxii</sup> Folglich nutzen sie auch mehr Parameter.

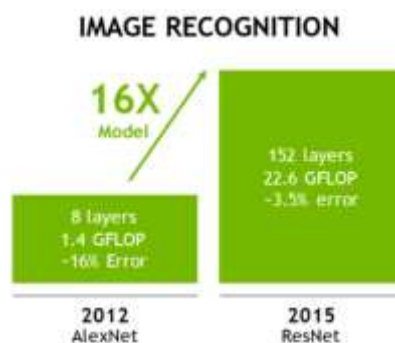


Abbildung 42: zunehmende Modellkomplexität<sup>56</sup>

VGGNet (VGG16 und VGG19) hat den ImageNet-Wettbewerb zwar nie gewonnen. Aufgrund seiner simplen Architektur (basierend auf AlexNet) war es dennoch sehr populär und fand entsprechend viele Nachahmer. VGG16 (VGG19) besitzt insgesamt 23 (26) Schichten, davon 16 (19) mit Parametern (Vgl. Tabelle 4). Ein weiterer Vorteil der simplen VGG-Architektur ist die Stapelung von kleinen 3x3 Convolutions ( $m = 3$ ). So lassen sich rezeptive Felder der Höhe 7x7 (wie ZFNet<sup>clxiii</sup>) bei deutlich weniger Parametern und erhöhter Diskriminierung erzielen.<sup>clxiv</sup>

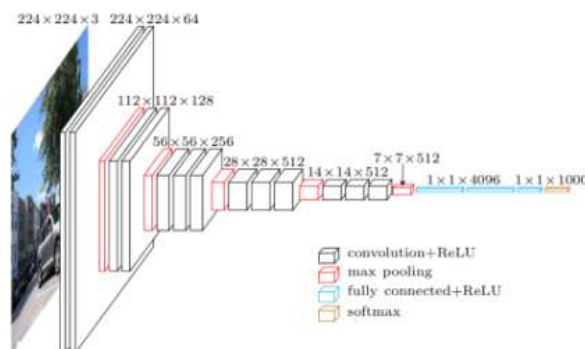


Abbildung 43: VGGNet Architektur<sup>57</sup>

<sup>55</sup> (Vgl. <https://knowyourmeme.com/memes/we-need-to-go-deeper>, Abgerufen am 11.6.19)

<sup>56</sup> (Quelle: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.4, Abgerufen am 11.6.19)

<sup>57</sup> (Quelle: <https://towardsdatascience.com/deep-learning-for-image-classification-why-its-challenging-where-we-ve-been-and-what-s-next-93b56948fcef>, Abgerufen am 11.6.19)

Der Gewinner des ImageNet-Wettbewerbs 2014 war jedoch nicht VGGNet, sondern InceptionV3. Die Architektur von VGGNet ist allerdings ähnlich aufgebaut wie die von InceptionV3: Es gibt ähnlich viele gewichtete Schichten und eine modulare Convolution-Strategie, die mit kleinen Convolution-Blöcken (1x1, 3,3 und 5x5) genauso wie das Inception-Modul (Vgl. Abbildung 44) arbeitet – mit der Besonderheit der 1x1-Convolution als sogenannte Flaschenhals-Schicht (*engl. bottleneck layer*), welche die Anzahl der Kanäle halbiert.<sup>clxv</sup>

#### 4.1.4 InceptionV3

Das gesamte InceptionV3-Netzwerk (Vgl. Abbildung 45) besteht neben der klassischen Eingabeschicht aus Convolution-Pooling-Folgen aus einem Stapel von neun Inception-Modulen *I* (Vgl. Abbildung 44 und Vgl. Tabelle 5; *Sonderfall I' bei anschließendem Pooling*) sowie zwei Seitenarmen, den Hilfsklassifikatoren (*engl. auxiliary classifiers*).

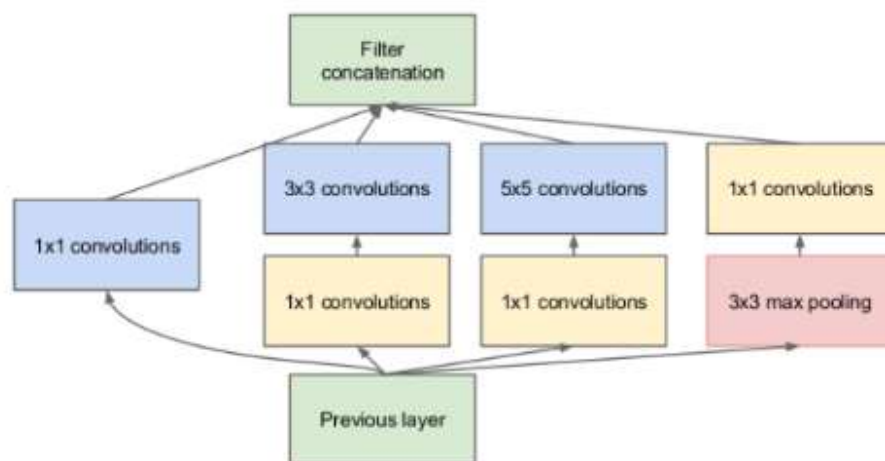


Abbildung 44: Das Inception-Modul<sup>clxvi</sup>

Drei weitere Besonderheiten von InceptionV3 sind die reduzierte Zahl an Pooling-Schichten (Vgl. Abbildung 44, *roter Kasten*; Vgl. Tabelle 5, *I'*)<sup>58</sup> sowie der Verzicht auf parameterreiche und teure FC-Schichten (bis auf die Klassifizierungsschicht) und die Ersetzung durch Global Average Pooling<sup>59</sup>.<sup>clxvii</sup> Aus diesen Gründen ist InceptionV3 besonders effizient und trotzdem präzise. Es hat eine ähnliche Tiefe und Genauigkeit wie VGGNet, aber deutlich weniger Parameter: 23 Millionen im Vergleich zu rund 140 Millionen (Vgl. Tabelle 4).

<sup>58</sup> (Vgl. <http://principlesofdeeplearning.com/index.php/2018/08/27/is-pooling-dead-in-convolutional-networks/>, Abgerufen am 11.6.19)

<sup>59</sup> (Vgl. <http://principlesofdeeplearning.com/index.php/a-tutorial-on-global-average-pooling/>, Abgerufen am 11.6.19)

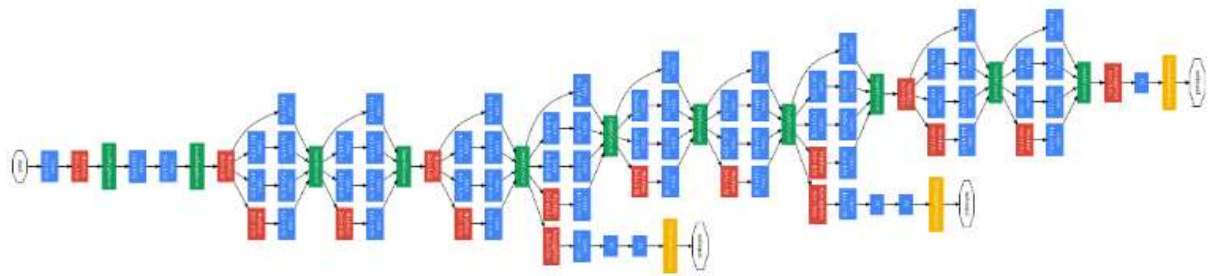


Abbildung 45: InceptionV3 Architektur (Quelle: Szegedy et al. 2014, S.4)

Noch mehr Schichten als InceptionV3 weisen ResNet und DenseNet auf, wobei DenseNet eine ähnlich modulare Struktur wie InceptionV3 benutzt. Sie haben aufgrund ihrer Tiefe (Komplexität) noch stärker mit dem Problem des verschwindenden Gradienten (*engl. vanishing gradient*) zu kämpfen.<sup>clxxiii</sup>

Spezielle Abkürzungen (*engl. skip connection*) innerhalb eines Residual-Blocks (*engl. residual block*) des Vorreiters ResNet erlauben das Überspringen von Schichten, indem sie die Identität der Eingabe an die nächste Schicht weitergeben (Vgl. Abbildung 46).<sup>clxxix</sup> Diese Querverbindungen gründen auf der Idee eines Netzwerks in einem Netzwerk<sup>clxxx</sup>. Anstatt als Entwickler immer nur eine Modellstruktur zu testen, werden dem Modell einfach alle Strukturmöglichkeiten zur Verfügung gestellt. Es wird ihm selbst überlassen, welche Operatoren-Folge die passende ist.<sup>clxxxi</sup>

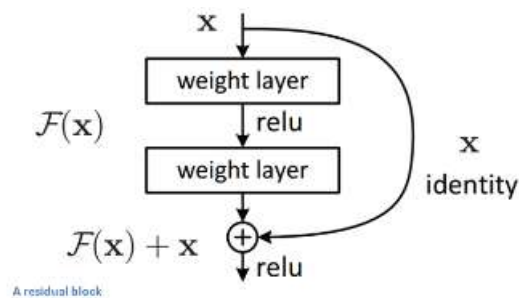


Abbildung 46: Der Residual-Block (Quelle: He et al. 2015, S.2)

