Brandenburg
University of Technology
Cottbus - Senftenberg

Künstliche Neuronale Netze in biomedizinischen Anwendungen

Anschel Julio Roman

Cottbus Mathematical Preprints
COMP # 20(2021)

Brandenburgische Technische Universität Cottbus - Senftenberg

Bachelorarbeit

Künstliche Neuronale Netze in biomedizinischen Anwendungen

vorgelegt am: 17. August 2021

Anschel Julio Roman

Studiengang: Wirtschaftsmathematik

Betreuung: Prof. Dr. Armin Fügenschuh
Zweitgutachten: Prof. Dr. Carsten Hartmann

Fakultät MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
Fachgebiet Ingenieurmathematik und Numerik der Optimierung

Abbildungsverzeichnis

1 Struktur eines neuronalen Netzwerks (vgl. [9, S.197], [11]) 10
2 Die ReLU-Funktion 12
3 Die Sigmoid-Funktion 12
4 3-dimensionaler Tensor (vgl. [28]) 22
5 Beispiel Verträglichkeit TensorFlow und Python-Code 22
6 Beispiel einer 2-dimensionalen Faltung ohne Kernel-Flip (vgl. [31],[32]) . . . 24
7 Beispiele zur Faltung 25
8 Beispiel Padding (vgl. [31]) 26
9 Beispiel MaxPooling mit Schrittweite 2 (vgl. [31]) 27
10 Struktur des eigenen Modells für 2 Klassen (vgl. [37]) 29
11 Ausgabe während einer Trainingseinheit in TensorFlow 31
12 Beispiel Konfusionsmatrix (vgl. [40],[39]) 32
13 Beispielbilder Interphase und Metaphase [Quelle: Medipan] 35
14 Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer ohne Moment 38
15 Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer mit einem Moment von 0.9 38
16 Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer mit einem Moment von 0.9 und mit Nesterov-Moment 38
17 Trainingsverläufe des eigenen Modells mit RMSprop 39
18 Trainingsverläufe des eigenen Modells mit dem ADAM-Optimierer 39
19 Trainingsverläufe des eigenen Modells mit dem NADAM-Optimierer und Data Augmentation 41
20 Trainingsverläufe des nicht vortrainierten VGG19 43
21 Trainingsverläufe des nicht vortrainierten InceptionV3 43
22 Trainingsverläufe des nicht vortrainierten Xception 44
23 Trainingsverläufe des nicht vortrainierten ResNet50 mit dem SGD-Optimierer (Mom.=0.9, Nesterov) 45
24 Trainingsverläufe des nicht vortrainierten ResNet50 mit ADAM 45
25 Trainingsverläufe des nicht vortrainierten EfficientNetB0 46
26 Trainingsverläufe des vortrainierten Xception 47
27 Trainingsverläufe des vortrainierten InceptionV3 48
28 Beispielbilder der 7 verschiedenen Klassen [Quelle: Medipan] 50
29 Konfusionsmatrizen für verschiedene Modelle für 7 Klassen 52
30 Trainingsverläufe verschiedener Modelle für 7 Klassen 53
31 Auswirkungen der einzelnen Parameter auf die angepassten Genauigkeiten (eigenes Modell) 56
32 Pruning des vortrainierten Xception mit verschiedenen Werten für konstante Spärlichkeit 58
33 Trainingsverläufe des vortrainierten Xception bei verschiedenen Werten für konstantes Pruning 59
34 Trainingsverläufe des vortrainierten Xception bei verschiedenen Werten für polynomielles Pruning 60
35 Pruning des vortrainierten InceptionV3 mit verschiedenen Werten für konstante Spärlichkeit 61

Tabellenverzeichnis

1 Ergebnisse Fine-Tuning der Filter des eigenen Modells 36
2 Ergebnisse Fine-Tuning der dichten Schichten des eigenen Modells 37
3 Ergebnisse der Tests mit verschiedenen Optimierern am eigenen Modell . . 40
4 Ergebnisse des Fine-Tunings von Lernrate und Moment mit VGG16 42
5 Ergebnisse der Tests mit nicht vortrainierten Keras Applications 46
6 Ergebnisse des Fine-Tunings der dichten Schichten im vortrainierten Xception 47
7 Ergebnisse der Tests mit dem vortrainierten InceptionV3 48
8 Ergebnisse der Tests mit vortrainierten Keras Applications 49
9 Ergebnisse der Tests mit Keras Applications und dem eigenen Modell für 7 Klassen 51
10 Genauigkeiten der Einzelerkennungmodelle 54
11 Ergebnisse der Tests mit der angepassten Evaluierung 57
12 Ergebnisse der polynomiellen Beschneidung des vortrainierten Xception . . 60
13 Ergebnisse der polynomiellen Beschneidung des vortrainierten InceptionV3 . 61

Inhaltsverzeichnis

1 Einleitung 6

2 Künstliche Intelligenz 7

3 Künstliche Neuronale Netze und Deep Learning 9
3.1 Die Aktivierungsfunktion . . . 10
3.2 Die Kostenfunktion . . . 13
3.3 Backpropagation . . . 16
3.4 Das stochastische Gradientenverfahren . . . 18
3.5 Optimierer . . . 19
3.6 Regularisierung . . . 20
3.6.1 L2- und L1-Regularisierung . . . 21
3.6.2 Dropout . . . 21
3.7 TensorFlow . . . 21

4 Convolutional Neural Networks 23
4.1 Faltung . . . 23
4.2 Padding . . . 25
4.3 Pooling . . . 26
4.4 Einführendes Beispiel . . . 27

5 Möglichkeiten zur Beurteilung eines Modells 30

6 Methodik 33
6.1 Aufteilung in Validierungs-, Trainings- und Testdaten . . . 33
6.2 Begrenzte Anzahl von Trainingsbildern und Epochen . . . 33
6.3 Programmierung in Google Colaboratory . . . 34
6.4 Darstellung der Ergebnisse . . . 34

7 Experimente und Ergebnisse 35
7.1 Fine Tuning . . . 35
7.2 Keras Applications und Transfer-Learning . . . 41
7.2.1 Tests mit nicht vortrainierten Modellen . . . 42
7.2.2 Tests mit vortrainierten Modellen . . . 46
7.2.3 Vergleich zwischen Modellen mit und ohne Pre-Weights . . . 49
7.3 Erweiterung auf sieben Klassen . . . 50
7.4 Anpassung der Evaluierung . . . 54
7.5 Pruning . . . 57

8 Diskussion 62
8.1 Nutzen des Fine-Tunings . . . 62
8.2 Nutzen des Transfer-Learnings . . . 62
8.3 Schwierigkeiten bei der Klassenerweiterung . . . 63
8.4 Berücksichtigung der biologischen Gegebenheiten . . . 64
8.5 Möglichkeiten zur Modellkomprimierung . . . 64

9 Fazit 65

A Dokumentation **71**
A.1 Allgemeine Hinweise 71
A.1.1 ipynb-Dateien 71
A.1.2 py-Dateien 71
A.2 Erläuterung der Programme 72
A.2.1 0-Material_von_Dr_Hiemann 72
A.2.2 1-Fine-Tuning 72
A.2.3 2-Keras_Applications_und_Transfer-Learning 72
A.2.4 3-Erweiterung_auf_sieben_Klassen 73
A.2.5 4-Anpassung_der_Evaluierung 73
A.2.6 5-Pruning 73
A.2.7 x-zu_erwartende_Ausgaben 74
A.3 Ergänzungen 74

1 Einleitung

Künstliche Intelligenz durchdringt heute nahezu jeden Bereich unseres Lebens und dabei stehen wir gerade erst am Anfang dieser Entwicklung. Die seit einigen Jahren bestehende große mediale Aufmerksamkeit befördert zudem die gesellschaftliche Auseinandersetzung mit dem Thema, sei es künstlerisch oder auch philosophisch, beispielsweise bei der Diskussion ethischer Fragen, welche der Einsatz von künstlicher Intelligenz notwendigerweise aufwirft. Auch in der Medizin spielt sie eine immer größere Rolle. Nutzen möchte man sie hier um mit ihrer Hilfe eine bessere medizinische Versorgung der Patienten und neue Therapiemethoden zu entwickeln. Ob als unterstützende Kraft an der Seite der Ärzte oder als Werkzeug für Hersteller von pharmazeutischen Prdukten, künstliche Intelligenz ist aus zukünftigen Planungen nicht mehr wegzudenken. Die potenziellen Anwendungsgebiete reichen von der Prozessautomatisierung bei der Medikamentenherstellung über die Früherkennung von Krankheiten bis zur Hilfestellung bei Entscheidungsfindungen in komplizierten Situationen. Der mögliche Nutzen von künstlicher Intelligenz wird dabei sowohl von wissenschaftlicher Seite als auch von Teilen der Wirtschaft intensiv erforscht.
Im Bereich der Auswertung medizinischer Bildaufnahmen konnte in einer Studie der Universitätshautklinik Heidelberg aus dem Jahr 2019 der Nutzen von künstlicher Intelligenz bei der Erkennung von schwarzem Hautkrebs nachgewiesen werden [1]. An der Stanford University wurde gezeigt, dass Algorithmen mit künstlicher Intelligenz bei der Erkennung verschiedener Lungenkrankheiten anhand von MRT-Bilden die Ärzte zum Teil sogar übertreffen konnten [2]. PwC belegte den potenziellen Nutzen der künstlichen Intelligenz bei der Entwicklung einer günstigeren Gesundheitsversorgung in Europa [3].
Diese Arbeit beschäftigt sich mit der Entwicklung effizienter Methoden für die Klassifizierung der Stadien von Zellen. Hierbei liegt der Fokus auf sogenannten Deep-Learning-Algorithmen. Diese haben sich unter anderem in der Bilderkennung als sehr leistungsfähig erwiesen und können genutzt werden um große Mengen von mikroskopischen Zellbildern in kurzer Zeit zu klassifizieren. Aufgezeigt werden Möglichkeiten zur Optimierung solcher Algorithmen mit dem Ziel, Genauigkeit und Speichergröße zu verbessern. Es wurden hierbei der Einfluss von verschiedenen Parametern auf die Performance eines Algorithmus untersucht und gegenübergestellt, verschiedene etablierte Modelle miteinander verglichen und eine Auswahl gängiger Methoden zur Modell-Optimierung getestet. Genutzt wurde die Software-Bibliothek TensorFlow, welche in die Programmiersprache Python eingebettet ist. Die Arbeit entstand in der Zusammenarbeit mit Herrn Dr. Rico Hiemann von Medipan. Sie liefert zudem in Vorbereitung auf die Präsentation der praktischen Ergebnisse einen Überblick über die wichtigsten Begriffe und die wesentlichen mathematischen Methoden und Zusammenhänge, die den genutzten Algorithmen und Optimierungstechniken zugrunde liegen.

2 Künstliche Intelligenz

Zu Beginn wird ein Überblick über die wichtigste Terminologie im Bereich der künstlichen Intelligenz gegeben. Es stellt sich zunächst die Frage, was genau künstliche Intelligenz überhaupt ist. Von Alan Turing, einem ihrer Väter, stammt das Zitat „A computer would deserve to be called intelligent if it could deceive a human into believing that it was human.“ [4] Er präsentierte 1950 mit dem Turing-Test eine Idee, wie man die Frage *Ist eine Maschine intelligent oder nicht?* beantworten könnte. Dabei war eine Maschine aus seiner Sicht dann als intelligent zu betrachen, wenn sie in der Lage wäre, einem Menschen vorzumachen, dass sie menschlich sei. [5, S.463]
Im Kontext der heutigen Beschäftigung mit dem Thema, bei der vor allem die praktischen Anwendungen der künstlichen Intelligenz von Bedeutung sind, scheint eine etwas pragmatischere Definition passend, nämlich als das Bemühen, intellektuelle Aufgaben, welche normalerweise von Menschen durchgeführt werden, zu automatisieren. Bis in die 80er Jahre glaubte man, dass eine genügend große Menge von expliziten Regeln für ein Computerprogramm ausreichend wäre, um das Level menschlicher Intelligenz erreichen zu können. Wie sich zeigte, ist diese sogenannte symbolische künstliche Intelligenz gut geeignet für wohldefinierte logische Probleme, wie beispielsweise die Programmierung eines virtuellen Schach-Gegners. Für komplexere Problemstellungen, wie Bilderkennung, Spracherkennung und Textübersetzung, war die Entwicklung eines neuen Ansatzes erforderlich, das **maschinelle Lernen**. Der Schlüssel liegt hier in der Einführung eines neuen Programmierparadigmas. Im klassischen Ansatz werden einem System Daten und ein Satz expliziter Regeln für deren Verarbeitung gegeben. Das System liefert dann Antworten in Abhängigkeit von den Eingangsdaten. Der neue Ansatz besteht nun darin, dem System Daten zu geben, und zu diesen Daten die entsprechenden Antworten. Das System liefert dann eine Menge von Regeln, die die gegebenen Antworten auf die Daten am besten erklären. Diese Regeln können auf neue Eingangsdaten angewendet werden, um passende Antworten zu generieren. [6, S.4,f.]

Das Kernproblem einer Machine-Learning-Aufgabe ist das Auffinden von nützlichen Repräsentationen der Daten, aus denen die durch statistische Zusammenhänge induzierten Regeln abgeleitet werden können. Zur Realisierung dieses als **Feature-Erkennung** bezeichneten Vorgangs kann das Machine-Learning-System aus einer vordefinierten Menge von mathematischen Operationen, dem Hypothesenraum, eine passende Regel auswählen. Dem Machine-Learning-Algorithmus wird in Form eines Feedback-Signals ein Bewertungsmaß für seine Fortschritte mitgegeben. Mittels dieses Signals passt der Algorithmus seine Arbeitsweise an. Diesen Vorgang nennen wir das **Lernen**. [6, S.6]
Ein Teilgebiet des maschinellen Lernens, bei dem **Schichten** sukzessiv bedeutsamerer Repräsentationen der Daten erlernt werden, bezeichnen wir als **Deep Learning**. Die Schichten der Datenrepräsentationen werden auch **versteckte Schichten** (hidden layer) genannt. Deep-Learning-Systeme können hunderte von versteckten Schichten verwenden. Im Unterschied dazu lernen andere Machine-Learning-Systeme in der Regel mit höchstens zwei versteckten Schichten. Ein Deep-Learning-Algorithmus lernt mittels einer Struktur, welche wir als **(künstliches) neuronales Netz** bezeichnen. Die einzelnen Elemente, aus denen die Schichten aufgebaut sind, heißen **(künstliche) Neuronen**. Die Verbindungen zwischen den Neuronen einer Schicht und den Neuronen der dahinterliegenden Schicht heißen **Gewichte**. Obwohl die künstlichen Neuronen in ihrer Funktionsweise an biologische Neuronen im Gehirn angelehnt sind, ist es interessant zu wissen, dass weder die Struktur noch die Arbeitsweise künstlicher neuronaler Netze in einer nachweisbaren Analogie zu biologischen neuronalen Netzen des Gehirns stehen. Die künstlichen neuronalen Netze (im weiteren Verlauf mit „neuronale Netze“ bezeichnet) sind lediglich *inspiriert* von ihren biologischen Namensgebern. [6, S.8]

Mathematisch betrachtet, symbolisieren die Neuronen Punkte im Netz, an denen bestimmte festgelegte Operationen durchgeführt werden, während die Gewichte die variablen Parameter sind, die einer Operationseinheit zugeführt werden.
Die einem Machine-Learning-Algorithmus zuzuführenden Antworten zu einem gegebenen Datensatz müssen im Vorfeld „per Hand“ generiert werden. Man nennt sie **Label**. Ein Label weist einem Datenpunkt die zugehörige wahre Antwort zu. Eingangsdaten und die zugehörigen Label bilden zusammen die **Trainingsdaten**. Das durch ein Machine-Learning-System generierte Ausgangssignal zu einem Eingangssignal wird die **Vorhersage** (prediction) genannt. Machine-Learning-Systeme können sowohl für Regressionprobleme als auch für Klassifikationsprobleme genutzt werden. Im weiteren Verlauf konzentrieren wir uns auf die Anwendung auf Klassifikationsprobleme.
Beim Einsatz von Machine-Learning-Algorithmen für die Klassifizierung von Daten unterscheidet man zwischen binärer Klassifizierung, also Problemen mit genau zwei möglichen Klassen, und Multiklassen-Klassifizierung im Falle von mehr als zwei vorhandenen Klassen. Es gibt verschiedene Arten von neuronalen Netzen, in dieser Arbeit liegt der Fokus auf vorwärtsgerichteten neuronalen Netzen (vgl. [7, S. 186,ff]).
Üblicherweise nennt man einen Deep-Learning-Algorithmus ein **Modell**. Den Versuch, einem Modell das Erlernen einer bestimmten Methode beizubringen, nennen wir passenderweise **Training**. Diejenigen Parameter, welche vom Programmierer vor Beginn einer Trainingseinheit festgelegt werden müssen, aber nicht während des Trainings verändert werden können, heißen **Hyperparameter** ([8]).

3 Künstliche Neuronale Netze und Deep Learning

Deep Learning ist das mehrstufige Erlernen nützlicher Reräsentationen von Daten. Input-Daten werden dazu mittels einer Sequenz von Datentransformationen einem Output zugeordnet. Das Feedbacksignal zur Bewertung des Algorithmus wird als Lösung eines Optimierungsproblems realisiert. Mittels einer Kostenfunktion wird in jedem Schritt der Fehler der aktuellen Gewichtskombination bewertet. Dieser Fehler wird minimiert, indem in jedem Schritt die Gewichte angepasst werden, sodass am Ende des Tranings zu den Trainingsbildern mit möglichst geringen Abweichungen gerade die in den Labeln vorgegebenen Ausganssignale erzeugt werden. Für die Fehlerminimierung werden die Ableitungsinformationen der Kostenfunktion benötigt, welche mithilfe einer Fehlerrückführung gewonnen werden.
Üblicherweise repräsentiert jedes der Neuronen der Output-Schicht genau eine der möglichen Klassen des Ausgangsproblems. Für ein gegebenes Eingangssignal $\mathbf{x}$ ist das zugehörige Label demnach von der Form

$$\mathbf{y_{true}}(\mathbf{x}) = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^K, \tag{3.1}$$

wobei die 1 an genau der Stelle steht, die in der Output-Schicht des Netzwerks der wahren Klasse des zugehörigen Eingangssignals entspricht und K die Anzahl der möglichen Klassen bezeichnet. Die Speicherung von Labeln in der Form (3.1) nennt man auch One-Hot-Kodierung der Klassen. Die von dem Netzwerk zu einem Datenpunkt erzeugte Vorhersage liegt also idealerweise möglichst nah an den Labeln $\mathbf{y_{true}}$ dran. Jede der versteckten Schichten führt dazu eine Transformation der Eingangsdaten durch. Diese wird durch die der Schicht zugehörigen Gewichtsparametern definiert. Das Lernen können wir somit betrachten als das Auffinden derjenigen Kombination von Gewichten, welche bei gegebenen Eingangsdaten zur Erzeugung der in den Labeln gespeicherten wahren Output-Vektoren führt. Die Gewichte werden durch Matrizen $\mathbf{A}^{(d)}$ repräsentiert, wobei $d \in \{1, \ldots, D\}$ und D die Anzahl der versteckten Schichten des Netzwerks plus die Ausgangsschicht bezeichnet. Die Matrix $\mathbf{A}^{(d)} = [a_{ij}^{(d)}]$ bildet dabei von der $(d-1)$-ten zur d-ten Schicht ab. Ein Gewicht $a_{ij}^{(d)}$ verbindet also das i-te Neuron der d-ten Schicht mit dem j-ten Neuron der $(d-1)-ten$ Schicht. Sei $\mathbf{x} := \mathbf{x}^{(0)} \in \mathbb{R}^p$ ein Datenpunkt in Form eines Vektors der Dimension p und bezeichne $x^{(d)}, d \in \{1, \ldots, D-1\}$ die in der d-ten versteckten Schicht erzeugte Datenrepräsentation. Im einfachsten Fall eines *linearen* neuronalen Netzes gilt dann

$$\begin{aligned} \mathbf{x}^{(1)} &= \mathbf{A}^{(1)}\mathbf{x} \\ \mathbf{x}^{(2)} &= \mathbf{A}^{(2)}\mathbf{x}^{(1)} \\ &\vdots \\ \mathbf{y_{pred}}(\mathbf{x}) = \mathbf{x}^{(D)} &= \mathbf{A}^{(D)}\mathbf{x}^{(D-1)}. \end{aligned} \tag{3.2}$$

Der Vektor $\mathbf{x}$ kann beispielsweise ein einzelnes Trainingsbild darstellen, dessen Pixelwerte aneinandergereiht wurden. In diesem Fall können wir p gerade als die Anzahl der Bildpixel auffassen. D.h. bereits der Input-Vektor selbst ist eine spezielle Repräsention des ursprünglichen Trainingsmaterials, z.B. eine Speicherung der einzelnen Bildpixel in einem

Vektor. Der Output-Vektor kann auch als

$$\mathbf{y_{pred}}(\mathbf{x}) = \mathbf{A}^{(D)}\mathbf{A}^{(D-1)} \cdots \mathbf{A}^{(2)}\mathbf{A}^{(1)}\mathbf{x} \tag{3.3}$$

geschrieben werden. Setzen wir $\tilde{\mathbf{A}} := \mathbf{A}^{(D)}\mathbf{A}^{(D-1)} \cdots \mathbf{A}^{(2)}\mathbf{A}^{(1)}$ vereinfacht sich dieser Ausdruck zu

$$\mathbf{y_{pred}}(\mathbf{x}) = \tilde{\mathbf{A}}\mathbf{x}. \tag{3.4}$$

[9, S.196] Es ist $\mathbf{y}_{pred} \in \mathbb{R}^K$. Bezeichnet $x_i^{(d)}$ das i-te Neuron der d-ten Schicht, dann wird dort die Operation

$$x_i^{(d)} = \sum_j a_{ij}^{(d)} x_i^{(d-1)} \tag{3.5}$$

durchgeführt. In der Literatur finden sich manchmal noch sogenannte *Bias-Vektoren* $b^{(d)}$, $d \in \{1, \ldots, D\}$, sodass für jedes Neuron die Berechnung von

$$x_i^{(d)} = \sum_j a_{ij}^{(d)} x_i^{(d-1)} + b_i^{(d)} \tag{3.6}$$

stattfindet (vgl. [10, S.4]). Wir beschränken uns aus Einfachheitsgründen auf die Betrachtung der Gewichtsmatrizen.

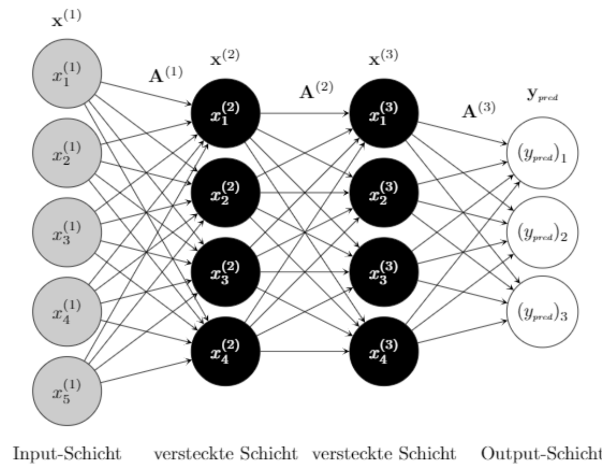


Abbildung 1: Struktur eines neuronalen Netzwerks (vgl. [9, S.197], [11])

3.1 Die Aktivierungsfunktion

Wenngleich ein lineares neuronales Netz einfach zu beschreiben ist, stellt sich die Linearität als zu einschränkend für die Lösung vieler Klassifikationsprobleme heraus. Oft liegen nichtlineare Zusammenhänge zwischen den Eingangsdaten und den gegebenen Antworten vor. Ein lineares neuronales Netz arbeitet wie ein lineares Reggressionsmodell und wäre deswegen nicht imstande, nichtlineare Zuordnungen zwischen Input und Output zu erlernen. Sei deswegen $f_i^{(d)}$ eine nichtlineare **Aktivierungsfunktion**, welche mit der in (3.4) beschriebenen linearen Abbildung verknüpft wird. Die an jedem Neuron stattfindende Operation kann nun durch

$$x_i^{(d)} = f_i^{(d)} \left(\sum_j a_{ij}^{(d)} x_i^{(d-1)} \right) \tag{3.1.1}$$

beschrieben werden. Für eine Schicht im Netz ergibt sich damit

$$\mathbf{x}^{(d)} = \mathbf{f}^{(d)} \left(\mathbf{A}^{(d)} \mathbf{x}^{(d-1)} \right). \tag{3.1.2}$$

Das durch das Netzwerk erzeugte Ausgangssignal wird zu

$$\mathbf{y_{pred}}(\mathbf{x}) = \mathbf{f}^{(D)} \left(\mathbf{A}^{(D)} (\ldots (\mathbf{f}^{(2)} (\mathbf{A}^{(2)} \mathbf{f}^{(1)} \left(\mathbf{A}^{(1)} \mathbf{x} \right) \ldots \right). \tag{3.1.3}$$

[9, S.197]. Die Aktivierungsfunktion ermöglicht es dem neuronalen Netz, sehr komplexe Informationen zu erkennen und komplizierte Zusammenhänge der Daten zu erlernen. Konnte der Outputvektor im linearen Fall in (3.3) nur als Polynom 1. Ordnung dargestellt werden, sind mittels nichtlinearer Aktivierungsfunktionen nun Darstellungen von beliebig komplexen Funktionen am Output möglich. Diese Eigenschaft wird im „Universal Approximation Theorem“ formal festgehalten. Insbesondere konnte für (vorwärtsgerichtete) neuronale Netze mit nicht-polynomiellen Aktivierungsfunktionen gezeigt werden, dass sie universale Funktionsapproximierer sind [12, S.1]. Wie wir später sehen werden, basiert die Erzeugung des Bewertungssignals für einen Deep-Learning-Algorithmus auf Ableitungsinformationen, daher spielen auch die Eigenschaften ihrer Ableitung eine wesentliche Rolle bei der Wahl einer geeigneten Aktivierungsfunktion.

Stufenfunktion
Ein intuitiv naheliegende Aktivierungsfunktion ist eine Stufenfunktion der Form

$$\operatorname{step}(x) = \begin{cases} 1, & x \geqslant 0 \\ 0, & x < 0. \end{cases} \tag{3.1.4}$$

Sie ist einfach zu implementieren und entspräche der Aktivierung eines biologischen Neurons das ebenfalls nur die beiden Zustände *An* oder *Aus* besitzt. Stufenfunktionen sind jedoch nur für binäre Klassifikationsprobleme geeignet. Ein weiteres Problem ist die Nichtdifferenzierbarkeit dieser Funktion. Dies ist von Nachteil bei der im vorherigen Kapitel beschriebenen Anpassung des Algorithmus durch das Bewertungs-Signal.

Lineare Aktivierung
Diese Form der Aktivierung löst zwar das Problem der Nichtdifferenzierbarkeit, allerdings nicht in einer zufriedenstellenden Weise, da die Ableitung stets eine vom Eingangssignal unabhängige Konstante ist. Außerdem bleibt hier die Linearität eines neuronales Netzes erhalten, gemeinsam mit den damit verbundenen Nachteilen.

Rectified Linear Unit (ReLU)
Diese nichtlineare Aktivierungsfunktion wird durch die Formel

$$\operatorname{ReLU}(x) := \max(0, x) \tag{3.1.5}$$

definiert. Der Vorteil gegenüber einer linearen Aktivierung liegt hier darin, dass nicht stets *alle* Neuronen angesteuert werden, sondern nur solche, an denen die Summe der Gewichte in (3.4) einen positiven Wert liefert.
Sigmoid
Die Sigmoid-Funktion ist durch

$$\operatorname{sigmoid}(x) := \frac{1}{1 + \mathrm{e}^{-x}} \tag{3.1.6}$$

gegeben. Diese Funktion hat eine Reihe günstiger Eigenschaften. Sie ist stetig differenzierbar und aufgrund ihrer „S“-Form dem Output-Signal einer Stufenfunktion ziemlich

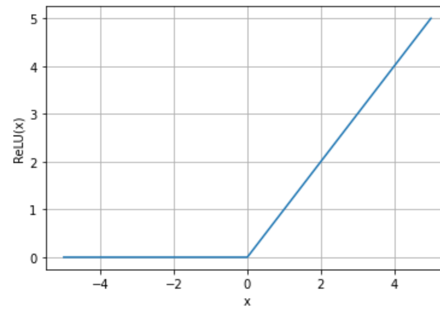


Abbildung 2: Die ReLU-Funktion

nahe. Außerdem bildet sie in das Intervall $(0, 1)$ ab. Das ist vor allem für binäre Klassifizierungsprobleme vorteilhaft. Für solche Probleme kann ein einzelnes Neuron am Output ausreichend sein, welches mit den beiden Werten 0 und 1 jeweils eine der beiden Klassen anzeigt. Daher wird hier oft die Sigmoid-Aktivierung für das Output-Neuron benutzt. Die auf diese Weise erzeugten Ausgangswerte können (informal) als „Wahrscheinlichkeiten" der Zugehörigkeit eines Eingangsdatenpunktes zu einer bestimmten Klasse interpretiert werden. In der Praxis wird diese Aktivierungsfunktion sehr häufig eingesetzt, dennoch hat sie auch Nachteile. Für sehr große und sehr kleine Werte des Eingangssignals ist die Ableitung der Sigmoid-Funktion besonders klein. Dies ist keine wünschenswerte Eigenschaft.

