HAW HAMBURG

MASTERTHESIS
Mike Perkovic

Methodenvergleich zum Einsatz von Deep Learning zur kamerabasierten Ermittlung des Schneebedeckungsgrades

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Mike Perkovic

Methodenvergleich zum Einsatz von Deep Learning zur kamerabasierten Ermittlung des Schneebedeckungsgrades

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informations- und Kommunikationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Dipl.-Kfm. Jörg Dahlkemper
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 3. Januar 2022

Mike Perkovic

Thema der Arbeit

Methodenvergleich zum Einsatz von Deep Learning zur kamerabasierten Ermittlung des Schneebedeckungsgrades

Stichworte

Kamera, Wetter, Schnee, Bedeckungsgrad, Deep Learning, Künstliches neuronales Netz, KNN, Erkennung, Algorithmus, Open Source, Python, OpenCV, TensorFlow, Keras, Neuronales Faltungsnetz, CNN

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Frage, welche Methode dafür geeignet ist, den Schneebedeckungsgrad auf Bildern optimal zu erkennen. Zur Realisierung wird Deep Learning in Kombination mit Faltungsnetzen angewendet. Es werden verschiedene Bilddatensätze und Netze verwendet wie VGG16, Xception und DenseNet201.

Mike Perkovic

Title of Thesis

Comparison of methods for using deep learning for camera-based snow cover determination

Keywords

Camera, Weather, Snow, Coverage, Deep Learning, Artificial Neural Network, ANN, Recognition, Algorithm, Open Source, Python, OpenCV, TensorFlow, Keras, Convolutional Neural Network, CNN

Abstract

This thesis deals with the question of which method is suitable for optimally detecting the degree of snow cover on images. For the realization, Deep Learning in combination with convolutional networks is applied. Different image datasets and networks are used, such as VGG16, Xception and DenseNet201.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation und Ziel

Im Rahmen von Arbeiten des Deutschen Wetterdienstes (DWD) [Bund 1952] werden Bilder von Wetterkameras daraufhin ausgewertet, ob und in welchem Umfang der Boden mit Schnee bedeckt ist. Dies erfolgt in der Regel manuell und es soll eine Lösung zur automatisierten Erfassung des Schneebedeckungsgrades gefunden werden. Anhand von Voruntersuchungen wurde die grundsätzliche Machbarkeit unter Einsatz von Machine Learning und Nutzung von Faltungsnetzen[1] nachgewiesen [Chaari 2021]. Parallel zu dieser Arbeit finden an der HAW Hamburg[2] weitere Untersuchungen im Bereich der klassischen Bildverarbeitung statt.

Deep Learning kommt in der Regel dann zum Einsatz, wenn sich Merkmale von großen Datenmengen nur schwer extrahieren lassen [Chollet 2018, S. 21]. Die Schneeerkennung von Bildern ist ein passendes Beispiel. Dabei soll Schnee nicht mit weißen Objekten oder Wolken verwechselt werden. Außerdem soll eine Erfassung mit variierenden Eigenschaften wie Belichtung, Kontrast und Helligkeit zuverlässig funktionieren. Die Erfolge von Deep Learning sind vielversprechend: Im medizinischen Bereich können Tumore anhand von Bildern erkannt werden [Zhao u. a. 2017]. Texte unterschiedlicher Fremdsprachen lassen sich auf Basis von CNNs übersetzen [Kutylowski 2017]. Ein weiterer Fortschritt ist die Manipulation von Gesichtern einzelner Personen in Bildern oder Videos (Deepfakes) [Nguyen u. a. 2021]. Menschen lassen sich auf Bilder erstellen [Wang 2019], die nicht existieren und nur schwer für das menschliche Auge von realen Personen zu unterscheiden sind. Ein Entgegenwirken zur Erkennung von Deepfakes ist in der Publikation [Dheeraj u. a. 2021] mit Hilfe von Deep Learning untersucht worden. Zudem können die verschiedensten Objekte mit Hilfe von Software [GitHub 2019b] in Echtzeit während einer Videoaufnahme erkannt werden.

[1] englisch: Convolutional Neural Network (CNN)
[2] Hochschule für Angewandte Wissenschaften Hamburg. Link: https://www.haw-hamburg.de/

Das Ziel dieser Arbeit ist der Vergleich und die Umsetzung von mehreren Methoden im Bereich des Machine Learnings zur Messung des Schneebedeckungsgrades am Beispiel von gegebenen Bilddatensätzen.

1.2 Allgemeine Vorgehensweise

Der Aufbau dieser Thesis beginnt mit den theoretischen Grundlagen, auf die diese Arbeit basiert. Anschließend werden Anforderungen zur Aufgabenstellung festgelegt und analysiert. Basierend dazu wird ein Konzept anhand der Anforderungen hergeleitet. Daraufhin erfolgen die Entwicklung und Umsetzung des grundlegenden Konzeptes. Die erzielten Ergebnisse werden ausgewertet und bewertet, um die optimale Lösung zu finden. Zum Abschluss folgt eine Zusammenfassung mit Ausblick über diese Arbeit.

2 Grundlagen

Dieses Kapitel befasst sich mit den theoretischen Grundlagen, die in dieser Arbeit zum Einsatz kommen. Dazu wird die Grundidee der künstlichen Intelligenz erläutert. Zudem wird unter anderem der Ablauf anhand eines künstlichen neuronalen Netzes (KNN) und CNNs erklärt. Des Weiteren werden zum aktuellen Stand der Technik vergleichbare Problemstellungen anhand von Veröffentlichungen vorgestellt. Abschließend werden die verwendeten Bilddatensätze sowie die Netzinfrastruktur präsentiert.

2.1 Einführung in die künstliche Intelligenz

Die künstliche Intelligenz (englisch: Artificial Intelligence; kurz: AI) beschäftigt sich mit der Automatisierung von intelligentem Verhalten [Dick 2019, vgl.]. Der Begriff entstand basierend auf der Vermutung, dass jede Sichtweise des menschlichen Lernens auf einen Computer beschrieben und simuliert werden kann [Dick 2019, vgl.]. Dazu zählen die Intelligenz des menschlichen Denkens und die kognitiven Fähigkeiten, die vom Gehirn aus abgeleitet werden. Erste Forschungen hierzu starteten in den 1950er Jahren, indem versucht wurde, Prozesse zu automatisieren, da diese das menschliche Verhalten widerspiegeln [Dick 2019, vgl.].

Daraus entwickelte sich im Laufe der Jahre ein eigenes Fachgebiet: das maschinelle Lernen (englisch: Machine Learning; kurz: ML) [Naqa u. a. 2015, vgl.]. Ein System basierend auf Machine Learning besteht aus computergestützten Algorithmen mit dem Ziel, Vorhersagen zu treffen. Ein Algorithmus wird dabei anhand von Daten für eine Vorhersage trainiert [Naqa u. a. 2015, vgl.]. Der wesentliche Vorteil, im Gegensatz zur klassischen Lösung ohne Einsatz von Machine Learning, ist die eigenständige Entscheidung des Algorithmus in der Vorhersage, sodass eine explizite Programmierung von Regeln nicht benötigt wird [Géron 2018, vgl., S. 5].

Die Vorgehensweise einer Klassifizierung lautet wie folgt: Bekannte Zusammenhänge werden mit Eingangsdaten in Verbindung gebracht und unbekannte Beziehungen durch das Erlernen von Strukturen gefunden [Géron 2018, S. 7-8, vgl.]. Dazu werden statistische Algorithmen verwendet, wie z. B. Entscheidungsbäume [Frosst und Hinton 2017], k-Nearest-Neighbor [Cunningham und Delany 2020] oder Support Vector Machines (SVM) [Hearst u. a. 1998], auf die nicht näher eingegangen wird. Die Lerntypen im Bereich des Machine Learnings werden wie folgt unterschieden [Géron 2018, vgl., S. 8-14]:

- Überwachtes Lernen: Merkmaldaten und Klassifikationsergebnisse sind bekannt, aus denen das System lernt.
- Unüberwachtes Lernen: Das System baut anhand von Daten ohne bekannte Ergebnisse eigenständig ein Modell auf, um diese optimal vorherzusagen.
- Verstärktes Lernen: Hierbei entwickelt das System eine Strategie und wird dabei über ein Belohnungssystem dann belohnt, wenn sich die Änderung am Ausgang positiv auswirkt.

Fortschritte im Bereich des Machine Learnings konnten in unterschiedlichen Anwendungen erzielt werden: Von der Betrugserkennung zur Vermeidung von Unterschriftfälschungen bis hin zur Strahlentherapie von Krebspatienten, um die korrekte Strahlendosis zu verabreichen [Naqa u. a. 2015].

Deep Learning (DL) ist ein Teilbereich des Machine Learnings [Chollet 2018, S. 4]. Mit ihrer Hilfe kann das Lernverhalten eines Systems mit Datenmengen gesteigert werden [Chollet 2018, vgl., S. 21]. Im Gegensatz zum klassischen Machine Learning können hierbei die Daten unstrukturiert sein, wie sie z. B. in Sprach-, Bild-, Audio- oder Textdaten vorkommen [Chollet 2018, vgl., S. 11-12]. Je umfangreicher verwendete Daten sind, desto rechenintensiver sind die Berechnungen eines Modells. Leistungsstarke Hardware kann das Training der Modelle beschleunigen [Chollet 2018, vgl., S. 20-21]. Dazu sind Grafikprozessoren[1] der Modelle Tesla V100 und A100 des Herstellers NVIDIA für Berechnungen im Bereich Deep Learning ausgelegt. DL-Modelle basieren auf künstlichen neuronalen Netzwerken, die im Unterabschnitt 2.1.1 näher erläutert werden. Im Vergleich zum klassischen Machine Learning übernimmt ein Deep Learning-Netz die Auswahl und Implementierung von Features selbstständig [Géron 2018, vgl., S. 18]. Dies hat den Vorteil, dass das manuelle Feature Engineering nicht mehr benötigt wird und optimale Ergebnisse

[1] englisch: Graphics Processing Unit (GPU)

mit unstrukturierten Daten zu erwarten sind [Géron 2018, vgl., S. 23]. In Abbildung 2.1 sind die Begriffe AI, ML und DL eingegliedert.

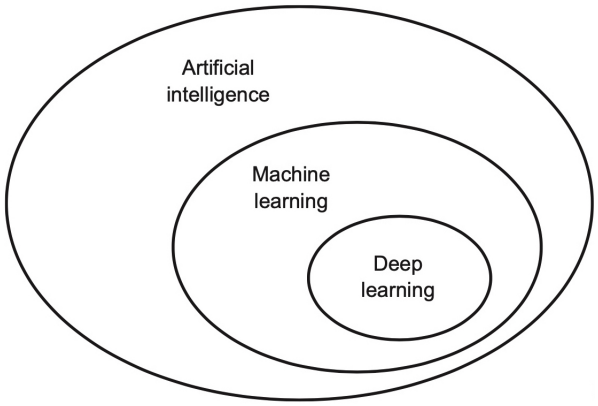


Abbildung 2.1: Einordnung der Begriffe für künstliche Intelligenz [Chollet 2018, S. 4]

2.1.1 Künstliche neuronale Netze

Künstliche neuronale Netze bestehen aus künstlichen Neuronen. Diese stellen eine mathematische Recheneinheit dar und bestehen aus verschiedene Varianten. In diesem Unterabschnitt sind Formeln genannt, die im Buch [Géron 2018] erwähnt sind. In Abbildung 2.2 sind drei ursprüngliche künstliche Neuronen abgebildet, die jeweils einen der binären Werte null und eins annehmen können. In dem Beispiel [Géron 2018, S. 257] sind alle Neuronen miteinander verknüpft, aus denen sich eine Disjunktion wie folgt durchführen lässt: $C = A \vee B$. Dabei wird das Neuron C erst aktiviert, wenn entweder Neuron A, Neuron B oder beide gleichzeitig aktiviert sind.

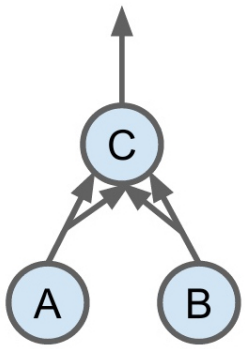


Abbildung 2.2: KNN für eine Disjunktion [Géron 2018, S. 257]

Eine andere Variante ist die LTU (englisch: Linear Threshold Unit) oder einfaches Perzeptron genannt [Géron 2018, S. 257-258]. Der Unterschied zum einfachen künstlichen Neuron sind Zahlenwerte, die ein LTU anstatt einer binären Größe am Eingang und Ausgang annehmen kann. Dabei wird jeder Eingangswert x_n mit einem Gewicht w_n berücksichtigt, woraus die gewichtete Summe aller Eingänge gebildet wird:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n = w^T \cdot x \tag{2.1}$$

Auf die Summe wird anschließend eine Aktivierungsfunktion angewendet.

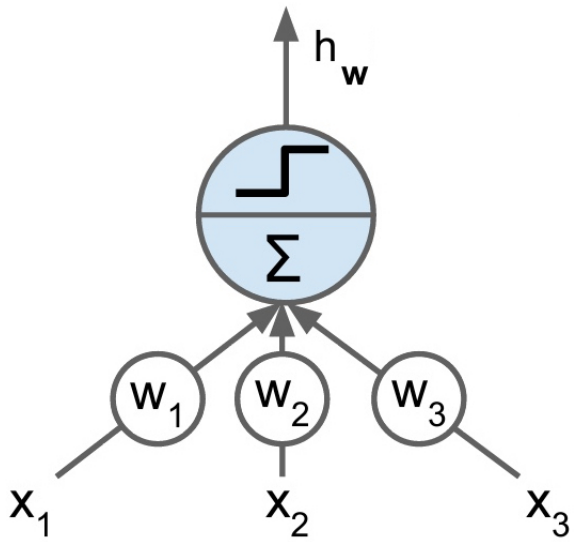


Abbildung 2.3: Linear Threshold Unit [Géron 2018, S. 257]

Eines dieser Funktionen ist die Stufenfunktion, wie am Beispiel in Abbildung 2.3 dargestellt. Dieser enthält den Schwellenwert null für die Entscheidung und gibt eine null aus, wenn $z < 0$ erfüllt ist und eine eins, falls $z \geq 0$ zutrifft:

$$heaviside(z) = \begin{cases} 0 \text{ wenn } z < 0 \\ 1 \text{ wenn } z \geq 0 \end{cases} \tag{2.2}$$

Die Ausgabe des LTUs lässt sich allgemein wie folgt definieren:

$$h_w(x) = Aktivierungsfunktion(w^T \cdot x) \tag{2.3}$$

Eine LTU lässt sich verwenden, um z. B. mit Hilfe des Algorithmus Support Vector Machines [Hearst u. a. 1998] Objekte anhand der Länge und Breite zu klassifizieren. Dabei werden die Gewichtungen w_n optimal für eine Vorhersage trainiert.

Mehrere LTUs lassen sich zu einem Perzeptron zusammenfassen [Géron 2018, S. 258]. In Abbildung 2.4 ist jeweils eine Eingabe- und Ausgabeschicht zu erkennen. Die Eingabeschicht enthält zwei Eingänge und ein Bias. Die Ausgabeschicht besteht aus drei LTUs. Dabei ist jede LTU am Eingang jeweils mit den Ausgängen des Bias-Neurons und allen Eingangsneuronen verbunden. Die letzte Schicht für die Klassifizierung wird als Dense Layer (fully-connected Layer) bezeichnet und lässt sich wie folgt beschreiben:

$$output = \varphi(W \cdot I + B) \tag{2.4}$$

Das Perzeptron von Abbildung 2.4 ist in der Lage, eine Klassifizierung von Daten für drei Klassen durchzuführen [Géron 2018, S. 258].

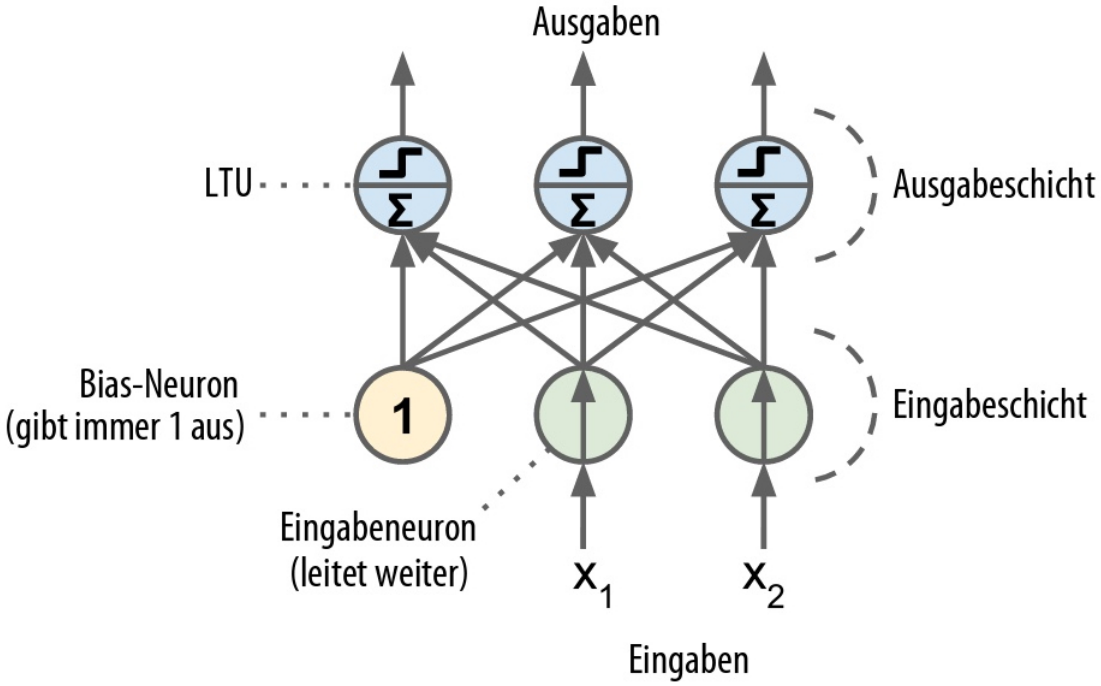


Abbildung 2.4: Perzeptron [Géron 2018, S. 258]

Das Perzeptron ist laut [Géron 2018, S. 259] wie folgt definiert:

$$w_{i,j}^{next,step} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \tag{2.5}$$

- η ist die Lernrate
- $w_{i,j}$ bezeichnet das Gewicht zwischen dem Neuron i und dem Ausgabeneuron j
- x_i ist der Wert des aktuellen Trainingspunktes
- y_j als Ausgabe und $\hat{y}_j$ als Zielausgabe des jeweiligen j-ten Ausgabeneurons für den aktuellen Trainingsdatenpunkt

Ein Perzeptron lässt sich zu einem mehrlagigen erweitern [Géron 2018, S. 261]. Dadurch erhält das Netz eine oder mehrere zusätzliche Hidden Layers, die sich zwischen der Eingabe- und Ausgabeschicht befinden, wie in Abbildung 2.5 dargestellt. Das dargestellte Netz wird als Deep Learning-Netz (DNN) bezeichnet [Géron 2018, S. 261].

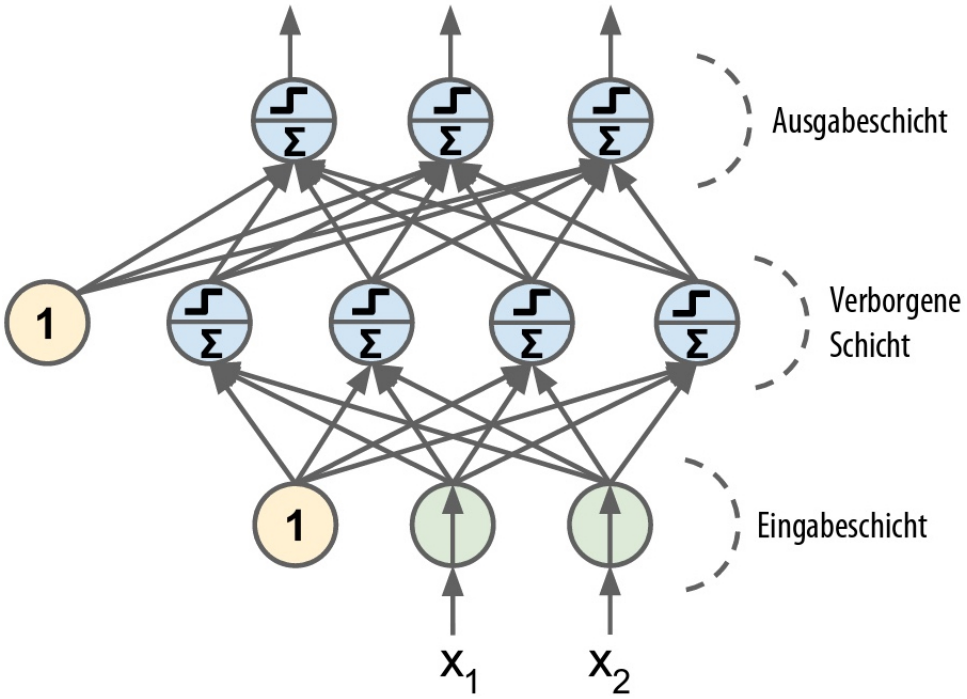


Abbildung 2.5: Mehrschichtiges Perzeptron [Géron 2018, S. 261]

2.1.2 Aktivierungsfunktionen

Die Aktivierungsfunktion stellt eine Übertragungsfunktion dar und hat das Ziel, Eingangswerte in einem gegebenen Wertebereich $\mathbb{W}$ zu beschränken sowie diese am Ausgang weiterzuleiten. Im Folgenden werden die Funktionen Sigmoid, ReLU und Softmax näher

erläutert. Die zugehörigen Formeln sind von [Chollet 2018] und [Géron 2018] entnommen.

Sigmoid

Die Sigmoid-Funktion kann den Wertebereich $\mathbb{W} = [0; 1]$ annehmen. Dieser ist wie folgt definiert [Géron 2018, S. 262]:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

Grafisch ist die Sigmoid-Funktion in Abbildung 2.6 abgebildet.

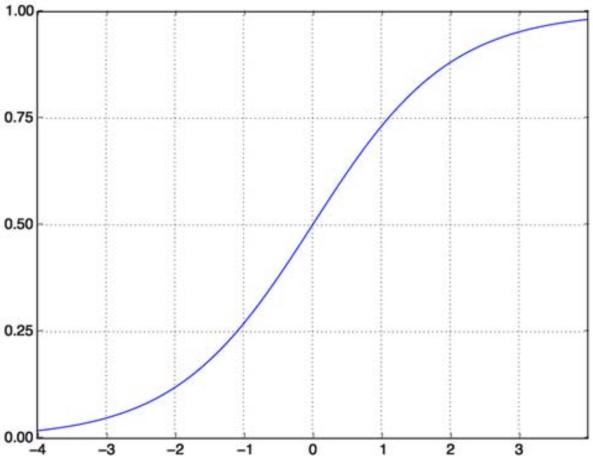


Abbildung 2.6: Sigmoid-Funktion [Chollet 2018, S. 71]

Sie galt aufgrund der in Gleichung 2.7 dargestellten Ableitung von Gleichung 2.6 lange Zeit als Standardfunktion. Die Herleitung ist für die Berechnung des Gradienten zuständig und von der nächsten Schicht abhängig [Géron 2018, S. 275-277]. Jedoch ist zu erkennen, dass der Gradient mit jeder Multiplikation aufgrund des Terms $(1 - f(x))$ immer kleiner wird und gegen null strebt:

$$f'(x) = f(x)(1 - f(x)) \tag{2.7}$$

Dies führt dazu, dass die ersten Schichten eines Netzes nicht erfolgreich trainiert werden können. Deshalb ist die Verwendung dieser Aktivierungsfunktion in tiefen Netzen proble-

matisch. Dies lässt sich als Problem des verschwindenden Gradienten (englisch: vanishing gradient problem) [Géron 2018, S. 275-276] bezeichnen.