#### 4.1.5 Architektur Übersicht

Name	Aufbau
AlexNet <sup>clxxii</sup>	$(C_{96}^{11;4}P_M)^{27 \times 27} - (C_{256}^{5;1}P_M)^{13 \times 13} - (C_{384}^{3;1}(C_{256}^2)^{3;1}P_M)^{6 \times 6 \times 256} - FC_{4096}^2 - FC_{1000}$
DenseNet121	$CP_M^{56 \times 56} - DCP_A^{28 \times 28} - DCP_A^{14 \times 14} - DCP_A^{7 \times 7} - DP_G^{1 \times 1} - FC_{1000}$
InceptionV3	$CP_M - C^2P_M - I^2P_M - II'I^2I'P_M - II^{8 \times 8 \times 2048} - P_G^{1 \times 1 \times 2048} - FC_{1000}$
LeNet	$(C^{5 \times 5 \times 6}P_A^{2 \times 2})_6^{14 \times 14} - (C^{5 \times 5 \times 16}P_A^{2 \times 2})_{16}^{5 \times 5} - C_{120}^{1 \times 1} - FC_{84} - FC_{10}$
ResNet50 <sup>clxxiii</sup>	$(C_{64}^{7;2}P_M^{3;2})_{64}^{56 \times 56} - 3(C_{64}^2)4(C_{128}^3)6(C_{256}^3)(3C_{512}^3)^{7 \times 7 \times 2048} - P_G^{1 \times 1 \times 2048} - FC_{1000}$
VGG16	$C^2P_{M_{64}} - C^2P_{M_{128}} - C^3P_{M_{256}} - C^3P_{M_{512}} - C^3P_{M_{512}} - FC_{4096}^2 - FC_{1000}$
VGG19	$C^2P_{M_{64}} - C^2P_{M_{128}} - C^4P_{M_{256}} - C^4P_{M_{512}} - C^4P_{M_{512}} - FC_{4096}^2 - FC_{1000}$
Xception	$C_{64}^2C_{128}^2C^2P_{M_{256}} - C^2P_{M_{728}} - (8C^3)C^2P_M - C^2P_G - FC_{2048} - FC_{opt} - FC_{1000}$

Tabelle 5: CNN-Architekturen (Eigene Darstellung)<sup>clxxiv</sup>

Eine Übersicht ausgewählter CNNs und ihrer Bausteine bietet Tabelle 5. Hochgestellte Indizes des Formats:  $(CP)^{m*m*f}$  beschreiben das Ausgabeformat  $m * m * f$  ( $m$ : Filter-Größe,  $f$ : Anzahl Filter), während der einfache Index mehrere gleiche Schichten hintereinander reiht:  $C^4 := CCCC$  oder  $FC_{4096}^2 := FC_{4096} - FC_{4096}$ . Tiefgestellte Indizes nennen die Anzahl enthaltener Einheiten:  $FC_{2048}$  ist eine vollständig verknüpfte Schicht mit 2048 Einheiten, während  $C_{256}^{5;1}$  und  $C^{5x5x256}$  (mit Parametern  $m = 5$ ;  $s = 1$ ;  $p = 0$ ;  $f = 256$ ) eine  $m * m$  Convolution-Schicht aus  $f$  Filtern beschreibt ( $f$ : Anzahl der Merkmalskarten).  $P_x$  nennt die Pooling-Variante ( $P_A$ : Average- (Mean)-Pooling,  $P_M$ : Max-Pooling,  $P_G$ : Global-Average-Pooling).

#### 4.1.6 Kompression von Modellen

Ein weiterer Trend moderner CNN-Implementierungen ist eine möglichst hohe Kompression (*engl. pruning*) des Modells, während die Genauigkeit gleichbleiben soll. Ein komprimiertes Modell kann effizienter und schneller trainiert werden, da die Speicherzugriffe und -Operationen (unter verringerter Präzision<sup>60</sup>) weniger Zeit und Energie kosten, was für den Betrieb auf mobilen Endgeräten sehr wichtig ist (Vgl. 4.1.7 Weitere Vergleichskriterien; Vgl. Abbildung 48). In einer verteilten Umgebung wird darüber hinaus weniger Kommunikation zwischen den Hardwareeinheiten benötigt.<sup>clxxv</sup>

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	<b>50x</b>	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	<b>363x</b>	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	<b>510x</b>	57.5%	80.3%

Abbildung 47: SqueezeNet-Kompression auf AlexNet (Quelle: Iandola et al. 2016, S.7)

Darüber hinaus ist ein kompakteres Modell leichter (effizienter) zu betreiben, da eine Aktualisierung schneller vonstattengeht und weniger Speicher (und Bandbreite) benötigt. SqueezeNet arbeitet nur mit 1/60 der Parameter-Anzahl im Vergleich zu AlexNet (Vgl. Tabelle 4); SqueezeNet verbraucht gerade einmal 1/510 des Speichers und besitzt eine vergleichbare Genauigkeit zu AlexNet (Vgl. Abbildung 47, *letzte Zeile*).

<sup>60</sup> (Vgl. <https://software.intel.com/en-us/articles/lower-numerical-precision-deep-learning-inference-and-training>, Abgerufen am 11.6.19)

#### 4.1.7 Weitere Vergleichskriterien

Neben der Genauigkeit eines Modells sind seine benötigten Ressourcen zum Training (Vorhersagen optimieren: Gewichtungen anpassen) und zum Betrieb (Vorhersagen treffen: Gewichtungen anwenden) von besonderer Relevanz.

Name	Fehlerrate	Trainingsdauer
ResNet38	10,76%	2,5 Tage
ResNet50	7,02%	5 Tage
ResNet101	6,21%	7 Tage
ResNet152	6,06%	10,5 Tage

Tabelle 6: ResNet-Trainings-Dauer<sup>61</sup>

Um die Komplexität von CNNs zu vergleichen, sind zunächst seine benötigten Ressourcen zu analysieren. Die Anzahl der Gewichtungsparameter wird häufig als Indiz der benötigten Trainingszeit und des Rechenbedarfs benutzt. Je mehr Parameter ein Modell verwendet, desto größer sind sein Speicherbedarf und die Zahl an Rechenoperationen. Der Ressourcenbedarf besitzt mehrere Ausprägungen:

1. *Rechenbedarf*: Anzahl der Rechenoperationen zur Bestimmung einer Vorhersage; in Fließkomma-Operationen pro Sekunde: (*engl. floating point operations per second, FLOPS*),
2. *Speicherbedarf*: Modellgröße, Parameter-Anzahl (Gewichtungskoeffizienten)<sup>62</sup>,
  - a. Präzision: Wertebereich und Repräsentation von Variablen<sup>clxxvi</sup>,
3. *Zeitbedarf*: benötigte Zeit bis zur Zielerreichung (Geschwindigkeit)
  - a. des Trainings: notwendige Zeit zur Konvergenz,
  - b. der Anwendung: Vorhersagen mit den Gewichtungen treffen,
4. *Energiebedarf*: Ressourcen (Energie, Zeit und Hardware), optimale Zuteilung (Effizienz), wobei Operationen erhöhter Präzision teurer sind (Vgl. Abbildung 48, *Memory*).

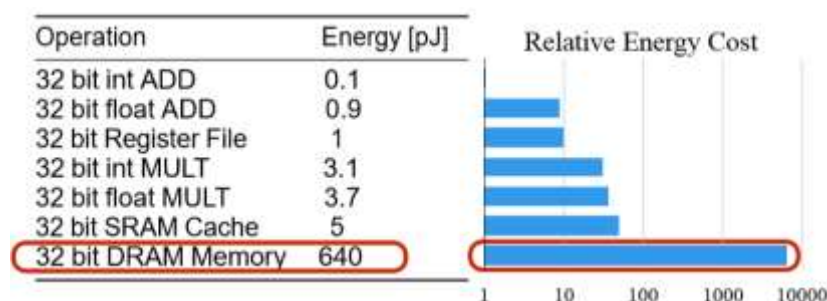


Abbildung 48: Energiekosten unterschiedlicher Rechenoperationen<sup>63</sup>

<sup>61</sup> (Quelle: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.6, Abgerufen am 11.6.19)

<sup>62</sup> (Vgl. [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.4, Abgerufen am 11.6.19)

<sup>63</sup> (Quelle: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.9, Abgerufen am 11.6.19)



## 4.2 Umgang mit unausgewogenen Datensätzen

Unausgewogene (nicht-balancierte) Datensätze sind dadurch gekennzeichnet, dass bestimmte Klassenbezeichnungen einen deutlich größeren Anteil des Datensatzes ausmachen (Vgl. Tabelle 8, Zeile *No\_Finding*).

Bei der Betrugserkennung (*engl. fraud detection*) sind die meisten Datenpunkte negativ (kein Spam). Ihr Anteil an den Gesamtdaten beträgt zwischen 90 und 99 Prozent, da es deutlich weniger positive Datenpunkte gibt. Für solch einen Datensatz ist die Korrektklassifizierungsrate keine geeignete Bewertungsmethode zur Beurteilung der Generalisierungsfähigkeit eines Modells, weil ein überangepasstes Modell trotz einer Genauigkeit von 99 Prozent schlicht den Anteil der größten Klassenbezeichnung (*engl. majority class*) zurückgibt. In solchen Fällen sollten andere Bewertungsmethoden herangezogen werden, zum Beispiel die Wahrheitsmatrix oder die Richtig-Positiv-Rate (Vgl. 3.1.7 Bewertung des Modells).