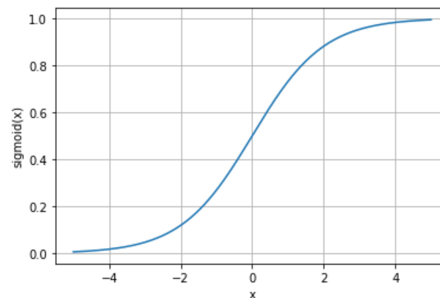


Abbildung 3: Die Sigmoid-Funktion

Softmax

Die Softmax-Funktion weist große Ähnlichkeiten mit der Sigmoid-Funktion auf. Sie wird meist für die Aktivierung der Output-Neuronen verwendet und eignet sich insbesondere für Multiklassifizierung. Es kann mittels der Softmax-Funktion ein K-dimensionaler Vektor erzeugt werden, der in jeder Komponente einen Wert zwischen 0 und 1 enthält und dessen Werte aufaddiert eine 1 ergeben. D.h. auch die Softmax-Funktion hat den Zweck, die „rohen" Werte der Neuronen der versteckten Schichten in als Wahrscheinlichkeiten interpretierbare Werte zwischen 0 und 1 zu konvertieren. Sie ist durch

$$\operatorname{softmax}(x)_i := \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}, \quad i = 1, \dots, K \tag{3.1.7}$$

gegeben. [13, 311,ff.], [14]

3.2 Die Kostenfunktion

In jedem Machine-Learning-System wird das Lernen des Algorithmus in ein mathematisches Optimierungsproblem überführt. Es wird die **Kostenfunktion** L minimiert, welche den Abstand zwischen dem wahren Label eines Datenpunktes zu der zugehörigen Vorhersage des Netzwerks misst. Es ist in der Praxis üblich, für Klassifizierungsprobleme die Kreuzentropie zu benutzen. Es sei $\mathbf{x}^n, n \in \{1, \dots, N\}$ einer von N gegebenen Datenpunkten. Wir bezeichnen

$$\begin{aligned} \mathbf{y}^n_{\mathbf{true}} &:= \mathbf{y}_{\mathbf{true}}(\mathbf{x}^n) \\ \mathbf{y}^n_{\mathbf{pred}} &:= \mathbf{y}_{\mathbf{pred}}(\mathbf{x}^n). \end{aligned}$$

Die Kreuzentropie ist dann gegeben durch

$$\mathrm{CE} := -\sum_{k=1}^{K} (y_{true})_k \log(y_{pred})_k. \tag{3.2.1}$$

Bei binären Klassifizierungsproblemen kann man mit nur einem Output-Neuron arbeiten. Hier nutzt man die **binäre Kreuzentropie**

$$\mathrm{CE}_{\mathrm{Bin}} := -\left[(y_{true})_k \log(y_{pred})_k + (1 - (y_{true})_k) \log(1 - (y_{pred})_k)\right]. \tag{3.2.2}$$

Um daraus nun eine passende Kostenfunktion für das Optimierungsproblem zu erhalten, bilden wir den Durchschnitt über die Kreuzentropien aller Trainingsdaten:

$$\mathrm{L}_{\mathrm{Bin}} := -\frac{1}{N} \sum_{n=1}^{N} \left[(y^n_{true})_k \log(y^n_{pred})_k + (1 - (y^n_{true})_k) \log(1 - (y^n_{pred})_k)\right] \tag{3.2.3}$$

Es bezeichnet hierbei N die Anzahl der Trainingsdaten. Für ein Multiklassifikationsproblem sprechen wir im Falle einer Kombination der Kreuzentropie (3.2.1) in der Kostenfunktion mit der durch die Softmax-Funktion erzeugten One-Hot-Darstellung der Klassen in (3.1.7) von der **kategorischen Kreuzentropie**. Analog zum binären Fall wird sie über den Durchschnitt aller Kreuzentropien gebildet:

$$\mathrm{L}_{\mathrm{Cat}} := -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \left[(y^n_{true})_k \log(y^n_{pred})_k\right] \tag{3.2.4}$$

[15] Die Kreuzentropie ist ein Konzept aus der Informationstheorie. Sie ist verwandt mit der Entropie. Diese ist ist wie folgt definiert.

Definition 1. Es sei X eine diskrete Zufallsvariable mit einer Wahrscheinlichkeitsverteilung P. Dann ist die Entropie $\mathbb{H}(P)$ von X definiert durch

$$\mathbb{H}(P) := -\sum_{k=1}^{K} P(X = k) \log P(X = k). \tag{3.2.5}$$

Die Entropie ist ein Maß für die Unsicherheit einer Zufallsvariable. Sie gibt den mittleren Informationsgehalt eines zufällig ausgewählten Elements des Ereignisraums an unter der Annahme, dass X die Verteilung P hat. Eine Entropie von 0 entspricht einer nicht vorhandenen Unsicherheit. Je höher die Wahrscheinlichkeit eines Ereignisses ist, desto weniger Informationsgehalt hat es, da es dann öfter auftritt. Mit anderen Worten, je konzentrierter die Dichtefunktion um einen Punkt ist, desto geringer ist die Unsicherheit. So hat beispielsweide eine gleichverteilte Zufallsvariable eine maximale Entropie von $-K \cdot \frac{1}{K} \log \frac{1}{K} = \log K$. Minimale Entropie erhält man im Falle einer Nadelimpulsfunktion.

Definition 2. Es sei X eine diskrete Zufallsvariable. Weiter seien P und Q zwei Wahrscheinlichkeitsverteilungen. Die **Kreuzentropie** ist gegeben durch

$$\mathbb{H}(P,Q) := -\sum_{k=1}^{K} P(X=k)\log Q(X=k). \tag{3.2.6}$$

Die Kreuzentropie können wir verstehen als ein Qualitätsmaß für ein Modell, mit dem wir eine Wahrscheinlichkeitsverteilung approximieren möchten. Im Kontext des maschinellen Lernen ist die wahre Distribution P der Eingangsdaten in (3.2.6) gegeben durch die Label $\mathbf{y_{pred}}$, während Q die durch das Modell erzeugte Approximation der wahren Antworten repräsentiert. Damit erhalten wir genau die Darstellung aus (3.2.1).
Herleiten lässt sich die Formel für die Kreuzentropie (3.2.6) aus der Maximum-Likelihood-Methode. Um das zu zeigen benötigen wir ein weiteres Konzept aus der Wahrscheinlichkeitstheorie.

Definition 3. Es sei wieder X eine diskrete Zufallsvariable und P und Q zwei Wahrscheinlichkeitsverteilungen. Die *Kullback-Leibler-Divergenz (KL-Divergenz)* ist gegeben durch

$$\mathbb{KL}(P,Q) := \sum_{k=1}^{K} P(X=k)\log\frac{P(X=k)}{Q(X=k)}. \tag{3.2.7}$$

Wir können (3.2.7) auch schreiben als

$$\begin{aligned}\mathbb{KL}(P,Q) &= \sum_{k=1}^{K} P(X=k)\left[\log P(X=k) - \log Q(X=k)\right]\\ &= \mathbb{E}_{X\sim P}\left[\log P(X=k) - \log Q(X=k)\right] \end{aligned} \tag{3.2.8}$$

[16].

Satz 1. Die Minimierung der Kreuzentropie ist äquivalent zur Maximierung der Log-Likelihood-Funktion.

Beweis. Es sei (ML) ein Multiklassifikationsproblem. Wir führen den Beweis für eine beliebige Klasse $k \in \{1,\dots,K\}$. Sei $Q(\mathbf{x};\boldsymbol{\theta})$ eine Familie von Wahrscheinlichkeitsverteilungen über dem selben Ereignisraum. Dabei sei $Q(\mathbf{y};\boldsymbol{\theta})$ ein Schätzer für die wahre Wahrscheinlichkeitsverteilug P der Daten, wobei $P(\mathbf{x})$ einem Datenpunkt $\mathbf{x}$ die Wahrscheinlichkeit zuordnet, zur Klasse k zu gehören. Dann können wir dementsprechend das Unterproblem $(\mathrm{ML_k})$ definieren als die Frage, mit welcher Wahrscheinlichkeit der gegebene Datenpunkt zur Klasse k gehört. Den Maximum-Likelihood-Schätzer für $(\mathrm{ML_k})$ definieren wir als

$$\begin{aligned}\boldsymbol{\theta}_{(\mathrm{ML_k})} &= \arg\max_{\boldsymbol{\theta}} Q(\mathbb{X};\boldsymbol{\theta})\\ &= \arg\max_{\boldsymbol{\theta}} \prod_{n=1}^{N} Q(\mathbf{x}^n;\boldsymbol{\theta})\\ &= \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} Q(\mathbf{x}^n;\boldsymbol{\theta}), \end{aligned} \tag{3.2.9}$$

wobei $\mathbb{X}$ die Gesamtheit aller gegebenen Datenpunkte $\{\mathbf{x}^1,\dots\mathbf{x}^N\}$ bezeichnet. Im letzten Schritt überführten wir die Maximum-Likelihod in die Log-Likelihood-Funktion. Wir teilen (3.2.7) durch N. Dieser Schritt ändert die Lösungsmenge des Optimierungsproblems nicht. Wir erhalten

$$\begin{aligned}\boldsymbol{\theta}_{(\mathrm{ML_k})} &= \arg\max_{\boldsymbol{\theta}} \sum_{n=1}^{N} \frac{1}{N} Q(\mathbf{x}^n;\boldsymbol{\theta})\\ \boldsymbol{\theta}_{(\mathrm{ML_k})} &= \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbb{X}\sim\hat{P}}\left[Q(\mathbf{x};\boldsymbol{\theta})\right]. \end{aligned} \tag{3.2.10}$$

Es bezeichnet $\hat{P}$ die durch die Daten $\mathbb{X}$ induzierte empirische Verteilung. Der Erwartungswert wird hier durch das empirische Mittel erwartungstreu geschätzt. Wir können die Maximierung der Likelihood-Funktion auch interpretieren als die Minimierung der Abweichung der beiden Wahrscheinlichkeitsverteilungen P und Q. Dies stellen wir mit der KL-Divergenz analog zu (3.2.8) dar und erhalten das Minimierungsproblem

$$\boldsymbol{\theta}_{(\mathrm{ML_k})} = \arg\min_{Q} \mathbb{KL}(\hat{P}, Q) = \mathbb{E}_{X\sim\hat{P}}\left[\log \hat{P}(\mathbf{x}) - \log Q(\mathbf{x})\right]. \tag{3.2.11}$$

Relevant für die Maximierung sind hier nur die Terme, die in Abhängigkeit vom Modell stehen. Wir erhalten somit

$$\begin{aligned}\boldsymbol{\theta}_{(\mathrm{ML_k})} &= \arg\min_{Q} -\mathbb{E}_{X\sim\hat{P}}\left[\log Q(\mathbf{x})\right] \\ &= \arg\min_{Q} -\sum_{k=1}^{K} \hat{P}(\mathbf{x}=k)\log Q(\mathbf{x}=k) \\ &= \arg\min_{Q} \mathbb{H}(\hat{P}, Q). \end{aligned} \tag{3.2.12}$$

□

(vgl. [7, S.131,ff.])

Bemerkung. In Beweis von Satz 1 wurde der Schätzer $\hat{P}$ der wahren Verteilung der Daten eingeführt. Damit stellt die Kostenfunktion eine *Approximation* der wirklichen Abweichung zwischen generierten und wahren Antworten dar.

Nutzt man für Regressionsprobleme üblicherweise die mittlere quadratische Abweichung

$$\mathrm{MQA} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{y}^{\mathbf{n}}_{\mathbf{true}} - \mathbf{y}^{\mathbf{n}}_{\mathbf{pred}})^2 \tag{3.2.13}$$

(vgl. [7, S.134]), so ist die Kreuzentropie für Klassifikationsprobleme die geeignetere Wahl. Das folgende Beispiel illustriert dies.

Beispiel 1. (Vergleich MQA und Kreuzentropie) Es sei ein binäres Klassifikationsproblem gegeben und $\mathbf{x} \in \mathbb{R}$ ein Trainingsdatenpunkt mit $\mathbf{y_{true}}(\mathbf{x}) = 1$. Die durch das Modell erzeugte Vorhersage sei $\mathbf{y_{pred}} = 10^{-5}$. Dann gilt

$$\begin{aligned}\mathrm{MQA} &= (1-10^{-5})^2 = 0,99998 \\ \mathrm{CE_{Bin}} &= -\left[1\log(10^{-5}) + (1-1)\log 10^{-5}\right] = 11,513.\end{aligned}$$

Die (binäre) Kreuzentropie führt also bei einer schlechten Vorhersage zu einer deutlich höheren Bestrafung als die mittlere quadratische Abweichung, was einen besseren Lern-Schritt durch das Optimierungsverfahren ermöglicht.

Die Kostenfunktion in einem neuronalen Netz hat einige bemerkenswerte Eigenschaften. Es konnte gezeigt werden, dass sie unabhängig von ihrer konkreten Definition stets nicht-konvex ist. Einer der Gründe liegt in der Symmetrie der neuronalen Netze. Sind die Neuronen der versteckten Schichten alle mit der gleichen Aktivierungsfunktion ausgestattet, dann können ein Minimum der Kostenfunktion erzeugende Gewichte permutiert werden, sodass sich der selbe Zielfunktionswert ergibt [17, S.1]. Diese verschiedenen Minima sind insbesondere lokal isolierte stationäre Punkte von L, wodurch L auf keinen Fall konvex sein kann.

Ein weiteres interessantes Phänomen ist, dass das Auffinden eines lokalen Minimierers oft bereits ausreichend ist. Es wurde gezeigt, dass bei großen Netzwerken mit vielen versteckten Schichten die meisten lokalen Minima gleich sind und die Wahrscheinlichkeit, ein schlechtes lokales Minimum zu finden mit zunehmender Größe des Netzwerks rasch abnimmt [18, S.1].

3.3 Backpropagation

Die im maschinellen Lernen genutzten Optimierungs-Algorithmen zur Minimierung der Kostenfunktion basieren auf Informationen der Ableitung. Um den Gradienten von L effizient zu bestimmen nutzen wir die **Backpropagation**. Diese Methode verdankt ihren Namen dem Umstand, dass die von den Gewichten hervorgerufenen Fehler vom Output ausgehend von hinten nach vorne durch das Netzwerk zurückverfolgt werden. Die in (3.13) dargestellte Struktur eines neuronalen Netzes entspricht einer Verkettung von Funktionen. Genau dies wird mithilfe der mehrdimensionalen Kettenregel der Differentialrechnung ausgenutzt. Die Methode kann im Prinzip für beliebige Funktionen angewandt werden, im Kontext des maschinellen Lernens beschreibt man mit Backpropagation aber in der Regel die Berechnung des Gradienten der Kostenfunktion. [7, S.204]
Wir untersuchen zur Vereinfachung im Folgenden ein Netzwerk mit genau einer versteckten Schicht. Betrachten wir mit $\mathbf{x}^n$ einen von N Datenpunkten, dann führt unser Netzwerk gemäß (3.13) die Operation

$$\mathbf{y_{pred}^n} = \mathbf{f}^{(2)}(\mathbf{A}^{(2)}\mathbf{f}^{(1)}(\mathbf{A}^{(1)}\mathbf{x^n}))$$

durch. Für eine bessere Übersicht führen wir folgende Notation ein:

$$\begin{aligned}
\mathbf{V} &:= \mathbf{A}^{(1)} \\
\mathbf{W} &:= \mathbf{A}^{(2)} \\
\mathbf{f} &:= \mathbf{f}^{(1)} \\
\mathbf{g} &:= \mathbf{f}^{(2)} \\
\mathbf{a}^n &:= \mathbf{V}\mathbf{x}^n \\
\mathbf{z}^n &:= \mathbf{f}(\mathbf{a}^n) \\
\mathbf{b}^n &:= \mathbf{W}\mathbf{z}^n \\
\mathbf{y_{pred}^n} &:= \mathbf{g}(\mathbf{b}^n)
\end{aligned}$$

Die Parameter unseres betrachteten Netzwerks werden definiert durch die Gewichtsmatrizen $\mathbf{V}$ und $\mathbf{W}$. Wir schreiben $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$ und drücken die Abhängigkeit der Output-Erzeugung von den Parametern mit $\mathbf{y_{pred}^n} = \mathbf{y_{pred}^n}(\boldsymbol{\theta})$ aus. Die über alle Eingangsdaten gemittelte Kreuzentropie ist dann gemäß (3.2.4) gegeben durch

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} L^n(\boldsymbol{\theta}), \tag{3.3.1}$$

mit

$$L^n(\boldsymbol{\theta}) := \sum_{k=1}^{K} (y_{true}^n)_k \log(y_{pred}^n(\boldsymbol{\theta}))_k. \tag{3.3.2}$$

Um $\nabla_{\boldsymbol{\theta}} L$ zu bestimmen, bemerken wir zunächst, dass nach der Summen- und der Faktorregel

$$\nabla_{\boldsymbol{\theta}} L = \frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} L^n \tag{3.3.3}$$

gilt, weshalb wir uns nun mit der Berechnung der einzelnen $\nabla_{\boldsymbol{\theta}} L^n$ befassen. Dazu benutzen wir das folgende Hilfsresultat.

Lemma 1. Es seien die Augangsneuronen softmax-aktiviert, mit anderen Worten es gelte $\mathbf{g}(\mathbf{x}) := \textbf{softmax}(\mathbf{x})$. Dann gilt in unserem Netzwerk

$$\nabla_{b_k^n} L^n = (y_{pred}^n)_k - (y_{true}^n)_k. \tag{3.3.4}$$

Beweis. Es sei t derjenige Index von $\mathbf{y^n_{true}}$ der die 1 enthält. Mit der Definition der Softmax-Funktion aus (3.1.7) erhalten wir

$$\begin{aligned}\nabla_{b_k^n} L^n &= \nabla_{b_k^n} \left(- \log \frac{e^{b_t^n}}{\sum_{l=1}^K e^{b_l{}^n}} \right) \\ &= \nabla_{b_k^n} \log \sum_{l=1}^K e^{b_l^k} - \nabla_{b_k^n} b_t^n.\end{aligned}$$

Mit den Ableitungsregeln für den Logarithmus folgt

$$\begin{aligned}\nabla_{b_k^n} L^n &= \frac{1}{\sum_{l=1}^K e^{b_l^n}} \nabla_{b_k^n} \sum_{l=1}^K e^{b_l^n} - \nabla_{b_k^n} b_t^n \\ &= \frac{e^{b_k^n}}{\sum_{l=1}^K e^{b_l^n}} - \nabla_{b_k^n} b_t^n \\ &= \text{softmax}(\mathbf{b}^n)_k - \mathbf{1}_{\{t=k\}} \\ &= (y_{pred}^n)_k - (y_{true}^n)_k.\end{aligned}$$

[19] □

Es bezeichne $\mathbf{W}_k$ die k-te Zeile von $\mathbf{W}$. Gemäß der Kettenregel erhalten wir

$$\nabla_{\mathbf{W}_k} L^n = \frac{\partial L^n}{\partial b_k^n} \nabla_{\mathbf{W}_k} b_k^n = \frac{\partial L^n}{\partial b_k^n} \mathbf{z}^n, \tag{3.3.5}$$

wobei wir ausgenutzt haben, dass aus der Definition von $\mathbf{b}^n$ folgt, dass $b_k^n = (\mathbf{W}_k)^\top \mathbf{z}^n$ und damit $\nabla_{\mathbf{W}_k} b_k^n = \nabla_{\mathbf{W}_k} (\mathbf{W}_k)^\top \mathbf{z}^n = \mathbf{z}^n$. Mit (3.3.4) folgt

$$\delta_k^{nw} := \frac{\partial L^n}{\partial b_k^n} = (y_{pred}^n)_k - (y_{true}^n)_k, \tag{3.3.6}$$

und wir erhalten das in der zweiten Schicht durch $\mathbf{W}_k$ erzeugte Fehlersignal

$$\nabla_{\mathbf{W}_k} L^n = \delta_k^{nw} \mathbf{z}^n. \tag{3.3.7}$$

$\boldsymbol{\delta}^{nw}$ gibt demnach an, wie sensitiv die Output-Signale der Ausgangsschicht auf Änderungen der Gewichte der Matrix $\mathbf{W}$ reagiert. Analog ergibt sich für die Inputschicht

$$\nabla_{\mathbf{V}_j} L^n = \frac{\partial L^n}{\partial a_j^n} \nabla_{\mathbf{V}_j} a_j^n := \delta_j^{nv} \mathbf{x}^n, \tag{3.3.8}$$

wobei analog zu oben $a_j^n = (\mathbf{V}_j)^\top \mathbf{x}^n$ genutzt wurde und $\mathbf{V}_j$ die j-te Zeile von $\mathbf{V}$ bezeichnet. Das von $\mathbf{V}_j$ in der ersten Schicht erzeugte Fehlersignal δ_k^{nv} errechnet sich nun aus

$$\delta_j^{nv} = \frac{\partial L^n}{\partial a_j^n} = \sum_{k=1}^K \frac{\partial L^n}{\partial b_k^n} \frac{\partial b_k^n}{\partial a_j^n} = \sum_{k=1}^K \delta_k^{nw} \frac{\partial b_k^n}{\partial a_j^n} \tag{3.3.9}$$

und δ_j^{nv} gibt an, wie sensitiv sich die Output-Signale bei Änderungen der Gewichtsmatrix $\mathbf{V}$ verhalten. Mit (3.1.1) ergibt sich

$$b_k^n = \sum_j w_{kj} f_j(a_j^n) \tag{3.3.10}$$

und damit

$$\frac{\partial b_k^n}{\partial a_j^n} = w_{kj} f_j'(a_j^n). \tag{3.3.11}$$

Einsetzen in (3.3.9) liefert

$$\delta_j^{nv} = \sum_{k=1}^{K} \delta_k^{nw} w_{kj} f_j'(a_j^n). \tag{3.3.12}$$

Insgesamt erhalten wir

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) &= \left[\nabla_{\mathbf{V}} L(\boldsymbol{\theta})\,, \nabla_{\mathbf{W}} L(\boldsymbol{\theta})\right]^\top \\ &= \frac{1}{N} \sum_{n=1}^{N} \left[\nabla_{\mathbf{V}} L^n(\boldsymbol{\theta})\,, \nabla_{\mathbf{W}} L^n(\boldsymbol{\theta})\right]^\top \\ &= \frac{1}{N} \sum_{n=1}^{N} \left[\boldsymbol{\delta}^{nv} \mathbf{x}^n,\, \boldsymbol{\delta}^{nw} \mathbf{z}^n\right]^\top . \end{aligned} \tag{3.3.13}$$

(vgl. [16, S.569,ff.]) In (3.3.12) können wir zwei wesentliche Eigenschaften der Backpropagation erkennen. Zum einen wird der in der Inputschicht produzierte Fehler mittels der Fehler in der zweiten Schicht und der zugehörigen Gewichtsmatrix berechnet. D.h. wir bewegen uns, wie eingangs beschrieben, von hinten nach vorne durch das Netzwerk. Zudem nutzten wir dazu für jedes Neuron j der ersten Schicht nur die Informationen bezüglich seiner unmittelbaren Nachbarn, was zur Effizienz des Algorithmus maßgeblich beiträgt. Die Backpropagation wird im Zusammenspiel mit der Berechnung der Vorhersage des Modells eingesetzt. Letztere wird beginnend mit dem Eingangssignal sukzessive durch die Komposition vieler Funktionen berechnet. Im Anschluss daran wird der durch die Vorhersage am Ausgangssignal entstandene Fehler auf dem selben Pfad zurückverfolgt. Die Berechnung der Vorhersage $\mathbf{y}^{\mathbf{n}}_{\mathbf{pred}}$ nennt man in diesem Zusammenhang auch Forwardpropagation. [16, S.571]

3.4 Das stochastische Gradientenverfahren

Das Training eines Machine-Learning-Systems kann mathematisch beschrieben werden als die Lösung der unrestringierten Optimierungsaufgabe

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{3.4.1}$$

mit

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} L^n(\boldsymbol{\theta}).$$

Das Lernen des Machine-Learning-Systems ist gleichbedeutend zur iterativen Anpassung der Gewichte mittels eines Optimierungsverfahrens zur Lösung von (3.4.1). Eine naheliegende Methode um $\boldsymbol{\theta}^*$ zu berechnen ist das Gradientenverfahren. Angewandt auf (3.4.1) ist die entsprechende Rechenvorschrift

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t), \tag{3.4.2}$$

wobei die Schrittweite $\eta > 0$ im Kontext des maschinellen Lernens die **Lernrate** genannt wird. Wesentlich ist hier die Tatsache, dass in jeder Iteration *alle* Trainingsdaten benutzt werden, um die Gewichte anzupassen. Die Rechenvorschrift (3.4.2) wird daher auch Batch Gradient Descent genannt. Um ein Modell mit einer hohen Verallgemeinerungsfähigkeit zu erhalten, benötigt man in der Praxis für dessen Training eine sehr hohe Anzahl von bis zu Millionen von Trainingsdaten. Daher ist die Berechnung von $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t)$ sehr rechenaufwendig. Genauer: die Rechenzeit liegt in $\mathcal{O}(N)$ [7, S.152]. Das klassische Gradientenverfahren ist somit in der Regel keine praktikable Methode für Machine-Learning-Probleme und muss durch Algorithmen ersetzt werden, die einen geringeren Aufwand bedeuten. Weit verbreitet sind hier verschiedene Varianten des **stochastischen Gradientenverfahrens** (SGD). Die Schlüsselidee liegt darin, den Gradienten $\nabla_{\boldsymbol{\theta}} L$ nicht expilizit auszurechnen, sondern ihn zu schätzen. Dazu wird nicht mehr wie in (3.4.2) in jeder Iteration der Durchschnitt über alle Trainingsdaten $n \in \{1, \ldots, N\}$ gebildet, sondern pro Schritt nur noch ein einzelner, zufällig ausgewählter Datenpunkt betrachtet. Die resultierende Rechenvorschrift wird zu

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} L^n(\boldsymbol{\theta}_t),$$

wobei n zufällig gleichverteilt aus $\{1, \ldots, N\}$ ausgewählt wird. Im Gegensatz zum klassischen Gradientenverfahren ist es bei SGD aufgrund der Auswertung von nur *einem* Datenpunkt pro Iteration keineswegs gesichert, dass das Verfahren nur Abstiegsrichtungen erzeugt [10, S.10]. Allerdings ist leicht zu erkennen, dass wegen

$$\begin{aligned} \mathbb{E}[\nabla_{\boldsymbol{\theta}} L^n] &= \sum_{n=1}^{N} \frac{1}{N} \nabla_{\boldsymbol{\theta}} L^n \\ &= \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} L^n \\ &= \nabla_{\boldsymbol{\theta}} L, \end{aligned} \tag{3.4.3}$$

$\nabla_{\boldsymbol{\theta}} L^n$ ein erwartungstreuer Schätzer von $\nabla_{\boldsymbol{\theta}} L$ ist. Eine häufig anzutreffende Variante des SGD ist das Mini-Batch-Gradientenverfahren, bei welchem in jedem Schritt nicht nur ein einzelner Datenpunkt sondern $N' \ll N$ viele Datenpunkte ausgewertet werden. Es werden hierbei in jeder Iteration N' viele Indizes $\{k^1, \ldots, k^{N'}\}$ zufällig und gleichverteilt aus $\{1, \ldots, N\}$ gewählt und die Rechenvorschrift

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \frac{\eta}{N'} \sum_{n=1}^{N'} \nabla_{\boldsymbol{\theta}} L^{k^n}(\boldsymbol{\theta}_t) \tag{3.4.4}$$

ausgeführt. Man bezeichnet die Menge $\{x^{k^1}, \ldots, x^{k^{N'}}\}$ der verwendeten Datenpunkten als Mini-Batch. Batch-Gradient-Descent hat gegenüber SGD ein besseres Konvergenzverhalten bezüglich lokaler Minima, daher stellt das Mini-Batch-Gradientenverfahren einen Kompromiss dar zwischen dem guten Konvergenzverhalten des klassischen Gradientenverfahrens und der numerischen Effizienz des SGD [20].
In der Praxis werden SGD und das Mini-Batch-Gradientenverfahren oft unter dem Begriff SGD zusammengefasst. Im Kontext des Deep Learning nennen wir eine vollständige Iteration des Algorithmus, in dem Forwardpropagation, Backpropagation und die Anpassung der Gewichtsparameter durchgeführt werden, eine **Epoche**.