ReLU

ReLU (Rectified Linear Unit) ist eine weitere Aktivierungsfunktion, die im Bereich Deep Learning zum Einsatz kommt [Chollet 2018, S. 71]. Der Zweck dieser Funktion ist die Projizierung jeglicher negativen Werte zu null. Eingesetzt wird dieser in der Regel als Hidden Layer oder Output Layer. Der Wertebereich von ReLU ist $\mathbb{W} = [0; +\infty]$, sodass keine Negativanteile zugelassen werden. ReLU ist wie folgt definiert:

$$\varphi(v) = max(0, v) \tag{2.8}$$

Grafisch lässt sich die ReLU-Funktion wie folgt darstellen:

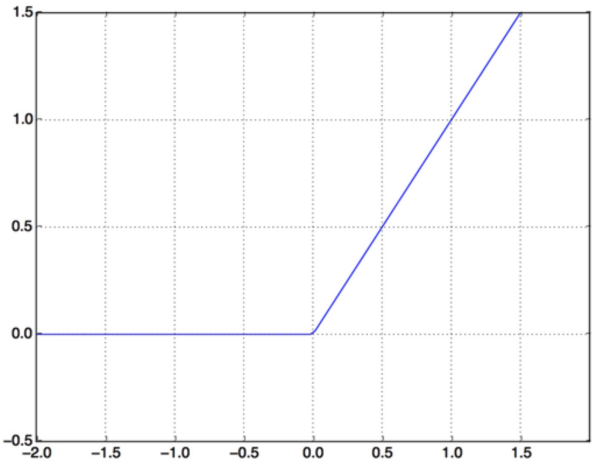


Abbildung 2.7: ReLU (Rectified Linear Unit) [Chollet 2018, S. 71]

Softmax

Die Softmax-Funktion [Géron 2018, S.140 - 141] wird in der Regel im Output-Layer eines Netzes verwendet. Dieser ist in der Lage, die Wahrscheinlichkeit der Klassenzugehörigkeiten für eine Eingabe x zu ermittelt. Dabei wird für jede Klasse eine Wahrscheinlichkeit

berechnet. Die Summe der Werte am Ausgang für alle Neuronen ergibt eins. Softmax ist in folgender Formel beschrieben:

$$\hat{p}_k = \sigma(s(x))_k = \frac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}} \tag{2.9}$$

wobei K die Anzahl der Klassen, $s(x)$ das Ergebnis jeder Klasse als Vektor für ein Datenpunkt x und $\sigma(s(x))_k$ die Wahrscheinlich der Zugehörigkeit vom Datenpunkt x sowie der Klasse k ist. Die höchste Wahrscheinlichkeit stellt hierbei die vorhergesagte Klasse dar. Diese kann aus dem Vektor $e^{s_k(x)}$ anhand des höchsten Wertes ermittelt werden:

$$\hat{y}_k = argmax(e^{s_k(x)}) \tag{2.10}$$

2.1.3 Verlustfunktionen

Die Verlustfunktion gibt das Verhältnis der Ausgangswerte eines Netzes zu den erwarteten Werten für Trainingsbeispiele an. Die Kostenfunktion hingegen beschreibt die Summe aller Verlustfunktionen für alle Trainingsdaten. In der Literatur werden die Begriffe Verlustfunktion und Kostenfunktion meist als Synonym verwendet. Eine Verlustfunktion kann dabei während der Trainingsphase beobachtet werden. Ziel ist es, den Wert während der Trainingsphase zu minimieren [Chollet 2018, vgl., S. 60].

Als Beispiel ist die kategorische Kreuzentropie (engl.: categorical crossentropy) zu erwähnen [Géron 2018, S. 142]. Diese Verlustfunktion kann zwischen zwei Wahrscheinlichkeitsverteilungen, einer geschätzten und tatsächlichen, die Kreuzentropie ermitteln. Als Ergebnis wird die durchschnittliche Menge in Bits angegeben, die benötigt wird, um das Resultat aus einer Anzahl von Wahrscheinlichkeiten zu ermitteln [Géron 2018, S. 142]. Unter der Annahme zweier diskreter Wahrscheinlichkeitsverteilungen wird die Funktion wie folgt beschrieben [Géron 2018, S. 142]:

$$H(p, q) = -\sum_{x} p(x) \log q(x) \tag{2.11}$$

wobei x eine Zufallsvariable, p und q jeweils Wahrscheinlichkeitsverteilungen sind.

Des Weiteren wird die mittlere quadratische Abweichung (kurz: MSE; englisch: Mean Squared Error) für Regressionsprobleme [Chollet 2018, S. 87] verwendet.

Diese ermittelt die quadratische Abweichung zwischen tatsächlichen und vorhergesagten Klassen. Der MSE ist wie folgt definiert:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 \tag{2.12}$$

2.1.4 Optimierungsverfahren

Ein Optimierungsverfahren hat das Ziel, die Gewichtungen in einem Netz während des Trainingsprozesses zu berechnen und zu trainieren [Géron 2018, vgl., S. 112 - 116]. Als Beispiel ist das Gradientenverfahren zu erwähnen. Hierzu soll das optimale Minimum oder Maximum anhand des Gradientenabstiegs gefunden werden. Abbildung 2.8 zeigt dabei die Verlustfunktion des Gradientenverfahrens:

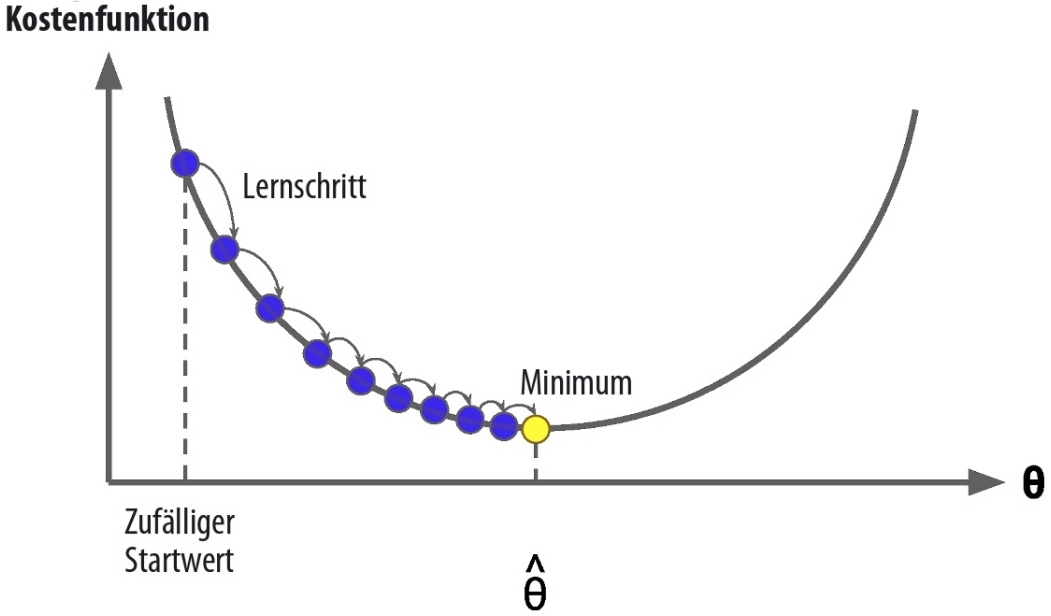


Abbildung 2.8: Gradientenverfahren [Géron 2018, S. 113]

Insgesamt werden zehn Trainingsepochen durchlaufen, wobei die zehnte das absolute Minimum erreicht und somit das optimale Ergebnis. Ein Optimierungsverfahren beinhaltet die Lernrate als einstellbaren Hyperparameter. Angenommen, die Lernrate wurde in Abbildung 2.8 optimal eingestellt. Wird die gleiche Prozedur mit einer geringeren Lernrate durchlaufen, so sind mehr Trainingsepochen nötig, um das Minimum zu erreichen. Durch einen höheren Wert der Lernrate wirkt sich das Training ebenfalls negativ auf das Ergebnis aus, da der Algorithmus divergiert [Géron 2018, S. 114].

Eine andere Variante des Gradientenverfahrens ist der Optimierer SGD (englisch: Stochastic Gradient Descent). Der Unterschied dabei ist, dass jede Iteration den Gradienten auf Basis eines zufällig ausgewählten Wertes z_t schätzt. Laut [Montavon u. a. 2012, S. 421 - 422] wird der Algorithmus wie folgt beschrieben:

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_t, w_t) \tag{2.13}$$

wobei $\{w_t, t = 1, ...\}$ von zufällig gewählten Werten z_t abhängt. Der stochastische Gradientenabstieg eines Perzeptrons ist wie folgt definiert:

$$w \leftarrow w + \gamma_t \begin{cases} y_t \Phi(x_t) \text{ if } y_t w^T \Phi(x_t) \leq 0 \\ 0 \text{ otherwise} \end{cases} \tag{2.14}$$

mit ihrer Verlustfunktion

$$Q_{perzeptron} = max\{0, -y\, w_T\, \Phi(x)\} \tag{2.15}$$

wobei $\Phi(x) \in \mathbb{R}_d$ die Features und $y = \pm 1$ den Klassen entsprechen.

Aufgrund der zufällig gewählten Werte kann sich der SGD-Optimierer kurz vor dem Erreichen des Minimums wieder entfernen. Die schrittweise Reduzierung der Lernrate während des Trainingsprozesses kann dem Problem entgegenwirken [Géron 2018, S. 119].

Ein weiterer Optimierer ist der leistungsstarke und effiziente Algorithmus Adam (Adaptive Moment Estimation) [Kingma und Ba 2015]. Dieser nutzt die Vorteile der nicht näher genannten Algorithmen AdaGrad [Géron 2018, S. 298 - 300] und RMSProp [Géron 2018, S. 300], der mit niedrigen Gradienten und nicht-stationären Umgebungen optimal funktioniert [Kingma und Ba 2015, vgl.]. Die Stufengröße während der Aktualisierungen wird anhand des Mittelwerts der Trägheitsterme von Gradienten laufend abgeleitet [Géron 2018, S. 300 - 302]. Dies macht den Algorithmus dynamischer. In [Kingma und Ba 2015] ist die Überlegenheit von Adam gegenüber AdaGrad, RMSProp und eine Variante des SGDs bewiesen. Der Adam-Algorithmus wird in [Géron 2018, S. 301] wie folgt beschrieben:

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J(\theta) \tag{2.16}$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta) \tag{2.17}$$

$$\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^T} \tag{2.18}$$

$$\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\beta_2^T} \tag{2.19}$$

$$\theta \leftarrow \theta + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon} \tag{2.20}$$

wobei T den ersten Iterationsindex darstellt und mit eins beginnt.

2.1.5 Phasen des Trainings

Ein Algorithmus lässt sich mit Hilfe von Daten für eine Vorhersage trainieren. Für das Training eines Modells werden Datensätze in Trainings-, Validierungs- und Testdaten unterteilt [Chollet 2018, vgl., S. 97 - 100]. Für das phasenweise Training eines Modells werden Trainingsdaten verwendet und nach jeder Trainingsperiode wird ein Trainingsscore anhand der Trainingsdaten ermittelt. Das Problem hierbei ist, dass die Trainingsgenauigkeit des Modells mit steigender Epochenanzahl zu 100 % tendiert [Chollet 2018, vgl., S. 139]. Das Verhalten kommt zustande, weil das Modell die bereits trainierten Daten zur Ermittlung der Genauigkeit immer wiederverwendet und auswendig lernt. Wenn das Modell auf unbekannte Daten getestet wird, kann die Genauigkeit der Testdaten keinen zuverlässigen Wert enthalten. Somit ist der Trainingsscore nicht dazu geeignet, das Training nach der optimalen Epoche zu überwachen. Aus diesem Grund werden separate Validierungsdaten eingesetzt. Nach jeder Epoche wird anhand von Validierungsdaten das Training überwacht. Ist die ermittelte Verlustfunktion der Validierung einer überwachten Metrik wie die Genauigkeit am niedrigsten, so ist dies die optimale Epoche und das Training kann beendet werden (siehe Abbildung 2.8). Mithilfe der Testdaten wird ein fertig trainiertes Modell anhand des Testscores bewertet.

2.1.6 Convolutional Neural Networks

Die begrenzte Sichtweise eines fully-connected KNNs lässt es nicht zu, Bilder in bestimmter Auflösung, wie z. B. VGA (640 × 480) Pixel, zu trainieren [Géron 2018, vgl., S. 361]. Diese enthalten eine zu hohe Anzahl an Parametern, die ein KNN nicht verarbeiten kann. Mehrfache Schichten lösen dieses Problem und deshalb kommt hierbei ein CNN zum Einsatz. Dieser besteht aus mehreren Layers, die teilweise miteinander verbunden sind [Gu u. a. 2017]. Das CNN wurde 1989 von LeCun eingeführt [LeCun u. a. 1989].

Convolutional Layer

Der Convolutional Layer ist wesentlicher Hauptbestandteil eines CNNs [Géron 2018, vgl., S. 361]. Dieser besteht aus mehreren Feature Maps mit jeweils fester Größe, deren Aufgabe es ist, unterschiedliche Merkmale und Muster zu finden [Géron 2018, vgl., S. 365]. Dazu werden Filter zur Identifizierung verwendet, wie laut Abbildung 2.9 in rot dargestellt. Jedes Neuron einer Feature Map enthält eine Gewichtung und ein Bias.

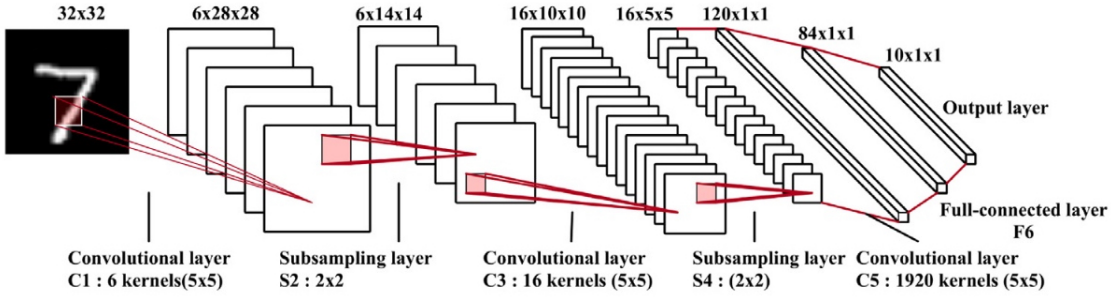


Abbildung 2.9: Schematischer Aufbau eines CNNs [Gu u. a. 2017, S. 355]

Grundlage von Berechnungen dieser Layers sind Kreuzkorrelationen, die mit einer Faltung vergleichbar ist. Dieser wird in diskreter Form laut [Koushik 2016] wie folgt beschrieben:

$$(f * g)(x) = \sum_{u=-\infty}^{\infty} f(u)g(x-u) \tag{2.21}$$

Abbildung 2.10 zeigt den Aufbau eines CNNs anhand von zwei Convolutional Layers und einem Input Layer. In der Eingabeschicht wird ein Bild im RGB-Format mit drei Kanälen als eine dreidimensionale Matrix eingeschleust. Dabei ist zu erkennen, dass Neuronen eines Convolutional Layers nicht mit allen der vorherigen Schicht verbunden sind.

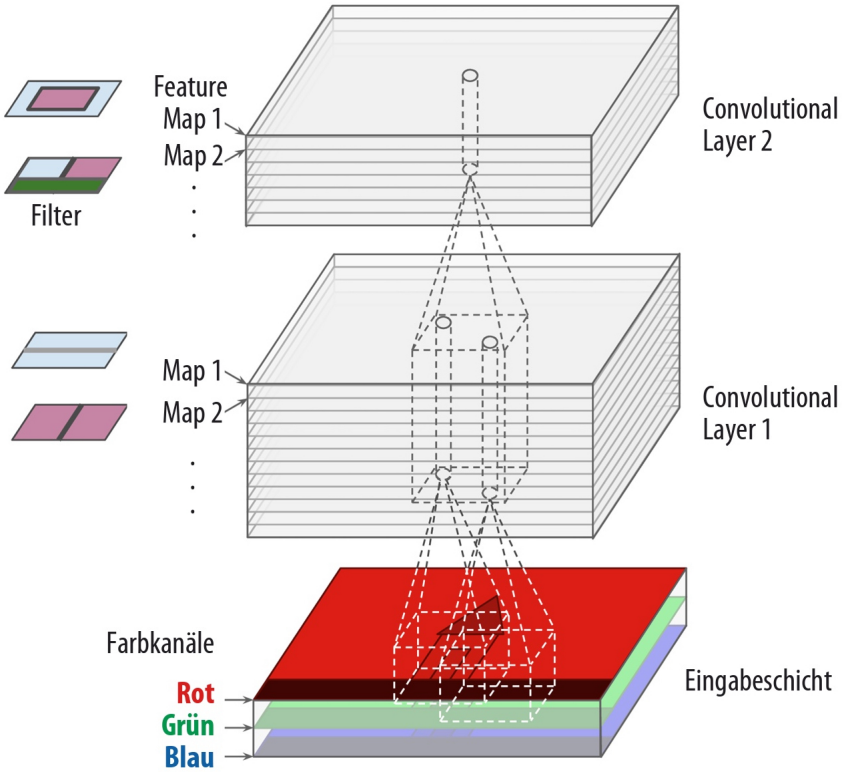


Abbildung 2.10: Convolutional Layers mit unterschiedlichen Feature Maps und RGB-Channels [Géron 2018, S. 365]

Mit der folgenden Formel nach [Géron 2018, S. 365 - 366] lässt sich die Ausgabe eines Neurons im Convolutional Layer berechnen und beschreiben:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \text{ mit } \begin{cases} i' = i' \times s_h + u \\ j' = j' \times s_w + v \end{cases} \tag{2.22}$$

wobei

- $z_{i,j,k}$ die Neuronausgabe (Zeile i, Spalte j, Feature Map k) des Convolutional Layers (Layer l) ist.
- s_h und s_w die vertikalen und horizontalen Schrittweiten sind. f_h und f_w sind die Höhe und Breite des Wahrnehmungsfelds. $f_{n'}$ ist die Anzahl der Feature Maps in der vorigen Schicht (Layer l – 1).

- $x_{i',j',k'}$ die Ausgabe des Neurons in Schicht $l-1$ mit Zeile i', Spalte j' und Feature Map k' (oder Kanal k') ist.
- b_k der Bias-Term von Feature Map k (in Schicht l) ist.
- $w_{u,v,k',k}$ das Gewicht der Verbindung zwischen einem beliebigen Neuron in Feature Map k der Schicht l und seiner Eingabe in Zeile u, Spalte v und Feature Map k' ist.

Pooling Layer

Mit Hilfe eines Pooling Layers lässt sich die Dimension eines Bildes reduzieren [Dias u. a. 2018, vgl., S. 144 - 154]. Dazu wird zum einen eine Filtergröße und zum anderen eine Schrittweite (stride) bekanntgegeben. Eine Aktivierungsfunktion wird dabei nicht festgelegt. Zudem wird zwischen Max- und Average-Pooling unterschieden [Dias u. a. 2018, vgl., S. 369 - 371]. In Abbildung 2.11 wird die Dimension eines Bildes von (6×6) Pixel auf (3×3) Pixel mit Hilfe der Filtergröße und Schrittweite von jeweils (2×2) Pixel verringert. Dabei wird das Max-Pooling angewendet, wodurch der maximale Wert innerhalb des Filters weitergeleitet wird. Mit Hilfe des Average-Poolings würde wie folgt der Mittelwert aller Werte dieser Filtermatrix gebildet und weitergeleitet werden:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.23}$$

Die Filtergröße beträgt üblicherweise (2×2) Pixel oder (3×3) Pixel, da sonst zu viele relevante Informationen verworfen werden [Dias u. a. 2018, S. 371].

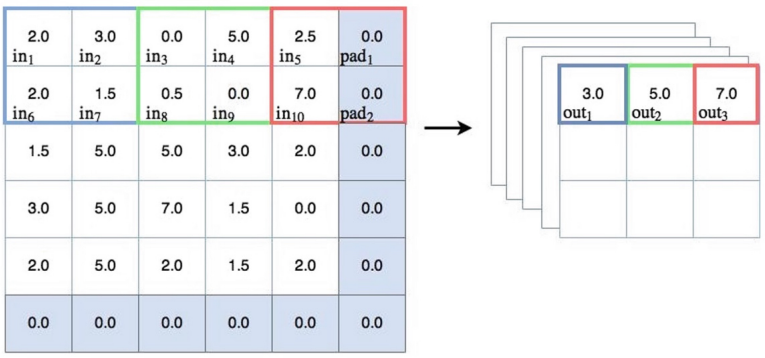


Abbildung 2.11: Anwendung von Max-Pooling [Dias u. a. 2018, S. 148]

Zusammengefasst erfüllt ein Pooling Layer folgende Aufgaben:

- Reduziert die Überanpassung
- Stellt eine Invarianz der Eingangsdaten her
- Reduziert die Anzahl der Parameter

Batch-Normalisierung

Die Batch-Normalisierung [Ioffe und Szegedy 2015] ist ein Verfahren, um das Problem der schwindenden Gradienten zu reduzieren. Die Technik kann vor der Durchführung einer Aktivierungsfunktion verwendet werden, z. B. vor dem Output-Layer zur Klassifizierung. Dabei wird die Berechnung während des Trainingsprozesses für jeden Batch[2] durchgeführt und für alle den Mittelwert um null gebildet [Ioffe und Szegedy 2015]. Die Varianz liegt dabei bis nahe eins [Ioffe und Szegedy 2015].

Die Vorgehensweise der Batch-Normalisierung wird laut [Ioffe und Szegedy 2015] nacheinander wie folgt durchgeführt:

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.24}$$

$$\sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{2.25}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \eta}} \tag{2.26}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathbf{BN}_{\gamma,\beta}(x_i) \tag{2.27}$$

wobei mit Hilfe von Gleichung 2.24 der Mittelwert eines Mini-Batches $B = \{x_{1\ldots m}\}$ ermittelt wird. Anschließend folgt die Ermittlung der Varianz in Gleichung 2.25. Mit Gleichung 2.26 wird eine Normalisierung durchgeführt. Die Parameter γ zur Skalierung und β für die Verschiebung der Schicht werden dabei erlernt.

[2] Stapel von Bildern, die während eines Trainingsprozesses verarbeitet werden.

Dropout

Dropout [Srivastava u. a. 2014] ist ein Verfahren, mit dem die Parameteranzahl eines Netzes reduziert wird, um Überanpassung zu reduzieren. Dabei werden während des Trainingsprozesses Ein- und Ausgangsverbindungen von zufälligen Neuronen n mit einer bekannten Wahrscheinlichkeit p als einstellbaren Hyperparameter entfernt. Die Ausgangsneuronen sind davon nicht betroffen. Nach jeder Trainingsepoche werden zufällige Neuronen verworfen. Dadurch wird vermieden, dass ein Netz zu genaue Merkmale erlernt. Abbildung 2.12 zeigt die Anwendung des Dropout-Verfahrens.

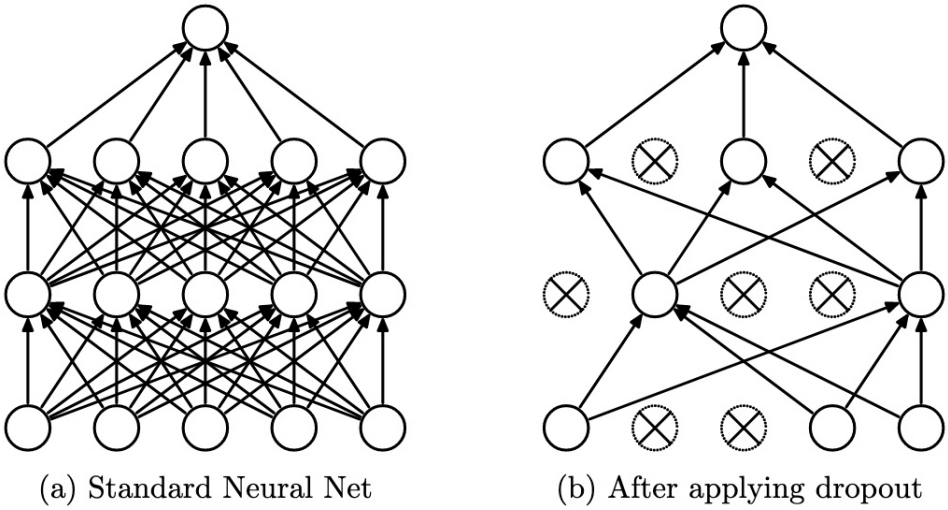


Abbildung 2.12: Dropout-Verfahren vor (a) und nach (b) der Anwendung [Srivastava u. a. 2014, S. 1930]

2.2 Stand der Technik

Dieser Abschnitt behandelt den aktuellen Stand der Technik. Hierzu werden verschiedene Publikationen vorgestellt, die Problemstellungen im Zusammenhang mit Schnee- und Wettererkennungen angehen. Zudem wird der ImageNet-Datensatz [Fei-Fei 2009] erläutert.