Die optimale Auswahl der Bewertungsmethode sollte durch das Klassifizierungsziel, den Datensatz sowie mögliche Fehlerarten und deren Tragweite bestimmt sein. Wenn das Ziel eine möglichst hohe Erkennungsrate bösartiger Tumore ist, dann ist die Trefferquote das optimale Kriterium. Für die Betrugserkennung ist hingegen die Genauigkeit das geeignetere Maß.<sup>clxxvii</sup>

Damit der ungleichen Verteilung Rechnung getragen wird, sollte die Zielfunktion um einen Strafterm erweitert werden, der eine Fehlklassifizierung der unterrepräsentierten Klasse stärker bestraft. Der Strafterm hat dabei die Form eines Wörterbuchs (*engl. dictionary*) und beinhaltet für jede Zielklasse einen Eintrag. Alternativ kann das Attribut `class_weights='balanced'`<sup>clxxviii</sup> anstelle einzelner Klassengewichtungen verwendet werden. Der balancierte Strafterm wird nach folgender Formel berechnet<sup>64</sup> (Vgl. 3.2.9 Trainingsphase #2 [Feinanpassung], Zeile 8):

$$(29) \quad \text{balanced} = \frac{\text{Gesamtanzahl der Datenpunkte}}{\text{Gesamtanzahl\_Klassen} * \text{Anzahl\_Datenpunkte}} \rightarrow \frac{42.862}{15 * x_i}^{65}$$

Resampling-Methoden<sup>clxxix</sup> werden auf nicht-balancierte Datensätze angewendet, um die Anzahl der unter- oder überrepräsentierten Datenpunkte einer Klasse künstlich zu erhöhen oder zu mindern (*engl. oversampling / undersampling*).<sup>clxxx</sup> Ein Beispiel für Oversampling ist SMOTE. Es erzeugt neue und künstliche Datenpunkte, die (abhängig von den vorhandenen Daten) einen möglichst kleinen Unterschied besitzen.<sup>clxxxi</sup>

<sup>64</sup> (Vgl. [https://scikit-learn.org/stable/modules/generated/sklearn.utils.class\\_weight.compute\\_class\\_weight.html](https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html), Abgerufen am 11.6.19)

<sup>65</sup> (Vgl. [https://github.com/scikit-learn/scikit-learn/blob/7813f7efb/sklearn/utils/class\\_weight.py#L8](https://github.com/scikit-learn/scikit-learn/blob/7813f7efb/sklearn/utils/class_weight.py#L8), Abgerufen am 11.6.19)



### 4.3 CXR - Implementierung

Einen Überblick über 300 verschiedene Deep-Learning-Implementierungen in der medizinischen Bildgebung (*engl. medical imaging*) bieten Litjens et al.. Sie beschäftigen sich mit allgemeinen Bildgebungsaufgaben, von der Klassifikation, Detektion, Segmentierung und Registrierung bis hin zu anderen Verfahren wie Bildwiedergabe und -generierung (*engl. image retrieval bzw. image generation*). Alle Anwendungsprobleme werden den verschiedenen Bildgebungsmethoden des Röntgens, Computertomographie (CT) und Magnetresonanztomographie (MRT oder auch Kernspintomographie genannt) und anderen (wie Mammographie und Ultraschall) zugeteilt. Des Weiteren findet eine Zuordnung der eingesetzten ML-Methoden statt (Vgl. Tabelle 7).<sup>clxxxii</sup>

Table 3: Overview of papers using deep learning techniques for chest x-ray image analysis.

Reference	Application	Remarks
Lo et al. (1995)	Nodule detection	Classifies candidates from small patches with two-layer CNN, each with $12.5 \times 5$ filters
Anavi et al. (2015)	Image retrieval	Combines classical features with those from pre-trained CNN for image retrieval using SVM
Bar et al. (2015)	Pathology detection	Features from a pre-trained CNN and low level features are used to detect various diseases
Anavi et al. (2016)	Image retrieval	Continuation of Anavi et al. (2015), adding age and gender as features
Bar et al. (2016)	Pathology detection	Continuation of Bar et al. (2015), more experiments and adding feature selection
Cicero et al. (2016)	Pathology detection	GoogLeNet CNN detects five common abnormalities, trained and validated on a large data set
Hwang et al. (2016)	Tuberculosis detection	Processes entire radiographs with a pre-trained fine-tuned network with 6 convolution layers
Kim and Hwang (2016)	Tuberculosis detection	MIL framework produces heat map of suspicious regions via deconvolution
Shin et al. (2016a)	Pathology detection	CNN detects 17 diseases, large data set (7k images), recurrent networks produce short captions
Rajkomar et al. (2017)	Frontal/lateral classification	Pre-trained CNN performs frontal/lateral classification task
Yang et al. (2016c)	Bone suppression	Cascade of CNNs at increasing resolution learns bone images from gradients of radiographs
Wang et al. (2016a)	Nodule classification	Combines classical features with CNN features from pre-trained ImageNet CNN

Tabelle 7: Deep-Learning für CXR (Quelle: Litjens et al. 2017, S.17)

Neben diesem umfassenden Überblick bietet die Arbeit von Baltruschat et al. die besten Ergebnisse und eine vorbildliche CNN-Architektur, da es neben den klassischen Bild-Attributen weitere Patientenattribute (z. B. Text-Merkmale, Vgl. Abbildung 49, *Patient Data*) in die Klassifikation mit einbezieht.<sup>clxxxiii</sup> Obwohl dieses Netzwerk mit 185.000 Bildern weniger trainiert wurde, besitzt es ein ähnliches Resultat wie das Vergleichsmodell, was den Vorteil der benutzen Text-Merkmale und des verdoppelten Eingabeformats von  $448 \times 448 \times 1$  hervorhebt.<sup>clxxxiv</sup>

Die Architektur der Modelle von Baltruschat et al. basiert auf ResNet, wobei verschiedene ResNet-Tiefen und Modellkombinationen verglichen werden.<sup>clxxxv</sup> Das beste Ergebnis liefert ResNet38 mit vergrößertem Eingabeformat unter Einbezug von Textmerkmalen. Um dem vergrößerten Eingabeformat Rechnung zu tragen, wurde vor der ursprünglichen Eingabeschicht eine weitere Convolution-Pooling-Folge eingesetzt.<sup>clxxxvi</sup> Nach dieser Folge entspricht das Ausgabeformat dem ursprünglichen Eingabeformat von ResNet und das restliche Netzwerk bleibt unverändert (bis auf die Klassifizierungs-Schicht im Verbund mit den Textmerkmalen am Ende).<sup>clxxxvii</sup>

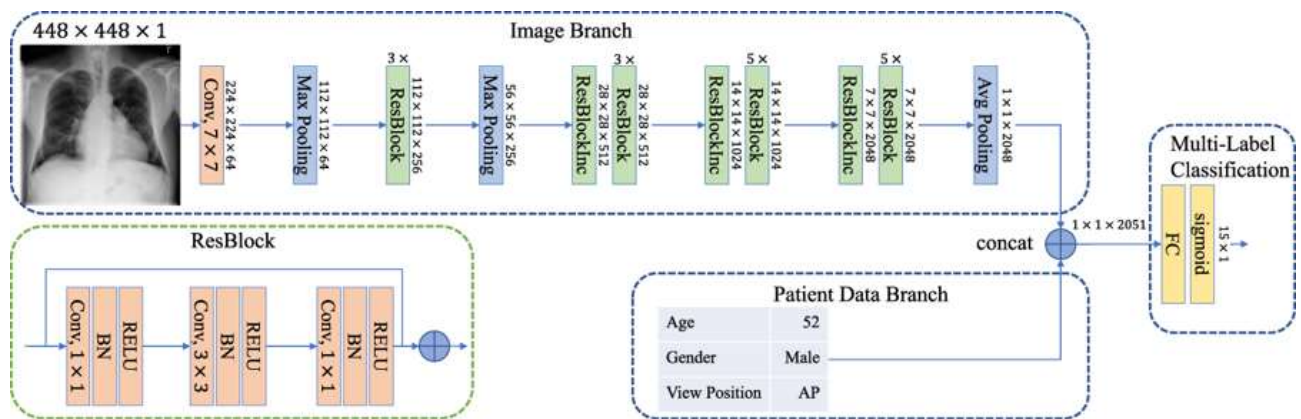


Abbildung 49: CNN unter Einbezug von Text-Merkmalen (Quelle: Baltrushat et al. 2019, S.3)

Neben der vergleichbaren Leistung zu Guendel et al. zeigt die Arbeit von Baltrushat et al. andere Anwendungsmöglichkeiten dieses Modells – beispielsweise als Merkmalsextrator (*engl. feature extractor*)<sup>clxxxviii</sup> zur Vorhersage des Patientenalters (*Regression*) oder die Unterscheidung der Ansichtsposition im Sinne einer binären Klassifikation zwischen *AP*- und *PA*-Aufnahmen (Vgl. Abbildung 49, *View Position*).