3.5 Optimierer

TensorFlow bietet eine Auswahl diverser Optimierungsalgorithmen für das Training eines Deep-Learning-Modells an. Um diese besser verstehen zu können, gehen wir zunächst kurz

auf den Begriff des **(klassischen) Moments** ein. Dabei handelt es sich um eine Technik zur Verbesserung des SGD-Verfahrens. Dazu wird der exponentiell gleitende Durchschnitt über alle bereits berechneten Gradienten der Zielfunktion gebildet und in einem Geschwindigkeitsvektor $\mathbf{v}$ gespeichert. Dieser wird dann für das Update der Parameter eingesetzt. Das Ziel dieser Methode ist die Verringerung des Hin- und Her-Springens des SGD-Verfahrens, welches aufgrund der Betrachtung einzelner (oder weniger) Datenpunkte auftreten kann. Sei $L = L(\boldsymbol{\theta})$ eine Kostenfunktion der Parameter $\boldsymbol{\theta}$. Dann wird das Moment für das klassische Gradientenverfahren über die Rechenvorschriften

$$\mathbf{v}_{t+1} := \boldsymbol{\mu}\mathbf{v}_t - \eta\nabla L(\boldsymbol{\theta}_t) \tag{3.5.1}$$

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \mathbf{v}_{t+1} \tag{3.5.2}$$

implementiert, wobei $\boldsymbol{\mu} \in [0, 1]$ der Momentenkoeffizient ist und $\eta > 0$ die Lernrate. [21, S.2],[7, S.296]
Eine abgewandelte Form des Moments ist das **Nesterov-Moment**, welches aus Nesterovs beschleunigtem Gradientenverfahren abgeleitet werden kann. Es ist durch

$$\mathbf{v}_{t+1} := \boldsymbol{\mu}\mathbf{v}_t - \eta\nabla L(\boldsymbol{\theta}_t + \boldsymbol{\mu}\mathbf{v}_t) \tag{3.5.3}$$

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \mathbf{v}_{t+1} \tag{3.5.4}$$

gegeben. Im Unterschied zum klassischen Moment wird der Gradient der Zielfunktion nach Anwendung des aktuellen Geschwindigkeitsvektors ausgewertet, was als eine Korrektur des klassischen Moments gedeutet werden kann. [21, S.2,f],[7, S.300]

Im Rahmen dieser Arbeit wurde mit mehreren der in TensorFlow bereitgestellten Optimierungsverfahren gearbeitet. Dazu zählen das klassische SGD-Verfahren ohne Moment (diese Variante wird auch **SGD vanilla** genannt) und mit Moment (klassisches Moment und/oder Nesterov-Moment).
Darüberhinaus wurden sogenannte **adaptive** Optimierungsverfahren verwendet. Diese zeichnen sich dadurch aus, dass im Gegensatz zu SGD die Lernrate nicht fix sondern variabel ist. Experimentiert wurde mit dem **RMSprop**-Algorithmus (vgl. [22]), dem **ADAM**-Optimierer (vgl. [23], [24]) und dem ADAM-Optimierer mit Nesterov-Moment, kurz: **NADAM** (vgl. [25]). Neben der variablen Lernrate ist eine weitere wichtige Eigenschaft dieser Optimierer die Verwendung von Momenten. Um die numerischen Schwierigkeiten aufgrund der Berücksichtigung *aller* zu einem früheren Zeitpunkt berechneten Gradienten zu umgehen, werden hier aber nur eine bestimmte Anzahl vergangener Gradienten in die Berechnung des Parameter-Updates miteinbezogen. [26]

3.6 Regularisierung

Trainieren wir ein Modell, so wünschen wir uns im Ergebnis seine Fähigkeit, *neue* Daten, die es zuvor noch nie gesehen hat, korrekt vorherzusagen. Diese Eigenschaft nennen wir **Generalisierung**. In diesem Zusammenhang ist der Begriff der **Überanpassung** von zentraler Bedeutung. Er beschreibt ein perfektes Vorhersagen der Trainingsdaten bei gleichzeitigem Versagen bzgl. neuer Daten. (Analog hierzu nennen wir ein Modell, das selbst die Trainingsdaten nicht gut vorhersagen kann, **unterangepasst**.) Eine Überanpassung kann auftreten, wenn das Modell zu komplex ist bzw. seine Kapazität (d.h. seine Fähigkeit, möglichst viele Funktionen nachbilden zu können) zu groß ist. Um die gute Generalisierungsfähigkeit eines Modells sicherzustellen, reichen die bisher vorgestellten Methoden noch nicht aus. Sie müssen um eine **Regularisierung** erweitert werden. Es gibt viele Regularisierungstechniken. Sie haben alle das Ziel, die Kapazität eines Modells zu verringern. Wir konzentrieren uns auf einige wesentliche Konzepte, die im Rahmen dieser Arbeit genutzt wurden. [7, S.110]

3.6.1 L2- und L1-Regularisierung

Diese Methoden beruhen auf einer Anpassung der Kostenfunktion L. Es wird eine von den Parametern $\boldsymbol{\theta}$ abhängige Straffunktion $\Omega = \Omega(\boldsymbol{\theta})$ zur eigentlichen Kostenfunktion $L = L(\boldsymbol{\theta})$ hinzuaddiert. Es lässt sich die Regularisierung durch Parameter-Norm-Bestrafung in seiner allgemeinen Form als

$$\tilde{L}(\boldsymbol{\theta}) := L(\boldsymbol{\theta}) + \alpha\Omega(\boldsymbol{\theta}) \tag{3.6.1}$$

schreiben, wobei der Regularisierungsfaktor α ein zu wählender Hyperparameter ist, der den Einfluss des Strafterms auf die neue Kostenfunktion $\tilde{L}$ gewichtet. Verwenden wir als Straffunktion eine Norm der Gewichtsparameter, so kann auf diese Weise eine zu hohe Komplexität des Modells bestraft werden, da die Straffunktion dann mit der Anzahl der Parameter wächst.[7, S.230]
Wir betrachten wieder den Fall ohne Bias-Vektoren, sodass $\boldsymbol{\theta}$ gerade die Gesamtheit aller Gewichte eines Modells beschreibt. Für die **L2-Regularisierung** erhalten wir damit die Vorschrift

$$\tilde{L}(\boldsymbol{\theta}) := L(\boldsymbol{\theta}) + \frac{\alpha_{L2}}{2}||\boldsymbol{\theta}||_2^2. \tag{3.6.2}$$

[7, S.230] Im Fall der **L1-Regularisierung** gilt

$$\tilde{L}(\boldsymbol{\theta}) := L(\boldsymbol{\theta}) + \alpha_{L1}||\boldsymbol{\theta}||_1. \tag{3.6.3}$$

[7, S.234]

3.6.2 Dropout

Diese Regularisierungsmethode beruht auf dem zufälligen Ausschalten von Neuronen. Dies wird über eine binäre Maske $\boldsymbol{\mu}$ realisiert, die in jeder Trainingsepoche auf zuvor festgelegte Schichten angewandt wird und angibt, welche der Neuronen inkludiert bleiben. Für jede Iteration wird die Maske zufällig neu geladen, sodass jedes Neuron mit einer Wahrscheinlichkeit p mitsamt seinen Verbindungen aus dem Netzwerk entfernt wird. Diese Wahrscheinlichkeit heißt **Dropout-Rate** und ist ein Hyperparameter, der vor dem Training festgelegt wird. Das Ziel ist die Minimierung von

$$\mathbb{E}_{\boldsymbol{\mu}} L(\boldsymbol{\theta}, \boldsymbol{\mu}), \tag{3.6.4}$$

wobei $L(\boldsymbol{\theta}, \boldsymbol{\mu})$ der nach Anwendung der Maske $\boldsymbol{\mu}$ resultierende Wert der Kostenfunktion L ist. Obwohl der obige Erwartungswert exponentiell viele Möglichkeiten für die Bildung von $\boldsymbol{\mu}$ beinhaltet, können wir einen erwartungstreuen Schätzer durch das Auswerten einer (kleinen) Teilmenge zufällig ausgewählter Realisierungen von $\boldsymbol{\mu}$ erhalten. [7, S.258,f]
Das Resultat ist ein spärlicher besetztes Netzwerk, dessen Schichten robuster, also unabhängiger voneinander, die Feature-Erkennung und die Klassifikation generieren können [27, S.4].

3.7 TensorFlow

In unseren bisherigen Betrachtungen haben wir uns auf die Repräsentation der Daten in Form von Vektoren und Matrizen beschränkt. Für viele Probleme wäre dies jedoch nicht praktikabel. Haben wir beispielsweise ein Rot-Grün-Blau-Bild gegeben, so ist es naheliegender für jeden der drei Kanäle eine Matrix mit den Helligkeit der Bildpixel als Einträge zu generieren. Dann bilden die drei Matrizen zusammen die dem Algorithmus zugeführte Repräsentation des Bildes. Eine solche Struktur ist ein 3-dimensionaler **Tensor**.

Allgemein können wir einen Tensor als eine Verallgemeinerung von Skalaren, Vektoren und Matrizen auf höhere Dimensionen verstehen. Ein Skalar ist ein Tensor der Dimension 0. Ein Vektor ist ein Tensor der Dimension 1 und eine Matrix ein Tensor mit Dimension 2. In der Praxis des maschinellen Lernens werden die Daten grundsätzlich in Form von Tensoren mittels Arrays gepeichert. In Python bedient man sich hierfür multidimensionaler Numpy-Arrays. [6, S.31,f.]

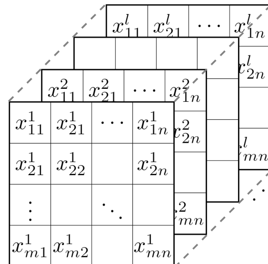


Abbildung 4: 3-dimensionaler Tensor (vgl. [28])

TensorFlow ist eine von Google entwickelte frei verfügbare Software-Bibiliothek, welche es dem Nutzer erlaubt, mit vergleichsweise wenig Programmcode komplexe Machine-Learning-Modelle zu erstellen, zu speichern und zu verwenden. Weit verbreitet ist insbesondere die Anwendung im Deep Learning durch die Programmierung von neuronalen Netzen. Der Name der Bibliothek lässt sich aus der ursprünglich zugrundeliegenden Arbeitsweise herleiten: Ein Deep-Learning-Modell entsteht durch die Konstruktion eines Berechnungsgraphen, welcher aus Tensoren und mathematischen Operationen besteht. Dieser wird im Rahmen einer Sitzung (session) anschließend ausgeführt. Mit der Version 1.5 wurde dieser Vorgang mithilfe der Eager Execution effizienter gestaltet. Diese ermöglicht eine direkte Eingabe und Ausführung der in Python eingegebenen TensorFlow-Codes. Damit können die TensorFlow-Befehle und reine Python-Operationen reibungslos miteinander arbeiten (siehe Abb. 5). Erst diese Arbeitsweise ermöglichst die zuvor erwähnte Behandlung der Tensoren als Numpy-Arrays. Mit der Version 2.0 kam die Integration der von François Chollet entwickelten Keras-Bibliothek in TensorFlow hinzu. [29, S.7]
Diese Programmierschnittstelle bietet den Vorteil, aufgrund ihrer einfachen Struktur sowohl für Experten als auch für Anfänger geeignet zu sein. [29, S.28]

```
import tensorflow as tf
import numpy as np

A = [[1, 0, 0],[-1, 0, 2]]
x = [[1],[1],[0]]
b = tf.matmul(A,B)
print(C)

tf.Tensor(
[[ 1]
 [-1]], shape=(2, 1), dtype=int32)
```

Abbildung 5: Beispiel Verträglichkeit TensorFlow und Python-Code

Mit dem TF-Befehls „matmul“ lassen problemlos Numpy-Arrays manipulieren.

4 Convolutional Neural Networks

Das Prinzip eines **faltenden neuronalen Netzes** (Convolutional Neural Network) findet vor allem in der Bildverarbeitung häufige Anwendung und führt dort oft zu hervorragenden Ergebnissen. Wie der Name suggeriert, basiert die Methode auf dem Prinzip der mathematischen Faltung.

4.1 Faltung

Die Faltung ist im Kontext allgemeiner Funktionen gegeben durch

$$(x * y)(t) := \int x(\tau)w(t-\tau)d\tau. \tag{4.1.1}$$

Es bezeichnet hierbei x ein von t abhängendes Eingangssignal welches mit dem sogenannten **Kernel** w gefaltet wird. Die Faltung ist eine spezielle lineare Funktion, welche eine große Rolle in der Signalverarbeitung spielt. Sie hat viele günstige algebraische Eigenschaften. Sie ist unter anderem linear, kommutativ, assoziativ und distributiv. [30, S.28]

Das diskrete Analogon lautet

$$(x * y)(t) := \sum_{k=-\infty}^{\infty} x(k)w(t-k). \tag{4.1.2}$$

Im Falle eines Eingangsbildes $\mathbf{I}$ liegt ein 2-dimensionales Gitter aus lauter Pixeln vor. Daher nutzt man hierfür einen 2-dimensionalen Kernel K und die entsprechende Verallgemeinerung der in (4.1.2) beschriebenen 1-dimensionalen Version. Sie lautet

$$(\mathbf{I} * \mathbf{K})(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{4.1.3}$$

Das Ergebnis einer Faltung von zwei Funktionen ergibt stets wieder eine Funktion, analog erhalten wir bei der Faltung einer Pixelmatrix mit einem 2-dimensionalen Kernel (also einer Matrix) wieder eine Pixelmatrix. Die Faltung ist kommutativ, d.h.

$$(\mathbf{I} * \mathbf{K})(i,j) = (\mathbf{K} * \mathbf{I})(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{4.1.4}$$

Die in Argument von $\mathbf{I}$ durchgeführten Subtraktionen bewirken eine Spiegelung des Kernels relativ zum Input $\mathbf{I}$. Dies ist wesentlich für die Kommutativität von $*$. In der Praxis der Bildverarbeitung wird diese Spiegelung, der Kernel-Flip, oft nicht durchgeführt. Die resultierende Operation

$$(\mathbf{I} * \mathbf{K})(i,j) = (\mathbf{K} * \mathbf{I})(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{4.1.5}$$

stellt deswegen streng genommen keine Faltung dar, wird in der Regel dennoch so genannt. Mathematisch korrekt bezeichnet man (4.1.5) als Kreuzkorrelation. Die diskrete Faltung können wir interpretieren als die Multiplikation einer Matrix mit einer anderen Matrix. [7, S.330,ff.]

Beispiel 2. (Faltung einer Matrix) Wir betrachten eine Bildmatrix $I \in \mathbb{R}^{7\times 7}$, welche mit dem Kernel $K \in \mathbb{R}^{3\times 3}$ gefaltet wird. Für jeden Pixel von I (außer den äußersten Zeilen und Spalten) werden folgende Schritte ausgeführt:

i. Der Kernel liegt so auf dem Pixel, dass dieser in der Mitte der Kernel-Matrix liegt.

ii. Jeder von dem Kernel überdeckte Pixel von I wird mit dem Wert des ihn überdeckenden Kernel-Eintrags multipliziert.

iii. Die daraus resultierenden Produkte werden alle aufaddiert.

iv. In der neuen Matrix $I * K$ wird ein neuer Pixel mit dem Wert dieser Summe erzeugt.

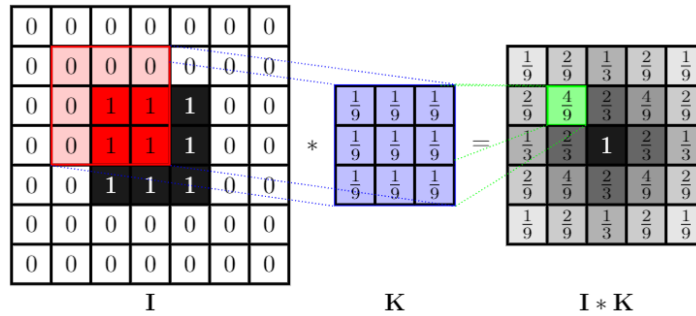


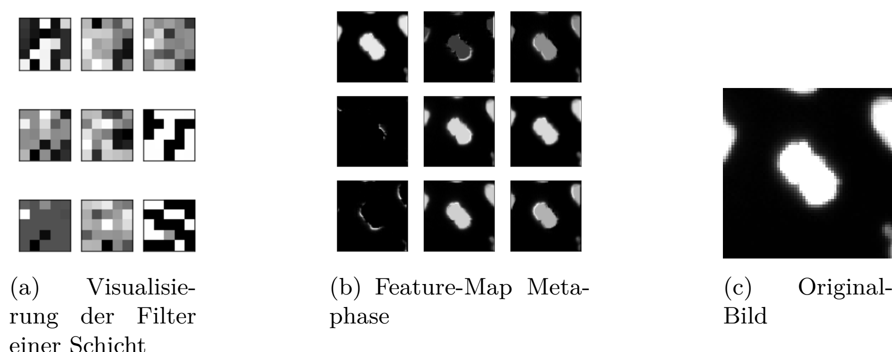
Abbildung 6: Beispiel einer 2-dimensionalen Faltung ohne Kernel-Flip (vgl. [31],[32])

Bei dem oben betrachteten Kernel handelt es sich um einen Glättungsfilter, der im Ergebnis das Originalbild unschärfer werden lässt.

Sei $I \in \mathbb{R}^{n \times n}$ eine Matrix und $K \in \mathbb{R}^{k \times k}$ ein Kernel. Dann gilt $I * K \in \mathbb{R}^{m \times m}$, wobei die Output-Dimension m gemäß

$$m = n - k + 1$$

gegeben ist [7, S.349].



(a) Visualisierung der Filter einer Schicht

(b) Feature-Map Metaphase

(c) Original-Bild

Abbildung 7: Beispiele zur Faltung

In Abbildung 7 (a) sehen wir einige der Filter der ersten Schicht eines einfachen Deep-Learning-Modells, welches dazu trainiert wurde, zwischen den beiden Klassen „Interphase“ und „Metaphase“ zu unterscheiden. Die Einträge der in diesem Fall 5×5-Kernel wurden hier mittels verschiedener Grautöne visualisiert. Das Ergebnis der Faltung mit diesen Filtern nennen wir die **Feature-Map**. In (b) sehen wir einen Ausschnitt der erzeugten Feature-Map. Insgesamt enthält die zugrundeliegende Faltungsschicht 64 solcher Filter, mit jeweils unterschiedlicher Konfiguration. Das ein-kanalige Originalbild zeigt eine Zelle in der Metaphase.
Die Funktionsweise eines faltenden neuronalen Netzes kann zusammenfassend beschrieben werden als das Erlernen nützlicher Kernel, mittels derer gehaltvolle Informationen aus den Daten extrahiert werden können.
Es gibt mehrere wesentliche Vorteile, die eine solche Methodik gegenüber einem klassischen neuronalen Netz ohne Faltung hat. Zum einen reduziert sich die Anzahl der Modellparameter dramatisch, da die Kernel in der Regel deutlich kleiner sind als die Eingangsmatrizen, d.h. die den Kernel repräsentierende Matrix ist spärlich besetzt. Dies führt zu einer deutlichen Verbesserung der Laufzeit. Ein weiterer Aspekt ist die Tatsache, dass ein und der selbe Kernel auf verschiedene Einträge der Inputmatrix angewandt wird. Das bedeutet, dass die gleichen Parameter mehrmals verwendet werden, was in einem Gegensatz zu traditionellen neuronalen Netzen steht, bei denen ein Gewicht für genau einen Eintrag verwendet wird und dann nie wieder. [7, S.330,ff.]

4.2 Padding

In Beispiel 2 konnten wir ein Problem beobachten, das bei der Faltung einer Matrix auftritt: Die Ausgangsmatrix ist kleiner als das Original. Um dies zu verhindern, bedienen wir uns des sogenannten **zero paddings**, d.h. es werden an den Rändern der Eingangsmatrix Nullen hinzugefügt, was es dem Kernel erlaubt auch teilweise außerhalb der eigentlichen Eingangsmatrix zu operieren. Würde man dies nicht tun, so würde unweigerlich nach jedem Faltungsfilter das Ausgangssignal immer weiter in der Dimension schrumpfen und man müsste die Kernel-Größe stets so klein wie möglich wählen, wodurch keine komplexeren faltenden Netze denkbar wären. Diesen Fall nennt man **valid padding**. Fügt man die Nullen in der Art und Weise zur Eingangsmatrix hinzu, dass die Ausgangsmatrix nach der Faltung gerade die selbe Größe hat, so sprechen wir von **same padding**. [31]

Beispiel 3. Gegeben sei eine Input-Matrix $I \in \mathbb{R}^{5\times5}$ mit

$$I = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0 & 0 \end{pmatrix}$$

Dann hat die durch das Padding generierte neue Matrix I' die Dimension 7×7.

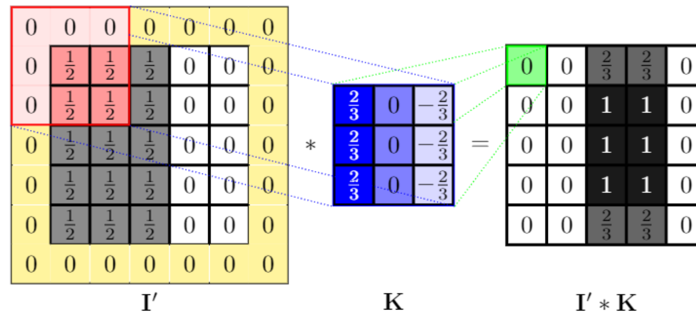


Abbildung 8: Beispiel Padding (vgl. [31])

Bei dem in Abb. 8 gezeigten Kernel handelt es sich um einen auf die Erkennung von Kanten spezialisierten Filter, welcher Übergänge von dunkel nach hell (von links nach rechts) hervorhebt.

4.3 Pooling

Die Pooling-Operation wird typischerweise im Anschluss einer Faltung durchgeführt. Die durch die Faltung entstandene Matrix wird so modifiziert, dass ihre Einträge gruppenweise in einer bestimmten Weise zusammengefasst werden. Typische Beispiele sind das Bilden des Durchschnittswerts aller in einer gewissen Nachbarschaft liegender Einträge (**Average Pooling**) oder das Bilden des Maximums ihrer Werte (**Max Pooling**).
Pooling bewirkt eine stärkere Invarianz der Datenrepräsentationen gegenüber kleinen Verschiebungen der Eingangswerte. Dies ist in vielen Objektklassifikations- Problemen eine wünschenswerte Eigenschaft [7, S.342]. Ist z.b. ein Bild mit einer Zelle in der Interphase gegeben, so sind für die Klassifizierung u.A. Helligkeit und Glattheit des Objekts sowie seine runde Form entscheidend für die Einteilung in die Klasse „Interphase“, jedoch nicht die Frage, ob sich die Zelle exakt in der Mitte oder ein klein wenig weiter links oder rechts davon befindet.
Jedoch gibt es auch Problemstellungen, wie die Objekterkennung, bei denen die genaue Position eines Objektes in einem Bild erfasst werden soll. Ein Modell mit Max-Pooling-Schichten nimmt deswegen implizit bereits an, dass es verschiebungsinvariante Operationen erlernen soll. Eine Pooling Operation reduziert die Dimension des Eingangssignals. Werden in jedem Schritt des Poolings k Einträge miteinander verrechnet und werden alle Nachbarschaften disjunkt abgearbeitet, dann hat das entstandene Ausgangssignal k-mal weniger Einträge. Damit reduziert sich der Rechenbedarf für die dahinterliegende Schicht. [7, S.342]
Die generische Grundstruktur eines faltenden Neuronalen Netzes können wir wie folgt zusammenfassen. Die Faltungsschichten zusammen mit den jeweils dahinterliegenden Pooling-Schichten sind für die Erkennung sinnhafter Strukturen des Eingangsbildes zuständig,

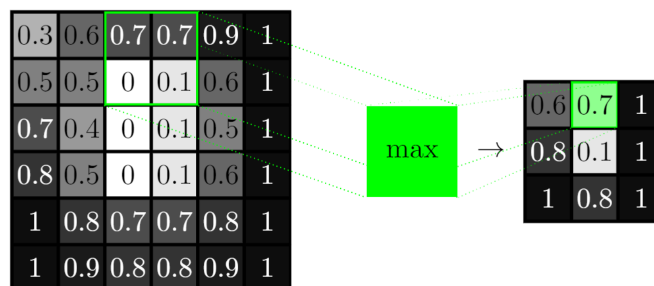


Abbildung 9: Beispiel MaxPooling mit Schrittweite 2 (vgl. [31])

während die dahinterliegenden **dichten Schichten** (dense/fully connected layers) zusammen mit der Ausgangsschicht die Klassifizierung auf der Grundlage der zuvor erkannten Features durchführen. [33]
Dichte Schichten sind eindimensional, d.h. sie bestehen aus genau einer langen Folge von Neuronen. Da das Ausgangssignal der letzten Faltungsschicht im Allgemeinen nicht eindimensional ist, ist an dieser Stelle eine Transformation ihres Ausgangssignals notwendig, welche als das **Flatten** bezeichnet wird. Der durch die letzte Faltungsschicht erzeugte Datentensor wird in einen langen Vektor umgewandelt, welcher dann an die dahinterliegende dichte Schicht weitergegeben werden kann. [34]

4.4 Einführendes Beispiel

Ein einfaches aber effektives Modell zur Klassifikation einer zentriert im Bild liegenden Zelle in die beiden Klassen „Interphase" und „Metaphase" wurde bereits in Kapitel 4.1 erwähnt. Erstellt wurde es für die Klassifikation von einkanaligen Schwarz-Weiß-Bildern, die aus dem Herausfiltern des Blau-Anteils der ursprünglich farbigen Abbildungen entstanden sind. Das Modell kommt ohne Pooling-Schichten aus und besteht aus den vier Faltungsschichten conv1, conv2, conv3, conv4 und den vier dichten Schichten dense1, dense2, dense3 und output. Hinter jeder zweiten Filterschicht und nach jeder versteckten dichten Schicht wurde ein Dropout nachgeschaltet mit einem Faktor von 0.05. Als Größe des in Kapitel 3.5 beschriebenen Batch wurde eine Anzahl von 8 Bildern festgelegt. Für die Aktivierung der Output-Neuronen wurde die Softmax-Funktion gewählt. Alle anderen Neuronen sind ReLU-aktiviert. In TensorFlow wird die Regularisierung der Gewichte für jede Schicht einzeln über den jeweiligen Regularisierungskoeffizienten konfiguriert und die resultierenden Strafterme auf die Kostenfunktion hinzuaddiert (vgl. [35]). In unserem Fall wurde in jeder Faltungsschicht und in jeder dichten Schicht eine L1-Regularisierung mit dem Faktor $\alpha_{L1} = 10^{-5}$ und eine L2-Regularisierung mit dem Hyperparameter $\alpha_{L2} = 10^{-4}$ festgeschrieben. Darüber hinaus wurden bei der Konstruktion des Modells weitere Hyperparameter konfiguriert, für jede Faltungsschicht unter anderem

die Anzahl der Kernel	K
die Größe der Kernel	F
die Schrittweite der Kernel	S
die Art des Paddings	P

und für jede der dichten Schichten

die Anzahl der Neuronen $\quad N$.