2.2.1 Binäre Wetterklassifizierung

Zwischen Sonne und Wolken

In der Veröffentlichung [Lu u. a. 2014] von 2014 werden die Wettereigenschaften sonnig und wolkig mit Hilfe eines erstellten Algorithmus auf Basis von Machine Learning und dem überwachten Lernen gelöst. Dazu wird ein passender Datensatz mit 10000 Bildern verwendet. Zur endgültigen Klassifikation wird ein Bild zunächst auf folgende Merkmale [f_{sk}; f_{sh}; f_{re}; f_{co}; f_{ha}] untersucht: Himmel, Schatten, Reflektion, Kontrast und Dunst. Zudem wird die Helligkeit des Bildes in dem Farbspektrum analysiert. Letztendlich konnte die normalisierte Genauigkeit auf 53,1 % gegenüber anderer Modelle wie SVM [Hearst u. a. 1998] gesteigert werden.

In 2015 wurde eine andere Publikation [Elhoseiny u. a. 2015] ebenfalls auf die Untersuchung der Wettereigenschaften zur Klassifizierung von sonnig und wolkig mit Hilfe des Deep Learnings vorgestellt. Dazu wurde ein CNN als Multi-Output-Modell verwendet, wie in Abbildung 2.13 dargestellt.

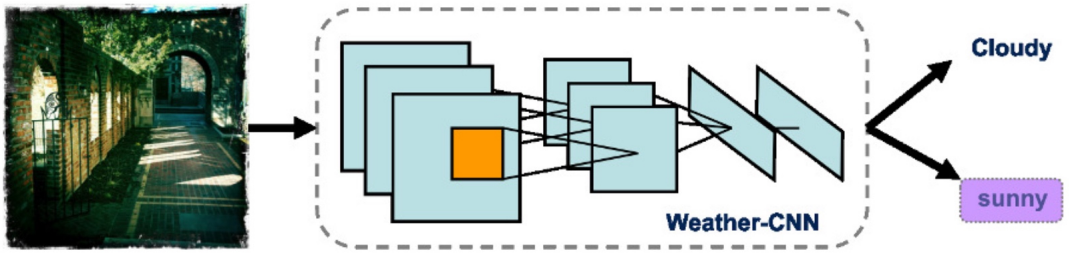


Abbildung 2.13: Wetterklassifizierung mit Hilfe von CNNs [Elhoseiny u. a. 2015, S. 1]

Das Netz basiert auf die in [Krizhevsky u. a. 2012] von A. Krizhevsky vorgestellte Netzarchitektur, das 2012 im Rahmen des Wettbewerbes zur Klassifizierung des ImageNet-Datensatzes die höchste Genauigkeit erzielte. Das in Abbildung 2.13 dargestellte Netz beinhaltet sieben Layers und eine Multi-Output-Schicht für die Ausgabe zur Unterscheidung des Bildes. Die normalisierte Genauigkeit konnte gegenüber der Veröffentlichung [Lu u. a. 2014] von 53,1 % auf 82,2 % gesteigert werden.

Die Autoren von [Lu u. a. 2014] veröffentlichen zwei Jahre später in 2016 ein Paper [Lu u. a. 2016], in der die gleiche Problemstellung auf Basis eines CNNs angegangen wurde. Hierbei erzielte das Netz deutlich höhere Genauigkeiten von über 90 %.

Satellitenbilder: Schnee oder Wolken?

In [Zhan u. a. 2017] wird die Vorhersage zur Unterscheidung zwischen Wolken und Schnee auf Satellitenbildern untersucht. Hierzu wurde das Deep Learning mit Hilfe von neuronalen Faltungsnetzen angewendet. Dabei erkennt das Netz eine Differenzierung zwischen Wolken, Schnee und Hintergrund. Das Ergebnis ist anhand eines Testbildes in Abbildung 2.14 dargestellt. Auf Basis von verschiedenen Daten wurde eine Relevanz (Precision) von 91,4 % und Sensitivität (Recall) von 91,5 % erzielt.

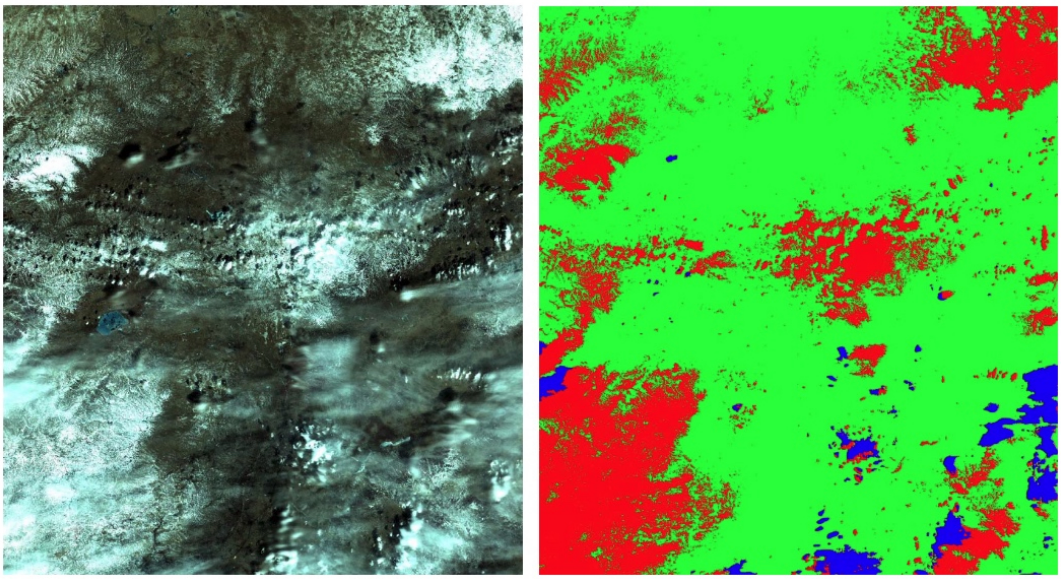


Abbildung 2.14: Unterscheidung zwischen Schnee (rot), Wolken (blau) und Hintergrund (grün) anhand eines Satellitenbildes [Zhan u. a. 2017, S. 1]

2.2.2 Schätzung der Wettereigenschaften

In einigen Publikationen wie [Ajayi und Wang 2019], [Zhao u. a. 2019], [Ibrahim u. a. 2019] und [Zhang u. a. 2016] wurde die Vorhersage von unterschiedlichen Wettereigenschaften untersucht. Dazu zählen folgende: Wolken, Regen, Dunst, Tag, Nacht, Nebel, Blendung, Reflektion und Schnee. Das in [Ibrahim u. a. 2019] vorgestellte Konzept verwendet vier Modelle auf Basis des Netzes ResNet50, die jeweils folgende Vorhersagen treffen, wie in Abbildung 2.15 dargestellt. Mitberücksichtigt werden zudem Metadaten von Bildern, z. B. die Uhrzeit. Anhand des Zeitpunktes lässt sich feststellen, ob ein Bild am Tag oder in der Nacht aufgenommen wurde.

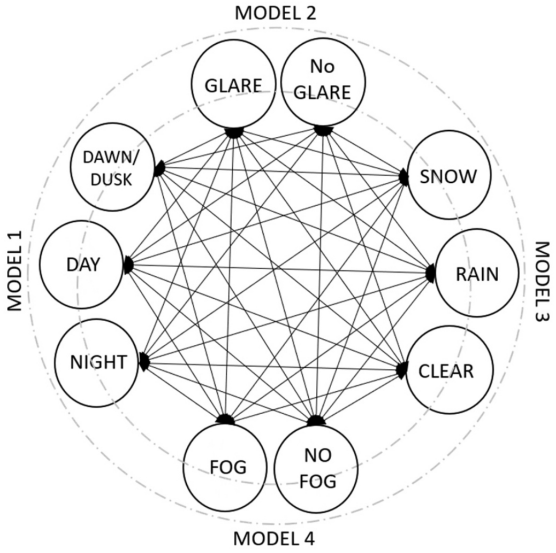


Abbildung 2.15: Wetter- und Lichtbedingungen [Ibrahim u. a. 2019, S. 7]

Es wurden 23865 gelabelte Bilder verwendet, auf denen hauptsächlich Straßen mit verschiedenen Wetterbedingungen zu erkennen sind. Abbildung 2.16 stellt einige Beispiele dar. Die höchste Durchschnittsgenauigkeit von 93 % wurde während der Klassenunterscheidung zwischen {Klarheit, Schnee, Regen} erzielt.

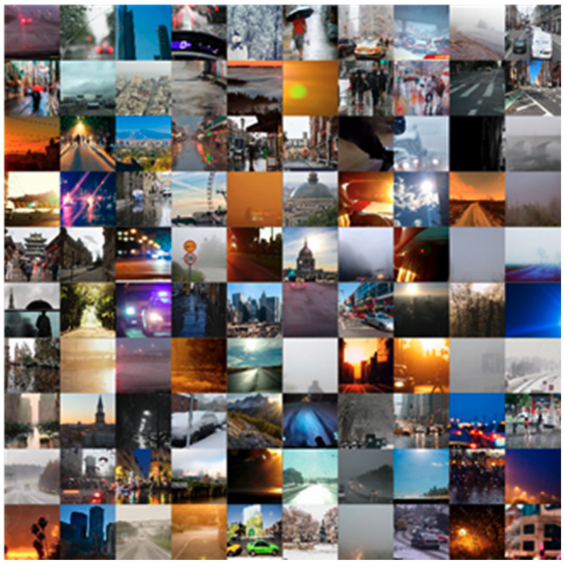


Abbildung 2.16: Beispiele der verwendeten Bilder [Ibrahim u. a. 2019, S. 9]

2.2.3 Datensätze

In [Chu u. a. 2016] wurden Bilder systematisch mit Hilfe eines Frameworks über Online-Dienste gesammelt. Dabei wurden Metadaten, wie z. B. Standort, Datum und Temperatur, mitberücksichtigt. Anschließend wurde mit dem aus Entscheidungsbäumen [Frosst und Hinton 2017] basierenden Klassifikator Random Forest versucht, eine Beziehung zwischen den Eigenschaften $\{sonnig, wolkig, verschneint, regnerisch, nebelig\}$ und den Bildern herzustellen sowie vorherzusagen. In Abbildung 2.17 sind einige Sehenswürdigkeiten zu erkennen.

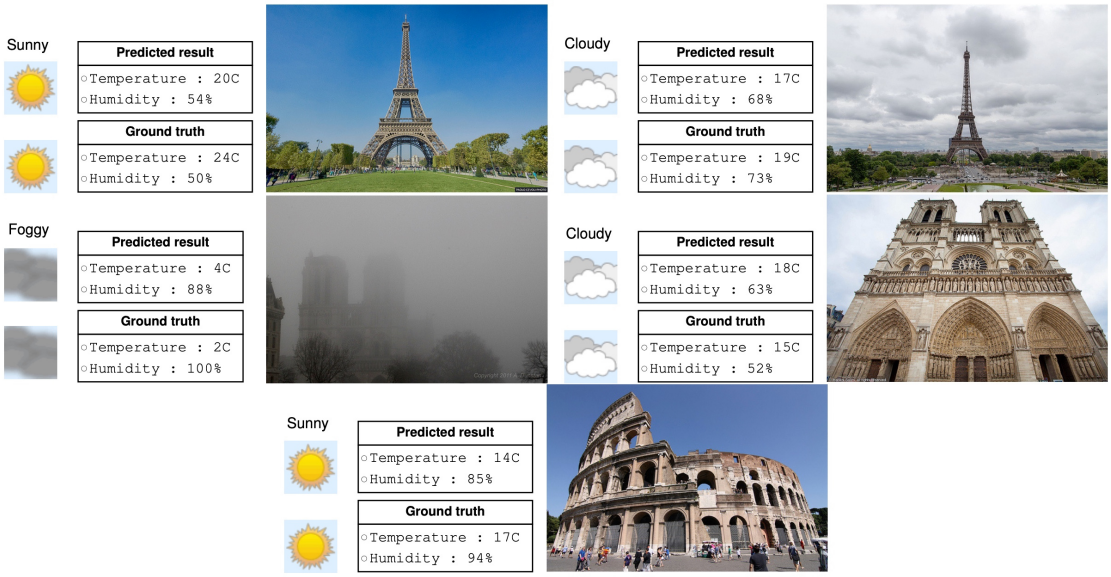


Abbildung 2.17: Vorhersagen verschiedener Sehenswürdigkeiten anhand des Datensatzes Image2Weather [Chu u. a. 2016, S. 144]

ImageNet [Fei-Fei 2009] ist ein öffentlich zugänglicher Datensatz mit 14197122 gelabelten Bildern (Stand: 19.12.2021). Dieser enthält mehr als 20000 Klassen von unterschiedlichen Kategorien. Eine Nutzung ist für Forschungszwecke erlaubt.

2.3 Ausgangssituation

2.3.1 Rechnerinfrastruktur

Die HAW Hamburg stellt im Rahmen dieser Arbeit einen Linux-Server zur Verfügung, der mit drei GPUs des Modells NVIDIA Tesla V100 ausgestattet ist. Diese Grafikkarte wurde speziell für das Maschine Learning entwickelt und erreicht jeweils eine Leistung von 149 Teraflops. Sie gehört aktuell (Stand: Mai 2021) zu den leistungsfähigen Grafikkarten für diesen Aufgabenbereich.

2.3.2 Bilddatensätze

Als Grundlage für eine Vorhersage des Schneebedeckungsgrades von unbekannten Bildern dienen Bilddatensätze. Bilder können auf unterschiedliche Arten gesammelt werden. Eine Möglichkeit besteht darin, existierende Bilddatenbanken zu verwenden. Eine andere ist, Bilddaten über Suchmaschinen zu sammeln, wie z. B. Google. Dies ist mit bestimmter freier Software [GitHub 2019a] auf Basis von Suchanfragen realisierbar. Alternativ können Daten gesammelt werden, indem z. B. eine Kamera installiert wird, welche Bilder mit einer vorgegebenen Bildrate abfotografiert und speichert. Die Aufnahme von Schnee ist jedoch abhängig von Standort, Temperatur und Jahreszeit.

In den folgenden Unterabschnitten werden Bilddatensätze vorgestellt, die für diese Arbeit zum Einsatz kommen.

muccam01

Der Datensatz muccam01 wurde vom DWD zur Verfügung gestellt und enthält 17337 ungelabelte Bilder. Auf den Bildern sind eine Flugzeuglandebahn, Wälder, eine Straße, Felder und der Himmel zu erkennen. Die Fotos sind von einer montierten Überwachungskamera mit einer Bildrate von ein Bild pro 15 Minuten aufgenommen. Es sind Tag- sowie Nachtbilder zu einer winterlichen Jahreszeit vorhanden. Die Bildgröße beträgt (6000×4000) Pixel.

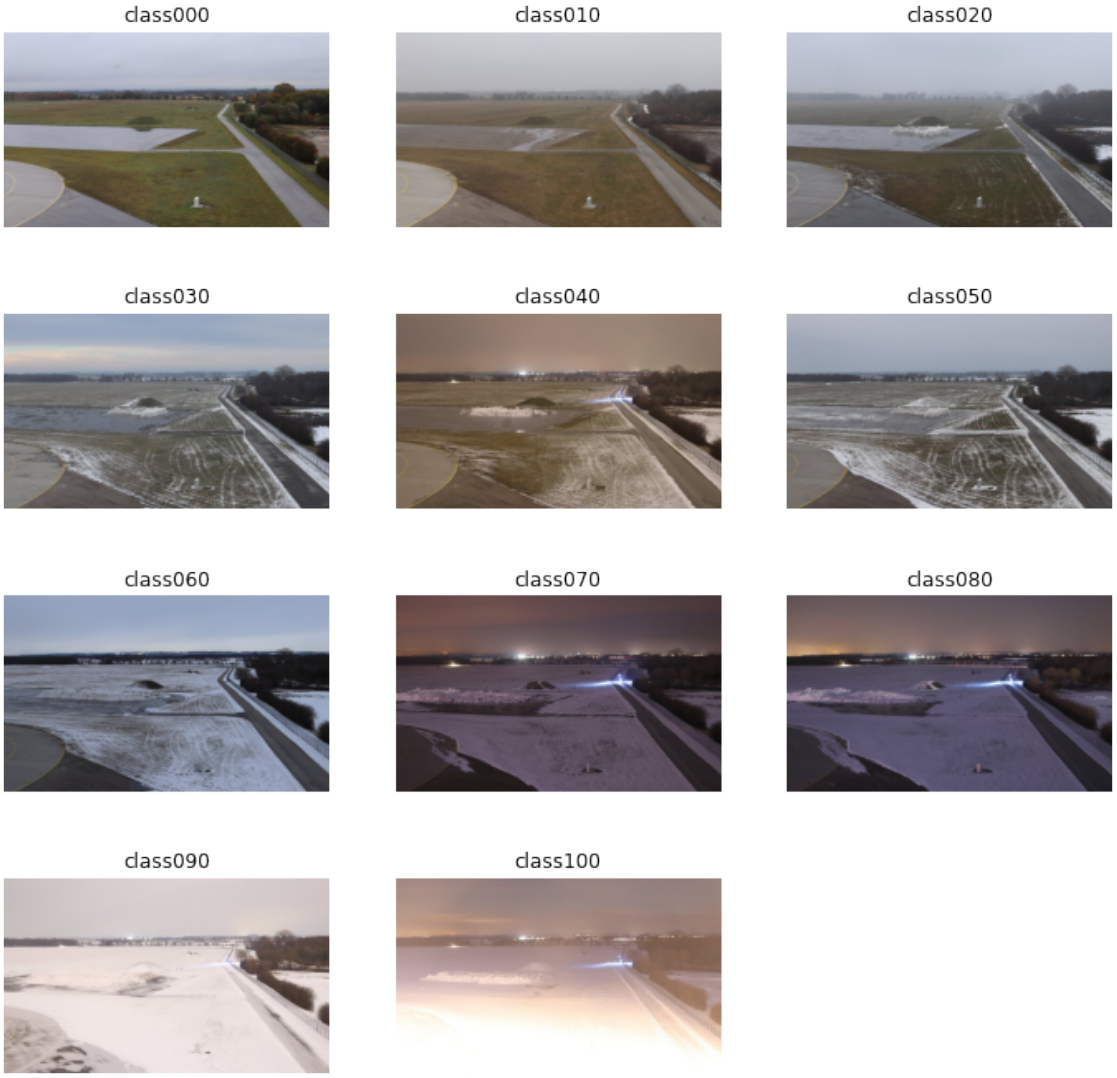


Abbildung 2.18: Beispielbilder des Datensatzes muccam01

Garten

Der vorhandene Datensatz enthält Aufnahmen von einer Indoor-Kamera und zeigt einen Garten im Außenbereich. Die Bilder sind tagsüber mit einer Bildrate von ein Bild pro Minute aufgenommen und bereits in fünf sowie elf Klassen unterteilt. Der Datensatz enthält 13218 gelabelte Bilder und die Bildgröße jedes einzelnen Bildes beträgt (440×480) Pixel.

Abbildung 2.19: Beispielbilder des Datensatzes Garten

bp_snowcam

Der Datensatz bp_snowcam wurde im Rahmen eines Bachelorprojektes an der HAW Hamburg erstellt [Everest u. a. 2021]. Diese zeigen unterschiedliche, mit und ohne Schnee, geprägte Bilder: Landschaften, Städte, Amateuraufnahmen und Webcambilder. Die gesammelten Bilder enthalten teilweise Text und Schriften. Der bereits gelabelte Bilddatensatz in elf Klassen besteht aus separat getrennten Tag- und Nachtbildern. Die Sammlung enthält 37000 tagsüber und 15926 während der Nacht aufgenommene Bilder. Die Bildgröße jedes Bildes beträgt (320×240) Pixel.

Abbildung 2.20: Beispielbilder des Datensatzes bp_snowcam (Tag)

Abbildung 2.21: Beispielbilder des Datensatzes bp_snowcam (Nacht)

Wasserkuppe

Der Datensatz Wasserkuppe zeigt Bilder, auf der eine Anlage, Bäume, der Himmel und eine Wiese zu erkennen sind. Er wurde vom DWD zur Verfügung gestellt. Aufgenommen wurden die Bilder von einer Outdoor-Webkamera der Marke Mobotix mit einer Bildrate von ein Bild pro 30 Minuten. Die Fotos sind in Tag und Nacht sowie in jeweils fünf Klassen aufgeteilt. Eine Aufteilung in elf Klassen ist nicht möglich, da die Anzahl der Bilder einiger Klassen zu gering ist. Die Bildgröße beträgt jeweils (2592×1944) Pixel.

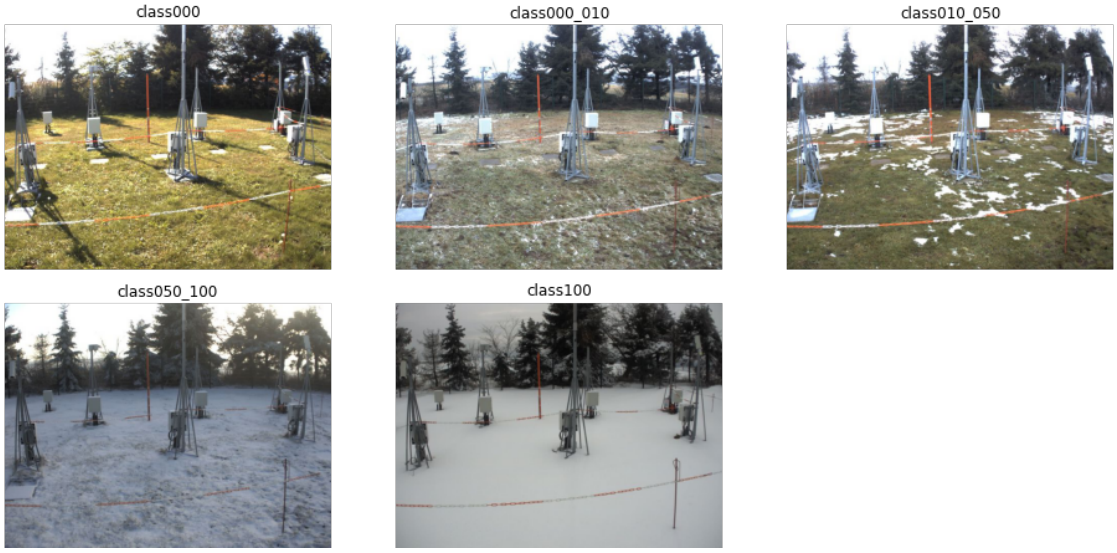


Abbildung 2.22: Beispielbilder des Datensatzes Wasserkuppe (Tag)

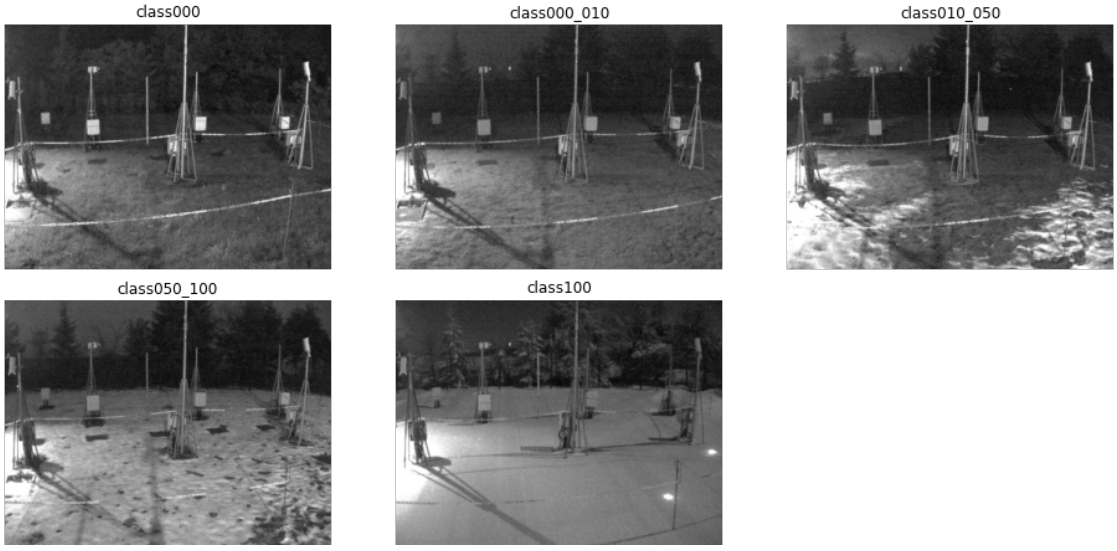


Abbildung 2.23: Beispielbilder des Datensatzes Wasserkuppe (Nacht)

3 Anforderungsanalyse

In diesem Kapitel werden Anforderungen festgelegt, die für den weiteren Verlauf der Arbeit berücksichtigt werden. Diese sind in festen Muss- und optionalen Kann-Anforderungen unterteilt. Eine Zusammenfassung der Anforderungen ist in Tabelle 3.1 zu finden.