## 4.4 Produktive ML-Einsatzfelder

Die Arbeit von Lee et al.<sup>clxxxix</sup> bietet einen Überblick über die Geschichte, Entwicklung und Anwendung von ML-Modellen im Bereich der medizinischen Bildgebung. Sie betonen die Abhängigkeit von einem guten Datensatz, dessen Datenpunkte aus verschiedenen Aufnahmegeräten stammen können. Lokale Unterschiede in der Bildgebung und Häufungen von Krankheitsbildern, stellen eine Herausforderung für die Zielverteilung des Datensatzes dar. Die Klassenbezeichnungen sollten verlässlich sein. Die NLP-Methoden zu ihrer Extraktion sind ein weiteres Optimierungspotenzial, das in dieser Arbeit nicht tiefer untersucht wird.<sup>66</sup> Andere Autoren kritisieren ebenfalls den Mangel eines ausreichend großen Datensatzes und die Qualität der Klassenbezeichnungen.<sup>cxc</sup>

Darüber hinaus stellen sich legale und ethische Fragen, während die Performanz von ML-Modellen von dem Bedarf wachsender Datenqualität abhängig ist. Im Idealfall bietet das ML-Modell eine Unterstützung des Workflows (genauere Diagnosen). Obwohl Anwendungsbeispiele für NLP und Spracherkennung sowie Berichterstellung genannt werden, finden sich keine Anwendungsbeispiele von maschinellem Sehen. Dessen abstrakte Beschreibung wird als Kollaborationsmittel betrachtet, um zwischen ablenkenden, wiederholenden und langweiligen Aufgaben zu vermitteln, anstatt Radiologen zu ersetzen.<sup>cxi</sup>

---

<sup>66</sup> (Vgl. <https://nihcc.app.box.com/v/ChestXray-NIHCC/file/220660789610>, S.1, Abgerufen am 11.6.19)

## 4.5 Modularisierung der ML-Pipeline

Dieser Abschnitt untersucht Einsparpotenziale und Voraussetzungen innerhalb der ML-Pipeline, um den Prozess der Wissensgenerierung (Musterextraktion mit Deep-Learning) effizienter und weniger iterativ zu gestalten und zu modularisieren. Transferlernen ist eine beliebte Methode der Computer-Vision Domäne. Sie überträgt die trainierten Gewichtungen (oder nur den Aufbau einer unveränderten Modellarchitektur) auf einen anderen Datensatz. Für die Auswahl eines geeigneten Modells stehen grundsätzlich drei Ansätze zur Verfügung (4.3 CXR-Implementierung).<sup>cxcii</sup>

1. Handgefertigt (*engl. handcrafted*)
2. Transferlernen
  - a. Von Anfang an (*engl. from scratch*)
  - b. Vortrainiert (*engl. pre-trained*)

### 4.5.1 Merkmalsextraktion auf andere Datensätze (Transferlernen)

Um das Transferlernen zu initialisieren, wird während des Modellaufbaus als Erstes eine neue Modellstruktur importiert und als Basismodell (`base_model`) festgelegt. Der Ausdruck `include_top` legt fest, ob die vollständig verknüpften Schichten am Ende des Basisnetzwerks übernommen werden, wobei entfernte FC-Schichten ersetzt werden müssen (Vgl. Abbildung 50, *hellblauer Kasten*). Eine FC-Schicht und die Ausgangsschicht werden der Ausgangsschicht des Ausgangsmodells hinten angehängt (Vgl. Abbildung 50, *roter Kasten unten*; Vgl. 3.2.3 Modellauswahl, Zeile 11-20: `base_model.output`). Danach wird die Initialisierungs-Strategie der Gewichtungen bestimmt; entweder von Anfang an, das bedeutet eine zufällige Initialisierung (`weights=None`), oder vortrainiert aus dem ImageNet-Datensatz (`weights='imagenet'`).<sup>cxciii</sup>

Der Lernprozess des Transferlernens ist mehrstufig: Zunächst werden die Gewichtungen eingefroren (`layer.trainable = False`) und vortrainiert (*engl. pre train*). Im Anschluss beginnt die Feinanpassung (*engl. fine tuning*). Die Schichten des Modells werden in mehreren Phasen aufgetaut, da das Modell sonst alles Gelernte vergisst, weil die neuen Schichten zufällig initialisiert sind. Das Einfrieren reduziert die trainierbaren Parameter auf einen Bruchteil der Gesamtmenge, während es nach dem Auftauen mehr trainierbare Parameter gibt.

Das Vortraining friert die Gewichte aller übernommenen Schichten ein. Es werden zunächst ausschließlich die neu hinzugefügten Schichten trainiert (Vgl. 3.2.3 Modellauswahl, Zeile 24f).

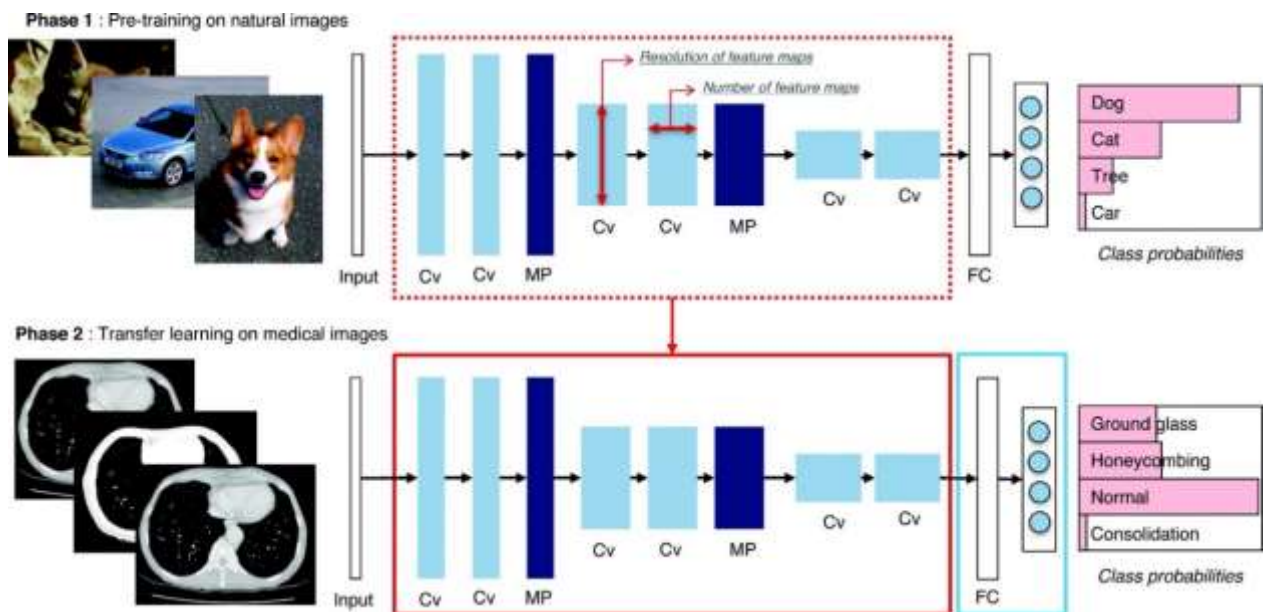


Abbildung 50: Vortraining und Transferlernen illustriert (Quelle: Chartrand et al. 2017, S.2127)

Nach dem Vortraining hebt die Feinanpassung, ausgehend von der Ausgangsschicht, die eingefrorenen Schichten (und enthaltene Gewichtungen) wieder auf (`layer.trainable = True`) und es werden mehr Schichten trainiert. Gegenüber dem Vortraining werden tiefere Schichten in den Lernprozess eingebunden (Vgl. 3.2.8 Tiefere Feinanpassung, Zeile 14f). Bei der Feinanpassung gibt es mehr trainierbare Parameter; die Beispiel-Implementierung wächst um 7 Millionen Parameter von 25.706.511 auf 32.785.935 Parameter der insgesamt 138.357.544 Parameter des Referenzmodells VGG16.

Bei Gegenüberstellung der Eingabedimension (Merkmalsraum) von ImageNet und dem NIH-Datensatz fällt auf, dass die beschränkte Eingabe (einer Implementierung wie InceptionV3) nur einen Teil der Bildpunkte eines Röntgenbildes abbildet:

$$(21) \quad \frac{268.203}{1.048.576} = 0,26 = 26 \%$$

Ein Datenpunkt des ImageNet-Datensatzes ist ein Farbbild (`input_shape = 299x299x3`) und besitzt mit 268.203 Pixeln deutlich weniger Merkmale gegenüber einem 1.048.576 Pixel großen Röntgenbild des NIH-Datensatzes (Vgl. Tabelle 8).