Folgende Werte wurden bei der Erstellung des Modells festgelegt:

Schicht	K	F	S	P	N
conv1	64	5	4	P_{same}	
conv2	32	5	4	P_{same}	
conv3	32	5	4	P_{same}	
conv4	32	5	4	P_{same}	
dense1					256
dense2					256
dense3					256
output					2

Obwohl es sich bei unserem Problem um ein binäres Klassifikationsproblem handelt, haben wir hier nicht ein Neuron, welches zwischen den beiden Klassen durch die Werte „0“ und „1“ unterscheidet sondern zwei Neuronen (eines für jede Klasse).
Die Form jeder Faltungsschicht ist bestimmt durch ihre Outputgröße O und die Anzahl K ihrer Kernel mit

$$(O \times O \times K).$$

Jeder der Kernel ist quadratisch und von der Größe $(F \times F)$. Damit lässt sich die Outputgröße O durch die Formel

$$O = \frac{I - F + 2 \cdot P}{S} + 1 \tag{4.4.1}$$

bestimmen. In unserem Fall gilt $P = P_{same}$ und P_{same} ist durch

$$P_{same} = \lfloor \frac{S\lceil \frac{I}{S} \rceil - I + F - S}{2} \rfloor \tag{4.4.2}$$

gegeben. Für conv1 folgt

$$\begin{aligned} P_{same} &= \lfloor \frac{4\lceil \frac{64}{4} \rceil - 64 + 5 - 4}{2} \rfloor \\ &= \frac{1}{2}. \end{aligned}$$

Damit erhalten wir für die Output-Größe von conv1

$$\begin{aligned} O &= \frac{256 - 5 + 2 \cdot \frac{1}{2}}{4} + 1 \\ &= 64 \end{aligned}$$

Insgesamt haben die Faltungsschichten des Modells folgende Formen:

conv1: (64,64,64)
conv2: (16,16,32)
conv3: (4,4,32)
conv4: (1,1,32)

Wir bezeichnen mit C die Tiefe der Schicht. Beispielsweise hat jedes der Eingangsbilder des Modells eine Tiefe von 1, da sie nur den Blau-Kanal besitzen. Ein RGB-Bild hat demnach eine Tiefe von 3. Die Anzahl d_conv der Parameter jeder Faltungs-Schicht des Modells lassen sich nun mittels der Formel

$$d_{conv} = (F \cdot F \cdot C + 1) \cdot K$$

berechnen, was auf die Werte

$$\begin{aligned} d_{conv1} &= (5 \cdot 5 \cdot 1 + 1) \cdot 64 &= 1664 \\ d_{conv2} &= (5 \cdot 5 \cdot 64 + 1) \cdot 32 &= 51232 \\ d_{conv3} &= (5 \cdot 5 \cdot 32 + 1) \cdot 32 &= 25632 \\ d_{conv4} &= (5 \cdot 5 \cdot 32 + 1) \cdot 32 &= 25632 \end{aligned}$$

führt.

Die Anzahl der Parameter in einer dichten Schicht berechnet sich aus

$$d_{dense} = (N_{in} + 1) + N_{out},$$

wobei N_{in} die Anzahl der Neuronen in der vorangegangenen Schicht bezeichnet und N_{out} gerade die Anzahl der Neuronen der dichten Schicht. Es folgt somit

$$\begin{aligned} d_{dense1} &= (32 + 1) \cdot 256 &= 8448 \\ d_{dense2} &= (256 + 1) \cdot 256 &= 65792 \\ d_{dense3} &= (256 + 1) \cdot 256 &= 65792 \end{aligned}$$

Da es sich bei der Ausgangsschicht output auch um eine dichte Schicht handelt, können wir die Anzahl d_{output}ihrer Parameter nach dem selben Schema ermitteln und erhalten

$$d_{output} = (256 + 1) \cdot 2 = 514.$$

(vgl. [36])

Es gibt verschiedene Möglichkeiten, ein (faltendes) neuronales Netz zu visualisieren. Die in Abb. 10 gezeigte Grafik hebt insbesondere die Größenverhältnisse zwischen den einzelnen Schichten hervor. Sie sind jedoch nur angedeutet, also nicht maßstabsgetreu, da dann z.B. die dichten Schichten sehr lang werden würden und die Überschaubarkeit der Darstellung verlorengehen würde.

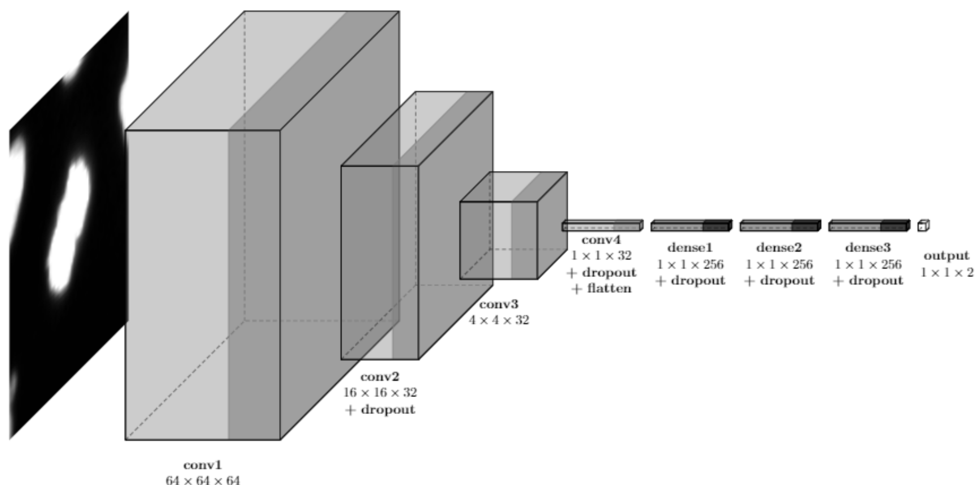


Abbildung 10: Struktur des eigenen Modells für 2 Klassen (vgl. [37])

5 Möglichkeiten zur Beurteilung eines Modells

Wesentlich für die Optimierung von Modellen ist eine tragfähige Methode zur Messung ihrer Leistungsfähigkeit. Dazu führen wir nun eine nützliche Terminologie ein.
Es sei zunächst ein Modell zur binären Klassifizierung der N Datenpunkte eines Datensatzes eingesetzt worden. Wir können o.b.d.A. die beiden Klassen mit „0“ (negativ) und „1“ (positiv) beschriften. Dann bezeichnen wir mit

TP (true positives)	die Anzahl der korrekt Klasse	„1“	zugeordneten Datenpunkte,
TN (true negatives)	die Anzahl der korrekt Klasse	„0“	zugeordneten Datenpunkte,
FP (false positives)	die Anzahl der falsch Klasse	„1“	zugeordneten Datenpunkte,
FN (false negatives)	die Anzahl der falsch Klasse	„0“	zugeordneten Datenpunkte.

Eine gängige Kennzahl ist die **Accuracy** (Genauigkeit). Sie berechnet sich aus

$$\frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{N}. \tag{5.1.1}$$

[38] Diese Kennzahl ist allerdings nur dann aussagekräftig, wenn der Datensatz, mit denen das Modell getestet wurde, ausbalanciert ist, also pro Klasse die selbe Anzahl von Datenpunkten enthält. Angenommen, es liegt im Test-Datensatz ein Verhältnis der Klassen „0“ und „1“ von 1 zu 9 vor, dann erreicht das Modell selbst, wenn es pathologischerweise *immer* die Klasse „1“ ausgibt, eine relativ hohe Accuracy von 90%.

Die **Precision**, gegeben durch

$$\frac{TP}{TP + FP}, \tag{5.1.2}$$

gibt den Anteil der tatsächlich positiven Datenpunkte bzgl. aller als positiv klassifizierten Datenpunkte an. [38]
Das folgende Beispiel verdeutlicht, warum diese Kennzahl aussagekräftiger sein kann, als die Accuracy.

Beispiel 4. Gegeben sei ein Datensatz, bestehend aus $N = 1000$ Zellbildern verschiedener Menschen. Wir stellen uns einen Klassifizierungsalgorithmus vor, dessen Aufgabe es ist, anhand der Bilder zu erkennen, ob ein Mensch mit einer bestimmten Krankheit infiziert ist oder nicht. Es seien 5 Menschen tatsächlich infiziert. Angenommen, unser Klassifizierungsalgorithmus stuft insgesamt 10 Menschen als krank ein, und zwar inklusive aller wirklichen Infizierten, dann beträgt die Präzision des Modells $5/(5 + 5) = 50\%$. Offensichtlich handelt es sich also um ein schlechtes Modell. Die Nicht-Ausbalanciertheit der Daten hat zur Folge, dass die Genauigkeit des Modells mit $(5 + 990)/1000 = 99,5\%$ trotzdem sehr hoch ist. Bedenkt man, welche weitreichende Konsequenzen die Einstufung einer Person als Infizierte oder Infizierter einer Krankheit haben kann, wird deutlich, dass in diesem Fall die Messung der Präzision des Modells weitaus bedeutsamer ist, als dessen Genauigkeit.

Mit TensorFlow lassen sich während einer Trainingseinheit in jeder Epoche mehrere Messgrößen überwachen. Zwei von ihnen werden dort mit **Accuracy** und **Loss** bezeichnet. Im Unterschied zu obiger Beschreibung, meint die Accuracy hier nicht die Genaugkeit bzgl. eines Test-Datensatzes sondern im Bezug auf die Trainingsdaten selbst. Der in jeder Iteration gemessene Loss, also der Wert der Kostenfunktion, ist gerade jener Wert, den das Modell beim Training durch seine Parameter-Updates verringern möchte. Dem Gegenüber stehen die **Validation-Accuracy** und der **Validation-Loss**. Diese beiden Werte werden auf Grundlage von Validierungsdaten gebildet, welche zwar während des Trainings zur

```
Epoch 00017: val_loss did not improve from 0.69351
Epoch 18/100
44/44 [==============================] - 54s 1s/step - loss: 5.1755e-04 - acc: 1.0000 - val_loss: 0.5661 - val_acc: 0.7217

Epoch 00018: val_loss improved from 0.69351 to 0.56611, saving model to bestmodel-0.5661-0018-0.722.h5
Epoch 19/100
44/44 [==============================] - 53s 1s/step - loss: 4.5685e-04 - acc: 1.0000 - val_loss: 0.4645 - val_acc: 0.7883

Epoch 00019: val_loss improved from 0.56611 to 0.46450, saving model to bestmodel-0.4645-0019-0.788.h5
Epoch 20/100
44/44 [==============================] - 54s 1s/step - loss: 4.0650e-04 - acc: 1.0000 - val_loss: 0.3988 - val_acc: 0.8550

Epoch 00020: val_loss improved from 0.46450 to 0.39883, saving model to bestmodel-0.3988-0020-0.855.h5
Epoch 21/100
44/44 [==============================] - 53s 1s/step - loss: 3.6731e-04 - acc: 1.0000 - val_loss: 0.3677 - val_acc: 0.8783

Epoch 00021: val_loss improved from 0.39883 to 0.36767, saving model to bestmodel-0.3677-0021-0.878.h5
Epoch 22/100
44/44 [==============================] - 53s 1s/step - loss: 3.3008e-04 - acc: 1.0000 - val_loss: 0.3402 - val_acc: 0.8950

Epoch 00022: val_loss improved from 0.36767 to 0.34016, saving model to bestmodel-0.3402-0022-0.895.h5
Epoch 23/100
20/44 [============>.................] - ETA: 26s - loss: 3.0930e-04 - acc: 1.0000
```

Abbildung 11: Ausgabe während einer Trainingseinheit in TensorFlow

Beurteilung des Modells herangezogen werden, den eigentlichen Lernfortschritt aber nicht beeinflussen.
In Abb. 11 sehen wir ein typisches Beispiel für die von TensorFlow generierte Ausgabe nach dem Start eines Modelltrainings. In diesem Fall handelt es sich um ein Modell, das für die Erkennung von zwei Klassen trainiert wird. Wir sehen hier, wie es bereits eine perfekte Anpassung der Gewichtsparameter an die Trainingsbilder erreicht hat. Oft ist ein Modell bereits nach wenigen Epochen in der Lage, die Trainingsdaten zuverlässig vorherzusagen. Die Steigerung der validierten Genauigkeit erfordert hingegen deutlich mehr Zeit. Wir sehen hier trotz eines kontinuierlichen Aufwärtstrends der Validation-Accuracy eine deutliche Lücke zwischen den beiden Werten. In diesem Stadium ist das Modell also noch überangepasst.

Für Multi-Klassifizierungsprobleme kann man die oben eingeführten Kennzahlen nur noch bezüglich der einzelnen Klassen ausrechnen (jeweils bezüglich der beiden Zustände „gehört zu dieser Klasse“ und „gehört nicht zu dieser Klasse“). Daher nutzt man für solche Probleme die **Konfusionsmatrix** um die Vorhersagen eines Modells zu untersuchen.

Beispiel 5. Abb. 12 zeigt das Beispiel einer Konfusionsmatrix. Der zugrundeliegende Datensatz hat pro Klasse genau 100 Elemente. Die Zeilenwerte summieren sich jeweils zu genau der Anzahl der dieser Klasse zugehörigen Datenpunkte auf. [39]
Aus dieser Matrix können wir einige Informationen direkt herauslesen. Wir können erkennen, dass das Modell die Klasse „B“ perfekt und die Klasse „D“ zumindest sehr gut vorhergesagt hat. Die anderen Werte lassen darauf schließen, dass es Schwierigkeiten hatte, die beiden Klassen „A“ und „C“ auseinanderzuhalten. Um diese Konfusionsmatrix zu **normalisieren**, dividieren wir jeden Wert durch die Summe der Werte über die jeweils zugehörigen Zeile. Auf diese Weise erhalten wir in jeder Zelle das Verhältnis von

$$\frac{\text{Anzahl der zur Spalte gehörigen vorhergesagten Klasse}}{\text{Anzahl der zur Zeile gehörigen wahren Klasse}}.$$

In unserem Beispiel würden wir hierfür alle Werte durch 100 teilen. [39]

Wahre Klasse \ Vorhergesagte Klasse	A	B	C	D
A	60	0	40	0
B	0	100	0	0
C	45	0	50	5
D	5	0	5	90

Abbildung 12: Beispiel Konfusionsmatrix (vgl. [40],[39])

Bei der **Top-k-Accuracy** handelt es sich um eine Abschwächung der „klassischen“ Genauigkeit, welche misst, wie oft genau die richtige Klasse vom Modell vorhergesagt wurde. Gerade im Bereich der Biologie gibt es aber nicht immer eindeutige Zuordnungen sondern es können auch fließende Übergänge zwischen verschiedenen Zuständen existieren. Um dies in unseren Beurteilungsmethoden für ein Klassifikationsmodell zu berücksichtigen, bietet es sich insbesondere bei mehr als zwei Klassen an, gewisse „naheliegende“ Fehler des Modells zu tolerieren. Wir bezeichnen mit K die Anzahl der Klassen in einem vorliegenden Klassifizierungsproblem und mit N die Gesamtanzahl der Testdaten. Für $k \in \{1, 2, \dots, K\}$ sei TP_k die Anzahl der richtigen Vorhersagen, wobei wir eine Vorhersage nun als „richtig“ ansehen, falls die wahre Klasse eines Datenpunkts $\mathbf{x}$ in der Menge der Indizes mit den k höchsten Einträgen der Vorhersage $\mathbf{y_{pred}}(\mathbf{x}) \in \mathbb{R}^K$ liegt. Dann berechnet sich die Top-k-Accuracy aus

$$\frac{TP_k}{N} \tag{5.1.3}$$

Aus dieser Definition folgt, dass die zuvor eingeführte Accuracy gerade der Top-1-Accuracy entspricht. [41]

Zuletzt halten wir eine weitere Konvention beim Trainieren von Deep-Learning-Modellen fest. Man dokumentiert den Trainingsverlauf, indem ein Plot erstellt wird, der nach Abschluss des Trainings aufgerufen werden kann. An der y-Achse werden zwei die Performance des Modells messende Werte abgetragen, wobei der eine auf den Trainingsdaten basiert und der andere auf den Validierungsdaten (z.B. Accuracy und Validation Accuracy). An der x-Achse wird die Anzahl der durchlaufenen Epochen angezeigt. Mit der Hilfe dieser **Lernkurven** lässt sich nach dem Training leicht auf einen Blick erkennen, ob das Modell überangepasst, unterangepasst oder gut angepasst ist. Gehen wir von einem Plot der Genauigkeiten aus, dann ist das Modell überangepasst, falls die Accuracy-Kurve deutlich über der Validation-Accuracy-Kurve verläuft. Trägt man die Loss- und Validation-Kurven ab, so verhält es sich genau umgekehrt. Ein gutes Modell lässt sich also anhand der Lernkruven daran erkennen, dass die Trainings- und die Validierungskurve der gemessenen Größe mit zunehmender Epochenanzahl immer ähnlicher verlaufen. [42]
Ein häufig anzutreffendes Phänomen sind fluktuierende Lernkurven. Diese können das Resultat numerischer Probleme sein, die durch den Gradienten der Verlustfunktion verursacht werden. Diese Probleme sind als „vanishing gradient“ und „exploding gradient“ bekannt (vgl. [43]). Ein weiterer Grund können zu hohe Lernraten sein. Sie führen dazu, dass der Minimierungsalgorithmus um ein lokales Minimum herum oszilliert [44]. Dies führt zu einer fluktuierenden Loss-Kurve und indirekt zu einer schwankenden Accuracy-Kurve.

6 Methodik

Zunächst fassen wir einige wichtige Rahmenbedingungen zusammen, die für alle durchgeführten Experimente Gültigkeit haben.

6.1 Aufteilung in Validierungs-, Trainings- und Testdaten

Zu Beginn wird das vorliegende Datenmaterial in Trainigs- und Testdaten aufgeteilt. Hier ist es gebräuchlich den größten Teil der Daten dem Training des Modell vorzubehalten während der andere Teil für den Test des trainierten Modells reserviert wird. Um eine unvoreingenommene Bewertung zu gewährleisten ist es wichtig, die Daten disjunkt aufzuteilen. Zur Verbesserung der Genauigkeit eines Modells ist es zudem verbreitet, auch bereits während des Trainings eine Messung seiner Performance durchzuführen. Dazu wird das Trainingsmaterial (disjunkt) aufgeteilt in die beiden Blöcke Training und Validierung. Trainiert wird nur mit dem ersten Block und bei jeder Epoche
wird die Genauigkeit des Modells auf Grundlage der Validierungsbilder gemessen. Dies bietet die Möglichkeit, bereits vor Beendigung des Trainings eine Aussage über die Leistungsfähigkeit des Modells zu treffen. Möchte man die Modellkonfiguration infolge einer verbesserungswürdigen Performance verändern, dann ist die Validierung während des Trainings deutlich zeitsparender als auf ein bereits komplett trainiertes Modell warten zu müssen. Es kann bei einer schlechten Einstellung der Hyperparameter passieren, dass das Modell grundsätzlich schlecht oder gar nicht funktioniert, was sich bei einem Zwei-Klassen-Problem im schlimmsten Fall in der Messung einer konstanten Genauigkeit von 0.5 äußert. In diesem Fall kann man das Training vorzeitig abbrechen und eine Veränderung der Konfigurationen vornehmen [45].
Im Rahmen dieser Arbeit wurde auf die Erstellung größerer Test-Datensätze verzichtet und stattdessen in den meisten Fällen die während des Trainings ausgegebenen validierten Kennzahlen als Grundlage für die Modell-Beurteilung genutzt, da diese grundsätzlich bereits aussagekräftig genug für eine Einschätzung der Leistungsfähigkeit eines Modells sind.

6.2 Begrenzte Anzahl von Trainingsbildern und Epochen

Prinzipiell wurde eine Menge von 1000 Bildern pro Klasse ausgewählt, welche im Verhältnis 0.7 zu 0.3 in die beiden Blöcke Training und Validierung aufgeteilt wurden. Ein zusätzlicher Test-Datensatz kam nur vereinzelt zum Einsatz.
Um eine hinreichend große Menge an Versuchen praktikabel umsetzen zu können, wurde in der Mehrheit der Experimente 100 Epochen lang trainiert. Je nach Modell und Problem kann das bereits genug sein, jedoch gibt es auch Modelle, die sehr langsam trainieren, also erst bei deutlich mehr als 100 Epochen in einen hohen Genauigkeitsbereich kommen. Man kann in der Regel aber schon nach 50-100 Epochen anhand der pro Epoche ausgegebenen Validation-Accuracy erkennen, ob man sich auf einem guten Weg befindet und es ist bereits möglich, mehrere Modelle mit leicht unterschiedlichen Parametereinstellungen miteinander zu vergleichen. Allgemein sind in der Praxis mehr als 1000 Epochen durchaus üblich [46]. Im Rahmen dieser Arbeit liegt die maximale Trainingsdauer bei 2000 Epochen. Es gibt auch die Option, vor Beginn des Trainings eine Schwelle festzulegen, ab der man keine Fortführung des Trainings aufgrund von mangelndem Fortschritt wünscht. Es ist beispielsweise möglich, dass man eine bestimmten Anzahl von Epochen definiert, nach der im Falle einer ausbleibenden Verbesserung der Validation-Accuracy das Training automatisch beendet wird. Diese Methode nennt man early stopping (frühes Stoppen).
TensorFlow bietet außerdem die Möglichkeit, die gegebene Menge an Trainings- und Validierungsbildern künstlich zu vergrößern. Die Originalbilder können dabei wahlweise unter

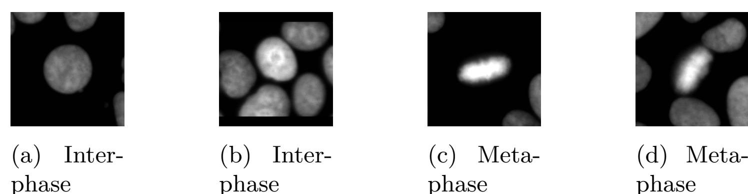
anderem gedreht, gespiegelt oder ihre Helligkeit verändert werden. Dies ist eine verbreitete Technik, genannt **Data Augmentation**, mit der man das durch einen kleinen Datensatz entstehende Handicap kompensieren kann.

6.3 Programmierung in Google Colaboratory

Alle durchgeführten Versuche wurden mit Colaboratory von Google realisiert. Dieses Produkt ist frei verfügbar und ermöglicht es, in Python geschriebene Programme zu erstellen und auszuführen und dabei die Rechenkapazität der von Google bereitgestellten Server zu nutzen. Dies hat mehrere Vorteile. Es ist nicht notwendig, alle für die Programmierung der neuronalen Netze notwendigen Pakete herunterzuladen und es erlaubt unabhängig von den vor Ort verfügbaren Ressourcen ein zeitlich effizientes Training. Die entscheidende Komponente sind hierbei die zur Verfügung gestellten Grafikkarten, die die Geschwindigkeit der notwendigen Bildverarbeitung im Vergleich zu CPUs deutlich erhöhen.

6.4 Darstellung der Ergebnisse

Am Beginn jedes durchgeführten Versuchs steht ein spezifisches Modell, das dann in verschiedener Weise abgeändert wird, um es zu verbessern. Für eine bessere Übersicht sind in jeder Darstellung (Tabelle oder Plot), welche die aus den Experimenten gewonnenen Werte (meist die Genauigkeit) gegenüberstellt, die Kennzahl(en) des Ausgangsmodells **grün** hervorgehoben und die besten erreichten Werte **rot**. Hierbei wurde grundsätzlich der beste Wert ausgewählt, der während einer Epoche des Trainings erreicht wurde.
Die während einer Trainingseinheit gemessenen Werte für die Kennzahlen sind grundsätzlich nicht exakt reproduzierbar. Es gibt immer gewisse Abweichungen, z.b. kann die beste gemessene Validation-Accuracy bei einer Wiederholung des Trainings (von Null an) des gleichen Modells nach eigenen Erfahrungen um bis zu 0.0015 im Vergleich zum vorangegangenen Training abweichen.



(a) Interphase (b) Interphase (c) Metaphase (d) Metaphase

Abbildung 13: Beispielbilder Interphase und Metaphase [Quelle: Medipan]

7 Experimente und Ergebnisse

7.1 Fine Tuning

Das in Kapitel 4.4 vorgestellte Modell für die Erkennung von zwei Klassen ging aus einem Fine-Tuning verschiedener Parameter hervor. Das bedeutet, es wurde von einem gegebenen Modell ausgehend versucht, mittels Veränderungen der Konfiguration eine bessere Genauigkeit zu erreichen. Beispielsweise hatte das ursprüngliche Modell eine andere Anzahl von Kerneln in seinen Faltungsschichten und eine geringere Anzahl von dichten Schichten. Dieses Grundmodell wurde von Dr. Hiemann entwickelt und bildet den Startpunkt aller im weiteren Verlauf vorgestellten Experimente. Das Tuning wurde mit Hilfe des von Keras bereitgestellten **Keras Tuner** realisiert. Er erlaubt es, Hyperparameter im Programmcode der Modellkonfiguration in zeitsparender Weise zu variieren. Dies ist notwendig, da ein Modell sie, im Unterschied zu den Gewichtsparametern, nicht während des Trainings erlernt [7, S.115]. Mit der Methode der Zufallssuche (random search) kann der Tuner für jeden zur Veränderung freigegebenen Hyperparameter (z.b. die Lernrate) die Werte einer vom Benutzer vordefinierten Menge in zufälliger Reihenfolge durchprobieren. Er trainiert dabei jedes mal das Modell neu und gibt jeweils die beste erreichte Validation-Accuracy aus. [47],[48]

Die Rahmenbedingungen für die Versuche sind wie folgt festgelegt:

Anzahl der Trainingsbilder:	1000 pro Klasse
Aufteilung Trainigdaten zu Validierungsdaten:	0.7 zu 0.3
Anzahl Epochen:	100

Einige Beispielbilder aus dem verwendeten Datensatz sind in Abb. 13 gezeigt.
Experimentiert wurde mit den folgenden Parametern:

- Anzahl der Filter
- Kernel-Größe und Schrittweite
- Anzahl und Konfiguration der dichten Sichten
- Lernrate
- Wahl des Optimierungsalgorithmus

Tuning der Filter:
Der erste Schritt ist die Verbesserung der Filterschichten durch Veränderungen der jeweiligen Kernelanzahlen. Die ursprüngliche Anzahl der Schichten von 4 wurde beibehalten.

Es wurden folgende Parameter fixiert:

Lernrate:	0.001
Kernel-Größe :	(5,5)
Schrittweite :	(4,4)
Anzahl der dichten Schichten :	2
Epochen:	80
Optimierer:	NADAM
Data-Augmentation:	keine

Mit der zufälligen Suche wurde eine Vielzahl möglicher Kombinationen untersucht. Als den zu überwachenden Zielwert wurde die Validation-Accuracy angegeben. Die besten 10 Ergebnisse sind in Tabelle 1 festgehalten.

Ergebnisse:

Anzahl Filter der Filterschichten	Val-Acc
(16,16,32,32)	0.9650
(16,32,32,32)	0.9633
(32,16,32,32)	0.9650
(32,16,32,64)	0.9650
(32,64,64,32)	0.9650
(64,16,32,32)	0.9650
(64,32,32,32)	0.9667
(64,32,32,64)	0.9650
(64,64,32,32)	0.9633
(64,64,64,32)	0.9650

Tabelle 1: Ergebnisse Fine-Tuning der Filter des eigenen Modells

Die originale Konfiguration war von einem anderen neuronalen Netz inspiriert, dem **VGG16**. Es gilt als eines der besten Modelle im Bereich der Objekterkennung. Seine Struktur ist dadurch gekennzeichnet, dass die Filterschichten in immer größer werdende Blöcke aufgeteilt sind und sich je Block die Anzahl der Kernel in den zugehörigen Schichten verdoppelt [49]. In unserem Fall erweist sich jedoch offenbar eine andere Filterkonfiguration als nützlicher. Dies kann an der Begrenztheit des Ausgangsproblems liegen. Während Modelle, wie das VGG16, für die Erkennung von bis zu Tausenden von Klassen konzipiert wurden, ist der Einsatzbereich unseres Modells natürlich sehr viel kleiner. Für unserer spezifisches Problem der Zellphasenerkennung zentrierter Zellen hat dass zur Folge, dass die für das eigene Modell gut geeigneten Hyperparameter nicht mehr stark vergleichbar sind zu den Strukturen anderer, deutlich komplexerer Modelle.
Wir sehen, dass die Veränderungen in der Genauigkeit nur sehr klein sind. Kombinieren wir das Filter-Tuning mit weiteren Hyperparametersuchen, so können sich unter Umständen die gewonnenen Verbesserungen addieren.