3.1 Eingangsdaten

Die zu berücksichtigten Bilddaten sind Farbbilder, die während der Tageszeit und Graubilder, die zur Nachtzeit aufgenommen wurden. Diese sollen eine Auflösung von (640×480) Pixel aufweisen. Dabei kann die Bildgröße der Bilder abweichen. Dies ist jeweils von den gesammelten oder vorhandenen Datensätzen abhängig.

3.2 Ausgabedaten

Bei der Ermittlung des Schneebedeckungsgrades sollen drei unterschiedliche Lösungsansätze bezüglich der Klassifizierung verglichen werden:

- Schneebedeckungsgrad in den folgenden fünf Klassen:
  - geschlossen (100 %)
  - durchbrochen ($>$ 50 % bis $<$ 100 %)
  - Schneeflecken ($>$ 10 % bis 50 %)
  - Schneereste ($>$ 0 % bis 10 %)
  - unbedeckt (0 %)

- Schneebedeckungsgrad in elf Klassen entsprechend der Prozentangabe von 0 % bis 100 % in Zehnerschritten
- Schneebedeckungsgrad als Prozentangabe

3.3 Klassifizierungsgüte

Es ist eine Genauigkeit (Accuracy) der Klassifizierung anhand von Testbildern mit gleichverteiltem Schneebedeckungsgrad von über 90 % zu erzielen. Bezüglich der in Abschnitt 3.2 festgelegten Ansätze handelt es sich hierbei jeweils um eine Single-Label-Mehrfachklassifizierung, wobei jedes Bild genau einer Kategorie zugeordnet wird. Die Anzahl von korrekt vorhergesagten Bildern wird mit der Summe aller, unabhängig, ob sie richtig oder falsch klassifiziert sind, dividiert.

Die Accuracy für mehrere Klassen wird laut [Manliguez 2016, vgl.] zusammenfassend wie folgt definiert:

$$ACC = \frac{1}{m} \sum_{j=1}^{n} x_{jj} \tag{3.1}$$

wobei m die Gesamtanzahl der Testbilder, n die Anzahl der Klassen, x_{jj} die Menge der korrekt klassifizierten Bilder einer Klasse (True Positive) ist. Der erste Index von x_{jj} ist die tatsächliche und der zweite die vorhergesagte Klasse.

Für den Einsatz von Regression soll die mittlere absolute Abweichung (englisch: Mean Absolute Error, kurz: MAE) von 10 % nicht überschritten werden. Der MAE ist ein statistischer Maßstab und gibt die Höhe der Abweichung von Vorhersagen an. Da während der Klassifizierung am Ausgang eine Prozentzahl ermittelt wird, handelt es sich hierbei um eine skalare Regression. Der MAE ist wie folgt [Géron 2018, S. 39] definiert:

$$MAE = \frac{1}{m} \sum_{i=1}^{m} \left| h(x^{(i)}) - y^{(i)} \right| \tag{3.2}$$

wobei m die Anzahl der Testbilder, *y(i)* die tatsächliche und $h(x^{(i)})$ die vorhergesagte Klasse entspricht.

3.4 Berechnungszeit

Die Berechnungszeit des Schneebedeckungsgrades (Feedforward) soll 200 ms auf einem Computer mit der CPU-Leistungsklasse *Intel Core i5* und gegebenenfalls in Kombination mit einer GPU auf Basis eines Eingangsbildes in VGA-Auflösung[1] nicht überschreiten.

3.5 Methodik

Im Bereich Machine Learning ist sich auf Ansätze zu beschränken, die sich auf Einzelbilder ohne Berücksichtigung der Zeitreihe orientieren. Insbesondere sind neben selbsterstellten Netzen unter Nutzung von CNNs vortrainierte Netze in Betracht zu ziehen. Zudem sind die Ausgabelayer jeweils für die geforderten Ausgaben, wie Klassen oder Regression, anzupassen und zu trainieren. Eine gleichzeitige Betrachtung der Ausgaben als Multi-Output-Modell ist denkbar. Ein zeiteffizientes Verfahren zur Anpassung des optimal geeigneten Verfahrens auf einen anderen Standort der Kamera ist zu entwickeln und dessen Funktion nachzuweisen.

3.6 Allgemeine Anforderungen

Die Lösung soll unter einem Linux-basierten Betriebssystem in der Programmiersprache Python entwickelt werden und es darf nur Open-Source-Software zur Vermeidung von Lizenzkosten eingesetzt werden. Im Bereich Machine Learning sind das Framework TensorFlow [Team 2015] für die datenstromorientierte Programmierung in Verbindung mit der Deep Learning-Bibliothek Keras [Chollet 2015] einzusetzen. Im Ergebnis soll auf der Grundlage von Kennzahlen die Eignung der betrachteten Verfahren bewertet werden.

Nachfolgend werden diese Anforderungen in Tabelle 3.1 zusammengefasst.

[1] Bildgröße: (640×480) Pixel

Tabelle 3.1: Anforderungsliste

ID	Kann/ Muss	Anforderung
1	M	Die Bilddaten bei Tag müssen Farbbilder sein.
2	M	Die Bilddaten bei Nacht müssen Graubilder sein.
3	K	Die Bilddaten sollen eine Auflösung von (640×480) Pixel haben.
4	M	Der Schneebedeckungsgrad soll entsprechend der Prozentangabe von 0 % bis 100 % in Zehnerschritten aus elf Klassen ermittelt werden.
5	M	Der Schneebedeckungsgrad soll als Prozentangabe von 0 % bis 100 % aus fünf Klassen (0 %, $>$ 0 % bis 10 %, $>$ 10 % bis 50 %, $>$ 50 % bis $<$ 100 %, 100 %) ermittelt werden.
6	K	Der Schneebedeckungsgrad soll als Prozentangabe ausgegeben werden.
7	M	Die Klassifizierungsgüte soll in Genauigkeit gemessen werden.
8	K	Der MAE soll ermittelt werden.
9	K	Die Genauigkeit von Testdaten soll über 90 % betragen.
10	K	Die MAE von Testdaten soll 10 % nicht überschreiten.
11	M	Die Berechnungszeit des Schneebedeckungsgrades soll 200 ms auf einer CPU der Leistungsklasse Intel Core i5 und gegebenenfalls in Kombination mit einer GPU nicht überschreiten.
12	M	Ein Modell soll ohne Berücksichtigung der Zeitreihe von Bilddaten trainiert werden.
13	M	Die Ausgabelayer sind jeweils für die Klassen und Klassifizierungsgüte laut Anforderung 4 – 8 anzupassen.
14	K	Ein Multi-Output-Modell soll realisiert werden.
15	M	Die Lösung soll unter einem Linux-basierten Betriebssystem in der Programmiersprache Python entwickelt werden.
16	M	Es muss Open-Source-Software verwendet werden.
17	M	Es müssen das Framework Tensorflow und die Bibliothek Keras verwendet werden.
18	M	Die Modelle müssen auf Grundlage von Kennzahlen nach dem optimalen Modell bewertet werden.
19	M	Vortrainierte Netze sollen unter Keras verfügbar sein.

4 Konzeption

Im folgenden Kapitel werden verschiedene vortrainierte Netze genannt, die in Keras verfügbar sind. Drei von ihnen werden für die Umsetzung ausgewählt und näher beschrieben. Im Anschluss wird die Vorgehensweise zur Umsetzung erläutert. Dabei werden geeignete Algorithmen für das Training des Modells genannt.

Um das Problem der Schneeerkennung zu lösen, muss das Netzwerk Bilder laut Anforderung 3 in Tabelle 3.1 mit einer bestimmten Auflösung verarbeiten können. Aufgrund der zu hohen Parameteranzahl der Bilder sind gewöhnliche KNNs nicht dazu geeignet, um das Problem der Schneeerkennung zu lösen. Aus diesem Grund werden CNNs eingesetzt.

4.1 Übersicht vortrainierter Netze in Keras

In Keras sind über eine Programmierschnittstelle unterschiedliche vortrainierte Netze verfügbar. Einige dieser Netze sind in Tabelle 4.1 aufgelistet. Die Top-Eins- und Top-Fünf-Genauigkeiten beziehen sich jeweils auf den ImageNet-Datensatz. Jedes Netz verfügt über eine individuelle Anzahl an Parametern, die sich in einem Bereich von 3538984 und 143667240 erstrecken. Die Tiefe eines Netzes beschreibt die Anzahl der Schichten vom Input Layer bis zum Output Layer, die laut Tabelle 4.1 mit einer Anzahl von 23 bis 572 reicht. Einige Netze enthalten keine Informationen über deren Tiefe, da sie Verzweigungen aufweisen, die aus mehreren unterschiedlichen Schichten bestehen und teilweise miteinander verknüpft sind. Des Weiteren ist die Zeit pro Ableitungsschritt einer nicht näher genannten CPU sowie GPU angegeben.

Tabelle 4.1: Verfügbare pretrained Modelle in Keras [Chollet 2015]. Auswahlentscheidung grün markiert.

Modell	Top-1-Genauigkeit [%]	Top-5-Genauigkeit [%]	Parameter	Tiefe	Zeit (CPU) [ms]	Zeit (GPU) [ms]
MobileNet	0,704	0,895	4253864	88	22,60	4,16
MobileNet	0,704	0,895	4253864	88	22,60	4,16
MobileNetV2	0,713	0,901	3538984	88	25,90	3,83
VGG16	0,713	0,901	138357544	23	69,50	4,16
VGG19	0,713	0,900	143667240	26	84,75	4,38
NASNetMobile	0,744	0,919	5326716	-	27,04	6,70
ResNet50	0,749	0,921	25636712	-	58,20	4,55
DenseNet121	0,750	0,923	8062504	121	77,14	5,38
ResNet50V2	0,760	0,930	25613800	-	45,63	4,42
DenseNet169	0,762	0,932	14307880	169	96,40	6,28
ResNet101	0,764	0,928	44707176	-	89,59	5,19
ResNet152	0,766	0,931	60419944	-	127,43	6,54
ResNet101V2	0,772	0,938	44675560	-	72,73	5,43
DenseNet201	0,773	0,936	20242984	201	127,24	6,67
InceptionV3	0,779	0,937	23851784	159	42,25	6,86
ResNet152V2	0,780	0,942	60380648	-	107,50	6,64
Xception	0,790	0,945	22910480	126	109,42	8,06
InceptionResNetV2	0,803	0,953	55873736	572	130,19	10,02
NASNetLarge	0,825	0,960	88949818	-	344,51	19,96

4.2 Auswahl vortrainierter Netze

In diesem Abschnitt findet eine Auswahl an vortrainierten Netzen statt, die für den weiteren Verlauf in dieser Arbeit zur Lösung der Problemstellung verwendet wird. Eine grafische Darstellung der vollständigen Netzstruktur eines vortrainierten Netzes kann in Keras mit Hilfe der Funktion `tensorflow.keras.utils.plot_model()` generiert werden.

4.2.1 VGG16

VGG16 ist ein bekanntes Netz, das in verschiedenen Büchern erwähnt wird [Géron 2018] [Chollet 2018]. Das Modell wurde 2014 von Andrew Zisserman und Karen Simonyan im Rahmen des ImageNet-Wettbewerbs vorgestellt [Simonyan und Zisserman 2015].

Abbildung 4.1 zeigt die Netzarchitektur von VGG16 mit einem Input Layer. Die Struktur besteht aus insgesamt fünf Blöcken: Die ersten beiden beginnen mit jeweils zwei Conv2D Layern und enden mit einem MaxPooling2D Layer. Die letzten drei Blöcke weisen jeweils drei Conv2D Layer und ein anschließender MaxPooling2D Layer auf. Das Netz verwendet dabei durchgehend 3×3 Filter. Eine andere Variante auf Basis von VGG16 ist VGG19, mit dem Unterschied, dass dieser zusätzliche Schichten enthält und somit eine tiefere Netzstruktur aufweist.

Das Netz beinhaltet laut Tabelle 4.1 die geringste Netztiefe von allen mit einer Anzahl von 23. Zudem enthält dieser die zweitgrößte Menge an Parametern vor VGG19. VGG16 ist in der Top-Fünf-Genauigkeit um 0,1 % genauer als VGG19. Die Berechnungszeit von VGG19 liegt aufgrund der Netztiefe jeweils auf der CPU sowie GPU im Vergleich zu VGG16 höher. Aufgrund des Bekanntheitsgrades in den Literaturen und der kurzen Rechenzeit, die laut Anforderung 11 in Tabelle 3.1 relevant ist, wird VGG16 ausgewählt.

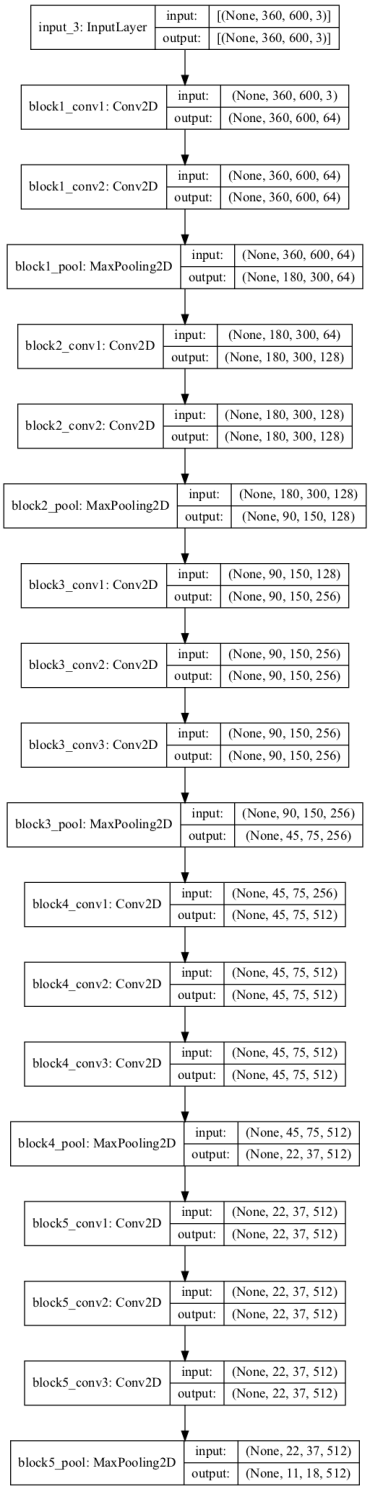


Abbildung 4.1: VGG16-Architektur

4.2.2 Xception

Xception [Chollet 2017] wurde 2017 von François Chollet veröffentlicht und ist aus der Inspiration des Netzes Inception entstanden. Xception verwendet tiefengetrennte Faltungen und mehrfache Verbindungen, wie in Abbildung 4.2 anhand der Netzstruktur abgebildet. Die Daten durchlaufen der Reihe nach den Eingangsdurchfluss, achtmal den mittleren Durchfluss und anschließend den Ausgangsfluss. Zudem erfolgt auf jeden Conv und SeperableConv Layer eine Batch-Normalisierung, die in Abbildung 4.2 nicht angezeigt ist [Chollet 2017].

Das Netz befindet sich in der Top-Eins- und Top-Fünf-Genauigkeit laut Tabelle 4.1 auf Platz drei. Die Parameteranzahl von 22910480 liegt in der unteren Hälfte. Mit einer Tiefe von 126 Schichten ist dieser im mittleren Bereich und daher tiefer als das Modell VGG16. Höhere Genauigkeiten als Xception bieten NASNetLarge und InceptionResNetV2. Beide dieser Netze weisen in der Berechnungsgeschwindigkeit langsamere Zeiten als Xception auf und beinhalten dabei eine höhere Parameteranzahl. Im Gegensatz zu VGG16 bietet Xception eine deutlich größere Genauigkeit an, ist in Bezug zur Netzstruktur tiefer und enthält weniger Parameter. Aufgrund der vielversprechenden und hohen Genauigkeit, die laut Anforderung 9 in Tabelle 3.1 erreicht werden soll, wird Xception ausgewählt.

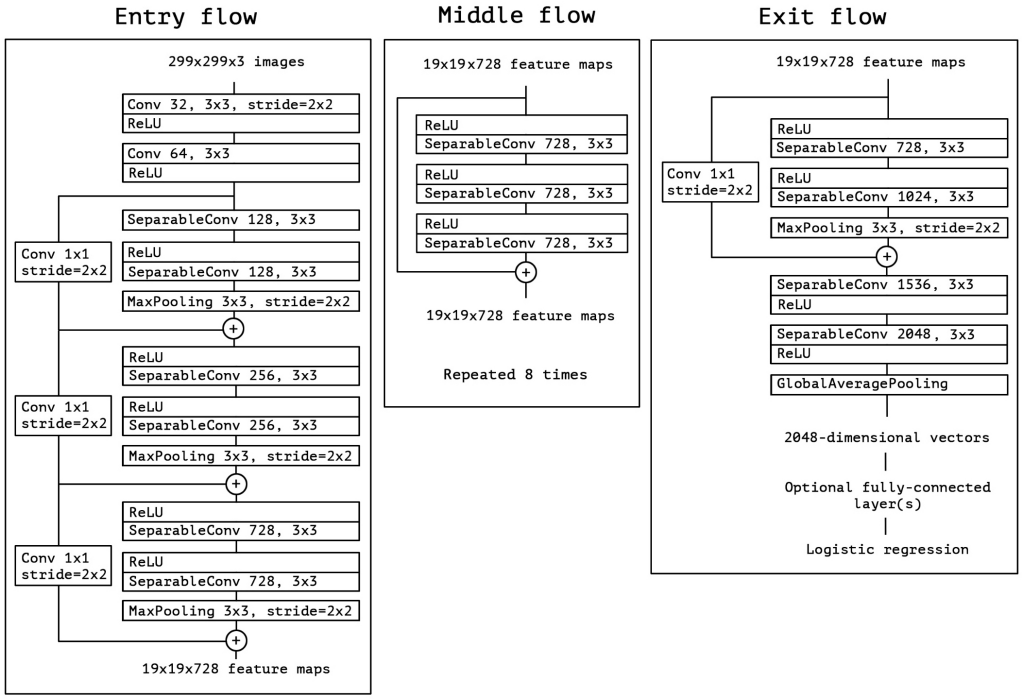


Abbildung 4.2: Xception-Architektur [Chollet 2017, S. 5]

4.2.3 DenseNet201

DenseNets haben laut [Huang u. a. 2018, S. 1] mehrere Vorteile: *„Sie dämpfen das Problem des verschwindenden Gradienten ab, festigen die Merkmalsausbreitung, fördern die Wiederverwendung von Merkmalen und reduzieren die Anzahl der Parameter erheblich“.*

DenseNet201 ist laut Tabelle 4.1 das zweittiefste Netz mit 201 Schichten. Dieses liegt jeweils um 1,7 % in der Top-Eins- und um 0,9 % in der Top-Fünf-Genauigkeit niedriger als Xception. DenseNet201 beinhaltet eine Parameteranzahl von 20242984, der im Vergleich zu VGG16 und Xception am niedrigsten ist. Das Netz basiert auf das DenseNet und ist in verschiedenen Ausprägungen verfügbar, die unterschiedlich viele Layers sowie Parameter beinhaltet. DenseNet201 weist die höchste Genauigkeit von allen verfügbaren DenseNet-Varianten auf.

Um ein breites Spektrum an unterschiedlichen Netzarchitekturen und Lösungen zu testen, wird DenseNet201 für den weiteren Verlauf mitberücksichtigt. In Abbildung 4.3 ist der grobe Aufbau eines DenseNets dargestellt.

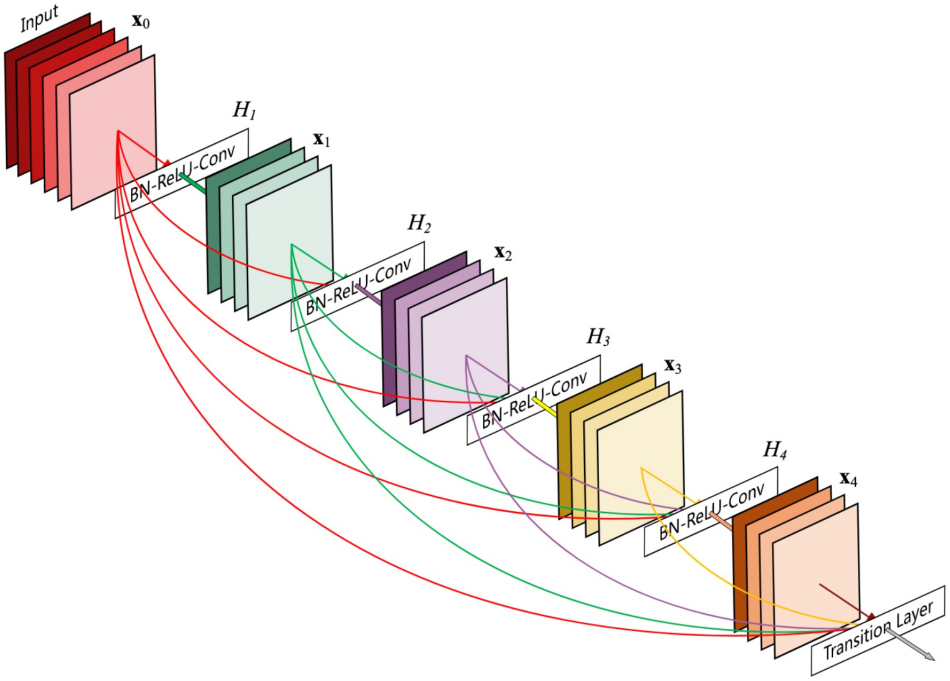


Abbildung 4.3: DenseNet-Architektur [Huang u. a. 2018, S. 1]

Da die Netze VGG16, Xception und DenseNet201 in Keras verfügbar sind, ist Anforderung 19 in Tabelle 3.1 erfüllt.

4.3 Erweiterung der Netze

In diesem Abschnitt werden vortrainierte Netze um zusätzliche Layer erweitert. Dadurch soll eine optimale Klassifikation der Ergebnisse ermöglicht werden.

4.3.1 Hidden Layers

Ein vortrainiertes Netz dient als Grundbasis für ein verwendetes Modell. Um die gewünschten Ausgabewerte laut Aufgabenstellung am Output Layer zu erhalten und zu optimieren, werden hierfür zusätzliche Hidden Layers hinzugefügt.

Die letzte Schicht eines vortrainierten Netzes verwendet ein 4D-Tensor wie folgt: (batch, x, y, f). Für eine Ermittlung der Ausgabewerte laut Anforderung 4 - 6 in Tabelle 3.1 müssen die Parameter x, y, und f zu einem eindimensionalen Vektor überführt werden. Dieser Vorgang wird Flattening genannt und der Batch soll dabei nicht mitberücksichtigt werden. Der Vektor wird eine Parameterzahl enthalten, die reduziert werden muss, um die Anzahl der Klassen am Ausgang zu erhalten.

Die Werte von Neuronen können negative sowie positive sein. Da der optimale Wert für die MAE null und die Genauigkeit minimal null betragen soll, kann der Wertebereich beschränkt werden. Hierzu eignet sich eine ReLU-Funktion, die in einem Dense Layer bekanntgegeben wird. Die Anzahl der Neuronen soll einen nicht zu geringeren Wert betragen, da zu viele relevante Informationen verworfen werden könnten. Für den weiteren Verlauf wird eine Anzahl von 512 gewählt.

Des Weiteren soll die Wahrscheinlichkeit einer Überanpassung reduziert werden. Dazu wird das Dropout-Verfahren mit einem Wahrscheinlichkeitswert von 20 % gewählt. Da hierbei ebenfalls zu viele Informationen verworfen werden könnten, wird dieser absichtlich gering gehalten, sodass er jedoch immer noch einer Überanpassung entgegenwirken kann.

Zum Abschluss werden Neuronenwerte, die zum jetzigen Zeitpunkt den Wertebereich $\mathbb{W} = [0; +\infty]$ betragen, vor der Ermittlung des entgültigen Wertes normalisiert. Dazu wird das in Abschnitt 2.1.6 beschriebene Verfahren Batch-Normalisierung noch vor der Entscheidung im Output Layer angewendet. Es folgt eine Zusammenfassung der Hidden Layers in Abbildung 4.4.