Außerdem kann InceptionV3 mit Graustufen nichts anfangen.<sup>67</sup> Die Bilder müssen stets farbig sein. Die Möglichkeit, mit Graustufen anstatt farbigen Bildern zu trainieren, stellt ein weiteres Optimierungspotenzial dar, das von Baltrushat et al. aufgegriffen wird (Vgl. 4.3 CXR-

<sup>67</sup> (Vgl. <https://stackoverflow.com/questions/45939561/is-it-possible-to-use-grayscale-images-to-existing-model>, Abgerufen am 11.6.19)

Implementierung).<sup>cxciv</sup> Die begrenzte Aufnahmefähigkeit verschiedener ResNet-Tiefen wird vor dem Hintergrund des limitierten Eingabeformats einer Modellneuentwicklung gegenübergestellt, welche mit einem größeren Eingabeformat (`input_shape = 448x448x1`) umgehen kann.<sup>cxcv</sup>

#### 4.5.2 Mehrere Stufen der ML-Pipeline als Aufgabenverteilung

Da der Prozess zur Erstellung eines fertigen Modells sehr zeitintensiv ist – abhängig von Modellgröße und Größe des Datensatzes – kann das Training bis zu einer hinreichenden Konvergenz je nach Hardware-Ausstattung mehrere Tage oder gar Wochen dauern (Vgl. Tabelle 6). Bevor ein Modell überhaupt produktiv eingesetzt werden kann, muss es genügend Wissen aus einem Datensatz gesammelt haben (trainiert sein) und dem Anwender möglichst nützliche Antworten liefern. Die notwendigen Schritte sind in Kapitel 3.1 dargestellt (Vgl. 3.1 ML-Pipeline): Die Pipeline beginnt bei der Datenerhebung und endet mit der Bewertung des Modells; dies wurde in Abschnitt 3.2 praktisch umgesetzt (Vgl. 3.2 CNN-Implementierung). Dabei fällt auf, dass dieser Prozess sehr iterativ ist. Das bedeutet, dass der Prozess je nach Bewertungsergebnis teilweise mehrfach komplett durchlaufen werden muss und dabei Ressourcen verschwendet werden.

Im schlimmsten Fall lautet die Antwort für ein nicht konvergierendes Modell, dass es nicht genügend Daten gibt, um die Zielverteilung abzubilden<sup>cxcvi</sup>. Darum sollte die ML-Pipeline in möglichst kleine Phasen geteilt werden, um Hypothesen schneller überprüfen zu können und dabei möglichst wenige Schritte zurückgehen zu müssen und möglichst viele einzusparen.

Ein Mittel zur Modularisierung der ML-Pipeline ist die Exportierung von Gewichtungen nach dem Abschluss einer Trainingsphase, beispielsweise nach dem Vortraining (Vgl. 3.2.11 Gewichte exportieren, Zeile 3 ff.). Diese Gewichtungen können zu einem späteren Zeitpunkt wieder importiert werden, um das Training fortzusetzen oder ein schiefgelaufenes Training zurückzusetzen und (mit anderen Parametern für die Feinanpassung) erneut zu starten. Dies hat den Vorteil, dass die ML-Pipeline nicht komplett von vorn durchlaufen werden muss und stellt ein großes Einsparpotenzial dar. Eine wichtige Voraussetzung zur Wiederaufnahme des Trainings ist die Wiederherstellung der Datenaufteilung (im Sinne einer Dateiliste nach *official split* oder mit dem `seed`-Parameter), sodass nach Import des Modells mit der gleichen Aufteilung weiter trainiert werden kann.<sup>68</sup> Außerdem können Modelle, die auf der gleichen Aufteilung basieren, besser verglichen werden.<sup>cxcvii</sup>

---

<sup>68</sup> (Vgl. <https://keras.io/preprocessing/image/>, Abgerufen am 11.6.19)

Die Wiederaufnahme eines Trainings, welches auf einer anderen Aufteilung basiert, ist schwer zu beurteilen. Der Grund dafür ist, dass Datenpunkte des ursprünglichen Trainingsdatensatzes (bereits gelernte Muster) nun im Validierungsdatensatz sein können und das Bewertungsergebnis verfälschen.

## Schlusswort

Diese Arbeit bietet einen praktischen Überblick über CNN-Referenzmodelle und deren Implementierung durch Transferlernen und Neuimplementierung. Vor dem Hintergrund des beschränkten Eingabeformats (`input_shape`) sind jedoch Kompromisse einzugehen, die mit der Erkenntnis über Modellstrukturen (die mehr Attribute berücksichtigen) und Datensatz-Eigenschaften (die unausgewogen sind und deren Format größer als im Referenzmodell ist) verbessert werden können, damit das perfekte Modell gebaut werden kann.

Die Diagnose maschinellen Lernens findet innerhalb der ML-Pipeline für CNN-Referenzmodelle und ihrer CNN-Bausteinen als Vorlage sowie in der Analyse des Datensatzes und in Trainings- und Klassifikationsergebnissen (zur Bewertung der Aufgabe) statt.

Um die Ergebnisse eines trainierten Modells bewerten zu können, ist Domänenwissen notwendig – was im Falle des gegebenen Radiologie-Datensatzes nicht gegeben ist. Als Bewertungsmethoden können lediglich die Metriken zur KKR u.a. (während des Trainings) sowie die Wahrheitsmatrix (nach dem Training) herangezogen werden.

Zu kritisieren am Ergebnis dieser Arbeit ist, dass die produktiven Einsatzfelder eines ML-Klassifikators nur aus Sicht eines Radiologen beschrieben werden. In einem Krankenhaus könnte ein CXR-Bildklassifikator z. B. die Diagnosen eines Radiologen priorisieren, damit dieser den Patienten zunächst auf die kritischsten Krankheitsbilder untersucht. Es sind vor allem falsch-negative Vorhersagen zu vermeiden, damit keine kranken Patienten nach Hause geschickt werden. Wie Radiologische Diagnosen eines Klassifikators verbessert werden können, bleibt weiterhin offen – dafür bietet diese Arbeit eine Diagnostik von CNN-Implementierungen und einen Bauplan um besser diagnostizierbare Modelle zu bauen.

Um ML produktiv einsetzen zu können, bedarf es zunächst einem Verständnis der Modell-Entscheidungen und ihrer Visualisierung, damit das Ansehen von ML-Entscheidungen (am Beispiel der Klassifikation) gesteigert wird.

Die meisten Modelle die trainiert wurden sind überhaupt nicht konvergiert (underfit) – was auf mangelndes Domänenwissen für unausgewogene Datensätze zurückzuführen ist. Die nicht-konvergierenden Modelle sind unter dem GitHub-Link in den Verzeichnissen *InceptionV3*, *VGG16* und *CNN from scratch* zu finden.



Diejenigen Modelle, die konvergiert sind, besitzen die Eigenschaft der Überanpassung (overfit) und können daher nicht als nützliche Modelle betrachtet werden. Das gesammelte Wissen über ihr mangelhaft großes Eingabeformat und die Möglichkeiten der Image-Augmentation zur Balancierung (*Resampling*) des Datensatzes jedoch schon. Diese Erkenntnisse konnten nicht vertieft untersucht werden, obwohl sie ein großes Verbesserungspotenzial der Konvergenz darstellen.

## Literaturverzeichnis

- Baltruschat, I., Nickisch, H., Grass, M., & Saalbach, A. (2019). Comparison of Deep Learning Approaches for Multi-Label Chest X-Ray Classification. *Scientific Reports*(9), S. 10. Abgerufen am 11. Juni 2019 von <https://www.nature.com/articles/s41598-019-42294-8.pdf>
- Bengio, Y., Courville, A., & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*(35), S. 1798 - 1828. doi:10.1109/TPAMI.2013.50
- Caruana, R., & Niculescu-Mizil, A. (2005). *An Empirical Comparison of Supervised Learning Algorithms Using Different Performance Metrics*. Abgerufen am 11. Juni 2019 von <http://www.cs.cornell.edu/~alexn/papers/comparison.tr.pdf>
- Chartrand, G., Cheng, P., Voront, E., & Tang, A. (2017). Deep Learning: A Primer for Radiologists. *RadioGraphics*, 7(37). Abgerufen am 11. 2019 Juni von <https://pubs.rsna.org/doi/pdf/10.1148/rg.2017170077>
- Chawla, N. V., Bowyer, K. W., Hall, L. C., & Kegelmeyer, W. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*(16), S. 321-357. Abgerufen am 11. Juni 2019 von <http://www.csee.usf.edu/~lohall/papers/smote.pdf>
- Chollet, F. (2017). Xception: Deep Learning with Depthwise Separable Convolutions. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (S. 1251-1259). Abgerufen am 11. Juni 2019 von [http://openaccess.thecvf.com/content\\_cvpr\\_2017/papers/Chollet\\_Xception\\_Deep\\_Learning\\_CVPR\\_2017\\_paper.pdf](http://openaccess.thecvf.com/content_cvpr_2017/papers/Chollet_Xception_Deep_Learning_CVPR_2017_paper.pdf)
- Dumoulin, V., & Visin, F. (2016). *A guide to convolution arithmetic for deep learning*. Abgerufen am 18. Mai 2019 von <https://arxiv.org/pdf/1603.07285>
- Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, S.37-54. Abgerufen am 19. April 2019 von [https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&cad=rja&uact=8&ved=2ahUKEwiCzezloODiAhXL\\_KQKHYYhUBzUQFjAHegQIAxAC&url=https%3A%2F%2Fwww.aaai.org%2FPapers%2FKDD%2F1996%2FKDD96-014.pdf&usg=AOvVaw0RmxCEkOXM9afVEgY\\_m5Y2](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=8&cad=rja&uact=8&ved=2ahUKEwiCzezloODiAhXL_KQKHYYhUBzUQFjAHegQIAxAC&url=https%3A%2F%2Fwww.aaai.org%2FPapers%2FKDD%2F1996%2FKDD96-014.pdf&usg=AOvVaw0RmxCEkOXM9afVEgY_m5Y2)
- Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow. Concepts, Tools and Techniques to Build Intelligent Systems*. Sebastopol: O'Reilly Media.