Tuning der dichten Schichten:
Im nächsten Schritt wurden die zuvor neu justierten Filter beibehalten und verschiedene Anpassungen der dichten Schichten untersucht. Die in Tabelle 2 dargestellten Ergebnisse zeigen, dass die ursprüngliche Konfiguration tatsächlich noch verbesserungswürdig war. Die beste Genauigkeit ist hier mit einer Anzahl von drei versteckten dichten plus die Ausgangsschicht erreicht worden. Im Unterschied zum Filter-Tuning, wurden hier manuell verschiedene Möglichkeiten durchprobiert. Grund dafür war das breite Spektrum von

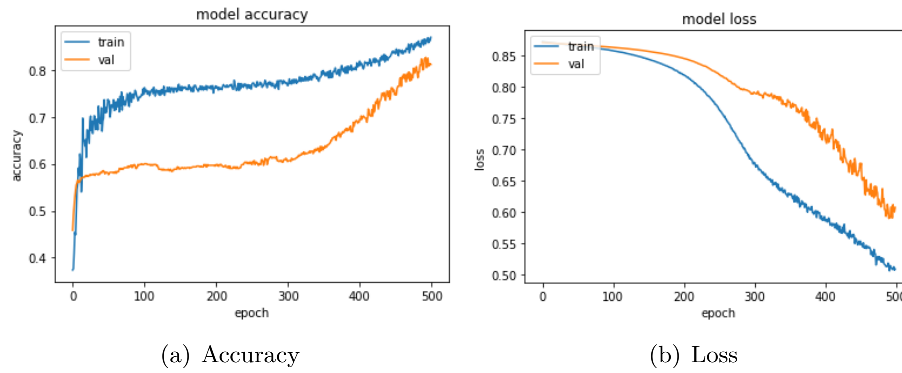
Möglichkeiten, das a priori nicht sicher einzuschränken war. Daher wurde hier sowohl mit der Anzahl der Neuronen, als auch mit der Anzahl der dichten Schichten experimentiert. Die Ergebnisse lassen darauf schließen, dass mehr Schichten (zumindest bis zu einer bestimmten Anzahl) tendenziell zu einer Verbesserung der Vorhersagen führt. Im Vergleich zum Filter-Tuning sind die Verbesserungen der Genauigkeit deutlicher ausgefallen.

Anzahl Neuronen der dichten Schichten	Val-Acc
(128,2)	0.9617
(256,2)	0.9617
(128,128,2)	0.9600
(256,256,2)	0.9667
(256,256,256,2)	0.9717
(256,256,256,128,2)	0.9650
(256,256,256,256,2)	0.9633

Tabelle 2: Ergebnisse Fine-Tuning der dichten Schichten des eigenen Modells

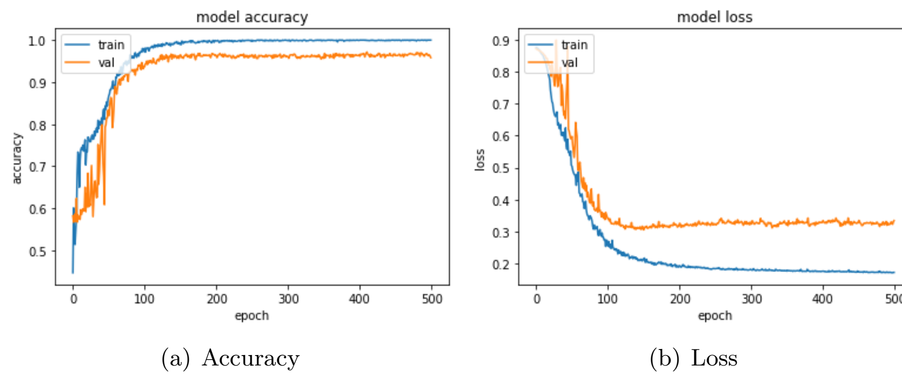
Optimierer im Vergleich:
Wesentlich für eine gute Performance ist eine geeignete Auswahl des Optimierungsalgorithmus. Die Kostenfunktion bildet ihre Form in Abhängigkeit von den Trainingsdaten und des vorliegenden Klassifizierungsproblems. Folglich ist es unwahrscheinlich, dass genau ein bestimmter Optimierer immer zu einer guten Performance des Modells führt. Außerdem kann die Wahl des Optimierers auch die Dauer des Trainings beeinflussen, etwa indem die Zeit bis zur Konvergenz gegen ein gutes lokales Minimum verkürzt wird [50].

Startpunkt des Optimizer-Tunings war das klassische SGD-Verfahren ohne Moment. Wir sehen in Abb. 14 den Trainingsverlauf des Modells während 500 Epochen. Auffällig ist, dass erst ab ca. 300 Epochen eine kontinuierliche Verbesserung der Validation Accuracy stattfindet, nachdem sie vorher lange in einem Plateau um 60% herum verbleibt. Dass die (auf den Trainingsbildern basierende) Accuracy gleichzeitig deutlich höher ist, lässt auf eine Überanpassung des Modells in dieser Periode schließen. Es ist prinzipiell immer zu erwarten, dass die Accuracy etwas höher als die Validation-Accuracy liegt, jedoch sollten sie möglichst nahe beieinander liegen. Analog können wir von der Loss-Kurve erwarten, dass sie immer mindestens leicht unterhalb der Validation-Loss-Kurve verläuft. Die Kostenwert-Kurve rechts zeigt eine nur langsame Verbesserung des Loss-Werts. Dies kann am Feststecken in einem schlechten lokalen Minimum liegen, was auf eine zu geringe Lernrate hindeutet. Trotz der Startschwierigkeiten ist anzunehmen, dass sich der Aufwärtstrend der Genauigkeit bzw. der Abwärtstrend des Loss-Werts bei einer höheren Zahl von Epochen fortsetzen würde.
Die Hinzunahme eines Moments von 0.9 erweist sich hier als eine kluge Maßnahme, um das eben beschriebene Verhalten zu kompensieren (vgl. Abb. 15). Das Modell erreicht bereits nach ca. 150 Epochen ein hohes Genauigkeitsniveau, dass dann aber ziemlich unverändert bleibt. Rechts sehen wir, dass sich auch der Loss-Wert bis zu einer ähnlicher Anzahl von Epochen wie die Validation Accuaracy verbessert und dann nicht mehr. Die Hinzunahme des Nesterov-Moments hat wiederum kaum einen Effekt (vgl. Abb. 16).
Der RMSprop-Algorithmus, ADAM und NADAM führen zu ähnlichen Ergebnissen (siehe Abb. 17,18).



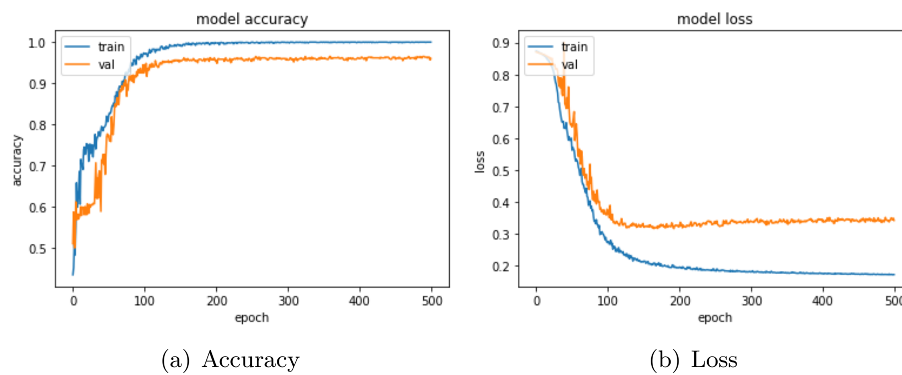
(a) Accuracy (b) Loss

Abbildung 14: Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer ohne Moment



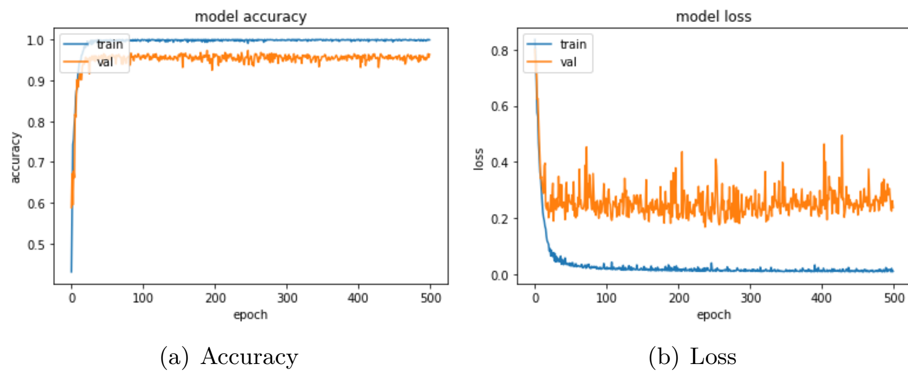
(a) Accuracy (b) Loss

Abbildung 15: Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer mit einem Moment von 0.9



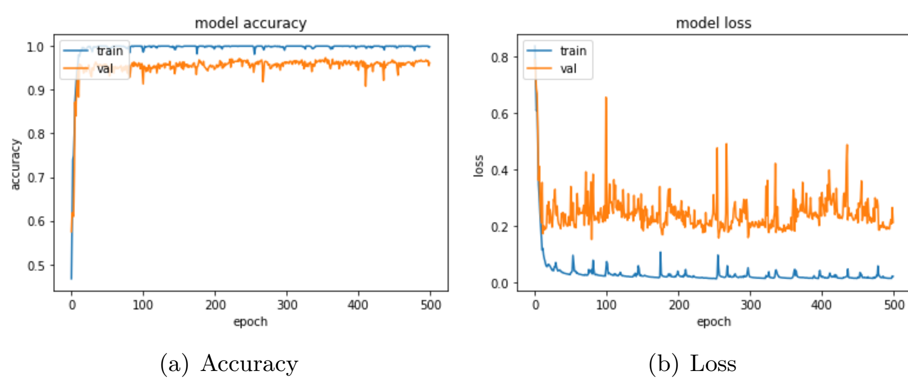
(a) Accuracy (b) Loss

Abbildung 16: Trainingsverläufe des eigenen Modells mit dem SGD-Optimierer mit einem Moment von 0.9 und mit Nesterov-Moment



(a) Accuracy (b) Loss

Abbildung 17: Trainingsverläufe des eigenen Modells mit RMSprop



(a) Accuracy (b) Loss

Abbildung 18: Trainingsverläufe des eigenen Modells mit dem ADAM-Optimierer

Ergebnisse:
In Tabelle 3 sind zusammenfassend die Ergebnisse des Optimierer-Tunings dargestellt. Das beste Ergebnis konnte mit NADAM eingefahren werden. Der Wert von 0.9717 ist gerade die Validation-Accuracy aus dem Tuning der dichten Schichten. Mit anderen Worten: Wir hatten keinen Erfolg bei dem Versuch, das Modell durch andere Optimierer als NADAM zu verbessern.

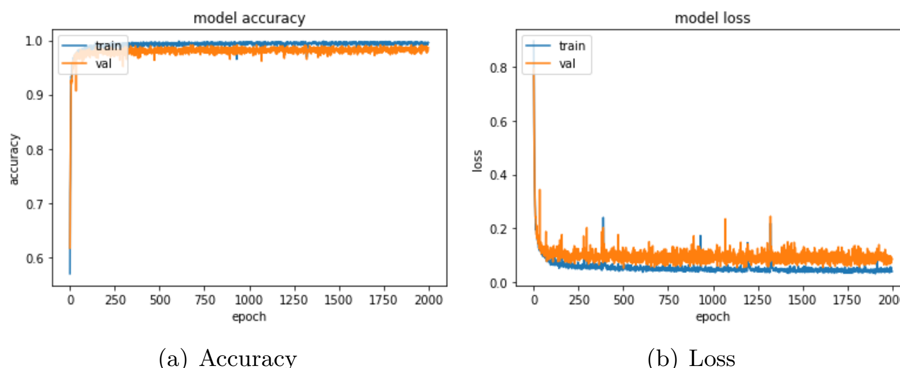
Optimierer	Val-Acc
SGD vanilla	0.8450
SGD (momentum=0.9, nesterov=false)	0.9700
SGD (momentum=0.9, nesterov=true)	0.9650
ADAM	0.9683
RMSprop	0.9680
NADAM	0.9717 (vgl. Tabelle 2)

Tabelle 3: Ergebnisse der Tests mit verschiedenen Optimierern am eigenen Modell

Data Augmentation:
Die Methode der Data Augmentation ist eine effektive Möglichkeit um die Genauigkeit eines Modells zu verbessern und eine möglicherweise geringe Anzahl von Trainingsbildern zu kompensieren. Da in den Experimenten mit nur 1000 Bildern je Klasse gearbeitet wurde, war zu hoffen, dass die durch das Tuning verbesserte Genauigkeit noch weiter erhöht werden kann. Die Vergrößerung des Trainingsmaterials an sich ist streng genommen kein Tuning des Modells, da wir seine Konfigurationen nicht anfassen. Folgende Operationen wurden für die Augmentierung freigeschaltet:

- horizontaler Flip
- vertikaler Flip
- Rotation (0° bis 180°)

Der Preis der Augmentierung ist eine längere Trainingsdauer, daher wurde es bei diesen drei Augmentierungsmethoden belassen. Grundsätzlich sind in TensorFlow noch weitere Operationen möglich.Mit diesen Einstellung konnte die Genauigkeit des aus den vorangegangenen Tuning-Einheiten hervorgegangene Modell von 0.9717 auf 0.9900 erhöht werden. In Abb. 19 sehen wir den Trainingsverlauf dieser Modellvariante, die 2000 Epochen lang trainiert wurde.
Über die eben vorgestellten Ergebnisse hinaus konnte in weiteren Experimenten festgestellt werden, dass die eingangs festgelegte Lernrate von 0.001 bereits gut gewählt war. Prinzipiell konnten mit verschiedenen Lernraten unterhalb von 0.001 gute Ergebnisse erzielt werden und nur marginale Differenzen der Validation-Accuracy bei verschiedenen Werten für die Lernrate in der Nähe von 0.0005 beobachtet werden. Die Kernel-Größe zusammen mit der Schrittweite der Kernel wurde beibehalten.



(a) Accuracy

(b) Loss

Abbildung 19: Trainingsverläufe des eigenen Modells mit dem NADAM-Optimierer und Data Augmentation

7.2 Keras Applications und Transfer-Learning

Mit den **Keras Applications** bietet sich die Möglichkeit an, von Forschern bereits entwickelte und getestete fertige Modelle zu benutzen. Sie sind mit dem „ImageNet“-Datensatz trainiert worden. Dabei handelt es sich um eine breite Sammlung von Bildern, die zum Training von Objekterkennungsmodellen genutzt werden können. Es ist verbreitet in der Forschung mit Deep Learning und für die Nutzer frei verfügbar. Laut eigenen Angaben, enthält das Datenpaket über 14 Millionen Bilder. [51]
Obwohl es in erster Linie Bilder von Gegenständen, Menschen, Tieren und Pflanzen enthält (vgl. [52]) werden mit diesen Bildern trainierte Modelle häufig auch in medizinischen Anwendungen genutzt, etwa zur Erkennung von Krebszellen (siehe z.b. [53]).
Es gibt nun zwei Möglichkeiten, sich die Keras Applications nutzbar zu machen.

1. Die Struktur des Modells wird zwar genutzt, trainiert wird es aber von Beginn an. D.h. am Anfang des Trainings werden die Gewichtsparameter zufällig initialisiert und das Modell erlernt die Feature-Erkennung von Null an. Vom Original abweichend geladen wird lediglich die Ausgangsschicht, bei welcher die Anzahl der Neuronen an den eigenen Bedarf angepasst werden kann.

2. Nur der Teil des Modells, der für die Feature-Erkennung zuständig ist, wird genutzt. Die restlichen (in diesem Zusammenhang als **Top-Layer** bezeichneten) Schichten, also insbesondere die hinter den Filter- und Pooling-Schichten liegenden dichten Schichten müssen dann selbst konfiguriert und hinzugefügt werden. Vor dem Training werden die vorgefertigten Gewichtsparameter (Pre-Weights) heruntergeladen und mit ihnen das Modell initialisiert.

[54] Das Transfer-Learning findet nur in der zweiten Variante statt. Unsere Hoffnung ist, dass die durch das Training mit ImageNet erlernten Strukturerkennungen der Modelle sich positiv auf die Zellphasenerkennung auswirken. Es ist anzunehmen, dass ein Modell, welches bereits mit Tausenden von verschiedenen Bildern von Objekten zur Klassifizierung trainiert wurde, auch ohne erneute Anpassung der Feature-Erkennung in der Lage ist, gute Ergebnisse zu liefern. Die grundlegenden Bestandteile einer Objekterkennung, wie die Detektion von Rändern, von Formen oder der Unterscheidung zwischen Hell und Dunkel, wurden schließlich bereits erlernt und sollten daher auch auf unser Klassifizierungsproblem anwendbar sein. Im Folgenden werden an einer Auswahl von Keras Applications durchgeführte Experimente mit der ersten Variante (ohne Transfer-Learning) und mit der zweiten Variante (mit Transfer-Learning) präsentiert und ein Vergleich zwischen den jeweiligen Ergebnissen gezogen.

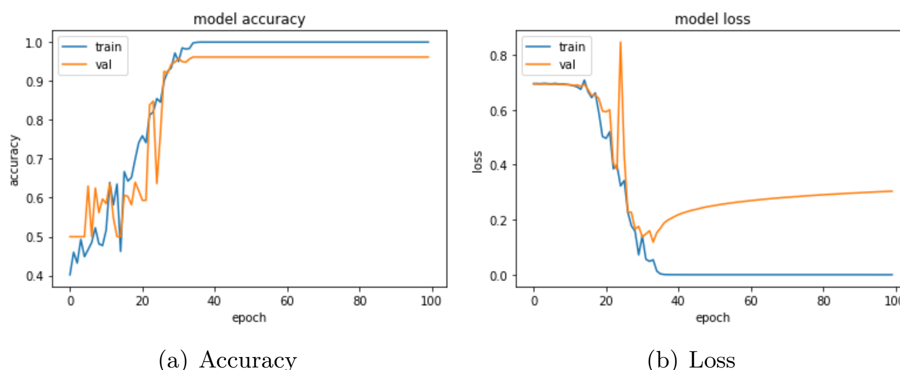
7.2.1 Tests mit nicht vortrainierten Modellen

Fine-Tuning mit VGG16:
Als geeigneter Optimierer für das VGG16-Modell hat sich das SGD-Verfahren mit Moment herausgestellt. Dies ist das Ergebnis einiger Tests mit anderen Optimierern, bei denen festgestellt werden konnte, dass beispielsweise ADAM nicht gut geeignet für das Modell ist. Um nun die Konfigurationen des SGD-Algorithmus zu optimieren, wurde wieder der Keras Tuner herangezogen. Mit ihm wurden verschiedene Werte für die Lernrate in Kombination mit dem Wert für das Moment ausgetestet. Die Ergebnisse sind in Tabelle 4 festgehalten. Als effektiv erweisen sich vor allem hohe Momente ab 0.8 während eine Verringerung der Lernrate 0.001 auf 0.0005 vergleichbare, wenn auch leicht schlechtere Genauigkeiten bewirkt.

Lernrate	**Moment**	**Val-Acc**
0.0005	0.7	0.8683
0.0005	0.8	0.9467
0.0005	0.9	0.9617
0.001	0.7	0.8867
0.001	0.8	0.9600
0.001	0.9	0.9650

Tabelle 4: Ergebnisse des Fine-Tunings von Lernrate und Moment mit VGG16

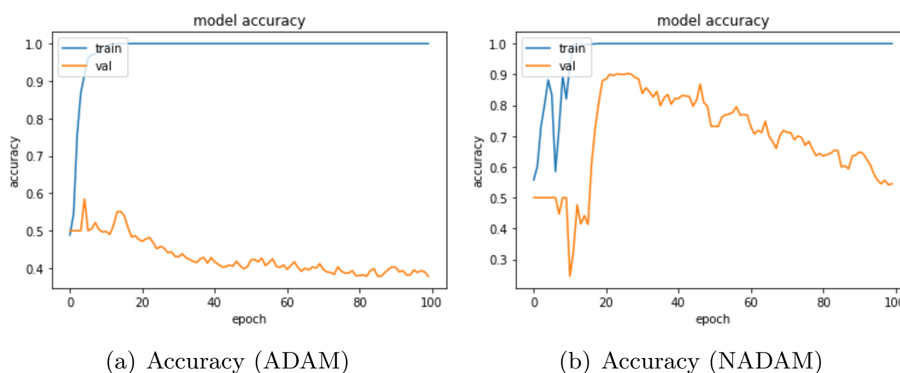
VGG19:
Mit SGD und den aus den Tests an VGG16 abgeleiteten Werten für die Lernrate und das Moment wurde auch das ähnliche Modell VGG19 implementiert. Es unterschiedet sich primär in der Anzahl der Schichten (19 statt 16) [55, S.3]. Die Trainingsverläufe sind in Abb. 20 zu sehen.



(a) Accuracy (b) Loss

Abbildung 20: Trainingsverläufe des nicht vortrainierten VGG19

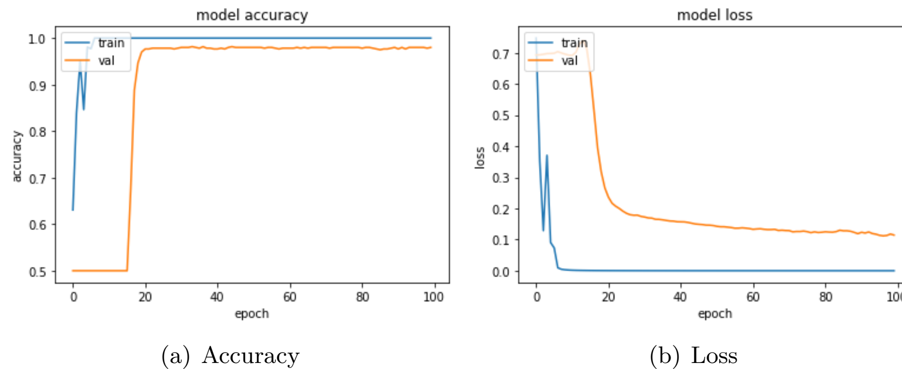
InceptionV3:
Dieses Modell basiert auf den in [56] vorgestellten Ideen. Während der Traininseinheiten konnte insbesondere eine deutlich geringere Trainingsdauer als bei den VGG-Modellen beobachtet werden. Gute Ergebnisse konnten mit NADAM erreicht werden (beste Val-Acc.: 0.9710), während ADAM versagt (beste Val-Acc.: 0.5750). Mit NADAM ist die beste Validation-Accuracy bereits nach ca. 20 Epochen zu beobachten, danach fällt die zugehörige Kurve ab, während die Trainingsdaten perfekt vorhergesagt wurden (vgl. Abb. 21). Dies lässt auf eine Überanpassung des Modells schließen. Sinnvollerweise können wir bereits in dem Plateau der Val-Acc.-Kurve nach 20 Epochen frühzeitig das Training stoppen und erhalten damit eine gute Performance. Wie in Kap. 3.6 beschrieben, kann eine Ursache der Überanpassung die zu hohe Komplexität des Modells (bei einer im Vergleich zu seinen Parametern geringen Anzahl von Trainingsdaten) sein. Daher ist anzunehmen, dass die Anzahl von insgesamt 2000 Trainingsbildern eher zu klein für InceptionV3 ist und wir die Überanpassung durch einen größeren Datensatz kompensieren könnten.



(a) Accuracy (ADAM) (b) Accuracy (NADAM)

Abbildung 21: Trainingsverläufe des nicht vortrainierten InceptionV3

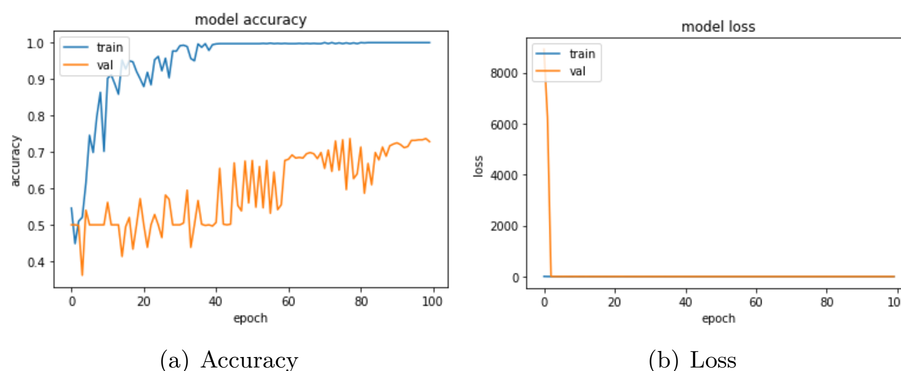
Xception:
Aufbauend auf der Inception-Architektur wurde bei diesem Modell eine modifizierte Form der Faltung benutzt, wodurch auf dem ImageNet-Datensatz leicht bessere Ergebnisse erzielt werden konnten als mit InceptionV3. [57]
In Abb. 22 sehen wir ein sehr zufriedenstellendes Trainingsverhalten ohne Fluktuationen und mit einem zügigen Erreichen eines hohen Plateaus der Validation-Accuracy. Auffällig ist, dass das Modell bis kurz vor Epoche 20 die Validierungsdaten nur „errät“ (also eine Val-Acc. von 0.5 erreicht) und die Genauigkeit dann unvermittelt steil nach oben springt. Trainiert wurde das Modell mit dem NADAM-Optimierer.



(a) Accuracy (b) Loss

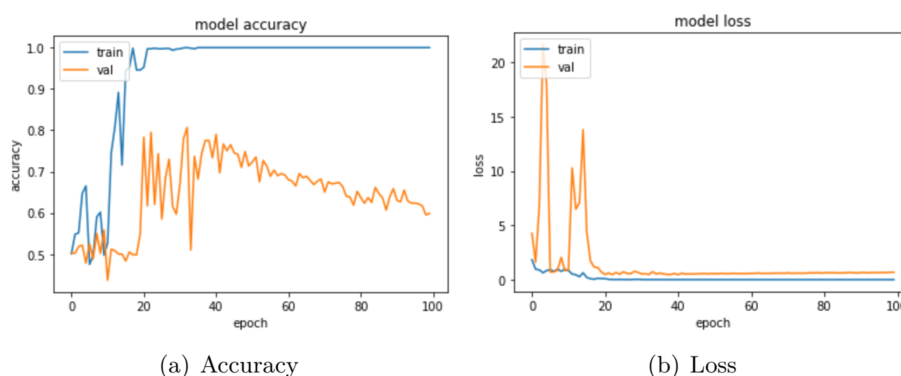
Abbildung 22: Trainingsverläufe des nicht vortrainierten Xception

ResNet50:
Die Entwicklung neuronaler Restnetzwerke (Residual Neural Networks) wurde durch das Problem der Degradierung (degradation problem) motiviert. Dabei handelt es sich um das in der Praxis beobachtete Phänomen, dass das Hinzufügen weiterer Schichten zu einem neuronalen Netzwerk ab einer hinreichend großen Anzahl bereits vorhandener Schichten zu einem Verlust an Genauigkeit führt. Dies kann allerdings nicht auf eine Überanpassung zurückgeführt werden, da der beobachtete Verlust an Genauigkeit überraschenderweise nicht nur bezüglich der Validierungsdaten sondern auch bezüglich der Trainingsdaten selbst stattfindet. Die Lösung dieses Problems liegt darin, die durch eine Schicht erzeugte Zurordnung $\mathbf{H}(\mathbf{x})$ durch die residuale Zuordnung $\mathbf{F}(\mathbf{x}) + \mathbf{x}$ mit $\mathbf{F}(\mathbf{x}) := \mathbf{H}(\mathbf{x}) - \mathbf{x}$ zu ersetzen. (vgl. [58, S.1,f.])
Verglichen wurde die Performance des Modells ResNet50 bei der Nutzung von SGD- und dem ADAM-Verfahren. In beiden Fällen konnten keine vergleichbar guten Ergebnisse erzielt werden, wie mit VGG16 (zumindest nicht nach 100 Epochen). Im Fall des ADAM-Optimierers wurde die beste Validation-Accuracy bei ca. 35 Epochen erreicht (Abb. 24(a)). Danach fällt die Validation-Accuracy-Kurve wieder. Auffällig sind die Loss-Kurven, welche über weite Strecken konstant sind. Jedoch zeigt Abb. 24(b), dass die Kurve im Fall des ADAM-Optimierers instabiler als mit SGD (Abb. 23(b)) ist. Dies zusammen mit dem tendenziell steigenden Verlauf der Val.-Accuracy in Abb. 23(a) deuten darauf hin, dass die Implementierung des Modells mit dem SGD-Verfahren nach einer längeren Trainingsdauer womöglich die bessere Wahl sein könnte als ADAM. Die Wahl des Moments in der SGD-Variante folgt dem Beispiel einer Implementierung verschiedener Residualer Netzwerke, die u.A. mit dem ImageNet-Datensatz trainiert wurden (vgl. [58, S.4]).