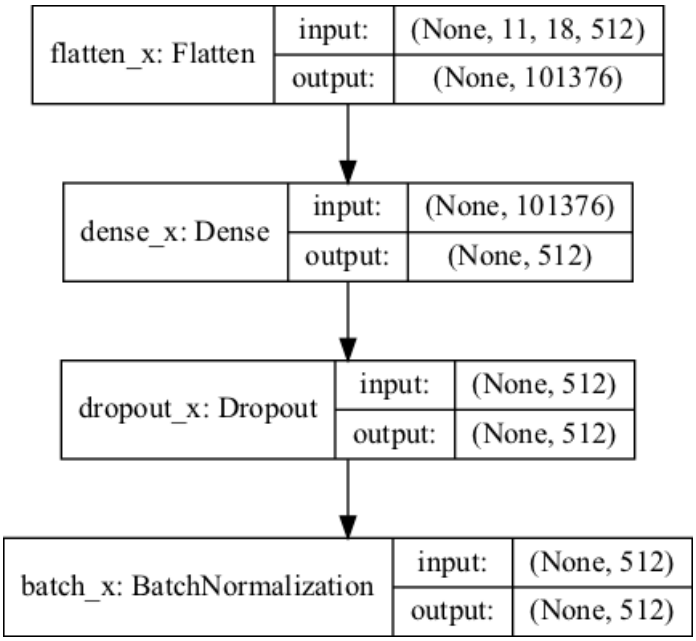


Abbildung 4.4: Zusätzliche Hidden Layers

4.3.2 Output Layers

Ein Netz schließt mit einem Output Layer als letzten Schritt ab, um das Ergebnis einer Klassifizierungsgüte auszugeben. Laut Anforderung 6 - 8 in Tabelle 3.1 soll dieser die Genauigkeit und den MAE jeweils als Prozentangabe ermitteln. Dazu wird ein Dense Layer für die Ausgabe verwendet. Zur Ermittlung der Accuracy beträgt die Anzahl der Neuronen gleich die der zu vorhersehenden Klassen. Dabei soll jedes Neuron eine Wahrscheinlichkeit zur Klassenzugehörigkeit ausgeben. Der höchste Wahrscheinlichkeitswert $\mathbb{W}$ stellt die Vorhersage des Eingangsbildes dar.

Zudem soll laut Gleichung 3.2 ein MAE-Wert in Prozent für alle Testdaten ermittelt werden. Dazu wird ein Neuron im Output Layer benötigt. In Abbildung 4.5 sind die verwendeten Output Layers dargestellt.

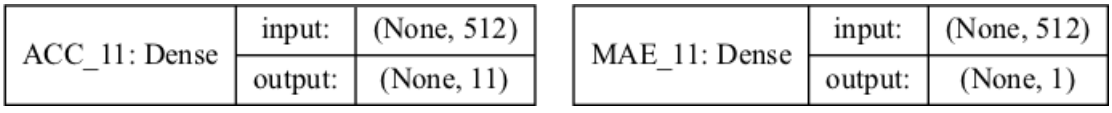


Abbildung 4.5: Output Layers zur Klassifizierung von elf Klassen (links) und zur Ermittlung des MAEs (rechts)

Abbildung 4.6 zeigt den vollständigen Aufbau des verwendeten Multi-Output-Modells mit Beispielwerten. Dieser beginnt mit einem Input Layer für die Einspeisung von Bildern. Dazu wird ein vierdimensionaler Tensor für die Eingabe verwendet. Dieser setzt sich der Reihe nach wie folgt zusammen: Batch, jeweils die Höhe und Breite in Pixel des Bildes sowie der Farbraum, wobei die drei für das RGB-Farbspektrum steht. Nach dem Input Layer folgen das vortrainierte Netz sowie die erweiterten Hidden Layers. Das Multi-Output-Modell schließt mit zwei Output Layern ab, worüber die Accuracy aus elf Klassen und der MAE als Prozentangabe ermittelt wird. Für die Feststellung der Genauigkeit von fünf Klassen wird ein separates Modell mit gleichem Aufbau verwendet, wobei der Output Layer zur Ermittlung des MAE fehlt, da die Klassen laut Anforderung 5 in Tabelle 3.1 nicht gleichmäßig verteilt sind. Mit der Verwendung eines Multi-Output-Modells ist Anforderung 14 in Tabelle 3.1 erfüllt.

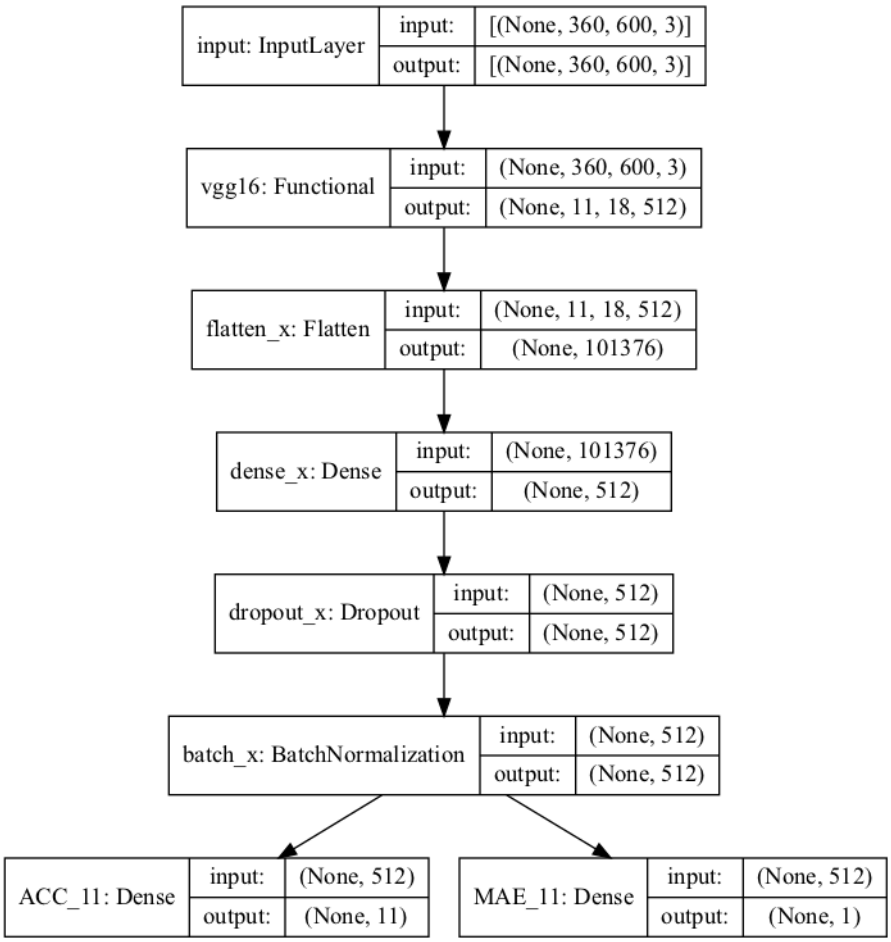


Abbildung 4.6: Aufbau des endgültigen Multi-Output-Modells zur Schneeerkennung für die in elf Klassen vorsortierten Datensätze

4.4 Algorithmus für das Training

Für das Training werden vortrainierte Gewichte anhand des Imagenet-Datensatzes verwendet. Es hat sich gezeigt, dass eine Nutzung von vortrainierten Gewichten optimale Ergebnisse liefern, wie in [Yosinski u. a. 2014] ausführlich beschrieben ist. Aus diesem Grund werden diese Gewichte verwendet, um die Anforderungen 9 - 10 in Tabelle 3.1 zu erreichen.

Als Optimierer für das Training wird der in Unterabschnitt 2.1.4 beschriebene Algorithmus Adam verwendet. Dieser ist in verschiedenen Literaturen [Chollet 2018] [Géron 2018] [Kingma und Ba 2015] vertreten und gilt als effizient sowie leistungsstark. Die optimale Lernrate wird, aufgrund der in Unterabschnitt 2.1.4 beschriebenen Probleme, im nächsten Kapitel ermittelt.

Zur Ermittlung der Genauigkeit soll eine Klasse aus mehreren ermittelt werden. Dazu eignet sich die Softmax-Aktivierungsfunktion. Diese wird, wie in Unterabschnitt 2.1.2 beschrieben, für den weiteren Verlauf verwendet. Als Verlustfunktion wird hierzu die kategorische Kreuzentropie eingesetzt. Für eine Feststellung des MAEs gibt es eine Vielzahl unterschiedlicher Aktivierungsfunktionen, die das Problem der Schneeerkennung lösen könnten. Eine Möglichkeit besteht darin, keine Aktivierungsfunktion zu verwenden. Eine nähere Untersuchung des optimalen Verfahrens wird während der Entwicklung und Umsetzung in Kapitel 5 ermittelt. Zusammenfassend zeigt die folgende Tabelle Kombinationen aus Klassifizierungsgüte, Neuronenanzahl des Output Layers, Aktivierungs- und Verlustfunktion. Diese eignen sich jeweils für die drei Ansätze laut Anforderung 4 - 6 in Tabelle 3.1. Einige dieser Kombinationen werden in [Chollet 2018, S. 114] erwähnt. Mit der Umsetzung nach Tabelle 4.2 wird Anforderung 13 in Tabelle 3.1 erfüllt.

Tabelle 4.2: Zusammenfassung der Algorithmen zur Ermittlung der Genauigkeiten und des MAEs

Güte	Klassen	Neuron(en) am Ausgang	Aufgabentyp	Aktivierungs-funktion des letzten Layers	Verlust-funktion
Accuracy	5 11	5 11	Single-Label-Mehr-fachklassifizierung	softmax	categorial-crossentropy
MAE	11	1	Regression (beliebige Werte)	*keine*, softplus, ReLU, tanh, sigmoid, linear	mse

4.5 Vorgehensweise

Für die Umsetzung der Anforderungen laut Tabelle 3.1 wird während der Softwareentwicklung nach drei Phasen vorgegangen. In der ersten Phase werden die Datensätze aus Unterabschnitt 2.3.2 nach den Anforderungen 4 - 5 in Tabelle 3.1 sortiert und nach Anforderung 1 - 3 verarbeitet. Anschließend werden Bilddaten für das Training vorbereitet. Näheres hierzu wird im nächsten Kapitel beschrieben.

Die zweite Phase beinhaltet die Implementierung des Modells mit den dazugehörigen Hidden und Output Layers. Hierzu sollen die genauen Bedingungen für die Protokollierung der Trainingsergebnisse sowie die Speicherung des trainierten Modells festgelegt werden. Anschließend erfolgt eine Kompilierung und schließlich das Training der Modelle auf Basis der Datensätze.

In der dritten und letzten Phase soll das Modell optimiert werden. Dazu wird zur optimalen Vorhersage Folgendes ermittelt:

- Lernrate des Algorithmus Adam
- Anzahl von Neuronen des versteckten Dense Layers, das in Unterabschnitt 4.3.1 beschrieben ist
- Aktivierungsfunktion für die Ermittlung des MAEs

Eine Zusammenfassung der Vorgehensweise ist in Abbildung 4.7 dargestellt.

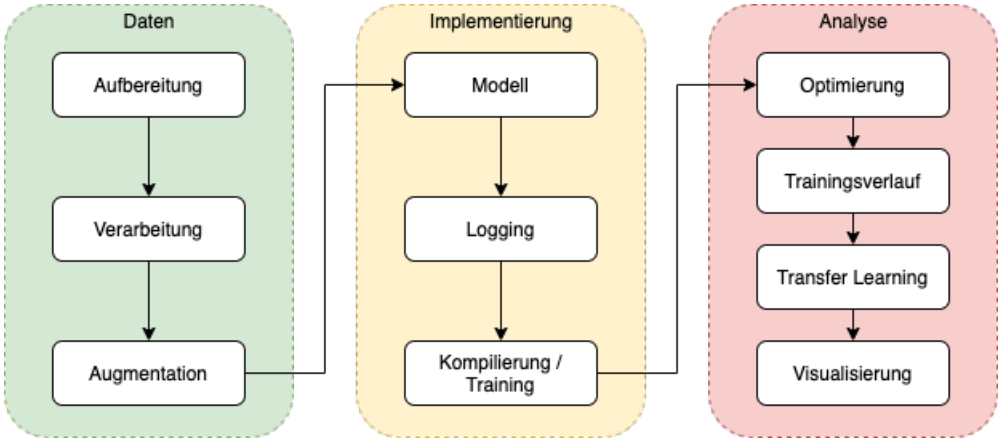


Abbildung 4.7: Vorgehensweise für die Umsetzung der Software

4.6 Bibliotheken und Entwicklungsumgebung

Zur Umsetzung des Python-Codes sind Bibliotheken erforderlich. Diese enthalten verschiedene Funktionen und Klassen für bestimmte Anwendungen, die über eine API verfügbar sind. Die festgelegten Funktionen sind in öffentlichen Dokumentationen einsehbar. Bibliotheken enthalten verschiedene Lizenzarten, auf die nicht näher eingegangen wird. Dennoch haben alle die Gemeinsamkeit, dass sie als Open Source verfügbar sind. Damit dürfen diese verwendet werden, ohne Lizenzkosten zu zahlen. Eine Liste der verwendeten Bibliotheken ist in Tabelle 4.3 aufgelistet. Durch ihre Verwendung sind Anforderung 15 - 17 in Tabelle 3.1 erfüllt.

Tabelle 4.3: Benötigte Bibliotheken

Bibliothek	Version	Lizenz	Beschreibung
TensorFlow	2.6.0	Apache-Lizenz, V2.0	Framework zur datenstromorientierten Programmierung
Keras	2.6	MIT-Lizenz	Deep-Learning-Bibliothek
NumPy	1.21.4	3-Klausel-BSD-Lizenz	Umgang mit Arrays, Vektoren, Matrizen
pandas	1.3.4	3-Klausel-BSD-Lizenz	Datenanalyse und -verwaltung
split-folders	0.4.3	MIT-Lizenz	Aufteilung in Trainings-, Validierungs- und Testdaten
Matplotlib	3.4.2	Matplotlib-Lizenz	Erstellung von Visualisierungen
scikit-learn	1.0.1	3-Klausel-BSD-Lizenz	Datenanalyse und -verwaltung
os		GNU General Public License (GPL)	Nutzung Funktionalität von Linux
PIL		Python Imaging Library license	Handhabung von Bilddateien
random, itertools, glob, shutil, warnings		Python Software Foundation License (PSFL)	Python-Module

Als Entwicklungsumgebung wird die web-basierte Open-Source-Software Jupyter Notebook [Pérez und Granger 2015] verwendet. Der Service wird als Server gestartet und über einen Browser, wie z. B. Firefox, mit Hilfe eines generierten Cookies auf eine IP-Adresse und Port-Nummer zugegriffen. Ein Vorteil von Jupyter Notebook ist, dass dieser unabhängig vom Betriebssystem ausgeführt und darauf zugegriffen werden kann. Projekte lassen sich in Dateien mit der Endung `.ipynb` im JSON-Format abspeichern. Diese enthalten verschiedene Blöcke mit Eingabe- und Ausgabedaten, auf denen Texte, Plots oder Codes enthalten sind. Jupyter Notebook unterstützt die Programmiersprache Python über einen Kernel.

5 Entwicklung und Umsetzung

Im folgenden Kapitel wird das Konzept aus Kapitel 4 in Python umgesetzt und entwickelt. Dazu gehören unter anderem die Vorverarbeitung und Aufbereitung der Bilddatensätze. Des Weiteren werden Netze und die Auswertung der Ergebnisse implementiert.

5.1 Daten

Voraussetzung für das Training eines Modells ist, dass die Datensätze den Klassen bereits zugeordnet sind. Das Modell versucht während des Trainings anhand der gelabelten Daten Zusammenhänge zu ermitteln und Neuronen zu gewichten. Je mehr Daten vorhanden sind, desto mehr Informationen kann ein Modell verarbeiten und desto genauer könnten die Vorhersagen werden. Jedoch wird die Trainingszeit des Modells dadurch umso länger andauern. Ein Datensatz sollte stets umfangreiche, vielfältige und gleichverteilte Daten von jeder Klasse enthalten [Géron 2018, vgl., S. 29].

5.1.1 Aufbereitung

Die Bilder des Datensatzes muccam01 werden mit Hilfe der freien Software ImageMagick [LLC 1990] auf eine Bildgröße von (600×360) Pixel reduziert. Der Zeitstempel wird entfernt, da das Modell eine Korrelation zwischen Monatszahl und Schneebedeckungsgrad erkennen könnte. Um eine Grundbasis für die Erfüllung der Anforderungen 4 - 6 in Tabelle 3.1 zu schaffen, werden die Bilder manuell in elf und fünf Klassen sortiert. Die Auswahl der Klassennamen erfolgt bewusst nach alphabetischer Reihenfolge, da sonst die Funktion `image_dataset_from_directory()` Bilder während der Vorverarbeitung falsch interpretiert werden. Nach folgender Vorgehensweise werden die Bilder in elf verschiedenen Klassenordnern einsortiert:

1. Bilder mit einem hohem Nebelanteil werden entfernt, da nicht eindeutig ist, ob Schnee vorhanden ist.
2. Fotos mit und ohne erkennbaren Schneeanteil werden voneinander getrennt.
3. Bilder ohne Schneeanteil werden der Klasse class000 zugewiesen.
4. Bilder mit Schneeanteil werden der Klasse class100 zugewiesen.
5. Bilder mit Schneeanteil werden nach der Klasse class050 abgeschätzt und einsortiert.
6. Die restlichen Bilder werden in den Klassen class010, class020, class030, class040 sowie class060, class070, class080, class090 nach grober Schätzung einsortiert.

Für eine Klassifizierung in fünf Klassen wird die gleiche Anzahl an Ordner verwendet, die bestimmte Bilder aus der Vorsortierung in elf Klassen wie folgt enthalten:

- class1_uncovered ∈ {class000}
- class2_snow_remnants ∈ {class010}
- class3_snow_patches ∈ {class020, class030, class040, class050}
- class4_broken ∈ {class060, class070, class080, class090}
- class5_closed ∈ {class100}

Des Weiteren werden die Datensätze bp_snowcam (Nacht) und Wasserkuppe (Nacht) in Grauwertbilder umgewandelt, um Anforderung 2 in Tabelle 3.1 zu erfüllen. Dies wird mit Hilfe von ImageMagick und folgendem Linux-Befehl durchgeführt:

Listing 5.1: Umwandlung von Aufnahmen in Grauwertbilder

```
1 magick mogrify -colorspace Gray *  # conversion to gray values
```

Der Zeitstempel des Bilddatensatzes Wasserkuppe wird nach Listing 5.2 vollständig entfernt und von der Bildgröße (2592 × 1944) Pixel auf (640 × 480) Pixel verkleinert.

Listing 5.2: Entfernung von Zeitstempel und Reduzierung der Bildgröße

```
1 mogrify -crop 2592x1944+12+9 *.jpg # cut pixel rows/columns
2 mogrify -resize 24.8% *.jpg        # resize images
```

5.1.2 Verarbeitung

Ein Problem von gelabelten Datensätzen kann die Unausgeglichenheit der Bilder in den jeweiligen Klassen sein [More 2016, vgl.]. Würde ein Modell mit einem unausgeglichenen Datensatz trainiert werden, so könnte das Modell dominierende Klassen während der Vorhersage bevorzugen, da Daten dieser Klasse häufiger vorkommen. Eine Möglichkeit ist, mehr Daten zu sammeln, um Minderklassen an der dominierendsten Klasse auszugleichen. Als eine andere Lösung bietet sich das Down- und Upsampling an [More 2016, vgl.]. Während dem Downsampling werden dominierende Klassen unterabgetastet, um die Bilder mit den der Unterklassen gleichzusetzen. Mit Anwendung des Upsamplings werden Klassen mit wenig vorhandenen Bildern den dominierenden angepasst. Dies kann etwa durch Kopieren der vorhandenen Bilder mit anschließender Veränderung der Kopien realisiert werden, z. B. durch geringere Neigung des Bildes. Dabei wird das Upsampling an Trainingsdaten und nicht an Validierungs- und Testdaten angewendet, da dies die Genauigkeitswerte zur Bewertung eines Modells verfälscht [Filter 2020].

Vor der Aufteilung aller Bilddatensätze in Trainings-, Validierungs- und Testdaten werden stark dominierende Klassen unter- und anschließend überabgetastet. Die Aufteilung erfolgt dabei mit verschiedenen Verhältnissen, die jeweils von den vorhandenen Daten abhängig sind. Realisiert wird die Aufteilung sowie das Upsampling mit Hilfe der Library split-folders [Filter 2020]. Listing 5.3 zeigt den Codeausschnitt:

Listing 5.3: Auslesen von Bildern aus Ordnern

```
splitfolders.fixed(input=INPUT_FOLDER,
                   output=OUTPUT_FOLDER,
                   seed=1337,                  # shuffle data
                   fixed=(val_fix, test_fix), # split val/test data
                   oversample=True,            # oversample train data only
                   group_prefix=None)          # default values
```

Die in Listing 5.3 dargestellte Funktion teilt einen Datensatz mit einer festgelegten Anzahl in Validierungs- und Testdaten auf. Die übrigen Daten werden für das Training verwendet. Dabei wird die Aufteilung anhand des Parameters `seed` zufällig ausgewählt, um Anforderung 13 in Tabelle 3.1 zu erfüllen. Anschließend wird das Oversampling an allen Klassen der Trainingsdaten angewandt. Dadurch wird die Bilderanzahl der Minderklassen zur dominierendsten Klasse gleichgesetzt. Trainingsdaten befinden sich im Ordner *train*, Validierungsdaten in *val* und Testdaten in *test*. Sobald die Aufteilung erfolgt ist, wird das Upsampling auf die Trainingsdaten angewendet. In Tabelle 5.1 ist eine

Zusammenfassung der endgültigen Bilddatensätze nach der Vorverarbeitung aufgelistet. Die nähere Vorgehensweise der Aufteilung ist im Anhang F dargestellt.

Tabelle 5.1: Datensätze nach der Vorverarbeitung. Das Verhältnis ist wie folgt angegeben: Trainings-, Validierungs- und Testdaten.

Datensatz	Tag / Nacht	Klassen	Verhältnis	Anzahl Bilder	Bildgröße [Pixel]
muccam01	gemischt	5	70:15:15	4500	600 × 360
		11	70:15:15	11000	
Garten	Tag	5	80:10:10	12660	440 × 480
		11	80:10:10	2750	
bp_snowcam	Tag	5	60:20:20	15000	320 × 240
		11	70:15:15	22000	
	Nacht	5	70:15:15	7500	
		11	70:15:15	8426	
Wasserkuppe	Tag	5	70:15:15	2000	640 × 480
	Nacht		70:15:15	1000	

Die Datensätze haben unterschiedliche Aufteilungsverhältnisse von 60:20:20, 70:15:15 und 80:10:10 für Trainings-, Validierungs- und Testdaten. Den höchsten Anteil weisen Trainingsdaten auf, da ein Modell aus vielen Daten lernen soll. Jeder Bilddatensatz ist mit einer unterschiedlichen Bildgröße von minimal 240 Pixel bis maximal 640 Pixel geprägt. Somit ist Anforderung 3 in Tabelle 3.1 teilweise erfüllt. Die Anzahl reicht jeweils von 1000 Bilder bis 22000 Bilder. Die endgültige Ordnerstruktur ist Abbildung 5.1 zu entnehmen. Der Placeholder {dataset} steht dabei für den jeweiligen Datensatznamen (muccam01, garten, bp_snowcam_day, bp_snowcam_night, wasserkuppe_day, wasserkuppe_night).

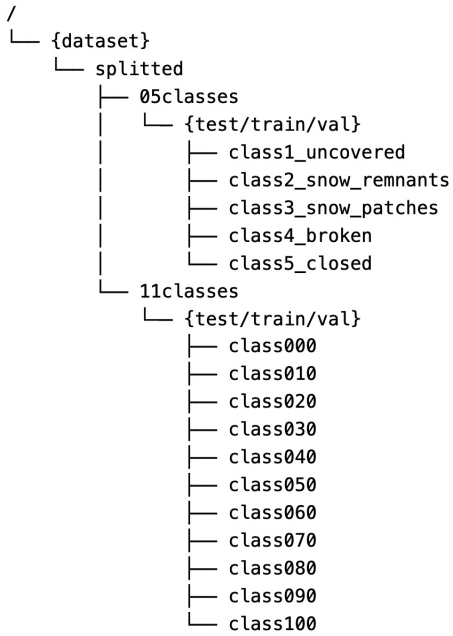


Abbildung 5.1: Ordnerstruktur der Bilddatensätze in fünf und elf Klassen

Die Funktion `image_dataset_from_directory` nimmt Bilder für die Verarbeitung auf, die in einem Ordner gespeichert sind und lädt diese im Zwischenspeicher des Programms in einem mehrdimensionalen Array. Der Codeteil aus Listing 5.4 wird jeweils auf Trainings-, Validierungs- und Testdaten angewendet.