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Von [www.deeplearningbook.org](http://www.deeplearningbook.org) abgerufen
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep Residual Learning for Image Recognition*. Abgerufen am 29. Dezember 2018 von <https://arxiv.org/abs/1512.03385>
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Identity Mappings in Deep Residual Networks*. Microsoft Research. Abgerufen am 11. Juni 2019 von <https://arxiv.org/abs/1603.05027>
- Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., . . . Adam, H. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Abgerufen am 11. Juni 2019 von <https://arxiv.org/pdf/1704.04861.pdf>
- Huang, G., Liu, Z., & van der Maaten, L. (2016). *Densely Connected Convolutional Networks*. Abgerufen am 11. Juni 2019 von <https://arxiv.org/abs/1608.06993.pdf>
- Hubel, D., & Wiesel, T. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*(160), S. 106-154. Abgerufen am 11. Juni 2019 von <https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1962.sp006837>
- Iandola, F., Han, S., Moskewicz, M., Ashraf#, K., Dally, W., & Keutzer, K. (2016). *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. Abgerufen am 11. Juni 2019 von <https://arxiv.org/pdf/1602.07360>
- Ioffe, S., & Szegedy, C. (2015). BatchNormalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Proceedings of the 32nd International Conference on Machine Learning*. Lille, France. Abgerufen am 31. Mai 2019 von <http://proceedings.mlr.press/v37/ioffe15.pdf>
- Kauderer-Abrams, E. (6. Juni 2017). *Quantifying Translation-Invariance in Convolutional Neural Networks*. Abgerufen am 2. Juni 2019 von <https://arxiv.org/pdf/1801.01450>
- King, G., & Zeng, L. (2001). Logistic Regression in Rare Events Data. *Political Analysis*, 9, 137-163. Abgerufen am 11. Juni 2019 von <https://gking.harvard.edu/files/gking/files/0s.pdf>
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems* 25, S. 9. Abgerufen am 18. Januar 2019 von <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE, November 1998*, 46. Abgerufen am 17. Januar 2019 von [https://www.researchgate.net/publication/2985446\\_Gradient-Based\\_Learning\\_Applied\\_to\\_Document\\_Recognition](https://www.researchgate.net/publication/2985446_Gradient-Based_Learning_Applied_to_Document_Recognition)

- Lee, J.-G., Jun, S., Cho, Y.-W., Lee, H., Kim, G., Seo, J., & Kim, N. (Jul-Aug 2017). Deep Learning in Medical Imaging: General Overview. *Korean Journal of Radiology*, S. 570-584. Abgerufen am 30. Dezember 2018 von <https://synapse.koreamed.org/DOIx.php?id=10.3348/kjr.2017.18.4.570>
- Lemaître, G., Nogueira, F., & Aridas, C. (2017). Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*(18), 1-5. Abgerufen am 11. Juni 2019
- Lin, M., Chen, Q., & Yan, S. (2014). *Network In Network*. Abgerufen am 29. März 2019 von <https://arxiv.org/pdf/1312.4400.pdf>
- Litjens, G., Kooi, T., Bejnordi, B., Setio, A., Ciompi, F., Ghafoorian, M., . . . Sanchez, C. (2017). A Survey on Deep Learning in Medical Image Analysis. Abgerufen am 11. Juni 2019 von <https://arxiv.org/pdf/1702.05747.pdf>
- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, S. 115-133. Abgerufen am 11. Juni 2019 von [https://www.aemea.org/math/McCulloch\\_Pitts\\_1943.pdf](https://www.aemea.org/math/McCulloch_Pitts_1943.pdf)
- Minh, V., Kavukcuoglu, K., Silver, D., Graves, A., & Antonoglou, I. (2013). *Playing Atari with Deep Reinforcement Learning*. Lake Tahoe, USA. Abgerufen am 11. Juni 2019 von <https://arxiv.org/abs/1312.5602>
- Müller, A., & Guido, S. (2017). *Einführung in Machine Learning mit Python*. Heidelberg: O'Reilly.
- Raschka, S., & Mirjalili, V. (2018). *Machine Learning mit Python und Scikit-learn und TensorFlow: Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analysis*. Frechen: mitp Verlags GmbH & Co. KG, Frechen.
- Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Cornell Aeronautical Laboratory, Psychological Review*, v65, No. 6, S. 386–408. Abgerufen am 29. April 2019 von <https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf>
- Russakovsky, O., Deng, J., Kraus, J., Krause, J., Satheesh, S., Ma, S., . . . Li, F.-F. (2014). *ImageNet Large Scale Visual Recognition Challenge*. Abgerufen am 1. Juni 2019 von <https://arxiv.org/pdf/1409.0575>
- Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. Abgerufen am 11. Juni 2019 von <https://arxiv.org/pdf/1801.04381>

- Simonyan, K., & Zisserman, A. (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Abgerufen am 3. März 2019 von <https://arxiv.org/https://arxiv.org/abs/1409.1556>
- Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. (2016). *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. Abgerufen am 11. Juni 2019 von <https://arxiv.org/pdf/1602.07261>
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Andrew, R. (2014). *Going deeper with convolutuions*. Abgerufen am 19. Januar 2019 von <https://arxiv.org/pdf/1409.4842.pdf>
- Tajbakhsh, N., Shin, J. Y., Gurundu, S. R., Hurst, T., Kendall, C. B., Gotway, M. B., & Liang, J. (2017). *Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?* Abgerufen am 2. April 2019 von <https://arxiv.org/pdf/1706.00712>
- Usama, F., Gregory, P.-S., & Padhraic, S. (15. September 1996). From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 3(17), S. 37-57.  
doi:<https://doi.org/10.1609/aimagv17i3.1230>
- Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and Tell: A Neural Image Caption Generator. *Proceedings of the IEEE conference on computer vision and pattern recognition*, (S. 3156-3164). Abgerufen am 11. Juni 2019 von [https://www.cv-foundation.org/openaccess/content\\_cvpr\\_2015/papers/Vinyals\\_Show\\_and\\_Tell\\_2015\\_CVPR\\_paper.pdf](https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Vinyals_Show_and_Tell_2015_CVPR_paper.pdf)
- YouTube Playlist: Stanford University CS231n (2017)* (2017). [Kinofilm]. Abgerufen am 2. Januar 2019 von <https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv>
- Zeiler, M., & Fergus, R. (2013). Visualizing and Understanding Convolutional Neural Networks. *European Conference on Computer Vision*, (S. 818-833). Abgerufen am 1. Juni 2019 von <https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>
- Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. (2017). *Learning Transferable Architectures for Scalable Image Recognition*. Von <https://arxiv.org/pdf/1707.07012> abgerufen



## Anhang

$$(22) \quad \text{Anteil an Gesamtdaten} = \frac{\text{Datenpunkte pro Klasse}}{\text{Anzahl Datenpunkte}}$$

1. Anzahl Datenpunkte	42.862 Bilder		Anteil an Gesamtdaten
2. Anzahl Klassen	len(labels) = 15		
3. Anzahl der Datenpunkte pro Klasse	Artelectasis	2015	$\frac{2015}{42862} = 4,7\%$
	Cardiomegaly	464	$\frac{464}{42862} = 1,08\%$
	Consolidation	633	$\frac{633}{42862} = 1,48\%$
	Edema	310	$\frac{310}{42862} = 0,72\%$
	Effusion	1804	$\frac{1804}{42862} = 4,21\%$
	Emphysema	461	$\frac{461}{42862} = 1,076\%$
	Fibrosis	366	$\frac{366}{42862} = 0,854\%$
	Hernia	53	$\frac{53}{42862} = 0,124\%$
	Infiltration	4640	$\frac{4649}{42862} = 10,83\%$
	Mass	1.043	$\frac{1043}{42862} = 2,43\%$
	No_Finding	28.000	$\frac{28000}{42862} = 65,33\%$
	Nodule	1.273	$\frac{1273}{42862} = 2,97\%$
	Pleural_Thickening	525	$\frac{525}{42862} = 1,22\%$
Pneumonia	161	$\frac{161}{42862} = 0,376\%$	
Pneumothorax	1.099	$\frac{1099}{42862} = 2,56\%$	
4. Anzahl der Merkmale je Datenpunkt	1024 * 1024 * 1 = 1024 <sup>2</sup> = 1.048.576		

Tabelle 8: Metriken des sortierten NIH-Datensatzes (Eigene Darstellung)

<b>Spezifikation</b>	<b>Kerne</b>	<b>Takt (MHz)</b>	<b>Speicher (-Bandbreite)</b>	<b>Operationen (TFLOPS)</b>	<b>Energie (TDP)</b>
Intel i9-9900K <sup>69</sup>	8	5000	16 MB (41,6 GB/s)	$\approx 0,5$ <sup>70</sup>	95W
NVIDIA RTX 2080 TI <sup>71</sup>	4352	1350	11 GB (616 GB/s)	13,45 <sup>72</sup>	250W
Google Cloud TPU v2 <sup>73</sup>	65536 MXU	700	64 GB (12,5 GB/s)	180 <sup>74</sup>	40W