(a) Accuracy (b) Loss

Abbildung 23: Trainingsverläufe des nicht vortrainierten ResNet50 mit dem SGD-Optimierer (Mom.=0.9, Nesterov)



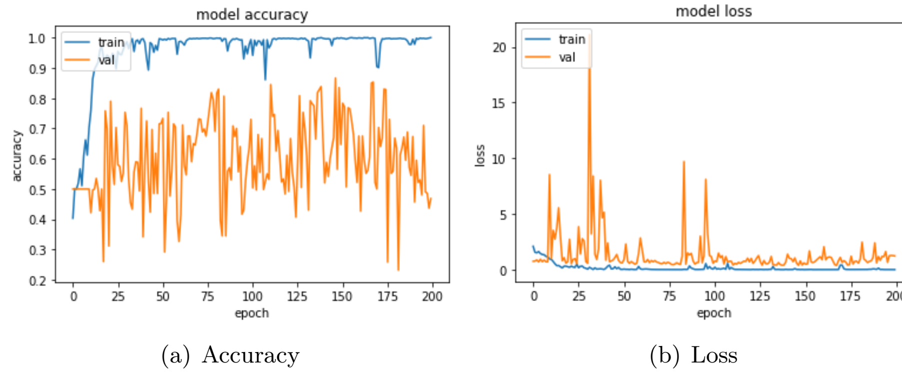
(a) Accuracy (b) Loss

Abbildung 24: Trainingsverläufe des nicht vortrainierten ResNet50 mit ADAM

EfficientNetB0:
Zuletzt wurden Tests mit einem Modell aus der Familie der **EfficientNets** durchgeführt. Hervorstechend bei diesen Modellen ist die Eigenschaft, trotz deutlich reduzierter Anzahl von Parametern eine bessere Genauigkeit auf dem ImangeNet-Datensatz erreicht zu haben, als die zuvor entwickelten gängigen Netzwerk-Architekturen. [59, S.1]
Für unseren Anwendungsfall hat sich das Modell jedoch nicht als besonders gut geeignet erwiesen. In Abb. 25 sehen wir ein sehr instabiles Trainingsverhalten, das keine Tendenz in Richtung eines Plateaus auf über 0.9 für die Val-Acc.-Kurve aufzeigt, wie einige der anderen getesteten Modelle.

Ergebnisse:

In der Kategorie der nicht vortrainierten Keras Applications stellte sich das Xception als die leistungsstärkste Wahl für die Erkennung unserer beiden Klassen heraus. Wie auch bezüglich ImageNet ist Xception scheinbar eine bessere Wahl als das Vorgängermodell Inception, wobei der Vorsprung in unserem Fall sogar deutlich größer ist.



(a) Accuracy (b) Loss

Abbildung 25: Trainingsverläufe des nicht vortrainierten EfficientNetB0

Keras Application (nicht vortrainiert)	Val-Acc
VGG16	0.9650
VGG19	0.9617
InceptionV3	0.9170
Xception	0.9800
ResNet50	0.8050
EfficientNetB0	0.8750

Tabelle 5: Ergebnisse der Tests mit nicht vortrainierten Keras Applications

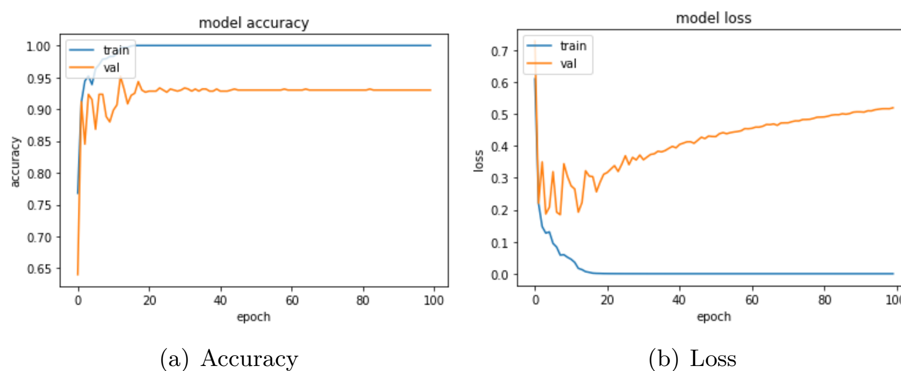
7.2.2 Tests mit vortrainierten Modellen

Bei der Implementierung vortrainierter Modelle ergibt sich eine neue Schwierigkeit. Wir müssen herausfinden, auf welche Weise wir die trainierbaren dichten Schichten konfigurieren sollen. Das Fine-Tuning der dichten Schichten im eigenen Modell hat gezeigt, dass dies keine triviale Aufgabe ist und einen deutlichen Unterschied in der Leistung des Modells machen kann.

Xception:
Als Ausgangspunkt wurde eine Anzahl von zwei versteckten dichten Schichten gewählt. Wie beim Fine-Tuning des eigenen Modells wurden mit dem Keras Tuner verschiedene Kombinationen mit der Zufallssuche getestet. Ein Auszug der besten Ergebnisse ist in Tabelle 6 abgebildet.
Wie wir sehen gibt es zwei Kombinationen, die (annähernd) gleich gute Validierungsgenauigkeiten liefern. Weiterverwendet wurde die Konfiguration (1024,512) der versteckten dichten Schichten, da sie eine leicht bessere Validation-Accuracy bewirkte (dies ist in der Ergebnistabelle nicht erkennbar, weil die Werte für eine bessere Übersicht gerundet wurden). Die Trainingsverläufe des Xception mit der besten Konfiguration der dichten Schichten ist in Abb. 26 zu sehen.

Neuronenanzahl 1. dichte Schicht	Neuronenanzahl 2. dichte Schicht	**Val-Acc**
256	256	0.9433
256	512	0.9450
512	128	0.9467
512	256	0.9467
512	512	0.9500
1024	128	0.9483
1024	256	0.9433
1024	512	0.9500

Tabelle 6: Ergebnisse des Fine-Tunings der dichten Schichten im vortrainierten Xception



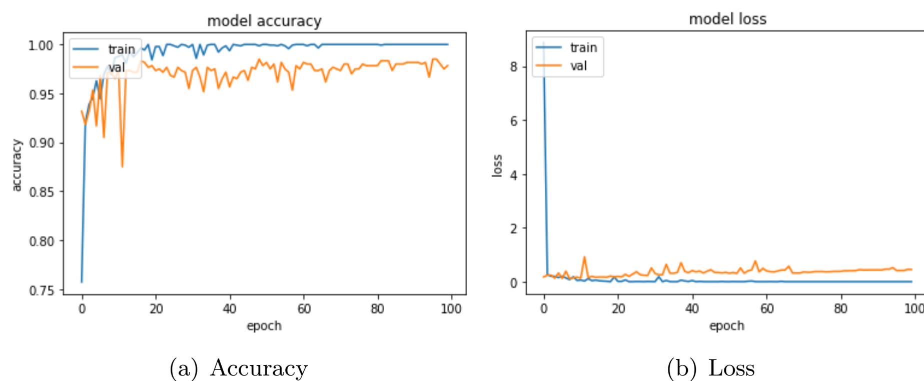
(a) Accuracy (b) Loss

Abbildung 26: Trainingsverläufe des vortrainierten Xception

InceptionV3:
An den zur Feature-Erkennung zuständigen Teil wurden als trainierbare Schichten eine dichte Schicht mit 1024 Neuronen plus anschließendem Dropout-Layer und die Ausgangsschicht nachgeschaltet. Experimentiert wurde mit diversen Optimierern, der Lernrate und der Dropout-Rate. In Tabelle 7 sind die entsprechenden Ergebnisse abgetragen. RMSprop schlägt bei einer Lernrate von 0.0005 die anderen Varianten. RMSprop wurde als letztes getestet, davor wurde die Dropout-Rate auf 0.2 fixiert. Es wird davon ausgegangen, dass eine Veränderung der Dropout-Rate bei den anderen Optimierern zu einem ähnlichen Ergebnis, also einem Abfallen der Validation-Accuracy, führen würde. Die in Abb. 27 dargestellten Kurven wurden beim Training des Modells mit den in der Ergebnistabelle rot markierten Konfigurationen erzeugt.

Optimierer	**Lernrate**	**Drop-Rate**	**Val-Acc**
ADAM	0.0001	0.2	0.9783
ADAM	0.001	0.2	0.9783
NADAM	0.0001	0.2	0.9750
NADAM	0.001	0.2	0.9783
NADAM	0.0005	0.2	0.9800
RMSprop	0.0001	0.2	0.9817
RMSprop	0.0005	0.1	0.9817
RMSprop	0.0005	0.2	0.9850
RMSprop	0.0005	0	0.9817

Tabelle 7: Ergebnisse der Tests mit dem vortrainierten InceptionV3



(a) Accuracy (b) Loss

Abbildung 27: Trainingsverläufe des vortrainierten InceptionV3

Ergebnisse:

Wir können diesmal einen deutlichen Vorsprung des InceptionV3 vor dem Xception-Modell beobachten.

Keras Application (vortrainiert)	**Val-Acc**
Xception	0.9500
InceptionV3	0.9850

Tabelle 8: Ergebnisse der Tests mit vortrainierten Keras Applications

7.2.3 Vergleich zwischen Modellen mit und ohne Pre-Weights

Das beste vortrainierte Modell (InceptionV3) konnte eine etwas genauere Performance liefern, als das beste nicht vortrainierte Modell (Xception). In den durchgeführten Experimenten konnte beobachtet werden, dass die Trainingsdauer eines vortrainierten Modells deutlich kleiner ist als die Trainingsdauer eines Modells ohne Pre-Weights. Dies ist nicht überraschend, da im Falle des Transfer-Learnings die für die Feature-Erkennung zuständigen Filter- und Pooling-Schichten alle eingefroren sind und somit nur die letzten selbst programmierten dichten Schichten trainiert wurden. Die Trainingsverläufe des vortrainierten und des nicht vortrainierten Xception sehen nicht grundverschieden aus. Jedoch können wir beobachten, dass die Lernkurven im letzteren Fall etwas instabiler verlaufen. Jedoch liegt die erreichte Validation-Accuracy des vortrainierten Modells unterhalb der des Modells ohne Vorgewichte (vgl. Abb. 22 und 26). Für InceptionV3 gilt, dass die vortrainierte Variante nicht mehr die Tendenz zu einem Abfall der Validation-Kurve hat, wie das beim nicht vortrainierten Modell der Fall war. Zudem hat dieses Modell, bei Initialisierung mit den Pre-Weights, eine deutlich höhere Validation-Accuracy (vgl. Abb. 21 und 27).

7.3 Erweiterung auf sieben Klassen

Für die praktische Anwendung ist ein Modell wünschenswert, dass es mehr als nur zwei Klassen von Zellstadien unterscheiden kann. Daher ist nun das Ziel, die Erkenntnisse aus den Tests für zwei Klassen zu übertragen auf die Implementierung von Modellen, welche sieben Klassen erkennen sollen. Unten sehen wir fünf Klassen, welche zum normalen Lebenszyklus einer Zelle gehören und zwei Klassen, welche einige potenziell auftretende Sonderfälle abdecken.

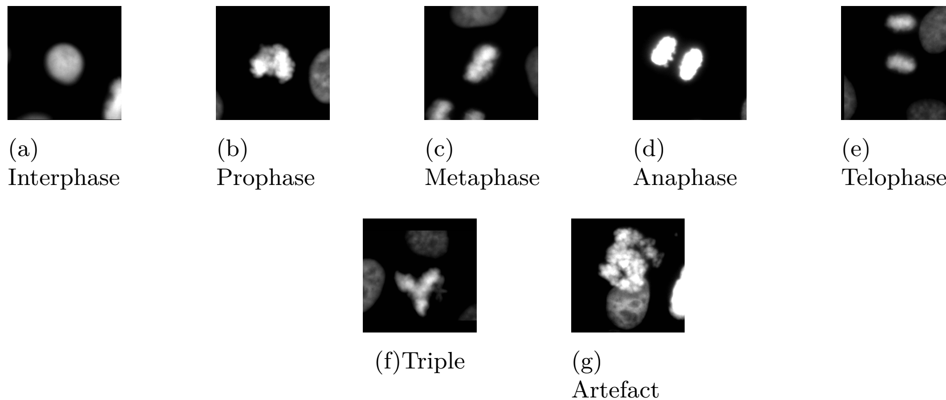


Abbildung 28: Beispielbilder der 7 verschiedenen Klassen [Quelle: Medipan]

Es wurde der Datensatz entsprechend erweitert und die Anzahl von 1000 Bildern je Klasse beibehalten sowie das Verhältnis von 0.3 zu 0.7 der Validierungs- und Trainingsdaten. Um die Trainingsdauer nicht zu stark zu verlängern, wurde prinzipiell auf eine Datenaugmentierung verzichtet. Getestet wurden wieder einige der Keras Applications und das eigene Modell. Die zuvor dargestellten Konfigurationen der Modelle und die Struktur der trainierbaren dichten Schichten von InceptionV3 und Xception gehen auf die im vorangegangenen Abschnitt vorgestellten Fine-Tunings zurück. Für VGG19 wurde die Konfiguration des InceptionV3 übernommen. Die Endkonfiguration des eigenen Modells für das Zwei-Klassen-Problem war im ersten Versuch nicht funktionsfähig (Val-Acc. = 0.1429) und wurde deswegen verändert. Angepasst wurde die Anzahl der Kernel auf (64,64,64,128) für die vier Filterschichten. Die Anzahl der versteckten dichten Schichten wurde auf eines reduziert und seine Neuronenzahl auf 144 festgelegt. Zudem wurde die Dropout-Rate auf 0.1 geändert. Diese neue Konfiguration ist das Resultat einiger analog zur in Kap. 7.1 vorgestellten Methodik durchgeführten Fine-Tunings, auf deren ausführliche Darstellung hier verzichtet wird. Die beste Validation-Accuracy der neuen Variante übertraf das ursprüngliche Modell (vor dem Fine-Tuning aus Kap. 7.1) um über 3%.
Im Unterschied zum Fall mit zwei Klassen, war das vortrainierte Xception nun um ca. 2% genauer als die nicht vortrainierte Variante. Dies stützt die Vermutung, dass das Transfer-Learning bei komplexeren Problemen nutzbringender wird. Es konnte beobachtet werden, dass das Training von Xception deutlich länger dauerte als das der anderen hier getesteten Modelle.

Auszug der Konfigurationen der getesteten Modelle:

VGG19	InceptionV3
Optimierer: RMSprop Lernrate: 0.0005 vortrainiert Top-Layer: (1024,7)	Optimierer: RMSprop Lernrate: 0.0005 vortrainiert Top-Layer: (1024,7)
Xception	**eigenes Modell**
Optimierer: NADAM Lernrate: 0.0005 vortrainiert Top-Layer: (1024,512,7)	Optimierer: NADAM Lernrate: 0.0005 nicht vortrainiert Top-Layer: (144,7)

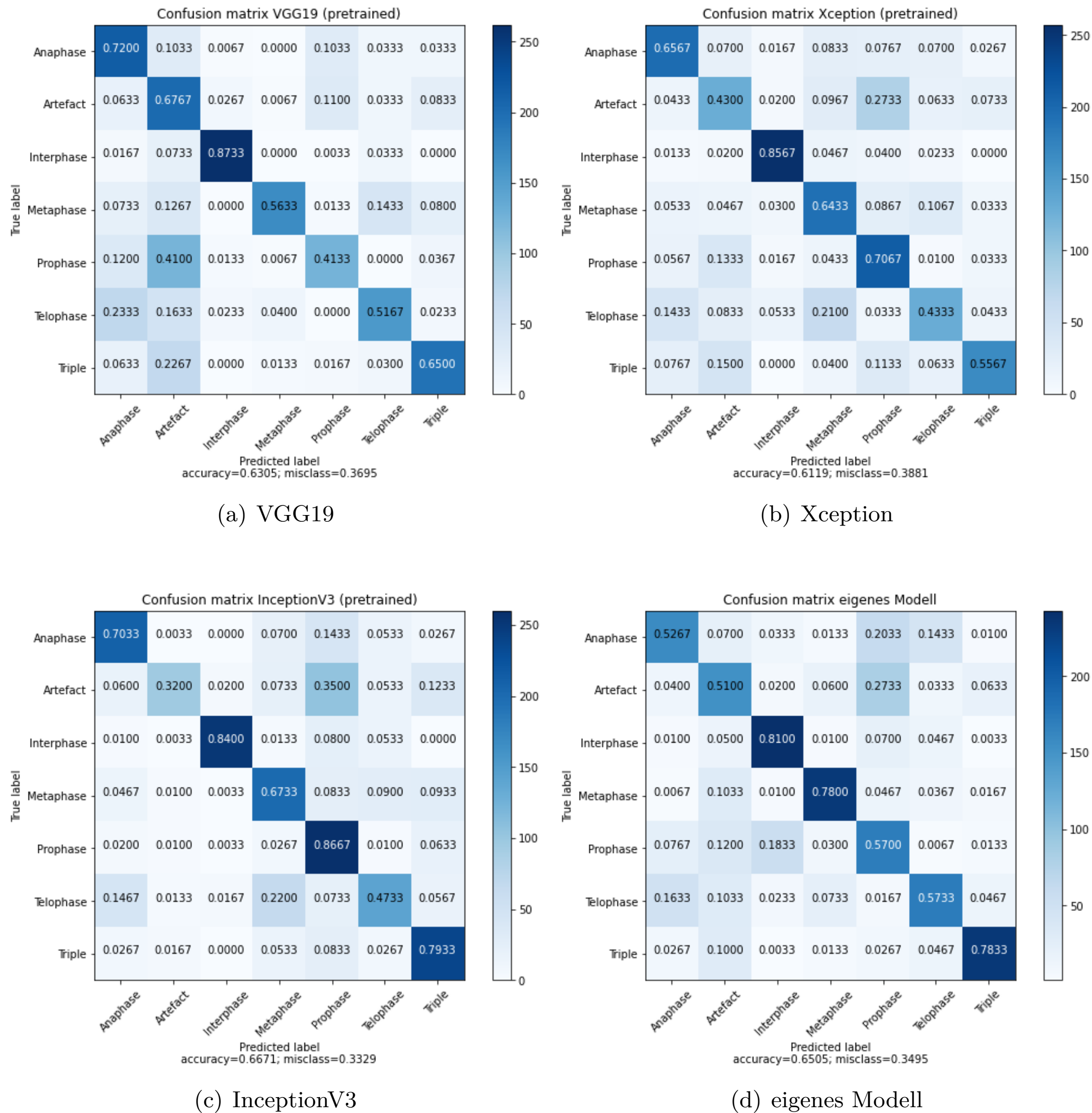
Ergebnisse:

Modell	Top-1-Accuracy	Top-2-Accuracy	Top-3-Accuracy
VGG19	0.6957	0.8667	0.9314
InceptionV3	0.6967	0.8567	0.9257
Xception	0.6323	0.8128	0.9198
eigenes Modell	0.6600	0.8543	0.9224

Tabelle 9: Ergebnisse der Tests mit Keras Applications und dem eigenen Modell für 7 Klassen

Als beste Modelle stellten sich VGG19 und InceptionV3 heraus. Überraschend ist, dass das eigene Modell dem vortrainierten Xception deutlich überlegen war. Alle Modelle konnten gute Werte der Top-3-Accuracy erreichen. Da es sich hier aber um ein Sieben-Klassen-Problem handelt, hat dies keine große Aussagekraft.
Um die Ergebnisse besser analysieren zu können, wurde zusätzlich nach jedem Training die entsprechende Konfusionsmatrix erstellt, wobei hier der Validierungsdatensatz als Grundlage genutzt wurde. Die resultierenden Matrizen sind in Abb. 29 gezeigt. Die Abweichungen der Werte für die Genauigkeit in den Plots der Konfusionsmatrizen von den Werten aus Tabelle 9 resultieren daraus, dass hier für die Berechnung die letzte Trainingsepoche ausgewertet wurde und nicht die beste. Wir halten einige Beobachtungen fest, die wir sofort ablesen können. Die Klasse „Interphase“ ist offensichtlich für alle genutzten Modelle vergleichsweise einfach zu erkennen, während die Klasse „Telophase“ unterdurchschnittlich gut detektiert wurde. Eine weitere Gemeinsamkeit der Modelle ist, dass zu den besonders oft auftauchenden falschen Zuordnungen die Verwechslung der wahren Klassen „Anaphase“ mit der falschen Klasse „Telophase“ gehört sowie die Verwechslung der wahren Klassen „Artefact“ mit der falschen Klasse „Prophase“. Bezüglich der anderen Klassen bzw. anderer Kombinationen von falschen Zuordnungen gibt es unter den Modellen teils sehr deutliche Unterschiede. Auffallend ist, dass VGG19 besonders oft die falsche Klasse „Artefact“ statt der wahren Klasse „Prophase“ vorhergesagt hat. Zudem ist die Erkennung von „Metaphase“ schlechter als bei allen anderen getesteten Modellen. Ausschlaggebend für das trotzdem insgesamt gute Abschneiden gegenüber den anderen Modellen sind unter anderem die höhere Genauigkeit bei der Erkennung von „Artefact“ und die geringere Häufigkeit falscher Vorhersagen der Klassen „Metaphase“ und „Prophase“. Die zueinander ähnliche Architektur der beiden Modelle InceptionV3 und Xception spiegelt sich auch in ihren Konfusionsmatrizen wieder. Sie haben oft bei den selben Kombinationen von Klassen eher gute oder schlechte Werte erreicht. Trotzdem ist InceptionV3 in fast allen Fällen

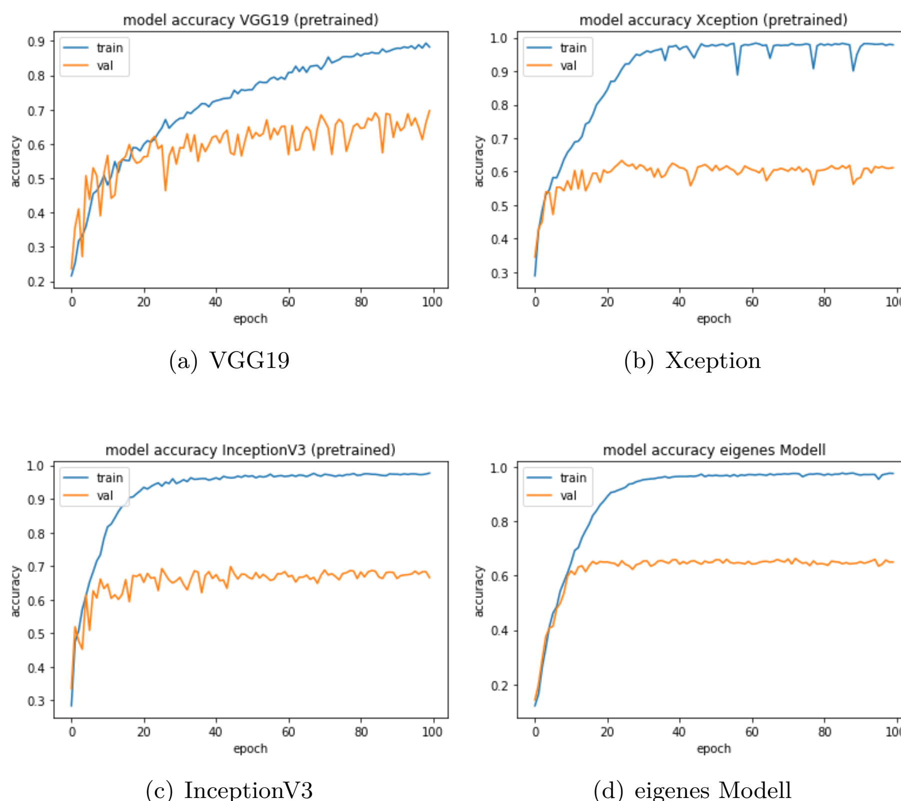
überlegen. Insbesondere die überdurchschnittlich gute Erkennung von „Prophase“ unterscheidet sich von den anderen Modellen. Eine deutliche Schwäche gegenüber Xception hat es lediglich bei der Erkennung von „Artefact“. Das eigene Modell hat am seltensten die Klasse „Anaphase“ richtig erkannt und am häufigsten die falschen Klassen „Prophase“ und „Telophase“ statt der wahren Klasse „Anaphase“ vorhergesagt. Darüber hinaus sind die Werte dieses Modells seiner Validation-Accuracy entsprechend nicht wesentlich schlechter als die der anderen Modelle.



(a) VGG19 (b) Xception

(c) InceptionV3 (d) eigenes Modell

Abbildung 29: Konfusionsmatrizen für verschiedene Modelle für 7 Klassen



(a) VGG19 (b) Xception

(c) InceptionV3 (d) eigenes Modell

Abbildung 30: Trainingsverläufe verschiedener Modelle für 7 Klassen

Für eine noch bessere Beurteilung der Frage, welche Klassen den Modellen grundsätzlich mehr Probleme bereiten als andere, wurden in einer Testreihe sieben Modelle jeweils darauf trainiert, genau eine der Klassen zu erkennen. Als Ausgangsmodell wurde aufgrund der relativ kurzen Trainingsdauer das mit dem ImageNet-Datensatz vortrainierte InceptionV3 ausgewählt und mit der in Kap. 7.2.2 vorgestellten Konfiguration implementiert. Jedes Modell wurde auf die Erkennung von zwei Klassen trainiert. Die erste Klasse ist eine der sieben Kategorien „Anaphase“, „Artefact“, „Interphase“, „Metaphase“, „Prophase“, „Telophase“ und „Triple“. Die andere Klasse entspricht jeweils der Nicht-Zugehörigkeit zur ersten Klasse. Das Trainingsmaterial für die erste Klasse besteht für jedes Modell aus 1000 Bildern. Der Trainingsdatensatz für die zweite Klasse enthält immer die sechs anderen Klassen, wobei jeweils 166 Bilder verwendet wurden, also je Modell 996 Bilder für die zweite Klasse. Das Ungleichgewicht zwischen den beiden Klassen wurde kompensiert mithilfe einer Methode aus der Programm-Bibliothek Scikit-Learn, welche die Klassen entsprechend ihrer Anzahl von Trainingsbildern unterschiedlich gewichtet. Die Trainigsdauer betrug hier jeweils wieder 100 Epochen. Die erreichten Genauigkeiten der Einzelerkennungsmodelle sind in Tabelle 10 zu sehen. Sie zeigen, dass die Klasse „Interphase“ mit deutlichem Abstand am einfachsten zu erkennen war. Dies entspricht auch den Beobachtungen, die wir bei den Sieben-Klassen-Modellen machen konnten. Auch das schlechte Abschneiden der „Telophase“-Erkennung deckt sich mit den Zahlen aus den Konfusionsmatrizen der Modelle für sieben Klassen.

Klasse	Val-Acc.
Anaphase	0.8829
Artefact	0.8311
Interphase	0.9498
Metaphase	0.7029
Prophase	0.8127
Telophase	0.6891
Triple	0.8428

Tabelle 10: Genauigkeiten der Einzelerkennungmodelle

7.4 Anpassung der Evaluierung

Um das Problem der fließenden Übergänge zwischen verschiedenen Phasen des Zellzyklus in die Beurteilung eines Modells mit einfließen zu lassen, kann es hilfreich sein, bestimmte Abstufungen der Begriffe „richtig“ und „falsch“ bei der Bewertung einer durch das Modell generierten Vorhersage zuzulassen. So macht es beispielsweise Sinn, eine Verwechslung der beiden Klassen „Interphase“ und „Prophase“ in gewisser Weise zu tolerieren, während eine Verwechslung von „Interphase“ und „Metaphase“ extra bestraft werden kann. Die Nichteindeutigkeit mancher Zellbilder, die bei einer großen Datenmenge zwangsläufig zumindest punktuell auftritt, kann zudem berücksichtigt werden, indem man eine Unterscheidung zwischen einer eindeutigen Vorhersage und einer uneindeutigen Vorhersage einführt. Diese Möglichkeiten wurden in einem Python-Skript implementiert. Dazu wurden drei Parameter eingeführt:

ε : Der Wert dieses Parameters definiert die Schwelle, ab der eine Vorhersage des Modells als uneindeutig betrachtet wird. Genauer: Sei $\tilde{\mathbf{y}}_{\mathbf{pred}}(\mathbf{x})$ der sortierte Output-Vektor für ein Eingangsbild $\mathbf{x}$, wobei $(\tilde{y}_{pred})_1 \geqslant (\tilde{y}_{pred})_2 \geqslant \ldots \geqslant (\tilde{y}_{pred})_K$ gilt und K die Anzahl der Klassen bezeichnet. Dann ist die Vorhersage $\mathbf{y}_{\mathbf{pred}}$ als „uneindeutig“ zu bewerten, falls $(\tilde{y}_{pred})_1 - (\tilde{y}_{pred})_2 < \varepsilon$ gilt. Da die Output-Neuronen prinzipiell zwischen 0 und 1 liegen, soll für den übergebenen Wert des Parameters gelten: $\varepsilon \in [0, 1]$.