Listing 5.4: Laden der Bilddaten von einem Ordner

```
1 train_ds = image_dataset_from_directory(DATASET_PATH + '/train',
2                                         seed=123,
3                                         image_size=(IMG_HEIGHT, IMG_WIDTH),
4                                         batch_size=BATCH_SIZE)
```

5.1.3 Augmentation

Die Datenaugmentation ist eine Technik, mit der sich Bilddaten transformieren und erweitern lassen [Géron 2018, S. 311]. Die Technik wird verwendet, um bspw. das Wackeln einer Kamera zu simulieren, weil diese während ungünstigen Wetterbedingungen unerwünschten Windkräfte ausgesetzt ist. Zudem bietet Datenaugmentation die Möglichkeit, um unterschiedliche Daten zu erzeugen. In Keras wird dazu die Funktion

`ImageDataGenerator` benutzt. Einstellmöglichkeiten für die Transformation sind z. B. Folgende: Rotation des Bildes in Grad, Spiegelung um eine Achse, Helligkeitsanpassung oder die Verarbeitungsart der entstandenen Lücken im Bild. Die eigentliche Datenaugmentation findet dabei während des Trainingsprozesses der Funktion `fit()` statt. Für jedes Bild im zusammengefassten Batch wird eine zufällige Transformation anhand von festgelegten Einstellungen durchgeführt. Anschließend werden die transformierten Bilder mit den Originalbildern ersetzt und nicht erweitert, wie der Name es vermuten lässt. Abbildung 5.2 zeigt vier zufällige Transformationen eines Bildes des Datensatzes muccam01 mit einer eingestellten Rotation von bis zu 15° in zufälliger Richtung. Nach der Drehung des Bildes entstehen Lücken, da Pixel verworfen werden. Um die gleiche Pixelanzahl des vom Eingabebildes am Ausgabebild wiederherzustellen, wird die Replication angewendet. Dabei werden an den entstandenen Lücken die Pixel der Ränder übernommen und nach dem Prinzip `aaaaaa|abcdefgh|hhhhhhh` gefüllt [Team 2015]. In Keras lautet dieser Füllalgorithmus „nearest“.

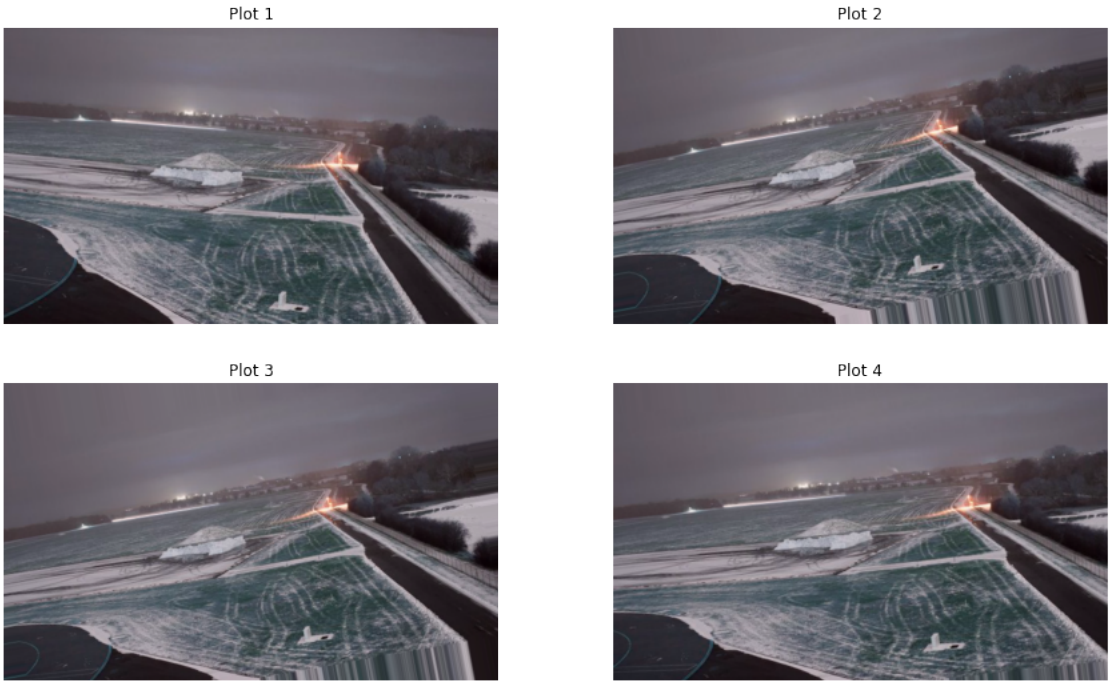


Abbildung 5.2: Anwendung der Datenaugmentation anhand eines Bildes vom Datensatz muccam01. Einstellungen: Drehung um bis zu 15°, Füllmodus „nearest"

`ImageDataGenerator` wird verwendet, um festzulegen, wie Daten während des Trainingsprozesses verarbeitet werden. Eine Vorverarbeitungfunktion, wie in Listing 5.5 dar-

gestellt, sorgt für eine Skalierung der Bilder und wird abhängig von dem dazugehörigen vortrainierten Netz bekanntgegeben.

Listing 5.5: Verwendete Funktion zur Datenaugmentation der Trainingsdaten

```
ImageDataGenerator(preprocessing_function=PREPROCESSING_FUNCTION,
                   rotation_range=4,        # rotation up to 4 degrees
                   width_shift_range=0.05,  # width shift range
                   height_shift_range=0.05, # height shift range
                   fill_mode='nearest')     # fill mode for gaps
```

5.2 Implementierung

In diesem Abschnitt wird die Implementierung der Netze in Python angewendet. Zudem wird die Protokollierung zur Speicherung der Modelle und Trainingsergebnisse vorgenommen. Anschließend wird das Modell kompiliert und trainiert.

5.2.1 Modell

Die Architektur des Modells aus Abbildung 4.6 soll umgesetzt werden. Dazu wird ein Multi-Output-Modell für die Klassifizierung in elf Klassen und zur Bestimmung des MAEs realisiert. Ein weiteres Single-Output-Modell für die Klassifizierung in fünf Klassen soll separat vorhanden sein, da der MAE für die in fünf Klassen unterteilten Datensätze nicht ermittelt wird.

Zu Beginn wird der Input-Layer laut Listing 5.6 für beide Modelle bekanntgegeben. Dieser enthält einen Übergabeparameter `shape`, der die Höhe und Breite des Bildes sowie das Farbspektrum angibt: drei Schichten für RGB-Bilder und ein Layer für Grauwertbilder. Gemischte Bilder mit unterschiedlichen Dimensionen erhalten statt der spezifischen Angabe zur Höhe und Breite den variablen Wert `None`. Für jeden der folgenden Schichten wird ein eindeutiger Name bekanntgegeben.

Listing 5.6: Input Layer

```
# input layer
input = layers.Input(shape=(IMG_HEIGHT, IMG_WIDTH, 3), name='input')
```

Nach dem Input Layer erfolgt die Anbindung des vortrainierten Netzes. Keras stellt dazu die Funktionen `VGG16()`, `Xception()` und `DenseNet201()` zur Verfügung. Diese beinhalten jeweils die Architekturen laut Abschnitt 4.2. Der Parameter `include_top` wird auf `False` gesetzt, um die letzte Schicht des vortrainierten Netzes, die für eine Klassifizierung zuständig ist, zu entfernen. In Listing 5.7 steht x für das Multi- und y für das Single-Output-Modell.

Listing 5.7: Vortrainiertes Netz

```
1 # build convnet
2 conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(
      IMG_HEIGHT, IMG_WIDTH, 3))
3 #conv_base = Xception(weights='imagenet', include_top=False, input_shape=(
      IMG_HEIGHT, IMG_WIDTH, 3))
4 #conv_base = DenseNet201(weights='imagenet', include_top=False, input_shape=(
      IMG_HEIGHT, IMG_WIDTH, 3))
5
6 x = conv_base(input) # connect input layer to convnet
7 y = conv_base(input) # connect input layer to convnet
```

An dieser Stelle wird das vortrainierte Netz eingefroren [Chollet 2018, vgl., S. 154], damit die darin enthaltenden ImageNet-Gewichtungen während des Trainings nicht überspielt werden. Dies wird wie folgt umgesetzt:

Listing 5.8: Einfrieren des vortrainierten Netzes

```
1 conv_base.trainable = False # freeze convnet
```

Für jedes Modell werden die in Unterabschnitt 4.3.1 festgelegen Hidden Layers im Anschluss an das vortrainierte Netz angebunden. Dies erfolgt der Reihe nach mit den Funktionen `Flatten()`, `Dense()`, `Dropout()` und `BatchNormalization()`. Die Dense Layers enthalten dabei nach Listing 5.9 jeweils eine Neuronenanzahl von 512 und die Dropout Layers einen Wert von 0,2.

Listing 5.9: Zusätzliche Hidden Layers

```
1 # additional hidden layers: model 1 (11 classes)
2 x = layers.Flatten(name='flatten_x')(x)
3 x = layers.Dense(512, activation='relu', name='dense_x')(x)
4 x = layers.Dropout(0.2, name='dropout_x')(x)
5 x = layers.BatchNormalization(name='batch_x')(x)
6
7 # additional hidden layers: model 2 (5 classes)
```

```
y = layers.Flatten(name='flatten_y')(y)
y = layers.Dense(512, activation='relu', name='dense_y')(y)
y = layers.Dropout(0.2, name='dropout_y')(y)
y = layers.BatchNormalization(name='batch_y')(y)
```

Zum Abschluss werden für die Ausgabe der Accuracy und MAE jeweils Dense Layers festgelegt. Dazu wird Tabelle 4.2 umgesetzt. Für eine Klassifizierung wird die Aktivierungsfunktion `softmax` und zur Ermittlung des MAEs laut Listing 5.10 keine Funktion verwendet. Alle Gewichtungen nach dem vortrainierten Netz sind für das Training freigegeben.

Listing 5.10: Output Layers

```
# output layers: model 1 (11 classes)
ACC_11 = layers.Dense(11, activation='softmax', name='ACC_11')(x)
MAE_11 = layers.Dense(1,                        name='MAE_11')(x)

# output layer: model 2 (5 classes)
ACC_05 = layers.Dense(5, activation='softmax', name='ACC_05')(y)

# initialize
model1 = Model(inputs=[input], outputs=[ACC_11, MAE_11])
model2 = Model(inputs=[input], outputs=[ACC_05])
```

5.2.2 Protokollierung

Für das Training eines Modells wird ein Callback erstellt. Dieser wird der Trainingsfunktion `train_model()` als Parameter übergeben und fasst Funktionen zusammen, die unter bestimmten Bedingungen ausgeführt werden. Dazu gehören:

- Speicherung des Modells: Mit Hilfe eines Checkpoints `ModelCheckpoint` wird ein Modell nach jeder Trainingsepoche gespeichert, wenn die Verlustfunktion der Validierung für die Metriken MAE und Accuracy niedriger sind als die der vorherigen Epoche. Die Speicherung nach der ersten Epoche erfolgt dabei immer. Das Modell wird in der Datei `model.h5` gespeichert. Diese beinhaltet die Architektur des Modells, Gewichtungen, Trainingseinstellungen und den Zustand des Optimierers.

- Speicherung des Trainingsverlaufs: Der Trainingsverlauf wird nach jeder Epoche in der Datei `history.log` im CSV-Format gespeichert oder erweitert, falls dieser vorhanden ist.
- Trainingsabbruch: Wenn sich der Wert einer Verlustfunktion (Validierung) für die Genauigkeit oder MAE nach drei Trainingsepochen nicht verringert hat, so wird das Training abgebrochen. Es ist zu erwarten, dass sich das Modell nach einer bestimmten Epoche nicht mehr verbessern lässt.

Die Modelle und Trainingsverläufe werden in folgendem Pfad gespeichert:

`logs/<pretrained network>/<dataset>/{05classes,11classes}/{ACC,MAE,HISTORY}/{model.h5,history.log}`

Die Realisierung des Callbacks erfolgt wie in Listing 5.11 dargestellt.

Listing 5.11: Callback zur Speicherung von Modell und Trainingsergebnissen

```
# set checkpoint
best_ACC = ModelCheckpoint(
                    filepath=CHECKPOINT_FILEPATH, # file path
                    monitor='val_ACC_loss', # reference to observed value
                    save_best_only=True, # save better model...
                    mode='min' # ... when value is smaller than previous one
)
# summary checkpoint + logger + early stopping when no improvements detected
callbacks = [
        CSVLogger(HISTORY_FILEPATH, # log history
                  append=True), # extend existed file, otherwise create
        EarlyStopping(monitor='val_ACC_loss', # reference value
                  patience=3), # stop training earlier
                  best_ACC
]
```

5.2.3 Kompilierung und Training

Keras bietet zudem eine Funktion MAPE (Mean Absolute Percentage Error) an, der die prozentuale Abweichung zwischen der vorhergesagten und tatsächlichen Klasse ermittelt. Dieser ist wie folgt definiert:

$$MAPE = \frac{1}{m} \sum_{i=1}^{m} \left| \frac{h(x^{(i)}) - y^{(i)}}{y^{(i)}} \cdot 100 \right| \tag{5.1}$$

Es ist zu erkennen, dass im Nenner der Term $y^{(i)}$ als tatsächliche Klasse angegeben ist. Die Funktion in Listing 5.4 indexiert während der Erfassung von Klassen die erste mit null. Folgende Gleichung zeigt die Problematik:

$$\lim_{x \to \infty} \frac{n}{x} = \infty \tag{5.2}$$

Eine Division durch null führt zu einem unendlichen Wert, daher wird MAPE nicht verwendet. Stattdessen wird die Metrik `mae` zur Ermittlung der Gesamtabweichungen für elf Klassen angewendet. Die Klassenabweichung kann hierbei in Prozent umgerechnet werden, da die in elf Klassen sortierten Datensätze bereits eine Gleichverteilung von 0 % bis 100 % darstellen.

Der folgende Code in Listing 5.12 wird zur Kompilierung des Modells verwendet. In `model.compile()` wird der Optimierer Adam, die Verlustfunktionen `categorical_crossentropy` für die Genauigkeit und `mse` für die MAE verwendet. Gleichzeitig werden die Ausgangsmetriken `accuracy` und `mae` für die Ermittlung der Werte bekanntgegeben.

Listing 5.12: Kompilierung des Modells

```
model.compile(
        optimizer=Adam(learning_rate=1e-5), # optimizer: Adam
        loss={ # initialize loss function
            'ACC_11': 'categorical_crossentropy',
            'MAE_11': 'mse'
        },
        metrics={ # initialize metric
            'ACC_11': 'accuracy',
            'MAE_11': 'mae'
        }
)
```

Nachdem das Modell kompiliert ist, wird das Training des Modells gestartet. Dazu wird die Funktion `train_model()` aufgerufen. Folgende Parameter werden dabei übergeben: das erstellte Modell, die Bildgeneratoren für das Training und der Validierung, das Callback aus Listing 5.11 und die Klassengewichtungen. Das Array `class_weights` dient zur Gewichtung für jede Klasse, der den Wert eins annimmt, da alle Bildsätze durch das Upsampling ausgeglichen sind.

Listing 5.13: Training des Modells

```
# train model
history = train_model(model, train_generator, val_generator, callbacks,
    class_weights)
```

Die erste Trainingsphase führt eine Gewichtsanpassung der Hidden und Output Layers durch. Um eine Feinabstimmung auf das Modell anzuwenden, wird dieser ein zweites Mal trainiert. Dazu wird jeweils der letzte Block von VGG16/Xception/DenseNet201 aufgetaut[1], um das Training der Gewichte für die dort enthaltenden Layers zuzulassen. In Listing 5.14 wird das vortrainierte Netz zuerst zur Gewichtsanpassung freigegeben. Anschließend wird jede Schicht des Netzes eingefroren, bis der erste Layer des letzten Blocks eines vortrainierten Netzes erreicht ist. Diese Schicht und alle darauffolgenden sind für das Training der Gewichte freigegeben. Dazu wird folgender Code implementiert:

Listing 5.14: Friere die letzten Schichten bis zur ersten des letzten Blocks vom vortrainierten Netz ein

```
conv_base.trainable = True # make pretrained net trainable
set_trainable = False

for layer in conv_base.layers: # for each layer in conv_base
    if layer.name == 'block5_conv1': # VGG16
    #if layer.name == 'block14_sepconv1': # Xception
    #if layer.name == 'conv5_block32_0_bn': # DenseNet201
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

5.3 Analyse

In diesem Abschnitt wird eine Optimierung des Trainings zur Ermittlung der Ergebnisse durchgeführt. Die Trainingsverläufe werden analysiert sowie das Transfer Learning für mehrere Datensätze angewendet. Anschließend wird eine Visualisierung anhand des Netzes VGG16 durchgeführt.

[1] Eine Gewichtsanpassung während dem Training wird gestattet.

5.3.1 Optimierung

Es stellt sich die Frage, welche Verlustfunktion zur Ermittlung der Metrik MAE optimale Ergebnisse ermittelt. Dazu wird ein Test durchgeführt, der das Modell mit Hilfe des Datensatzes Wasserkuppe (Tag) auf Basis von VGG16 trainiert. Die Resultate sind in Abbildung 5.3 dargestellt. Es ist zu erwarten, dass die Verlustfunktionen zu Beginn eine steile Steigung aufweisen und mit jeder Epoche stetig abfällt. Die Funktionen ReLU und Sigmoid erfüllen diese Voraussetzungen nicht. Anhand der lila Kurve ist zu erkennen, dass das Auslassen der Aktivierungsfunktion die optimale Wahl ist.

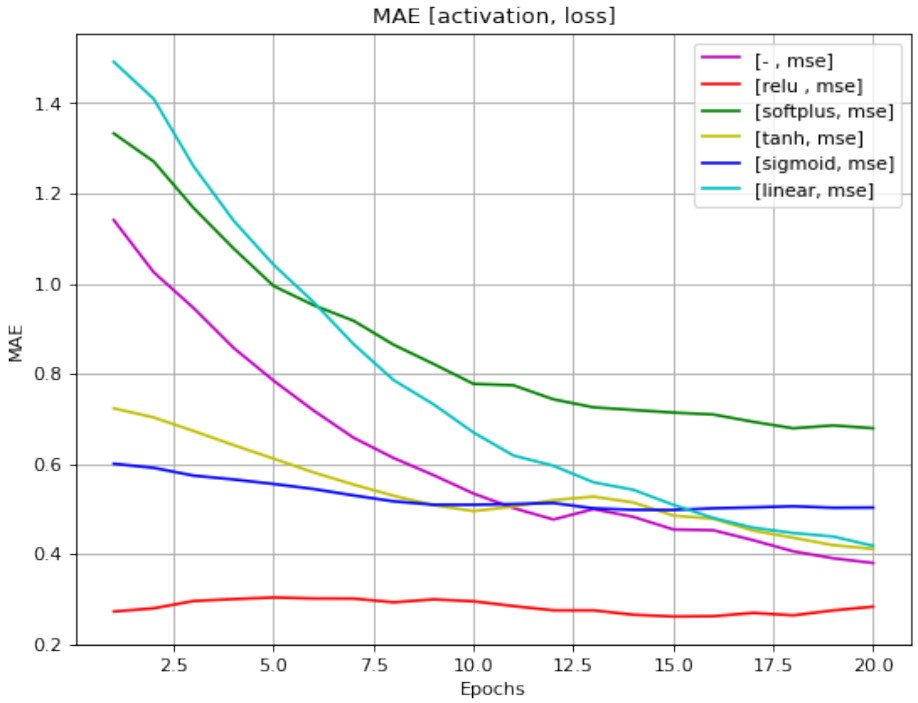


Abbildung 5.3: Verlustfunktionen auf Basis unterschiedlicher Aktivierungsfunktionen für die Trainingsphase. Verwendet wird der Datensatz *Wasserkuppe (Tag)*.

Es soll die optimale Lernrate für den Trainingsalgorithmus Adam ermittelt werden. Dazu werden unterschiedliche Lernraten anhand des Datensatzes bp_snowcam (Tag) auf Basis von VGG16 getestet. Die Ergebnisse sind der Abbildung 5.4 zu entnehmen. Hierbei zeigt sich, dass die dunkelblaue Kurve der Verlustfunktion mit einer Lernrate von $1 \cdot 10^{-5}$ am niedrigsten und somit optimal verläuft. Die nächstliegenden Lernraten $2 \cdot 10^{-5}$ und $1 \cdot 10^{-6}$ weisen schlechtere Verläufe auf.

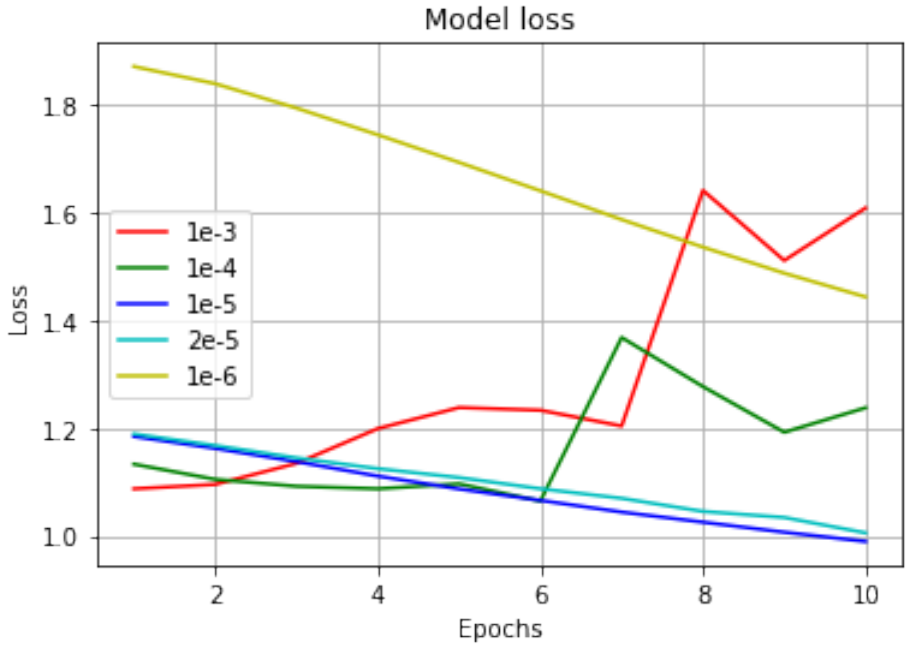


Abbildung 5.4: Verlustfunktionen auf Basis unterschiedlicher Lernraten für die Trainingsphase zur Ermittlung der Genauigkeit. Verwendet wird der Datensatz bp_snowcam (Tag).

5.3.2 Trainingsverlauf

Im folgenden Abschnitt wird der Trainingsverlauf eines Netzes auf Basis von VGG16 mit muccam01 dargestellt und analysiert. Dabei wird das Modell mit der maximalen Epochenanzahl von 20 trainiert. Alle in VGG16 enthaltene Layers sind von Epoche null bis zehn eingefroren. Ab Phase elf wird die Feinabstimmung durchgeführt. Der Kurvenverlauf ist mit einer Glättungsfunktion [Chollet 2018, vgl., S. 90] für eine optimale Darstellung überarbeitet.

Genauigkeit

Die folgenden Grafiken zeigen Kurvenverläufe der Genauigkeiten und Verlustfunktionen für jeweils fünf sowie elf Klassen. Es ist zu erkennen, dass die Validierungskurve der Verlustfunktion für fünf Klassen niedriger verläuft als die für elf Klassen. Dadurch hat sich

das Training positiver für die Klassifizierung in fünf Klassen ausgewirkt. Mit jeder Epoche sinkt die Kurvensteigung der Verlustfunktionen. In Epoche 20 ist die Steigung der Validierungskurven beinahe null und die optimale Phase ist ermittelt. Die Verlustfunktionen der Trainingskurven streben dabei gegen null, wie bereits in Unterabschnitt 2.1.5 erläutert.

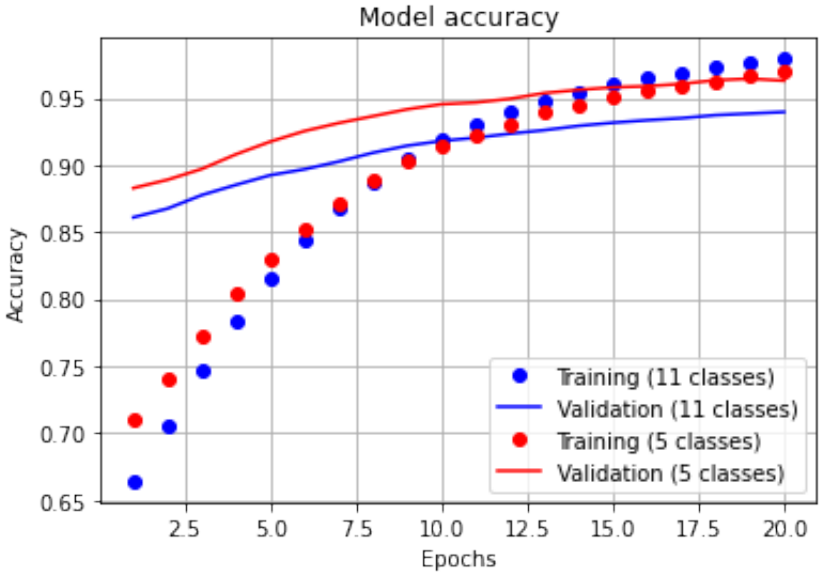


Abbildung 5.5: Genauigkeiten für Training und Validierung

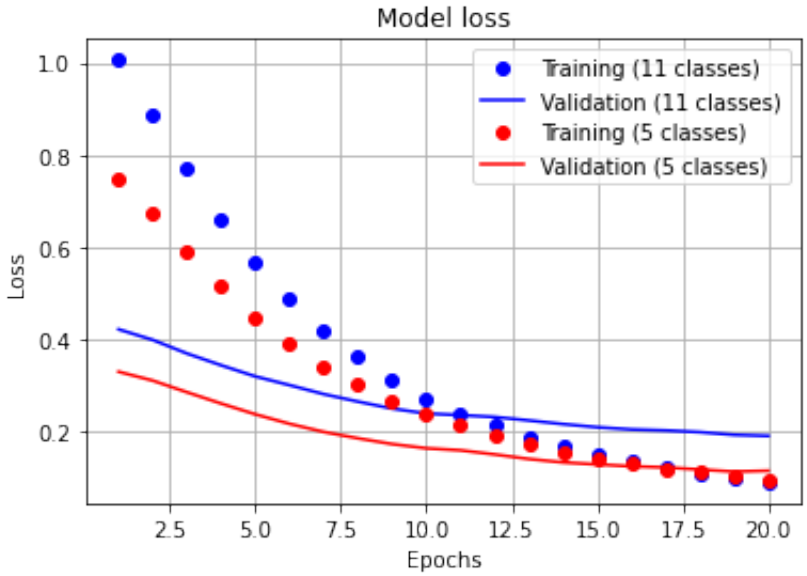


Abbildung 5.6: Verlustfunktionen für Training und Validierung

MAE

In Abbildung 5.7 sind die Verläufe des MAEs für Trainings- und Validierungsdaten dargestellt. Abbildung 5.8 zeigt die jeweiligen Verlustfunktionen. Die Verlustfunktion der Validierung erreicht mit Epoche eins den Höchststand und fällt im weiteren Verlauf stetig ab, somit wirkt sich das Training mit jeder weiteren Epoche positiv aus. Die Steigung dieser Kurve hat den Wert null mit Epoche 20 nicht erreicht, somit kann das Training fortgesetzt werden, um optimale Ergebnisse zu erzielen.

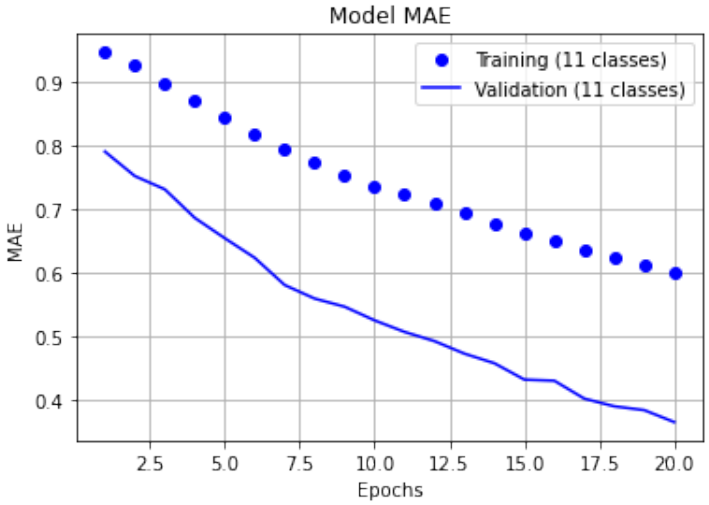


Abbildung 5.7: MAE für Training und Validierung

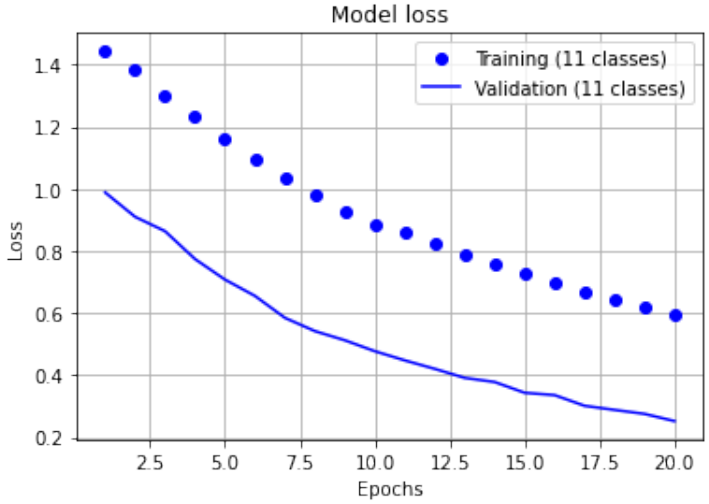


Abbildung 5.8: Verlustfunktionen für Training und Validierung

5.3.3 Transfer Learning mehrerer Datensätze

Offen bleibt die Frage, ob sich alle in Unterabschnitt 2.3.2 genannten Bilddatensätze auf ein Modell mittels Transfer Learning übertragen lassen. Hierzu werden zwei Tests durchgeführt. Zum einen wird ein Netz auf den Datensatz muccam01 und dann auf Wasserkuppe (Tag) trainiert. Zum anderen wird das Training anhand der beiden Datensätze gestartet, die zuvor zu einem Datensatz fusioniert werden. Als Testdaten dienen jeweils muccam01-Daten. Dabei wird die Genauigkeit der Klassifizierung in fünf Klassen ermittelt. Das Training erfolgt mit einer Epochenanzahl von 20 für jeden Datensatz. Abbildung 5.9 und Abbildung 5.10 zeigen den Fortschritt des ersten Tests nach dem Training.

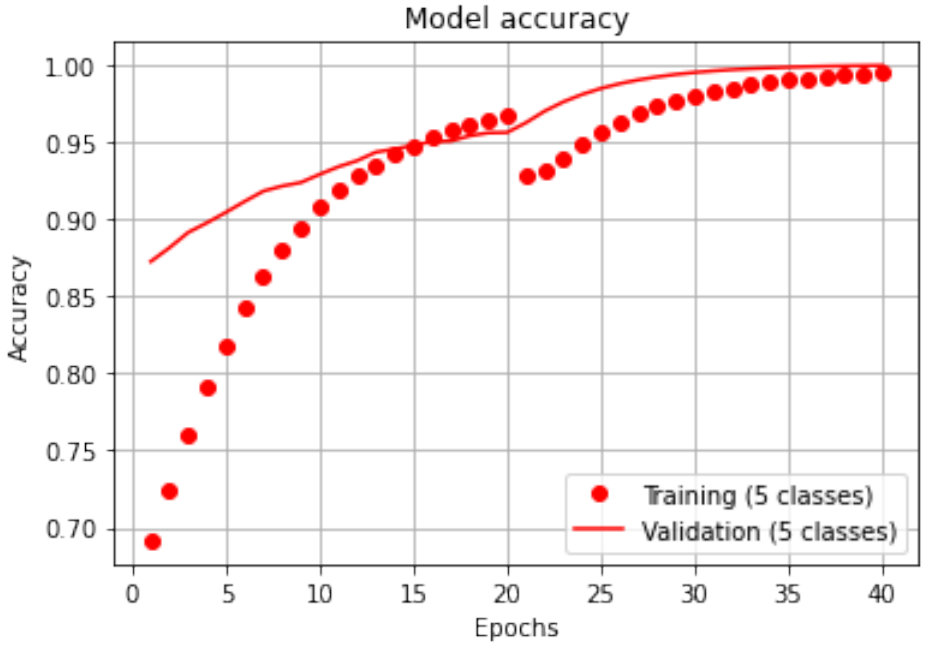


Abbildung 5.9: Trainingsverlauf des Datensatzes muccam01 bis zur 20. Epoche und Wasserkuppe (Tag) ab der 21. Epoche auf Basis von VGG16

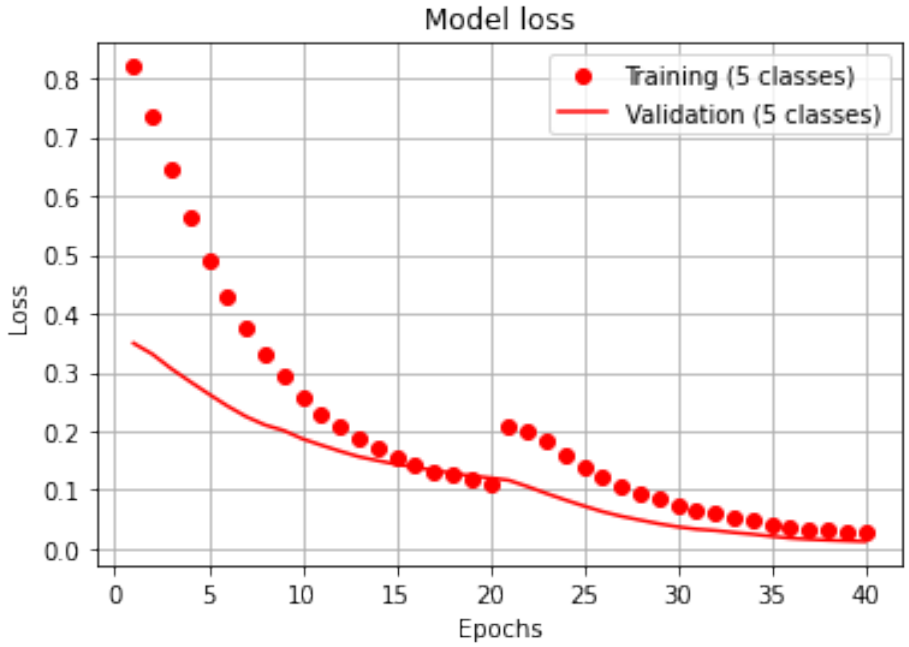


Abbildung 5.10: Trainingsverlauf des Datensatzes muccam01 bis zur 20. Epoche und Wasserkuppe (Tag) ab der 21. Epoche auf Basis von VGG16

In Abbildung 5.10 ist ein Sprung von Epoche 20 auf Epoche 21 zu erkennen, denn dort findet ein Wechsel der Datensätze für das Training statt. Die optimale Epoche in der ersten Hälfte des Trainings beträgt 20 und die der zweiten Phase 40. Die Validierungsgenauigkeit der ersten Hälfte beträgt 96 % und die der zweiten tendiert gegen 100 %.

Eine Auswertung der Testergebnisse wird in Kapitel 6 behandelt.

5.3.4 Visualisierung

In diesem Abschnitt wird eine Visualisierung durchgeführt, um herauszufinden, wie ein CNN Schnee erkennt und welche Merkmale dieser dabei extrahiert. Dazu wird als Beispiel das VGG16-Modell und ein mit Schnee bedecktes Testbild (Abbildung 5.11) des Datensatzes „Wasserkuppe (Tag)“ als Eingabe verwendet.

Abbildung 5.11: Beispielbild für die Visualisierung

Alle fünf Blöcke, aus denen das VGG16-Netz besteht, werden visualisiert. Das Bild mit drei RGB-Kanälen wird zuerst von einem 2D-Convolutional-Layer mit der Bezeichnung *block1_conv1* des VGG16-Netzes verarbeitet. Dabei entstehen während der Verarbeitung des Bildes aus Abbildung 5.11 ($640 \times 480 \times 64$) Feature Maps für den ersten Block, mit denen sich 64 Bilder darstellen lassen. Im zweiten Block verdoppelt sich die Anzahl der Feature Maps und der dritte Block enthält die vierfache Menge. Im vierten und fünften Block sind jeweils ($640 \times 480 \times 512$) Feature Maps enthalten. Aufgrund von Platzmangel

werden für jeden Block 64 Bilder gezeigt. In folgender Abbildung wird der erste von fünf dargestellt.

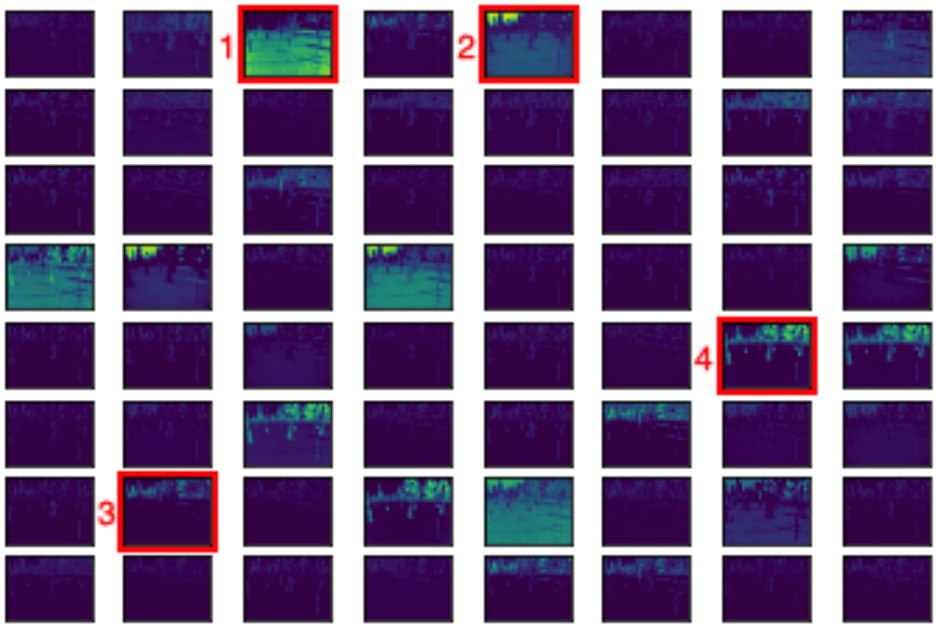


Abbildung 5.12: Visualisierung von Feature Maps des ersten Blocks (VGG16)

Im ersten Block in Abbildung 5.12 sind Feature Maps mit vielen unterschiedlichen Merkmalen zu erkennen, die das Netz versucht zu extrahieren und optisch erkennbar sind. Nachfolgend werden die in rot markierten und nummerierten Feature Maps erläutert:

- Feature Map 1: Der vollständig mit Schnee bedeckte Boden ist zu erkennen.
- Feature Map 2: Der Himmel ist stärker zu erkennen als der Rest des Bildes.
- Feature Map 3: Der Horizont ist heller dargestellt.
- Feature Map 4: Es werden Bäume und die Anlage hervorgerufen.

Ein Großteil der Feature Maps sind außerdem dunkel dargestellt. Es werden nun Block zwei bis vier abgebildet.

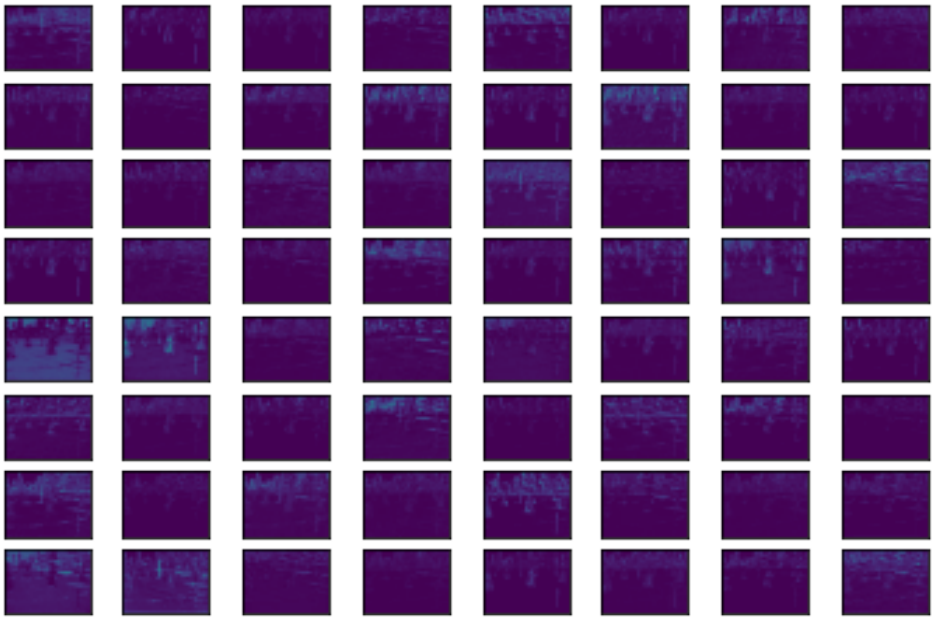


Abbildung 5.13: Visualisierung von Feature Maps des zweiten Blocks (VGG16)

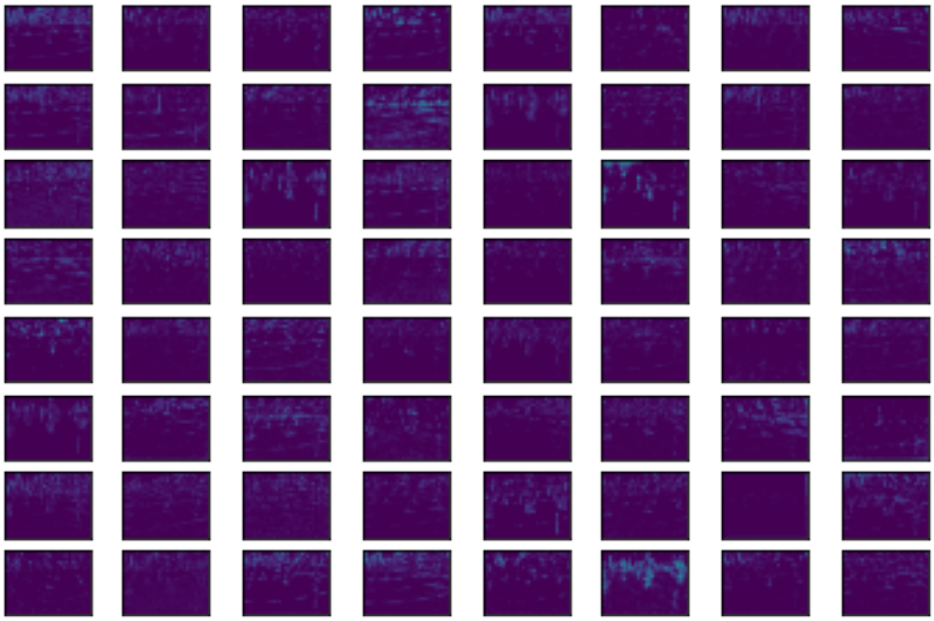


Abbildung 5.14: Visualisierung von Feature Maps des dritten Blocks (VGG16)

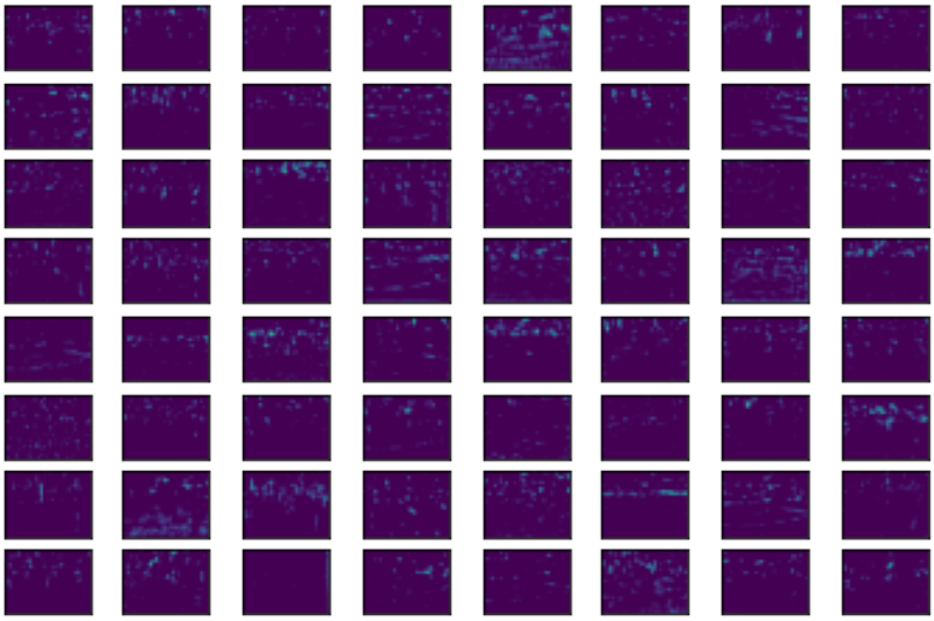


Abbildung 5.15: Visualisierung von Feature Maps des vierten Blocks (VGG16)

Im zweiten, dritten und vierten Block sind jeweils schwache Hervorhebungen verschiedener Bereiche zu erkennen. Eine spezifische Merkmalsinterpretation ist hierbei nicht festzustellen. Es folgt der fünfte und letzte Block, wie in Abbildung 5.16 dargestellt.

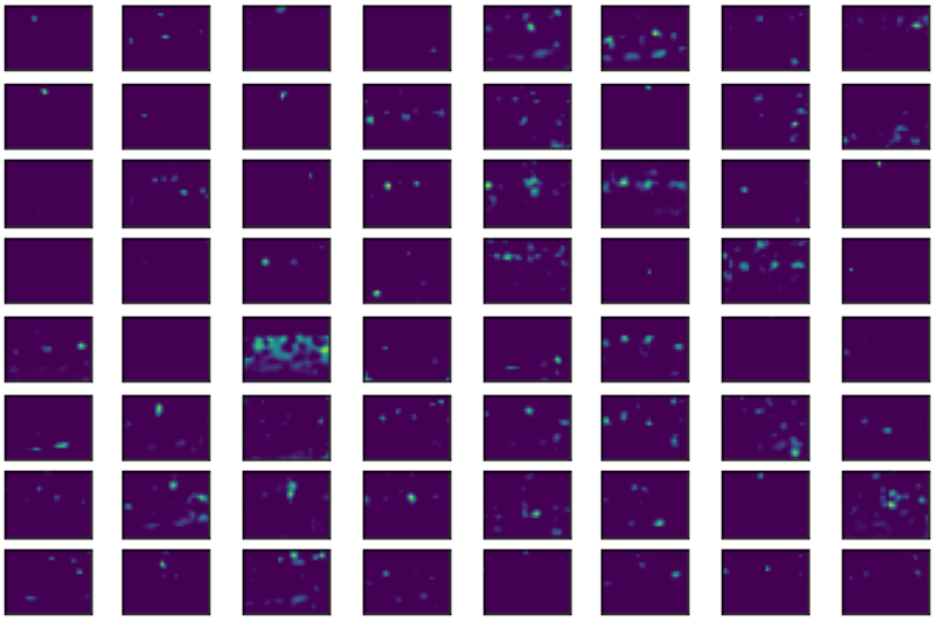


Abbildung 5.16: Visualisierung von Feature Maps des fünften Blocks (VGG16)

Im fünften und gleichzeitig letzten Block sind vereinzelte, helle Stellen deutlicher über die Feature Maps verteilt dargestellt als in den Blöcken zwei bis vier. Zusammenfassend lässt sich feststellen, dass die Merkmale mit jedem Block spezifischer und präziser ermittelt werden. Die letzten Schichten enthalten daher relevante Merkmale für eine Vorhersage.

6 Auswertung

In diesem Kapitel wird die Leistung der Modelle anhand von Testdaten aller Bilddatensätze aus Unterabschnitt 2.3.2 beurteilt. Dabei werden separate, per Transfer Learning übertragende und fusionierte Datensätze trainiert. Zudem wird die Laufzeit der Vorhersage eines Bildes ermittelt. Abschließend werden die Anforderungen in Tabelle 3.1 bewertet.

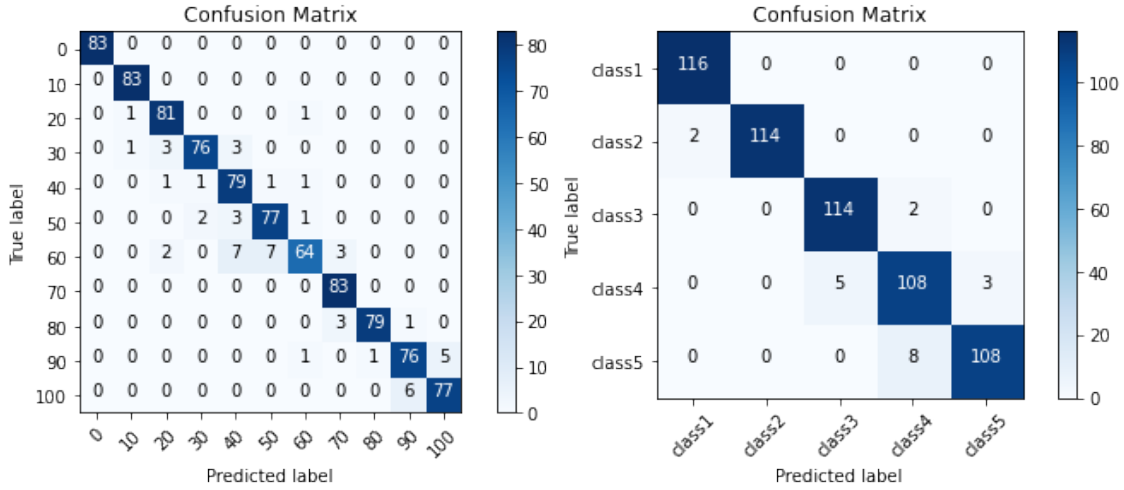
6.1 Analyse der Vorhersagen anhand eines Datensatzes

Die Funktion `predict(test_generator)` lässt sich verwenden, um die Vorhersagen der Klassen anhand von Testbildern zu bestimmen. Dabei wird ein Testgenerator als Argument übergeben. Anschließend berechnet die Funktion `accuracy_score(Y_TRUE, Y_PRED)` anhand der ermittelten Vorhersagen und den bekannten tatsächlichen Klassen die Gesamtgenauigkeit laut Gleichung 3.1. Der MAE wird mit `evaluate()` laut Gleichung 3.2 ermittelt.

Ein Netz auf Basis von VGG16 erreicht mit dem Datensatz muccam01 folgende Ergebnisse:

- Genauigkeit von 96,55 % mit der Klassifizierung in fünf Klassen.
- Genauigkeit von 93,98 % mit der Klassifizierung in elf Klassen.
- MAE von 28,62 % mit den in elf Klassen vorsortierten Daten.

Die Ergebnisse der Vorhersagen für die Klassifizierung in fünf und elf Klassen lassen sich in einer Konfusionsmatrix darstellen. Die höchste Abweichung in Abbildung 6.1a beträgt vier Klassen anhand eines Bildes zwischen der tatsächlichen Klasse class020 und der vorhergesagten Kasse class060. In Abbildung 6.1b weichen 20 Bilder von 580 Bildern um eine Klasse ab.



(a) Elf Klassen (b) Fünf Klassen

Abbildung 6.1: Konfusionsmatrix

Aus den tatsächlichen und vorhergesagten Klassen lassen sich folgende Kennzahlen ermitteln: Precision, Recall und F1-Score. Grundlage dazu sind folgende Kennzahlen: TP (True Positive), FP (False Positive) und FN (False Negative). Das True/False deutet auf die tatsächliche Klasse und Positive/Negative auf die der vorhergesagten Klassen hin. Mit folgenden Formeln [Manliguez 2016] werden die prozentualen Anteile für jede Klasse ermittelt:

$$FN_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} x_{ij} \tag{6.1}$$

$$FP_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} x_{ji} \tag{6.2}$$

$$TP_i = x_{ii} \tag{6.3}$$

dabei ist i die vorhergesagte und j die tatsächliche Klasse. n gibt die Gesamtanzahl aller Klassen an und x die jeweilige Anzahl der klassifizierten Bilder im Verhältnis zur tatsächlichen Klassifizierung.

Die Presicion gibt den Anteil der positiv klassifizierten Klassen an, die tatsächlich korrekt sind. Der Recall bestimmt den Anteil der tatsächlichen Klassen, die gefunden wurden. Der F1-Score setzt sich aus der Presicion und dem Recall zusammen, um beide dieser

Werte zu bewerten. Diese lassen sich wie folgt [Manliguez 2016] beschreiben:

$$\text{presicion}_i = \frac{TP_i}{TP_i + FP_i} \tag{6.4}$$

$$\text{recall}_i = \frac{TP_i}{TP_i + FN_i} \tag{6.5}$$

$$\text{f1-score}_i = \frac{2}{\frac{1}{presicion_i} + \frac{1}{recall_i}} \tag{6.6}$$

In Tabelle 6.1 erreicht die Klasse class000 einen F1-Score von eins. Testbilder ohne Schneeanteil werden daher vollständig erkannt. Die Klasse class060 weist den niedrigsten F1-Score von 0,85 auf.

Tabelle 6.1: Ermittlung der Precision, Recall und F1-Score für die Klassifizierung in elf Klassen

Klasse	precision	recall	f1-score	images
class000	1,00	1,00	1,00	83
class010	0,98	1,00	0,99	83
class020	0,93	0,98	0,95	83
class030	0,96	0,92	0,94	83
class040	0,86	0,95	0,90	83
class050	0,91	0,93	0,92	83
class060	0,94	0,77	0,85	83
class070	0,93	1,00	0,97	83
class080	0,99	0,95	0,97	83
class090	0,92	0,92	0,92	83
class100	0,94	0,93	0,93	83

Die Klassen class1_uncovered und class2_snow_remnants beinhalten laut Tabelle 6.2 jeweils einen F1-Score von 0,99. Klasse class4_broke weist dabei den niedrigsten F1-Score von 0,93 auf.

Tabelle 6.2: Ermittlung der Precision, Recall und F1-Score für die Klassifizierung in fünf Klassen

Klasse	precision	recall	f1-score	images
class1_uncovered	0,98	1,00	0,99	116
class2_snow_remnants	1,00	0,98	0,99	116
class3_snow_patches	0,96	0,98	0,97	116
class4_broken	0,92	0,93	0,92	116
class5_closed	0,97	0,93	0,95	116

6.2 Training mit separaten Datensätzen

Die Modelle auf Basis von VGG16, Xception und DenseNet201 werden jeweils mit separaten Datensätzen aus Unterabschnitt 2.3.2 trainiert. Die Ergebnisse anhand von Testdaten sind in den folgenden Tabellen aufgelistet:

Tabelle 6.3: Testergebnisse des Datensatzes muccam01

Netz	Klassen	Accuracy [%]	MAE [%]
VGG16	5	96,55	-
	11	93,98	28,62
Xception	5	95,52	-
	11	94,09	20,68
DenseNet201	5	95,52	-
	11	94,63	23,85

Tabelle 6.4: Testergebnisse des Datensatzes Garten

Netz	Klassen	Accuracy [%]	MAE [%]
VGG16	5	99,04	-
	11	96,10	30,29
Xception	5	98,82	-
	11	95,67	24,31
DenseNet201	5	99,14	-
	11	96,97	24,01

Tabelle 6.5: Testergebnisse des Datensatzes bp_snowcam (Tag)