Tabelle 9: Hardware im Detailvergleich (Eigene Darstellung)<sup>75</sup>


---

<sup>69</sup> (Vgl. <https://www.intel.com/content/www/us/en/products/processors/core/i9-processors/i9-9900k.html>, Abgerufen am 11.6.19)

<sup>70</sup> (Vgl. <https://weborus.com/intel-core-i9-9900k-performance-gflops/>, Abgerufen am 11.6.19)

<sup>71</sup> (Vgl. <https://www.nvidia.com/de-de/geforce/graphics-cards/rtx-2080-ti/#specsmodal>, Abgerufen am 11.6.19)

<sup>72</sup> (Vgl. <https://www.gamestar.de/artikel/steigende-preise-bei-nvidia-rtx-2080-ti2070-bis-gtx-580570-im-vergleich,3333987,seite3.html>, Abgerufen am 11.6.19)

<sup>73</sup> (Vgl. <https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu>, Abgerufen am 11.6.19)

<sup>74</sup> (Vgl. <https://cloud.google.com/tpu/>, Abgerufen am 11.6.19)

<sup>75</sup> (Quelle: [http://cs231n.stanford.edu/slides/2017/cs231n\\_2017\\_lecture15.pdf](http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf), S.18, Abgerufen am 11.6.19)



## Endnoten

- 
- <sup>i</sup> (Vgl. Goodfellow et al. 2016, S.102)
- <sup>ii</sup> (Vgl. Wirth/Hipp, 2000, S.2)
- <sup>iii</sup> (Vgl. Goodfellow et al. 2016, S.12-26)
- <sup>iv</sup> (Bengio et al. 2013)
- <sup>v</sup> (Vgl. Goodfellow et al. 2016, S.96)
- <sup>vi</sup> (Vgl. Raschka/Mirjalili, 2016, S.410-418)
- <sup>vii</sup> (Vgl. Goodfellow et al. 2016, S.200)
- <sup>viii</sup> (Vgl. Raschka/Mirjalili, 2016, S.390)
- <sup>ix</sup> (Vgl. Goodfellow et al. 2016, S.200)
- <sup>x</sup> (Vgl. Goodfellow et al. 2016, S.104)
- <sup>xi</sup> (Vgl. Fayyad, 1996, S.40)
- <sup>xii</sup> (Vgl. Raschka/Mirjalili, 2016, S.26)
- <sup>xiii</sup> (Vgl. Goodfellow et al. 2016, S.98)
- <sup>xiv</sup> (Vgl. Müller/Guido, 2017, S.123)
- <sup>xv</sup> (Vgl. Goodfellow et al. 2016, S.104)
- <sup>xvi</sup> (Vgl. Minh et al. 2013)
- <sup>xvii</sup> (Vinyals et al. 2015)
- <sup>xviii</sup> (Vgl. Goodfellow et al. 2016, S.97-101)
- <sup>xix</sup> (Vgl. Goodfellow et al. 2016, S.98)
- <sup>xx</sup> (Vgl. Goodfellow et al. 2016, S.99)
- <sup>xxi</sup> (Vgl. Goodfellow et al. 2016, S.103)
- <sup>xxii</sup> (Vgl. Müller/Guido, 2017, S.27)
- <sup>xxiii</sup> (Vgl. Goodfellow et al. 2016, S.97)
- <sup>xxiv</sup> (Vgl. Goodfellow et al. 2016, S.99)
- <sup>xxv</sup> (Vgl. Müller/Guido, 2017, S.28)
- <sup>xxvi</sup> (Vgl. Raschka/Mirjalili, 2016, S.51)
- <sup>xxvii</sup> (Vgl. Goodfellow et al. 2016, S.14)
- <sup>xxviii</sup> (Hubel/Wiesel, 1962)
- <sup>xxix</sup> (Raschka/Mirjalili, 2016, S.56-71)
- <sup>xxx</sup> (McCulloch/Pitts, 1943)
- <sup>xxxi</sup> (Vgl. Raschka/Mirjalili, 2016, S.42)
- <sup>xxxii</sup> (Rosenblatt, 1957)
- <sup>xxxiii</sup> (Vgl. Raschka/Mirjalili, 2016, S.43)
- <sup>xxxiv</sup> (Vgl. Raschka/Mirjalili, 2016, S.386)

- 
- xxxv (Vgl. Raschka/Mirjalili, 2016, S.418 ff.)
- xxxvi (Vgl. Goodfellow et al. 2016, S.108)
- xxxvii (Vgl. Raschka/Mirjalili, 2016, S.56f).
- xxxviii (Vgl. Raschka/Mirjalili, 2016, S.44)
- xxxix (Vgl. Raschka/Mirjalili, 2016, S.45)
- xl (Raschka/Mirjalili, 2016, S.59-63)
- xli (Vgl. Raschka/Mirjalili, 2016, S.56f)
- xlII (Vgl. Raschka/Mirjalili, 2016, S.59)
- xlIII (Vgl. Raschka/Mirjalili, 2016, S.57 ff.)
- xliv (Vgl. Raschka/Mirjalili, 2016, S.56-59)
- xlV (Vgl. Raschka/Mirjalili, 2016, S.56)
- xlvi (Vgl. Raschka/Mirjalili, 2016, S.57 ff.)
- xlVII (Raschka/Mirjalili, 2016, S.81-84)
- xlVIII (Vgl. Goodfellow et al. 2016, S.167)
- xlIX (Vgl. Goodfellow et al. 2016, S.173)
- I (Vgl. Müller/Guido, 2017, S.101)
- II (Vgl. Raschka/Mirjalili, 2016, S.93)
- III (Vgl. Müller/Guido, 2017, S.101f)
- IIII (Vgl. Müller/Guido, 2017, S.100f)
- IV (Vgl. Müller/Guido, 2017, S.101)
- IV (Vgl. LeCun et al. 1998, S.8)
- IVI (Russakovsky et al. 2014)
- IVII (Vgl. Szegedy et al. 2014)
- IVIII (Vgl. Goodfellow et al. 2016, S.339)
- lix (Vgl. Raschka/Mirjalili, 2016, S.385)
- lx (Vgl. Raschka/Mirjalili, 2016, S.64f)
- lxi (Vgl. Raschka/Mirjalili, 2016, S.66-71)
- lxii (Vgl. Raschka/Mirjalili, 2016, S.386)
- lxiii (Vgl. Raschka/Mirjalili, 2016, S.507f)
- lxiv (Vgl. Goodfellow et al. 2016, S.165)
- lxv (Vgl. Raschka/Mirjalili, 2016, S.490)
- lxvi (Vgl. Hubel/Wiesel, 1962, S.109 ff.)
- lxvII (Vgl. Goodfellow et al. 2016, S.336)
- lxvIII (Quelle: Goodfellow et al. 2016, S.336)
- lxix (Vgl. Goodfellow et al. 2016, S.335 ff.)
- lxx (Vgl. Dumoulin, 2016, S.6)
- lxxi (Vgl. Goodfellow et al. 2016, S.329f)
- lxxii (Vgl. Goodfellow et al. 2016, S.330f) und (Vgl. Goodfellow et al. 2016, S.254 ff.)
- lxxiii (Vgl. Goodfellow et al. 2016, S.331 ff.) und (Vgl. Goodfellow et al. 2016, S.253f)
- lxxiv (Vgl. Goodfellow et al. 2016, S.334f)
- lxxv (Vgl. Dumoulin, 2016, S.12)
- lxxvi (Vgl. Raschka/Mirjalili, 2016, S.494f)

- 
- lxxvii (Vgl. Raschka/Mirjalili, 2016, S.497)
- lxxviii (Vgl. Raschka/Mirjalili, 2016, S.444-451)
- lxxix (Vgl. Goodfellow et al. 2016, S.221)
- lxxx (Vgl. Goodfellow et al. 2016, S.14)
- lxxxi (Quelle: Raschka/Mirjalili, 2016, S.450)
- lxxxii (Vgl. Raschka/Mirjalili, 2016, S.500f)
- lxxxiii (Vgl. Müller/Guido, 2017, S.101)
- lxxxiv (Vgl. Goodfellow et al. 2016, S.354f)
- lxxxv (Vgl. Müller/Guido, 2017, S.49)
- lxxxvi (Goodfellow et al. 2016, S.234-237)
- lxxxvii (Goodfellow et al. 2016, S.231-234)
- lxxxviii (Goodfellow et al. 2016, S.258-268)
- lxxxix (Goodfellow et al. 2016, S.453-458)
- xc (Ioffe/Szegedy, 2015)
- xcI (Vgl. Müller/Guido, 2017, S.65)
- xcii (Vgl. Raschka/Mirjalili, 2016, S.141)
- xciii (Vgl. Raschka/Mirjalili, 2016, S.94 ff.)
- xciv (Vgl. Raschka/Mirjalili, 2016, S.141)
- xcv (Vgl. Raschka/Mirjalili, 2016, S.141 ff.)
- xcvi (Vgl. Raschka/Mirjalili, 2016, S.504 ff.)
- xcvii (Vgl. Goodfellow et al. 2016, S.240f)
- xcviii (Vgl. Raschka/Mirjalili, 2016, S.385)
- xcix (Vgl. Goodfellow et al. 2016, S.177)
- c (Vgl. Raschka/Mirjalili, 2016, S.442)
- ci (Vgl. Fayyad et al. 1996, S.37 ff.)
- cii (Vgl. Fayyad et al. 1996, S.41)
- ciii (Vgl. Fayyad et al. 1996, S.39)
- civ (Vgl. Fayyad et al. 1996, S.41)
- cv (Vgl. Fayyad et al. 1996)
- cvi (Vgl. Geron, 2017, S.84) und (Vgl. Raschka/Mirjalili, 2016, S.93-96)
- cvi (Vgl. Raschka/Mirjalili, 2016, S.322-326)
- cviii (Vgl. Dumoulin, 2016, S.6)
- cix (Vgl. CS231n, 2017, Lecture 2)
- cx (Vgl. Müller/Guido, 2017, S.112)
- cxI (Vgl. Raschka/Mirjalili, 2016, S.205f)
- cxii (Vgl. Raschka/Mirjalili, 2016, S.205f)
- cxiii (Vgl. Müller/Guido, 2017, S.238)
- cxiv (Vgl. Raschka/Mirjalili, 2016, S.208f)
- cxv (Vgl. Raschka/Mirjalili, 2016, S.206-210)
- cxvi (Vgl. Raschka/Mirjalili, 2016, S.208)
- cxvii (Vgl. Raschka/Mirjalili, 2016, S.207)
- cxviii (Vgl. Müller/Guido, 2017, S.238)