α : Dies ist der **Duldungsfaktor**. Er definiert die Gewichtung, mit der die Verwechslung bestimmter Klassen in der angepassten Evaluierung toleriert wird. Es soll $\alpha \in [0, 1]$ gelten. Hat der Parameter den Wert 0, dann wird jede falsche Zuordnung des Modells als „falsch“ betrachtet. Gilt $\alpha \in (0, 1]$, so können die im Vorfeld als vertretbar definierten Verwechslungen in die Endbewertung als „teilweise richtige“ Vorhersagen einfließen, wobei $\alpha = 1$ bedeutet, dass die eigentlich falsche Vorhersage als „komplett richtig“ gewertet wird.

β : Der **Bestrafungsfaktor**. Er definiert die Gewichtung, mit der sich die Verwechslung bestimmter Klassen negativ auf das angepasste Genauigkeitsmaß auswirkt. Es soll $\beta \leqslant 0$ gelten. Nullsetzen des Parameters bedeutet, dass keine falsche Zuordnung extra bestraft wird. $\beta < 0$ bewirkt eine Bewertung der vorher festgelegten schlimmen Verwechslungen als „besonders falsch“. Der Wert für β sollte nicht zu klein gewählt werden, da das die angepasste Endbewertung des Modells zu stark verzerren würde.

Es sei $\mathbb{X}$ ein Test-Datensatz mit N Elementen. Wir bezeichnen mit

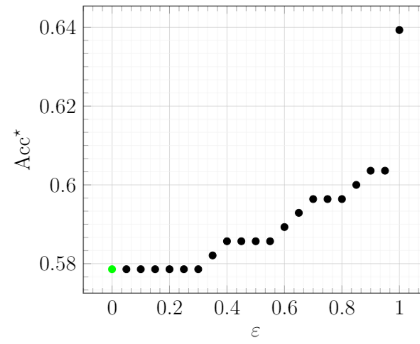
TP die Anzahl der korrekten Vorhersagen,
e die Anzahl der als „richtig“ gewerteten Vorhersagen obwohl falsch,
a die Anzahl der geduldeten Verwechslungen und mit
b die Anzahl der schlimmen Verwechslungen.

Dann können wir angepasste Genauigkeitsmaße gemäß

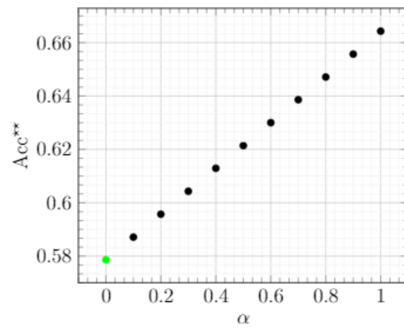
$$\mathrm{Acc}^{\star} := (TP + e)/N \tag{7.4.1}$$

$$\mathrm{Acc}^{\star\star} := (TP + e + \alpha a + \beta b)/N \tag{7.4.2}$$

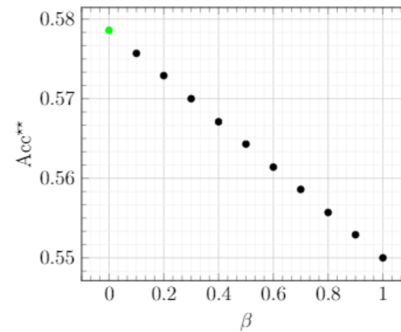
definieren. Wir können $\mathrm{Acc}^{\star}$ als die Genauigkeit unter Berücksichtigung ungenauer Vorhersagen verstehen. Ist die zum zweitgrößten Wert zugehörige Klasse richtig (die Vorhersage damit also falsch) und weicht der Wert nur unterhalb der Schwelle ε vom Wert des stärksten Output-Neurons ab, dann gilt die Vorhersage als „richtig“. In der Python-Implementierung wurde nach dem gleichen Prinzip auch der Fall erfasst, dass das drittstärkste Output-Neuron nur weniger als ε schwächer aktiviert ist als das stärkste Neuron und die wahre Klasse anzeigt. $\mathrm{Acc}^{\star\star}$ ist die angepasste Genauigkeit unter Berücksichtigung der uneindeutigen Vorhersagen *und* unter Einbeziehung der (vorher festgelegten) bestimmten akzeptablen bzw. besonders schlimmen Verwechslungen.
Im Folgenden werden die Ergebnisse einiger Tests mit diesem Programm anhand des eigenen Modells vorgestellt. Ein Test-Datensatz mit einer Anzahl von 280 Bildern (welche nicht bereits als Trainings- oder Validierungsbilder genutzt wurden) bildete die Grundlage der neuen Modell-Evaluierung. Getestet wurde das eigene Modell für sieben Klassen. Die klassische Genauigkeit bezüglich des neuen Test-Datensatzes betrug weniger als die während des Trainings gemessene Validation-Accuracy. Das liegt womöglich daran, dass der Test-Datensatz wesentlich kleiner ist als der Validation-Datensatz, welcher 2100 Bilder für das Sieben-Klassen-Problem enthielt.
Zunächst wurde untersucht, welche Auswirkungen Veränderungen der Parameter ε, α und β auf die beiden neuen Genauigkeitsmaße haben. Abb. 31(a) zeigt das Verhalten von $\mathrm{Acc}^{\star}$ in Abhängigkeit von ε, wobei hier die Parameter α und β auf Null gesetzt wurden. Es kann beobachtet werden, dass die angepasste Genauigkeit $\mathrm{Acc}^{\star}$ ungefähr in einer treppenförmigen Art und Weise steigt, je höher der Wert für ε gewählt wird. Abb. 31(b) und (c) zeigen die Auswirkungen von Veränderungen des Duldungsfaktors bzw. des Bestrafungsfaktors auf das Genauigkeitsmaß $\mathrm{Acc}^{\star\star}$, wobei jeweils die anderen beiden Parameter auf Null gesetzt wurden. Der zugehörigen Formel aus (7.4.2) entsprechend, sind es in beiden Fällen lineare Zusammenhänge.



(a) $\alpha = 0$, $\beta = 0$



(b) $\varepsilon = 0$, $\beta = 0$



(c) $\varepsilon = 0$, $\alpha = 0$

Abbildung 31: Auswirkungen der einzelnen Parameter auf die angepassten Genauigkeiten (eigenes Modell)

Anschließend wurde eine Auswahl möglicher Kombinationen von ε, α und β festgelegt. Die jeweils von dem Python-Programm zurückgegebenen Werte sind in Tabelle 11 abgebildet.

ε	α	β	TP	e	a	b	Acc	Acc*	Acc**
0	0	0	162	0	24	8	0.5786	0.5786	0.5786
0.5	0	0	162	2	16	6	0.5786	0.5857	0.5857
0	1	0	162	0	24	8	0.5786	0.5786	0.6643
0.5	1	0	162	2	16	6	0.5786	0.5857	0.6429
0	0	-1	162	0	24	8	0.5786	0.5786	0.5500
0.5	0	-1	162	2	16	6	0.5786	0.5857	0.5643
0	1	-1	162	0	24	8	0.5786	0.5786	0.6357
0.5	1	-1	162	2	16	6	0.5786	0.5857	0.6214

Tabelle 11: Ergebnisse der Tests mit der angepassten Evaluierung

7.5 Pruning

Pruning (Beschneiden) ist eine Methode, welche auf verschiedene Weise ein trainiertes Modell effizienter machen kann. Unter anderem lässt sich damit die Größe des Modells, also sein Speicherplatzbedarf, reduzieren. Das vereinfacht insbesondere die Nutzung von Deep-Learning-Modellen auf dem Smartphone. Beim Pruning können wir entweder die Anzahl der Neuronen reduzieren (Neuron Pruning) oder die Anzahl der Gewichte (Weight Pruning). In beiden Fällen ist grundsätzlich mit einem Verlust an Genauigkeit zu rechnen. Die Schwierigkeit besteht deswegen darin, eine gute Balance zwischen Reduktion des Speicherbedarfs und der Verschlechterung der Performance zu finden. Wie wir noch sehen werden, kann es aber sogar passieren, dass das Modell nach dem Beschneiden eine etwas bessere Genauigkeit vorweist.
Wir werden nun näher auf das Weight-Pruning eingehen. Hierbei werden die Werte bestimmter Gewichte auf Null gesetzt, sie werden also faktisch herausgelöscht. Eine einfache Art des Weight Pruning ist es, einen Spärlichkeitsfaktor (sparsity) festzulegen. Liegt dieser z.B. bei 0.5, dann wird die Hälfte der Gewichte gelöscht und zwar gerade diejenigen mit den kleinsten Werten [60]. In TensorFlow trainieren wir zunächst das Modell wie gehabt und feintunen anschließend mit dem beschnittenen Modell, d.h. wir trainieren noch einmal. Nach diesem Schritt erhalten wir allerdings erst ein Modell, dass größer als das Original ist. Für den Pruning-Vorgang ist die Verwendung zusätzlicher Variablen notwendig, die nun noch entfernt werden müssen. Außerdem verbrauchen die seriell (also in Bits) gespeicherten Gewichte, selbst mit dem Wert Null, immer noch Speicherplatz. Dies wird mit einem Kompressionsalgorithmus, dem strip pruning gelöst. [61]

Für die Experimente zum Pruning wurde eine Auswahl der in den beiden vorangegangenen Unterkapitel benutzt. Die Rahmenbedingungen sind die selben, wie in den vorangegangenen Versuchen.

vortrainiertes Xception:
Zunächst wurde die einfachere Variante der Beschneidung ausgetestet, das **constant Pruning** (konstantes Beschneiden), bei dem vor dem Training eine globale Pruning-Rate festgelegt und über das gesamte Training beibehalten wird. Es wird also gleich während der ersten Epoche das Ausgangsmodell mit dieser Rate beschnitten und dies bis zum Ende des Trainings so beibehalten. In Abb. 32 sehen wir die Ergebnisse, wobei hier der Zusammenhang zwischen erreichter Validation-Accuracy und der Pruning-Rate hervorgehoben wurde. Gut zu sehen ist, dass sich die Genauigkeit für einige der Pruning-Raten sogar verbessern ließ. Jedoch gibt es eine Schwelle, die irgendwo im Intervall $(0.7, 0.8)$ liegt, ab der offenbar zu viele der Gewichte gelöscht sind und es dem Modell nicht mehr möglich ist, vernünftige

Vorhersagen zu liefern. In Abb. 33 ist eine Auswahl von Genauigkeits-Lernkurven dargestellt. Deutlich erkennbar ist der Einfluss der Beschneidung auf das Trainingsverhalten des Modells. Er schlägt sich in einer immer stärkeren Fluktuation der Genauigkeit nieder, je stärker beschnitten wird.

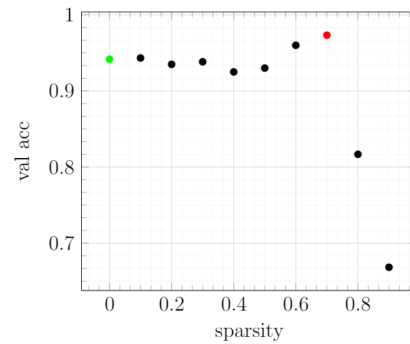
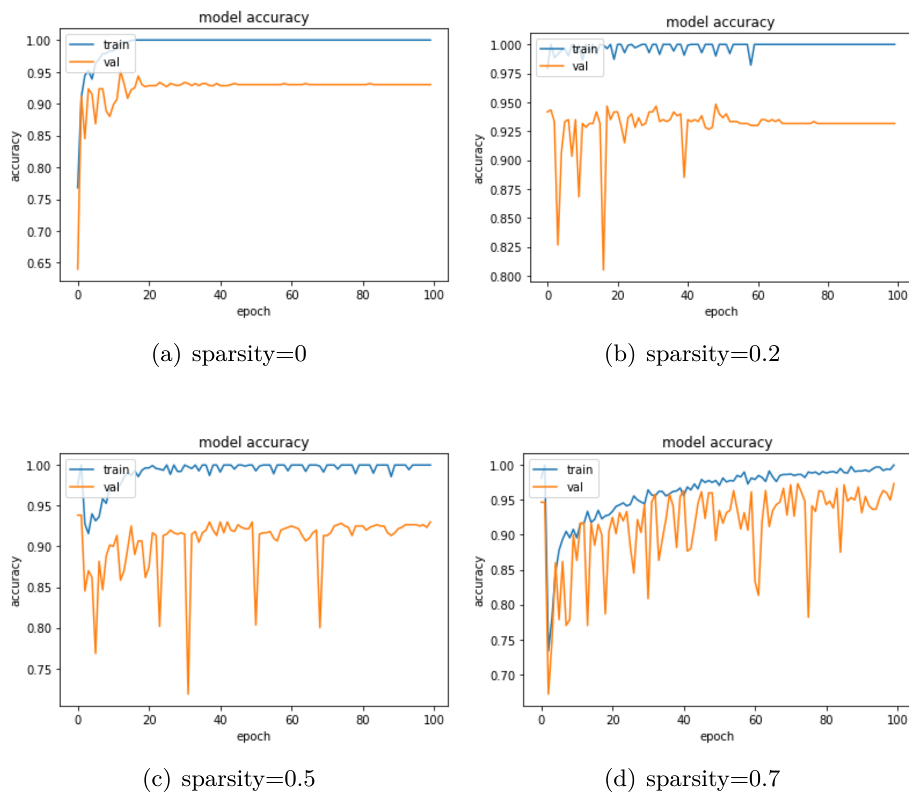


Abbildung 32: Pruning des vortrainierten Xception mit verschiedenen Werten für konstante Spärlichkeit



(a) sparsity=0 (b) sparsity=0.2

(c) sparsity=0.5 (d) sparsity=0.7

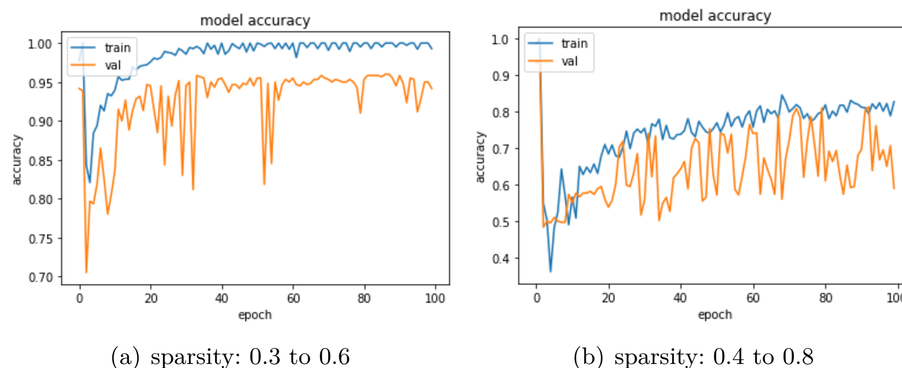
Abbildung 33: Trainingsverläufe des vortrainierten Xception bei verschiedenen Werten für konstantes Pruning

Überraschenderweise war festzustellen, dass die Speichergröße des beschnittenen und komprimierten Modells in allen Fällen die selbe war. Intuitiv war es naheliegend, anzunehmen, dass die Speicherplatzreduktion mit steigender Pruning-Rate zunimmt. Die ursprüngliche Speichergröße beläuft sich auf 112.5 MB; die konstant beschnittenen Modelle sowie die polynomiell beschnittenen Modelle haben alle eine Größe von 92 MB. Das entspricht einer Reduktion um 18.2%.
Neben der konstanten Beschneidung gibt es die Möglichkeit ein **polynomielles Pruning** durchzuführen. Hier wird eine Startepoche festgelegt, bei der mit einem bestimmten Anfangsfaktor für die Beschneidung gestartet wird und eine Endepoche bei der dann mit einem anderen Beschneidungsfaktor geendet wird. In unserem Fall wurden verschiedene Kombinationen von Anfangs- und Endfaktor des Prunings getestet und dabei immer die erste Trainingsepoche als Anfangsepoche und die letzte Trainingsepoche als die Endepoche des Pruning gewählt. Die Ergebnisse sehen wir in Tabelle 12.

initiale Spärlichkeit	finale Spärlichkeit	Val-Acc
0	0	0.9417
0.3	0.6	0.96
0.3	0.7	0.97
0.4	0.7	0.9733
0.4	0.8	0.8083
0.5	0.8	0.8117

Tabelle 12: Ergebnisse der polynomiellen Beschneidung des vortrainierten Xception

Auch hier können wir eine Verbesserung der Validation-Accuracy bei einigen der Konfigurationen beobachten. Wir können sehen, dass das Intervall der Pruning-Faktoren weder zu groß noch zu klein sein darf.



(a) sparsity: 0.3 to 0.6

(b) sparsity: 0.4 to 0.8

Abbildung 34: Trainingsverläufe des vortrainierten Xception bei verschiedenen Werten für polynomielles Pruning

vortrainiertes InceptionV3:
Die Ergebnisse der Experimente mit dem vortrainierten InceptionV3 sind weitestgehend vergleichbar mit denen von Xception. Abb. 35 zeigt ein ähnliches Verhalten beim konstanten Beschneiden wie zuvor, wenngleich das InceptionV3 bereits ohne Pruning ein relativ instabiles Trainingsverhalten bzgl. der Validation-Accuracy aufweist. Eine Spärlichkeit von 80% und mehr führt zu einem Versagen des Modells. Im Unterschied zu den vorangegangenen Experimenten mit Xception konnte hier die Validation-Accuracy nicht mithilfe des polynomiellen Pruning verbessert werden. Das ursprüngliche Modell hat eine Speichergröße von 228 MB. Die beschnittenen Modelle haben alle eine Größe von 156 MB. Das entspricht einer Reduktion um 31,6%.

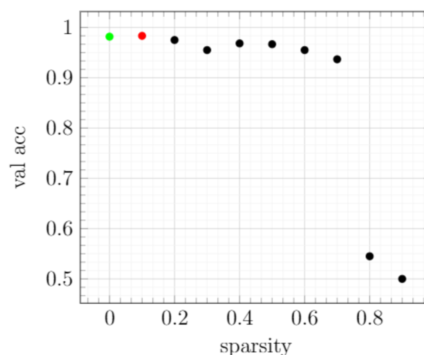


Abbildung 35: Pruning des vortrainierten InceptionV3 mit verschiedenen Werten für konstante Spärlichkeit

initiale Spärlichkeit	finale Spärlichkeit	Val-Acc
0	0	0.9817
0.3	0.6	0.9633
0.3	0.7	0.9517
0.4	0.7	0.9550
0.4	0.8	0.5000
0.5	0.8	0.5000

Tabelle 13: Ergebnisse der polynomiellen Beschneidung des vortrainierten InceptionV3

8 Diskussion

8.1 Nutzen des Fine-Tunings

Ein Problem, auf das man zwangsläufig bei der Optimierung von Deep-Learning-Modellen stößt, ist die Tatsache, dass es keine allgemeingültige Gesetzmäßigkeiten gibt, die eine eindeutige Wahl von Hyperparametern implizieren. Daher ist es auch nach der Durchführung von Fine-Tuning schwierig zu beurteilen, ob das Modell bzgl. des betrachteten Hyperparameters wirklich optimal ist. Allein der Begriff „optimal“ ist in diesem Kontext schwer zu formalisieren. Bezogen auf die in Kapitel 7.1 präsentierten Ergebnisse könnte eine passende Definition in etwa wie folgt lauten: Ein oder mehrere Hyperparameter eines Modells werden bezüglich einer vor Beginn der Tests definierten (diskreten und endlichen) Menge von möglichen Hyperparameter-Werten so optimiert, dass eine maximale Validation-Accuracy während des Trainings erreicht wird. Der zuvor beschriebene Ansatz bestand darin, die Modelloptimierung in einzelne Unterexperimente aufzuteilen, bei denen jeweils ein oder wenige Parameter getestet wurden, und die damit gewonnenen Parameterwerte sukzessive weiterzuverwenden. Die Ergebnisse aus Abschnitt 7.1 zeigen, dass das eine sinnvolle Methodik sein kann. Jedoch ist es notwendig, die Anzahl an Möglichkeiten von Hyperparameterkonfigurationen so zu beschränken, dass die Tests noch praktikabel sind. Daher ist eine Vorab-Recherche hilfreich, damit man vor Beginn der Tests bereits einen Ansatzpunkt hat, von dem aus man starten kann.

Ein Aspekt der Hyperparametersuche mit dem Keras Tuner, welcher kritisch hinterfragt werden sollte, ist die Auswahl des besten Ergebnisses (in unserem Fall der besten Validation-Accuracy) nach einer in der Regel reduzierten Anzahl von Epochen. Da die Abweichungen der Genauigkeit bei verschiedenen Konfigurationen teilweise sehr klein sind und weil eine Trainingseinheit bei Wiederholung nie identische Werte hervorbringt, ist es theoretisch möglich, dass auch die Rangfolge der besten Hyperparameter-Kombinationen nach einer Sitzung des Tuners bei erneuter Durchführung leicht verändert sein könnte. Es ist in der Praxis jedoch nicht sinnvoll, jede denkbare Eventualität in dieser Hinsicht in die Tests mit einzubinden. Klüger wäre eine klare Maßgabe zu Beginn eines Projektes, die definiert, in welcher Größenordnung man die durch die zufällige Natur der neuronalen Netze unvermeidbaren Unsicherheiten akzeptiert. In unserem Fall könnte dies z. B. eine Zahl $\varepsilon = 0.015$ sein, da dies der in Kap. 6 erwähnten beobachteten Abweichung bei wiederholtem Training entspricht.

Über die vorgestellten Experimente hinaus kann man noch weitere Hyperparameter in die Suche nach besseren Modellkonfigurationen einbeziehen. Zudem beschränkt sich der Einsatz des Keras Tuners in dieser Arbeit auf die zufällige Suche. TensorFlow bietet hierzu noch weitere Methoden an.

Des Weiteren können die vordefinierten Parametermengen vergrößert sowie die Größe des Modells in Form von weiteren Schichten noch erhöht werden.

8.2 Nutzen des Transfer-Learnings

Erwartet wurde hier eine klar erkennbare Überlegenheit der Modelle, welche bereits mit Vorgewichten ausgestattet wurden, gegenüber jenen ohne Pre-Weights. Intuitiv liegt diese Annahme nahe, da die vortrainierten Modellen mit einem sehr großen Datensatz mit vielen verschiedenen Klassen trainiert wurden und dabei bereits sehr gute Ergebnisse abliefern konnten. Die Ergebnisse aus Kap. 7.2 lassen jedoch darauf schließen, dass Modelle mit Vorgewichten nicht grundsätzlich immer besser arbeiten. Das könnte daran liegen, dass unser dort betrachteter Anwendungsfall mit zwei Klassen so simpel ist, dass der Vorzug eines bereits mit Millionen von Bildern aus tausenden Klassen trainierten Modells nicht besonders stark zur Geltung kommen kann. Da der Unterschied zwischen einer Zelle in

der Interphase und einer in der Metaphase optisch ziemlich deutlich ist, können die hierfür hilfreichen Feature-Erkennungen relativ leicht antrainiert werden, wie auch die sehr guten Ergebnisse des einfachen eigenen Modells zeigen. Vermutlich wächst der Vorteil durch das Transfer-Learning mit der Komplexität des vorliegenden Problems. Der Vorsprung des besten vortrainierten Modells gegenüber dem besten nicht vortrainierten Modell, betrug 0.0050. Außerdem war die Trainingsdauer bei den vortrainierten Modellen geringer als bei den nicht vortrainierten. Dem gegenüber steht, dass zunächst herausgefunden werden musste, welche Konfiguration der Top-Layer überhaupt zu einem guten Ergebnis führt. Insofern ist der Programmieraufwand in Fall der vortrainierten Modelle deutlich höher als bei den nicht vortrainierten Modellen. In beiden Fällen mussten (wie auch beim Fine Tuning) mit den Optimierungsalgorithmen experimentiert werden. Bei der Struktur der nicht vortrainierten Modellen musste darüber hinaus aber lediglich die Anzahl der Neuronen in der Output-Schicht angepasst werden. Aus den in Kapitel 7.2 vorgestellten Ergebnissen lässt sich somit keine eindeutige Empfehlung darüber ableiten, ob der Einsatz von nicht vortrainierten oder von vortrainierten Modellen nutzbringender ist. Die Effektivität eines vortrainierten Modells, angewandt auf ein benutzerspezifisches Problem, hängt stark von der Konfiguration der trainierbaren Schichten ab. Es konnte gezeigt werden, dass dies gut mittels eines Fine-Tunings realisierbar ist. Die zuvor beschriebenen Implikationen des Fine-Tunings können also auf diesen Fall übertragen werden. Es müssen aber noch weitere Überlegungen berücksichtigt werden. Es ist in Anbetracht des deutlich höheren Speicherverbrauchs der Keras Applications (vgl. [54]) im Vergleich zum einfachen Modell (3.8 MB für das Sieben-Klassen-Problem) eine legitime Frage, inwieweit die nicht sehr viel höheren Validierungsgenauigkeiten der Keras Applications deren Einsatz rechtfertigen. Selbst das getestete EfficientNetB0 hat einen deutlich höheren Bedarf an Speicherplatz als das eigene Modell. Sowohl beim Zwei-Klassen- als auch beim Sieben-Klassen-Problem der Zellphasenklassifikation konnte das eigene Modell mit den anderen Modellen mithalten.

8.3 Schwierigkeiten bei der Klassenerweiterung

Grundsätzlich ist es zu erwarten, dass ein Problem mit der Anzahl der Klassen schwieriger wird. Der Abfall der erreichten Validierungsgenauigkeiten von ca. 30% bei vergleichbarer Anzahl von Trainingsbildern je Klasse war allerdings überraschend stark. Die in der praktischen Umsetzung größte Hürde bei der Klassenerweiterung war die wesentlich längere Trainingsdauer aufgrund der höheren Anzahl an Trainingsbildern. Daher stellen die in dieser Arbeit vorgestellten Ergebnisse Empfehlungen dar, wie man ein Modell konfigurieren könnte, bevor man es dann über eine lange Zeit (womöglich bis zu einigen Wochen) trainieren kann. Ausschlaggebend für das Erreichen einer deutlich höheren Validation-Accuracy als in den vorgestellten Experimenten ist ein größerer Datensatz. Medipan stehen solche mit über 10000 Bildern zur Verfügung, somit können wir optimistisch sein, das dies auch praktisch gelingen kann.
Die Konfusionsmatrizen ermöglichten uns bereits eine Analyse der Performance verschiedener getesteter Modelle. Dort konnten wir sehen, dass im Wesentlichen genau die Klassen besonders verwechslungsgefährdet sind, die auch optisch ähnlich zueinander sind. In dieser Hinsicht ist das Verhalten der Modelle wenig überraschend. Die Ergebnisse der Tests mit den Einzelerkennungsmodellen aus Abschnitt 7.3 geben einen Eindruck darüber, welche der Klassen schwerer zu erkennen sind als andere. Somit könnte dies genutzt werden um das Sieben-Klassen-Problem etwas abzuschwächen und gerade die besonders schwer zu detektierenden Klassen herauszunehmen, sofern dies für den Anwendungsfall noch sinnvoll ist. Beispielsweise könnte man eine Zusammenfassung der beiden Klassen „Anaphase“ und „Telophase“ in Betracht ziehen, was wahrscheinlich einen Gewinn an gemessener Genauigkeit bewirken würde, allerdings bei gleichzeitigem Verlust an Information.

8.4 Berücksichtigung der biologischen Gegebenheiten

Das Problem der fließenden Übergänge zwischen verschiedenen Zellstadien wurde mithilfe einer angepassten Evaluierungsmethode beantwortet. Die Aussagekraft der vorgestellten Genauigkeitsmaße hängen von der Wahl der Parameter ab. Hierfür ist ein medizinischer Sachverstand gefragt. Es ist also sinnvoll, dass jemand vom Fach genau beurteilt, inwieweit bestimmte Abweichungen der Modell-Vorhersage von der wahren Klasse tolerierbar oder unbedingt zu vermeiden sind. Die durchgeführten Experimente aus Abschnitt 7.4 liefern einen ersten Eindruck über die Auswirkungen der Parameterwahl. Besonders die Definition der Schwelle, ab der eine Vorhersage als ungenau deklariert wird, scheint schwierig einstellbar zu sein.
Natürlich sind noch weitere Optionen als ein verändertes Genauigkeitsmaß denkbar, etwa die Erstellung von Zwischenklassen. So könnte beispielsweise eine Klasse „Anaphase/Telophase“ diejenigen Zellbilder enthalten, die nicht eindeutig der einen oder der anderen Klasse zuzuordnen sind. Das Problem bei diesem Ansatz ist jedoch, dass es unter Umständen zu einer deutlichen Unterrepräsentierung einiger Klassen kommen kann. In dem von Medipan zur Verfügung gestellten Datensatz ist beispielsweise die Klasse „Interphase“ bereits deutlich überrepräsentiert. Teilt man ohnehin schwach vertretene Klassen noch in Zwischenklassen auf, steigert sich die Nicht-Ausbalanciertheit der Daten enstprechend weiter.

8.5 Möglichkeiten zur Modellkomprimierung

Die durchgeführten Tests des Weight-Pruning-Verfahrens ergaben eine relativ hohe Speichergrößenreduktion. Die Tatsache, dass die Genauigkeit der beschnittenen Modelle die der unbeschnittenen Varianten manchmal sogar übertreffen, spricht ebenfalls für diese Methode. Jedoch haben die durchgeführten Experimente gezeigt, dass es notwendig ist, vor Beginn des eigentlichen Prunings einige Vorab-Tests durchzuführen, um ein völliges Versagen des resultierenden beschnittenen Modells zu vermeiden. Dies kann genau im Stil der in Kap. 7.5 präsentieren Experimente vonstattengehen. Die Schwelle, ab welcher der Pruning-Faktor zu hoch ist, variiert von Modell zu Modell. Daher kann auch hier keine allgemeingültige Regel für die Konfiguration des Beschneidungsverfahrens herangezogen werden. Die vorgestellten Experimente zeigen einen Ausschnitt über alle möglichen Varianten des Prunings. Es sind noch weitere Test-Werte für den konstanten Pruning-Faktor sowie mehr Kombinationen für Anfangs- und Endfaktor beim polynomiellen Pruning denkbar. Durch die Ergebnisse erhalten wir aber für beide Modelle eine gute Vorstellung, in welchem Bereich die optimale Konfiguration der Beschneidung jeweils liegen könnte.
Ein relevanter Faktor für den Nutzen dieses Verfahrens ist auch der voraussichtliche Einsatzort des Modells. Handelt es sich um einen schnellen PC, so hat eine Speichergrößenreduktion im 2-stelligen MB-Bereich keinen besonders großen Nutzen. Für den Einsatz auf dem Smartphone kann es hingegen durchaus sinnvoll sein, so viel unnötigen Speicherplatzverbrauch wie möglich zu vermeiden. Der Programmier- und Zeitaufwand für die Durchführung einer Gewichtsbeschneidung ist vergleichbar mit dem des Fine-Tunings eines bis zweier Hyperparameter. Da die Lernkurven der beschnittenen Modelle eine maximal sehr leicht steigende Tendenz aufweisen, können wir davon ausgehen, dass die Wahl der Epochenanzahl von 100 ausreichend ist.
Neben dem Pruning gibt es noch weitere Möglichkeiten, um die Effizienz eines Modells bezüglich seines Speicherplatzbedarfs zu steigern. Zu den gängigen Verfahren gehören Quantisierung und Clustering. Insbesondere Quantisierungsmethoden lassen sich gut mit dem Pruning kombinieren [62].

9 Fazit

Das Ziel dieser Arbeit war die Erforschung von Möglichkeiten der Optimierung von Deep-Learning-Modellen zur Erkennung verschiedener Zellstadien. In allen betrachteten Teilbereichen der Modelloptimierung konnten die entsprechenden Techniken erfolgreich eingesetzt werden. Der Fokus lag hierbei auf den Konzepten des Fine-Tunings und Transfer-Learnings sowie auf dem Pruning-Verfahren und einer Anpassung der Modell-Evaluierung. Das Fine-Tuning unseres eigenen Modells ermöglichte eine deutliche Verbesserung seiner Validation-Accuracy. Die Verbesserungen der Konfigurationen von Faltungsschichten und dichten Schichten sowie eine geeignete Wahl von Optimierungsalgorithmen für den Lernvorgang des Modells haben sich dabei als hilfreiche Techniken herausgestellt. Die durchgeführten Experimente gaben einen Einblick in die Komplexität der Modelloptimierung und haben gezeigt, dass jeder der vielen vorhandenen Hyperparameter potentiell die Leistungsfähigkeit eines Modells beeinflussen kann.
Es wurde ein Zusammenhang zwischen Fine-Tuning und Transfer-Learning hergestellt und die Erkenntnis gewonnen, dass der erfolgreiche Einsatz bereits vortrainierter Modelle von einer geeigneten Konfiguration der Top-Layer und einer sinnvollen Wahl der Hyperparameter abhängen. Überraschend waren die Erkenntnisse aus dem Vergleich zwischen dem Training etablierter Modelle ohne Pre-Weighs und solchen mit Pre-Weights. Hier konnte keine deutliche Überlegenheit der vortrainierten Modelle festgestellt werden. Nichtsdestotrotz konnte mit der kürzeren Trainingsdauer der vortrainierten Modelle ein Vorteil des Einsatzes von Transfer-Learning beobachtet werden. Interessant war ebenso die Tatsache, dass das einfach aufgebaute eigene Modell weder für das Zwei-Klassen-Problem noch für das Sieben-Klassen-Problem eine wesentlich schlechtere Performance ablieferte als die um ein Vielfaches komplexeren Keras Applications.
Für den Übergang von zwei auf sieben Klassen konnten Ergebnisse generiert werden, aus denen man Empfehlung ableiten kann für den Anwendungsfall mit einem größeren Datensatz als dem im Rahmen dieser Arbeit genutzten von 1000 Bildern je Klasse. Mit den Konfusions-Matrizen wurde eine verbreitete Methode zur Evaluierung eines Multi-Klassen-Modells vorgestellt und auf unseren Problemfall angewandt. Der Nutzen dieser Evaluierungstechnik konnte bestätigt werden und es wurden, aus den Ergebnissen abgeleitet, weitere Ansatzpunkte, wie die Erstellung von Zwischenklassen oder das Weglassen einzelner Klassen, diskutiert.
Die angepasste Evaluierung stellt einen Versuch dar, die aus der Klassenerweiterung resultierenden Probleme durch fließende Übergänge und optisch schwer auseinanderzuhaltende Klassen von Zellstadien in die Beurteilung eines Modells mit einzubetten, indem die hierfür zu starre Definition der klassischen Genauigkeit bzw. Validation Accuracy verändert wurde. Miteinbezogen in die Evaulierung wurde neben der Frage, ob eine Vorhersage in einem gewissen Sinn uneindeutig ist oder nicht auch die Überlegung, inwieweit die Verwechslung bestimmter Klassen anders betrachtet werden sollte als einfach nur „falsch“. Die präsentierten Vorschläge stellen einen Ausgangspunkt für mögliche Erweiterungen der Anpassungen dar.
Um nicht nur die Genauigkeit eines Modells sondern auch andere Kennzahlen, wie die Speichergröße, in die Modelloptimierung aufzunehmem, wurde eine gängige Methode ausgewählt und an den zuvor mit selbst konstruierten Top-Layern ausgestatteten Keras Applications getestet. Die Ergebnisse waren zufriedenstellend insofern, als dass die Speichergröße tatsächlich reduziert werden konnte. Zudem konnte die Erkenntnis gewonnen werden, dass die Wahl des Pruning-Faktors im Fall des konstanten Beschneidens bzw. der Pruning-Faktoren bei der polynomiellen Beschneidung nicht einfach ist und eine Reihe von Versuchen notwendig ist, bevor man ein zumindest annähernd optimales Ergebnis gewinnen kann.
Zuletzt ist eine wichtige Erkenntnis aus dieser Arbeit die Tatsache, dass es für die Wahl von

guten Hyperparameterwerten für die Modelloptimierung im Deep Learning keine wohldefinierte Anleitung gibt, die viele oder gar alle möglichen Anwendungsfälle abdeckt. Vielmehr ist es so, dass wir auf der Grundlage von aus Literatur- und Internetrecherche gewonnenen Erkenntnissen einerseits und anderseits auch aufgrund von Erfahrungswerten, bisweilen auch einer gewissen Intuition folgend, unsere Entscheidungen treffen müssen in der Hoffnung, dass wir uns auf einem guten Weg befinden und mit der Bereitschaft, das eine oder andere Mal auch den Weg in eine potentielle Sackgasse auszutesten.

Literatur

[1] Titus Brinker, Achim Hekler, Alexander Enk, Joachim Klode, Axel Hauschild, Carola Berking, Bastian Schilling, Sebastian Haferkamp, Dikr Schadendorf, Tim Holland-Letz, Jochen Utikal, Christof von Kalle, and Collaborators. Deep learning outperformed 136 of 157 dermatologists in a head-to-head dermoscopic melanoma image classification task. 2019.

[2] Pranav Rajpurkar, Jeremy Irvin, Robyn L. Ball, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Daing, Aarti Bagul, Curtis P. Langlotz, Bhavik N. Patel, Kristen W. Yeom, Katie Shpanskaya, Francis G. Blankenberg, Jayne Seekins, Amrhein , Timothy J., David A. Mong, Safwan S. Halabi, Evan J. Zucker, Andrew Y. Ng, and Matthew P. Lungren. Deep learning for chest radiograph diagnosis: A retrospective comparison of the CheXNeXt algorithm to practicing radiologists. 2018.

[3] PwC. Sherlock in Health - How artificial intelligence may improve quality and efficiency, whilst reducing healthcare costs in Europe. 2017.

[4] Alberto Romero. 7 Famous AI Quotes Explained . 2021. https://towardsdatascience.com/7-famous-ai-quotes-explained-782dda72d2c5 aufgerufen am 16. 08. 2021.

[5] Ayse Pinar Saygin, Ilyas Cicekli, and Varol Akman. Turing Test: 50 Years Later. 2000.

[6] François Chollet. Deep Learning with Python. 2018.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning . 2017. https://www.deeplearningbook.org/ aufgerufen am 19. 5. 2021.

[8] Jason Brownlee. What is the Difference Between a Parameter and a Hyperparameter? 2017. https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/ aufgerufen am 08. 08. 2021.

[9] Steven L. Brunton and J. Nathan Kutz. Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. 2019.

[10] Catherine F. Higham and Desmond J. Higham. Deep Learning: An Introduction for Applied Mathematicians . 2018.

[11] Drawing neural network with tikz . https://tex.stackexchange.com/questions/153957/drawing-neural-network-with-tikz aufgerufen am 16. 08. 2021.

[12] Moshe Leshno and Shimon Schocken. Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function . 1991.

[13] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. 2020.

[14] Ayyüce Kızrak. Comparison of Activation Functions for Deep Neural Networks . 2019. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 aufgerufen am 14. 5. 2021.

[15] Kiprono Elijah Koech. Cross-Entropy Loss Function . 2018. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e aufgerufen am 16. 5. 2021.

[16] Kevin P. Murphy. Machine Learning: A probabilistic view . 2012.

[17] Johanni Brea, Berfin Simsek, Bernd Illing, and Wulfram Gerstner. Weight-space symmetry in deep networks gives rise to permutation saddles, connected by equal-loss valleys across the loss landscape . 2019.

[18] Anna Choromanska, Mikael Henaf, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks . 2015.

[19] Thomas Kuribel. Derivative of the Softmax Function and the Categorical Cross-Entropy Loss . 2021. https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1, aufgerufen am 22. 5. 2021.

[20] Sushant Patrikar. Batch, Mini Batch and Stochastic Gradient Descent . 2019. https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a aufgerufen am 25. 5. 2021.

[21] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning . 2013.

[22] Mahesh Chandra Mukkamala and Matthias Hein. Variants of RMSProp and Adagrad with Logarithmic Regret Bounds . 2017.

[23] Diederik P. Kingma and Jimmy Lei Ba. Adam: A Method for Stochastic Optimization . 2015.

[24] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond . 2018.

[25] Ange Tato and Roger Nkambou. Improving Adam Optimizer . 2018.

[26] Sanghvirajit. A complete Guide to Adam and PMSprop Optimizer . 2020. https://medium.com/analytics-vidhya/a-complete-guide-to-adam-and-rmsprop-optimizer-75f4502d83be aufgerufen am 07. 08. 2021.

[27] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting . 2014.

[28] How to draw 3d matrix using tikz . https://tex.stackexchange.com/questions/516073/how-to-draw-3d-matrix-using-tikz aufgerufen am 17. 06. 2021.

[29] Tony Holdroyd. TensorFlow 2.0 Quick Sart Guide . 2019.

[30] Herbert Süße and Erik Rodner. Bildverarbeitung und Objekterkennung - Computer Vision in Industrie und Medizin . 2014.

[31] Md Shahid. Convolutional Neural Network . 2019. https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529 aufgerufen am 16. 06. 2021.

[32] Drawing a convolution with Tikz . https://tex.stackexchange.com/questions/437007/drawing-a-convolution-with-tikz aufgerufen am 16. 06. 2021.

[33] MK Gurucharan. Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network . https://www.upgrad.com/blog/basic-cnn-architecture/ aufgerufen am 06. 07. 2021.

[34] Jiwon Jeong. The Most Intuitive and Easiest Guide for Convolutional Neural Network . 2019. https://towardsdatascience.com/the-most-intuitive-and-easiest-guide-for-convolutional-neural-network-3607be47480 aufgerufen am 06. 07. 2021.

[35] Keras. Layer weight regularizers . https://keras.io/api/layers/regularizers/ aufgerufen am 08. 08. 2021.

[36] Afshine Amidi and Shervine Amidi. Convolutional Neural Networks cheatsheet . https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks aufgerufen am 16. 06. 2021.

[37] HarisIqbal88. PlotNeuralNet . https://github.com/HarisIqbal88/PlotNeuralNet aufgerufen am 16. 06. 2021.

[38] Kartik Nighania. Various ways to evaluate a machine learning model's performance. 2018. https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15 aufgerufen am 09. 07. 2021.

[39] Steven Simske. Meta-Analytics. 2019. https://www.sciencedirect.com/topics/computer-science/confusion-matrix aufgerufen am 10. 07. 2021.

[40] How to construct a confusion matrix in LaTeX? . https://tex.stackexchange.com/questions/20267/how-to-construct-a-confusion-matrix-in-latex aufgerufen am 10. 07. 2021.

[41] Rushabh Nagda. Evaluating models using the Top N accuracy metrics. 2019. https://medium.com/nanonets/evaluating-models-using-the-top-n-accuracy-metrics-c0355b36f91b aufgerufen am 19. 07. 2021.

[42] Jason Brownlee. How to use Learning Curves to Diagnose Machine Learning Model Performance. 2019. https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/ aufgerufen am 18. 07. 2021.

[43] Sayon Dutta. Intelligent Signals: Unstable Deep Learning. Why and How to solve them? 2017. https://medium.com/@sayondutta/intelligent-signals-unstable-deep-learning-why-and-how-to-solve-them-295dc12a7fb0 aufgerufen am 19. 07. 2021.

[44] Hafidz Zulkifli. Understanding Learning Rates and How It Improves Performance in Deep Learning . 2018. https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 aufgerufen am 20. 07. 2021.

[45] Jason Brownlee. What is the Difference Between Test and Validation Datasets? . 2017. https://machinelearningmastery.com/difference-test-validation-datasets/ aufgerufen am 07. 07. 2021.

[46] Jason Brownlee. Difference Between a Batch and an Epoch in a Neural Network. 2019. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/ aufgerufen am 14. 07. 2021.

[47] Keras. The Tuner classes in Keras Tuner. https://keras.io/api/keras_tuner/tuners/ aufgerufen am 18. 07. 2021.

[48] Keras. RandomSearch Tuner. https://keras.io/api/keras_tuner/tuners/random/#randomsearch-class aufgerufen am 18. 07. 2021.

[49] Rohit Thakur. Step by step VGG16 implementation in Keras for beginners. 2019. https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c aufgerufen am 18. 07. 2021.

[50] Nhu Hoang. Full review on optimizing neural network training with Optimizer . 2020. https://towardsdatascience.com/full-review-on-optimizing-neural-network-training-with-optimizer-9c1acc4dbe78 aufgerufen am 13. 08. 2021.

[51] . 2020. https://image-net.org/about.php aufgerufen am 19. 07. 2021.

[52] ImageNet. ImageNet Large Scale Visual Recognition Challenge 2014 (ILSVRC2014) . https://image-net.org/challenges/LSVRC/2014/browse-synsets.php aufgerufen am 19. 07. 2021.

[53] Deogratias Mzurikwao, Muhammad Usman Khan, Oluwarotimi Williams Samuel, Jindrich Jr. Cinatl, Mark Wass, Martin Michaelis, Gianluca Marcelli, and Chee SiangAng. Towards image-based cancer cell lines authentication using deep neural networks .

[54] Keras. Keras Applications. https://keras.io/api/applications/ aufgerufen am 19. 07. 2021.

[55] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition . 2015.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, and Jonathon Shlens. Rethinking the Inception Architecture for Computer Vision . 2015.

[57] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions . 2017.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition . 2015.

[59] CMingxing Tan and Quoc V. V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks . 2020.

[60] Ranjeet Singh. Pruning Neural Networks . 2019. https://towardsdatascience.com/pruning-deep-neural-network-56cae1ec5505 aufgerufen am 07. 07. 2021.

[61] TensorFlow. TensorFlow Model Optimization Toolkit-Pruning API . 2019. https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras aufgerufen am 08. 07. 2021.

[62] TensorFlow. Pruning in Keras example . https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras aufgerufen am 08. 08. 2021.

A Dokumentation

Im Folgenden werden die im Rahmen dieser Arbeit verwendeten Programme vorgestellt. Es sei darauf hingewiesen, dass aufgrund der Menge an Tests nicht jedes einzelne Experiment in Form eines eigenen separaten Skripts festgehalten wurde und manche der Codes zur Zeiteinsparung überschrieben wurden. Der Anhang stellt einen Auszug der wichtigsten Skripte dar.

A.1 Allgemeine Hinweise

Jedes Programm liegt in zweifacher Ausführung vor, einmal als ipynb-Datei, welche mit Google Colaboratory im Browser geöffnet werden kann, und einmal als Python-Skript, welches lokal auf dem PC ausführbar ist. Die Programme sind im Ordner **scripts** zu finden. Der Inhalt dieses Ordners ist den Unterkapiteln aus Kap. 4 entsprechend strukturiert. Es sind einige Punkte bei der Ausführung der Programme zu beachten.

- An allen Stellen im Programmcode, an denen ein Dateipfad angegeben wird, muss der eigene stattdessen eingesetzt werden.
- Der Aufbau der Ordner **train**, **test** und **Einzelerkennung** darf nicht verändert werden.

A.1.1 ipynb-Dateien

Für die Programmierung mittels Colaboratory rufe man zunächst die folgende Seite auf und beachte die dort bereitgestellten Informationen:
`https://colab.research.google.com/notebooks/welcome.ipynb?hl=de`
Zwingende Voraussetzung für die Nutzung dieses Dienstes ist ein Google-Account. Darüber hinaus sind jedoch keine weiteren Registrierungen notwendig. Der Gebrauch ist kostenlos, allerdings nicht unbegrenzt. Google behält sich vor, je nach Auslastung der Server, die Nutzung der GPU-Einheiten zu beschränken, falls man über einen längeren Zeitraum bereits viel Kapazität genutzt hat. Dann kann man dort zwar weiterhin programmieren, aber darf nur noch CPU-Einheiten nutzen, was die Geschwindigkeit der Programmausführung deutlich verlangsamt.

A.1.2 py-Dateien

Wir beginnen mit einer kurzen Anleitung zur Installation der erforderlichen Pakete. Man beachte, dass die nachfolgenden Schritte für die Ausführung auf einem Windows-Betriebssystem gelten. Für weiterführende Informationen besuche man die Seite
`www.tensorflow.org/install`
Nach erfolgreicher Installation der aktuellen Python-Version und dem pip-Installer müssen über die CMD-Konsole folgende Befehle ausgeführt werden:

1. **pip install –user virtualenv** (optional)
2. **pip install –upgrade tensorflow**
3. **pip -U scikit-learn**
4. **pip install pillow**
5. **pip install -U matplotlib**
6. **pip install keras-tuner**
7. **pip install tensorflow-model-optimization**

A.2 Erläuterung der Programme

Als Vorlage für die im weiteren Verlauf präsentierten Programme diente ein von Dr. Hiemann bereitgestelltes Skript, welches das in Kap. 7.1 erwähnte Ausgangsmodell für das Fine-Tuning beinhaltet. Ein Teil des Codes der Programme ist überall (fast) gleich. Das gilt vor allem für den Import der notwendigen Pakete und den Aufruf der Trainingsdaten. Wir werden nun für jeden Ordner kurz auf die dort gespeicherten Dateien eingehen. Im weiteren Verlauf meint ein Dateiname ohne Endung immer das gleichnamige Python-File bzw. das entsprechende Colab-Notebook.

A.2.1 0-Material_von_Dr_Hiemann

- **a1ANAMito** ist das Ausgangsmaterial, welches ein generisches Modell für die Einteilung der Zellen in sieben Klassen enthält. Nach der Inkludierung der Pakete erfolgt die Definition der Unterprogramme für den Aufruf der Trainingsbilder. Anschließend erfolgt die Definition der Unterprogramme für die Erstellung der Modellarchitektur sowie die Festlegung einiger Parameter für das Training, wie z.B. die Art der Kostenfunktion und der Optimierungsalgorithmus für dessen Minimierung. Anschließend werden diese Unterprogramme aufgerufen und weitere Konfigurationen für das Training festgelegt, z.b. die Berechnung der Klassengewichte, welche einen womöglich unausgeglichenen Datensatz kompensieren kann. Nun wird das Training über die den **model.fit**-Befehl eingeleitet und das Modell anschließend abgespeichert. Nach diesem Schema ist auch der größte Teil der nachfolgenden Programme aufgebaut.

- **bestmodel-0.6837-0094-0.792.h5** und **bestmodel-0.7077-0081-0.783.h5** sind die Modelle, welche nicht über den **model.save**-Befehl erstellt werden, sondern bereits während des Trainings erzeugt werden und die besten in einer Epoche erreichten Gewichte bezüglich der erreichten Validation-Accuracy enthalten.

A.2.2 1-Fine-Tuning

- **eigenes_Modell_2_Klassen** enthält die finale Konfiguration der Modellarchitektur und der Trainingsparameter für das aus dem in Kap. 7.1 vorgestellten Fine-Tuning hervorgegangene Modell.

- **eigenes_Modell_2_Klassen.h5** ist das entsprechende Modell, welches die Modellarchitektur und die zugehörigen Gewichte (der letzten Trainingsepoche) beinhaltet. Es wurde 100 Epochen lang trainiert.

- **Filter-Tuning_eigenes_Modell_2_Klassen** enthält die Konfigurationen für die Durchführung des in Kap. 7.1 beschriebenen Filter-Tunings am eigenen Modell.

A.2.3 2-Keras_Applications_und_Transfer-Learning

- **Keras_Applications_ohne_pre_weights** enthält die Codes, welche für das Training der in Kap. 7 vorgestellten Modelle ohne Vorgewichte genutzt werden können. Zu beachten ist, dass man zum Wechseln zwischen den Modellen die entsprechend im Python-File gekennzeichneten Zeilen auskommentieren muss. Das gilt auch für das nächste Skript.

- **Tuning_Lernrate_Moment_VGG16_untrained** enthält den Code für die Durchführung des in Kap. 7.2.1 vorgestellten Tuning von Lernrate und Moment für das nicht vortrainierte VGG16.

- **Keras_Applications_mit_pre_weights** enthält die Konfigurationen für die beiden in Kap. 7.2.2 vorgestellten Modelle, welche dort zusammen mit den auf dem ImageNet-Datensatz erlernten Vorgewichten geladen werden. Die hier festgehaltenen Einstellungen bezüglich der trainierbaren Top-Layer sind jeweils das Endresultat aus den zuvor durchgeführten Experimenten an ihren Hyperparametern.

- **Tuning_der_dichten_Schichten_Xception_pretrained_2_Klassen** enthält die Konfigurationen für das in Kap. 7.2.2 beschriebene Fine-Tuning der Neuronenanzahl in den beiden trainierbaren versteckten dichten Schichten des vortrainierten Xception.

- **Xception_pretrained_2_Klassen.h5** ist das daraus resultierende Modell, welches 100 Epochen lang trainiert wurde.

- **InceptionV3_pretrained_2_Klassen.h5** ist das aus den in Kap. 7.2.2 beschriebenen Tests hervorgegangene InceptionV3-Modell mit Vorgewichten für die Erkennung von zwei Klassen.

A.2.4 3-Erweiterung_auf_sieben_Klassen

- **eigenes_Modell_7_Klassen** enthält die aus den in Kap. 7.3 erwähnten Experimenten resultierende finale Architektur und Trainingskonfiguration für das eigene Modell zur Erkennung von sieben Klassen.

- **eigenes_Modell_7_Klassen.h5** ist das entsprechende 100 Epochen lang trainierte Modell.

- **Keras_Applications_7_Klassen** enthält den Code für die in Kap. 7.3 gezeigten vortrainierten Keras Applications, welche auf die Einteilung in sieben Klassen trainiert werden können. Auch hier müssen zum Wechseln zwischen den einzelnen Modellen, die entsprechend (im Python-File) gekennzeichneten Zeilen auskommentiert werden. Das gilt auch für das nächste Skript.

- **Einzelerkennung** ist im gleichnamigen Unterordner zu finden und enthält das Programm für die Erstellung der zum Ende des Kap. 7.3 erwähnten Modelle zur Erkennung einzelner Klassen. Im gleichnamigen Ordner sind auch die während des Tranings der sieben Einzelerkennungsmodelle erzeugten Konfusionsmatrizen zu finden.

A.2.5 4-Anpassung_der_Evaluierung

- **angepasste_Evaluierung** ist das Programm zur in Kap. 7.4 präsentierten Anpassung der Modell-Evaluation.

A.2.6 5-Pruning

- **Constant_Pruning** enthält das Skript für die Durchführung der konstanten Beschneidung an den beiden Modellen InceptionV3 und XceptionV3. Hier sind die entsprechend markierten Zeilen nach Bedarf auszukommentieren.

- **Polynomial_Pruning** Dieses Skript ist fast identisch zum vorangegangenen Skript. Der einzige Unterschied liegt in der Definition des durchzuführenden Prunings.

- **InceptionV3_pretrained_2_Klassen_after_strip_pruning_constant_01.h5** und **Xception_pretrained_2_Klassen_after_strip_pruning_constant_07.h5** sind zwei der aus einer konstanten Beschneidung hervorgegangenen komprimierten Modelle. Man vergleiche die Speichergrößen mit denen der beiden Modelle aus dem Ordner 2.

A.2.7 x-zu_erwartende_Ausgaben

In diesem Ordner befindet sich eine Auswahl von Ausgaben, die an verschiedenen Stellen eines Codes zu erwarten sind. Die Zugehörigkeiten sind aus den jeweiligen Dateinamen einfach abzuleiten. Die Bilder sind den vorangegangenen Ordnern entsprechend sortiert. Erstellt wurden sie in Colab. Man vergleiche die in
1-3-model_summary_eigenes_Modell_2_Klassen abgebildeten Zahlen mit den in Kap. 4.4 berechneten Werten. Auf die in **3-3-Ausgabe_Confusion-Matrix** gezeigten Genauigkeitsmaße wird nicht weiter eingegangen, da sie im Rahmen dieser Arbeit keine Rolle gespielt haben. Für weiterführende Informationen besuche man beispielsweise folgende Seite:
`https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9`

A.3 Ergänzungen

Abschließend gehen wir auf die restlichen Inhalte des Datei-Anhangs ein. Im Ordner **logs** werden die während des Trainings generierten Gewichtsparameter gepeichert. In **models** werden die Modelle über den **model.save**-Befehl gespeichert. In **tuner_results** werden die Zwischenergebnisse der Test-Einheiten mit dem Keras Tuner gespeichert. Diese Vorgaben sind ein Vorschlag, grundsätzlich können über den selbst einzugebenden Dateipfad an den entsprechenden Stellen der Programm-Codes auch andere Ziele eingegeben werden. Der Ordner **train** enthält die Trainingsbilder für den Großteil der Modelle. In **test** befindet sich der Test-Datensatz für die angepasste Evaluierung. **Einzelerkennung** enthält die Trainingsbilder für die Einzelerkennungmodelle.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelor-Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Cottbus, den 17.08.2021 ....

IMPRESSUM

Brandenburgische Technische Universität Cottbus-Senftenberg
Fakultät 1 | MINT - Mathematik, Informatik, Physik, Elektro- und Informationstechnik
Institut für Mathematik
Platz der Deutschen Einheit 1
D-03046 Cottbus

Professur für Ingenieurmathematik und Numerik der Optimierung
Professor Dr. rer. nat. Armin Fügenschuh

E fuegenschuh@b-tu.de
T +49 (0)355 69 3127
F +49 (0)355 69 2307

Cottbus Mathematical Preprints (COMP), ISSN (Print) 2627-4019
Cottbus Mathematical Preprints (COMP), ISSN (Online) 2627-6100

www.b-tu.de/cottbus-mathematical-preprints
cottbus-mathematical-preprints@b-tu.de
doi.org/10.26127/btuopen-5630