- 
- cxix (Vgl. Müller/Guido, 2017, S.108), (Vgl. Müller/Guido, 2017, S.121) und (Vgl. Raschka/Mirjalili, 2016, S.64)
- cxx (Müller/Guido, 2017, S.196-201)
- cxxi (Vgl. Müller/Guido, 2017, S.159f)
- cxxii (Vgl. Caruana/Niculescu-Mizil, 2005, S.1)
- cxxiii (Goodfellow et al. 2016, S.246-252)
- cxxiv (Vgl. Raschka/Mirjalili, 2016, S.216-220)
- cxxv (Vgl. Raschka/Mirjalili, 2016, S.220)
- cxxvi (Vgl. Raschka/Mirjalili, 2016, S.211-216)
- cxxvii (Vgl. Raschka/Mirjalili, 2016, S.222)
- cxxviii (Vgl. Raschka/Mirjalili, 2016, S.220 ff.)
- cxxix (Vgl. Raschka/Mirjalili, 2016, S.220 ff.)
- cxxx (Raschka/Mirjalili, 2016, S.224-227)
- cxixi (Vgl. Goodfellow et al. 2016, S.108-114)
- cxixii (Vgl. Müller/Guido, 2017, S.235)
- cxixiii (Vgl. Raschka/Mirjalili, 2016, S.211-214)
- cxixiv (Vgl. Müller/Guido, 2017, S.30)
- cxixv (Vgl. Raschka/Mirjalili, 2016, S.220f)
- cxixvi (Vgl. Raschka/Mirjalili, 2016, S.223)
- cxixvii (Vgl. Raschka/Mirjalili, 2016, S.223)
- cxixviii (Vgl. Raschka/Mirjalili, 2016, S.223f)
- cxixix (Vgl. Litjens et al. 2018, S.24f)
- cxli (LeCun et al. 1998, S.7)
- cxlii (Vgl. Krizhevsky et al. 2012, S.5f)
- cxliii (Krizhevsky et al. 2012, S.5)
- cxliiii (Huang et al. 2016, S.4)
- cxliv (Szegedy et al. 2014, S.5 ff.)
- cxlv (Szegedy et al. 2016) [InceptionResNetV2]
- cxlvi (LeCun et al. 1998, S.7)
- cxlvii (LeCun et al. 1998, S.9), während  $C_{120}^{1x1}$  vollständig mit der vorherigen Schicht verbunden ist
- cxlviii (Howard et al. 2017)
- cxlix (Sandler et al. 2018)
- cl (Zoph et al. 2017)
- cli (He et al. 2015, S.4)
- clii  $(C_{64}^{7;2} P_M^{3;2})_{64}^{56x56} - 3(C_{64}^2)4(C_{128}^3)6(C_{256}^3)3(C_{512}^3)^{7x7x2048} - P_G^{1x1x2048} - FC_{1000} = 7C + 13(3C) = 46C$
- cliii (He et al. 2016)
- cliv (Xie et al. 2016)
- clv (Iandola et al. 2016, S.6)
- clvi (Simonyan & Zisserman, 2014, S.3)
- clvii  $C^2 P_{M_{64}} - C^2 P_{M_{128}} - C^3 P_{M_{256}} - C^3 P_{M_{512}} - C^3 P_{M_{512}} - FC_{4096}^2 - FC_{1000} = 2C + 2C + 3C + 3C + 3C = 13C$
- clviii  $C^2 P_{M_{64}} - C^2 P_{M_{128}} - C^4 P_{M_{256}} - C^4 P_{M_{512}} - C^4 P_{M_{512}} - FC_{4096}^2 - FC_{1000} = 2C + 2C + 4C + 4C + 4C = 16C$
- clix (Chollet, 2017, S.1255)
- clx  $C_{64}^2 C_{128}^2 C^2 P_{M_{256}} - C^2 P_{M_{728}} - (8C^3)C^2 P_M - C^2 P_G - FC_{2048} - FC_{opt} - FC_{1000} = 4(2C) + 8(3C) + 2(2C) = 36C$

- 
- clxi Alle CNN-Architekturen wurden selbst implementiert und deren Merkmale in Tabelle 4 eingetragen, nicht importierbare Modelle und deren Eigenschaften wurden aus dem jeweiligen Paper des Referenzmodells entnommen Vgl. Endnoten in der Namens-Spalte Quelle: [https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow/tree/master/cnn%20basic%20architecture%20\(vgl.%20\[4.1\]%20Tabelle%205%20B6\)](https://github.com/faust-prime/X-ray-images-classification-with-Keras-TensorFlow/tree/master/cnn%20basic%20architecture%20(vgl.%20[4.1]%20Tabelle%205%20B6))
- clxii (Vgl. Goodfellow et al. 2016, S.164f)
- clxiii (Zeiler/Fergus, 2013)
- clxiv (Vgl. Simonyan & Zisserman, 2014, S.1 ff.)
- clxv (Szegedy et al. 2014, S.2)
- clxvi (Quelle: Szegedy et al. 2014, S.4)
- clxvii (Vgl. Szegedy et al. 2014, S.5)
- clxviii (Vgl. Szegedy et al. 2014, S.5)
- clxix (Vgl. Raschka/Mirjalili, 2016, S.387)
- clxx (Lin et al. 2014)
- clxxi (Vgl. Goodfellow et al. 2016, S.198)
- clxxii mit  $P_{s=2}^{3 \times 3}$  und  $C_{96}^{11 \times 4} := 2DConvolution(f = 11; stride = 4; no. of filters = 96)$
- clxxiii  $3(C_{64}^2)$  ist die Kurzschreibweise eines Residual-Blocks mit der Convolution-Folge:  $C_{64}^{1 \times 1} C_{64}^{3 \times 3} C_{64 \times 4}^{1 \times 1}$
- clxxiv Nach den Erkenntnissen von Tabelle 4: CNN-Überblick.
- clxxv (Vgl. Iandola et al. 2016, S.1)
- clxxvi Unter Beachtung der Grenzen des Wertebereichs. Eine FP32-Variable (mit der Einteilung in Vorzeichen, Exponent und Mantisse nach IEEE 754) wird beispielsweise durch 32 Bit Speicher repräsentiert und besitzt nur die halbe (einfache) Genauigkeit (*engl. single precision*) gegenüber einer FP64-Variable mit 64 Bit Speicherbedarf (*engl. double precision*).
- clxxvii (Vgl. Raschka/Mirjalili, 2016, S.229)
- clxxviii (King & Zen, 2001)
- clxxix (Lemaître et al. 2017)
- clxxx (Vgl. Raschka/Mirjalili, 2016, S.228 ff.)
- clxxxi (Chawla et al. 2002)
- clxxxii (Litjens et al. 2017)
- clxxxiii (Baltrushat et al. 2019)
- clxxxiv (Vgl. Baltrushat et al. 2019, S.8)
- clxxxv (Vgl. Baltrushat et al. 2019, S.7)
- clxxxvi (Vgl. Baltrushat et al. 2019, S.3)
- clxxxvii (Vgl. Baltrushat et al. 2019, S.9)
- clxxxviii (Vgl. Baltrushat et al. 2019, S.6)
- clxxxix (Lee et al. 2017)
- cxc (Vgl. Litjens et al. 2017, S.27)
- cxc i (Vgl. Lee et al. 2017, S.580)
- cxc ii (Vgl. Tajbaksh et al. 2017, S.1f)
- cxc iii (Vgl. Tajbaksh et al. 2017, S.2f)
- cxc iv (Vgl. Baltrushat et al. 2019, S.7)
- cxc v (Vgl. Chartrand et al. 2017, S.2129)
- cxc vi (Vgl. Litjens et al. 2017, S.27)
- cxc vii (Vgl. Baltrushat et al. 2019, S.2)