Netz	Klassen	Accuracy [%]	MAE [%]
VGG16	5	71,90	-
	11	37,90	37,93
Xception	5	68,24	-
	11	37,14	26,24
DenseNet201	5	69,25	-
	11	36,66	27,38

Tabelle 6.6: Testergebnisse des Datensatzes bp_snowcam (Nacht)

Netz	Klassen	Accuracy [%]	MAE [%]
VGG16	5	64,77	-
	11	32,67	54,04
Xception	5	62,97	-
	11	27,84	33,07
DenseNet201	5	65,29	-
	11	33,95	39,31

Tabelle 6.7: Testergebnisse des Datensatzes Wasserkuppe (Tag)

Netz	Klassen	Accuracy [%]
VGG16	5	100
Xception	5	98,75
DenseNet201	5	98,75

Tabelle 6.8: Testergebnisse des Datensatzes Wasserkuppe (Nacht)

Netz	Klassen	Accuracy [%]
VGG16	5	100
Xception	5	95,71
DenseNet201	5	97,14

Es ist zu erkennen, dass das Xception-Modell anhand der Datensätze muccam01, Garten und bp_snowcam (Nach/Tag) die optimalen MAE-Ergebnisse aufweist. Das VGG16-Netz berechnet die geringsten MAE-Werte. Der niedrigste MAE von allen Datensätzen beträgt 20,68 % für muccam01-Daten. Somit ist Anforderung 10 in Tabelle 3.1 nicht erfüllt.

Alle Datensätze weisen während der Klassifizierung in fünf Klassen höhere Genauigkeiten auf als in elf Klassen. VGG16 erreicht mit Wasserkuppe (Tag/Nacht), bp_snowcam und muccam01 die höchsten Genauigkeiten. Die Datensätze Garten und bp_snowcam (Tag) funktionieren mit DenseNet201 optimal. Xception liefert die niedrigsten Genauigkeiten. Mit dem Datensatz bp_snowcam (Nacht) wurden die niedrigsten Genauigkeiten von allen erzielt. Anforderung 9 in Tabelle 3.1 ist teilweise erfüllt, da die Accuracy der Datensätze muccam01, Wasserkuppe (Tag und Nacht) und Garten jeweils mindestens 90 % beträgt.

6.3 Transfer Learning mehrerer Datensätze

In Tabelle 6.9 ist eine Auflistung der Testergebnisse dargestellt. Die Genauigkeiten haben sich für die Vorhersage der muccam01-Testdaten nach dem Transfer Learning bei allen Modellen deutlich verschlechtert. Dies lässt vermuten, dass die Modelle ihre bereits vorhandenen Gewichtungen neu antrainieren und dabei die Vorhersagefähigkeit des Datensatzes muccam01 verlieren. Dieses Phänomen hat einen eigenen Begriff und wird Catastrophic Interference (z. Dt.: Katastrophales Vergessen) [Morales u. a. 2019, vgl., S. 64 - 75] genannt.

Tabelle 6.9: Testergebnisse mit muccam01-Testdaten auf Basis des trainierten Datensatzes muccam01 und Wasserkuppe (Tag) im Anschluss

Netz	Klassen	Accuracy [%] (nach Epoche 20)	Accuracy [%] (nach Epoche 40)
VGG16	5	95,69	76,90
Xception	5	95,69	50,17
DenseNet201	5	94,83	65,69

Der zweite Test wird mit einem fusionierten Datensatz, bestehend aus muccam01 und Wasserkuppe (Tag) durchgeführt. Die Ergebnisse sind in Tabelle 6.10 aufgelistet.

Tabelle 6.10: Testergebnisse mit muccam01-Testdaten auf Basis des trainierten und fusionierten Datensatzes muccam01 und Wasserkuppe (Tag)

Netz	Klassen	Accuracy [%]
VGG16	5	95,86
Xception	5	95,00
DenseNet201	5	95,52

Die Accuracy in Tabelle 6.10 ist deutlich genauer als die durchgeführte Methode des Transfer Learnings (siehe Tabelle 6.9). Das VGG16-basierte Netz schneidet dabei optimal ab: 0,86 % genauer als Xception und um 0,34 % akkurater als DenseNet201. Daraus lässt

sich schließen, dass ein Training mit einem fusionierten Datensatz anstelle des Transfer Learnings zu bevorzugen ist, wenn ein universelles Modell für mehrere Datensätze benötigt wird.

Zum Abschluss werden alle in Unterabschnitt 2.3.2 genannten Bilddatensätze zusammengefasst und trainiert. Alle Bilder werden dabei auf die Größe (320×240) Pixel skaliert. Die Genauigkeiten der jeweiligen Modelle sind Tabelle 6.11 zu entnehmen.

Tabelle 6.11: Bewertung aller in Unterabschnitt 2.3.2 genannten Datensätze als fusionierten Datensatz

Netz	Klassen	Accuracy [%]
VGG16	5	78,76
Xception	5	77,03
DenseNet201	5	77,19

Das VGG16-basierte Modell erzielt dabei das optimale Ergebnis: 1,73 % genauer als Xception und um 1,57 % akkurater als DenseNet201.

6.4 Laufzeiten

Die Laufzeiten der akkuratesten Modelle aus Abschnitt 6.2 und die auf Basis der gemeinsamen Datensätze aus Tabelle 6.11 werden auf die Laufzeit getestet. Verwendet wird ein Computer mit dem Prozessor Intel Core i5-1038NG7 [Intel 2020]. Die CPU besitzt vier Kerne und acht Threads mit einer Grundtaktfrequenz von 2 GHz sowie einer maximalen Taktfrequenz von 3,8 GHz. In Tabelle 6.12 sind die Laufzeiten aufgelistet. Dabei wurde ein zufälliges Testbild ausgewählt, die Laufzeiten anhand der Funktion `time.process_time()` für jeweils 20 Vorhersagen ermittelt und davon der Mittelwert gebildet.

Tabelle 6.12: Laufzeiten für die Vorhersage eines Bildes

		Datensätze		
		VGG16	Xception	DenseNet201
Datensätze	muccam01	2,69 s	1,19 s	2,49 s
	Garten	2,42 s	1,21 s	2,47 s
	bp_snowcam	0,96 s	0,47 s	1,04 s
	Wasserkuppe	3,67 s	1,66 s	3,68 s

Es ist zu erkennen, dass die Laufzeiten von Xception optimal sind. Je größer die Bildgröße des Eingabebildes ist, desto länger ist die Laufzeit. Keines der Netze erreicht eine Zeit von 200 ms oder niedriger, somit ist Anforderung 11 in Tabelle 3.1 nicht erfüllt.

Alle Modelle wurden auf Grundlage von Kennzahlen bewertet, somit ist Anforderung 18 in Tabelle 3.1 erfüllt.

6.5 Bewertung

Zum Abschluss dieses Kapitels werden die Anforderungen aus Tabelle 3.1 bewertet. Dabei können diese als erfüllt, nicht erfüllt oder teilweise erfüllt markiert sein. In Tabelle 6.13 befindet sich eine Zusammenfassung.

Tabelle 6.13: Bewertung der Anforderungen von Tabelle 3.1

ID	Bemerkung	Erfüllt?
1	Alle Tagbilder sind RGB-Farbbilder.	Ja
2	Alle Nachtbilder sind Graubilder, ausgenommen die des Datensatzes muccam01.	Teilweise
3	Der Datensatz Wasserkuppe besitzt als einziger eine Auflösung von (640×480) Pixel.	Teilweise
4	Der Schneebedeckungsgrad ist für Datensätze mit einer Vorsortierung in elf Klassen ermittelt.	Ja
5	Der Schneebedeckungsgrad aus fünf Klassen für jeden Datensatz ist ermittelt.	Ja
6	Eine Prozentangabe des Schneebedeckungsgrades ist ermittelt.	Ja
7	Ein Modell ermittelt die Accuracy.	Ja
8	Ein Modell ermittelt die MAE aus fünf Klassen.	Ja
9	Mit einigen Datensätzen wurde eine Genauigkeit von über 90 % erreicht.	Teilweise
10	Der niedrigste ermittelte MAE beträgt 20,68 %. Ein MAE von 10 % oder niedriger wurde nicht erreicht.	Nein
11	Die geringste Laufzeit beträgt 470 ms. Eine Laufzeit von 200 ms oder niedriger wurde nicht erreicht.	Nein
12	Alle Bilder wurden während der Vorverarbeitung in zufälliger Reihenfolge berücksichtigt.	Ja
13	Alle Ausgabelayer der Modelle wurden entsprechend der Aufgabenstellung angepasst.	Ja
14	Ein Multi-Output-Modell für Daten mit einer Vorsortierung in elf Klassen wurde realisiert.	Ja
15	Die Lösung ist in Jupyter Notebook realisiert.	Ja
16	Die verwendete Software und Bibliotheken sind Open Source.	Ja
17	Es werden TensorFlow und Keras verwendet.	Ja
18	Alle Modelle sind auf Basis der Kennzahlen Genauigkeit und MAE bewertet.	Ja
19	VGG16/Xception/DenseNet201 sind in Keras verfügbar.	Ja

7 Zusammenfassung und Ausblick

In dieser Arbeit wurden Modelle auf Basis von CNNs zur Erkennung des Schneebedeckungsgrades erfolgreich entwickelt und umgesetzt.

Es werden die vortrainierten Netze VGG16, Xception und DenseNet201 als Grundlage verwendet. Diese enthalten jeweils ImageNet-Gewichtungen, da sich gezeigt hat, dass dadurch optimale Ergebnisse zu erzielen sind. Anhand der vier Datensätze muccam01, bp_snowcam, Wasserkuppe und Garten wurden Modelle trainiert und bewertet. Es sind Genauigkeiten von mindestens 96,55 % für drei Datensätze ermittelt worden. Für den Datensatz bp_snowcam mit unterschiedlich und vielseitig ausgeprägten Bildern ist eine Höchstgenauigkeit von 71,90 % mit VGG16 erreicht. Eine Klassifizierung aus fünf Klassen erzielte höhere Genauigkeiten als aus elf Klassen. Ein Transfer Learning mehrerer Datensätze nacheinander auf ein Modell hat gezeigt, dass dies nicht gut funktioniert. Die zuvor trainierten Gewichtungen eines Datensatzes werden während des nächsten Trainings für einen anderen Datensatz überspielt und somit vergessen. Als Lösung dazu liefert ein Training anhand eines gemeinsamen Datensatzes, bestehend aus einer Fusionierung der gewünschten Datensätze, deutlich bessere Ergebnisse. Die Ermittlung des optimalen MAEs liefert einen Wert von 20,68 % und verfehlt den Richtwert von 10 %. Xception konnte mit keinem der vier Datensätze den höchsten Genauigkeitswert ermitteln, jedoch lieferte dieser mit drei Datensätzen den optimalen MAE-Wert. VGG16 konnte mit Wasserkuppe während Tag und Nacht jeweils eine Genauigkeit von 100 % erreichen. DenseNet201 konnte anhand des Datensatzes Garten die besten Ergebnisse in der Klassifizierung aus fünf und elf Klassen sowie die Ermittlung des MAEs erzielen. Mit einer Fusionierung aller Datensätze konnte VGG16 die beste Genauigkeit von 78,76 % erzielen. Die trainierten Modelle können verwendet werden, um den Schneebedeckungsgrad vorherzusagen. Je höher die Auflösung eines Bildes ist, desto länger dauert die Vorhersage. Xception konnte zu jedem Datensatz die schnellsten Vorhersagen ermitteln. Die Vorhersagezeit hält sich auf Basis eines Intel-i5-Prozessors mit bis zu 3,68 s im Rahmen. Mit Hilfe einer leistungsstärkeren CPU oder einer zusätzlichen GPU kann die Vorhersagezeit verringert werden. Im

Durchschnitt ist VGG16 zur Ermittlung der Genauigkeit des Schneebedeckungsgrades die optimale Wahl.

Für zukünftige Arbeiten zu dem Thema können weitere Netze auf die Genauigkeit und MAE überprüft werden. Das Problem des Vergessens beim Transfer Learning kann näher untersucht werden. Zur weiteren Optimierung der Vorhersagen können Metadaten mit berücksichtigt werden. Der Einsatz von Sensoren für einen Standort in Kombination mit CNNs ist ebenfalls denkbar.

Literaturverzeichnis

[Ajayi und Wang 2019] Ajayi, G. O. ; Wang, Z.: *Multi-Class Weather Classification from Still Image Using Said Ensemble Method*. IEEE. 2019. – URL https://ieeexplore.ieee.org/document/8704783

[Bund 1952] Bund: *Deutscher Wetterdienst*. 1952. – URL https://www.dwd.de. – Zugriffsdatum: 05.06.2021

[Chaari 2021] Chaari, R.: *Kamerabasierte Messung des Schneebedeckungsgrades*. Studienarbeit. 2021

[Chollet 2015] Chollet, F.: *Keras*. 2015. – URL https://keras.io/api/applications. – Zugriffsdatum: 05.07.2021

[Chollet 2017] Chollet, F.: *Xception: Deep Learning with Depthwise Separable Convolutions*. arXiv. 2017. – URL https://arxiv.org/abs/1610.02357

[Chollet 2018] Chollet, F.: *Deep Learning with Python*. Manning Publications, 2018 (9781617294433)

[Chu u. a. 2016] Chu, W.-T. ; Zheng, X.-Y. ; Ding, D.-S.: *Image2Weather: A Large-Scale Image Dataset for Weather Property Estimation*. IEEE. 2016. – URL https://ieeexplore.ieee.org/document/7545010

[Cunningham und Delany 2020] Cunningham, P. ; Delany, S. J.: *k-Nearest Neighbour Classifiers: 2nd Edition (with Python examples)*. arXiv. 2020. – URL https://arxiv.org/abs/2004.04523

[Dheeraj u. a. 2021] Dheeraj, J. C. ; Nandakumar, K. ; Aditya, A. V. ; Chethan, B. S. ; Kartheek, G. C. R.: *Detecting Deepfakes Using Deep Learning*. IEEE. 2021. – URL https://ieeexplore.ieee.org/document/7545010

[Dias u. a. 2018] Dias, C. A. ; Bueno, J. C. S. ; Borges, E. N. ; Botelho, S. S. C. ; Dimuro, G. P. ; Lucca, G. ; Fernandéz, J. ; Bustince, H. ; Junior, P. L. J. D.: *Using the Choquet Integral in the Pooling Layer in Deep Learning Networks*. Springer, 2018

[Dick 2019] Dick, S.: Artificial Intelligence. URL https://hdsr.mitpress.mit.edu/pub/0aytgrau/release/2, 2019. – Forschungsbericht

[Elhoseiny u. a. 2015] Elhoseiny, M. ; Huang, S. ; Elgammal, A.: *WEATHER CLASSIFICATION WITH DEEP CONVOLUTIONAL NEURAL NETWORKS*. 2015

[Everest u. a. 2021] Everest, T. ; Jodi, D. ; Pérez Duarte, A. M. ; Putra, K. ; Tödt, S.: *Snow Coverage Detection through Deep Learning*. Bachelorprojekt. 2021

[Fei-Fei 2009] Fei-Fei, L.: *ImageNet*. 2009. – URL https://image-net.org. – Zugriffsdatum: 04.12.2021

[Filter 2020] Filter, J.: *split-folders 0.4.3*. 2020. – URL https://pypi.org/project/split-folders. – Zugriffsdatum: 03.07.2021

[Frosst und Hinton 2017] Frosst, N. ; Hinton, G.: *Distilling a Neural Network Into a Soft Decision Tree*. arXiv. 2017. – URL https://arxiv.org/abs/1711.09784

[Géron 2018] Géron, A.: *Praxiseinstieg - Machine Learning mit Scikit-Learn and TensorFlow*. 1. Auflage. O'REILLY, 2018

[GitHub 2019a] GitHub: *Google Images Download*. 2019. – URL https://github.com/hardikvasa/google-images-download. – Zugriffsdatum: 25.07.2021

[GitHub 2019b] GitHub: *Mask R-CNN for Object Detection and Segmentation*. 2019. – URL https://github.com/matterport/Mask_RCNN. – Zugriffsdatum: 25.07.2021

[Gu u. a. 2017] Gu, J. ; Wang, Z. ; Kuen, J. ; Ma, L. ; Shahroudy, A. ; Shuai, B. ; Liu, T. ; Wang, X. ; Wang, G. ; Cai, J. ; Chen, T.: *Pattern Recognition*, Elsevier Ltd, 2017

[Hearst u. a. 1998] Hearst, M.A. ; Dumais, S.T. ; Osuna, E. ; Platt, J. ; Scholkopf, B.: *Support vector machines*. IEEE. 1998. – URL https://ieeexplore.ieee.org/document/708428

[Huang u. a. 2018] Huang, G. ; Liu, Z. ; Maaten, L. van der ; Weinberger, K. Q.: *Densely Connected Convolutional Networks*. arXiv. 2018. – URL https://arxiv.org/abs/1608.06993

[Ibrahim u. a. 2019] Ibrahim, M. R. ; Haworth, J. ; Cheng, T.: WeatherNet: Recognising Weather and Visual Conditions from Street-Level Images Using Deep Residual Learning. In: *ISPRS* (2019), S. 18

[Intel 2020] Intel: *CPU Intel Core i5-1038NG7*. 2020. – URL https://ark.intel.com/content/www/de/de/ark/products/196594/intel-core-i5-1038ng7-processor-6m-cache-up-to-3-80-ghz.html. – Zugriffsdatum: 07.12.2021

[Ioffe und Szegedy 2015] Ioffe, S. ; Szegedy, C.: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv. 2015. – URL https://arxiv.org/abs/1502.03167

[Kingma und Ba 2015] Kingma, D. P. ; Ba, J. L.: *Adam: A Method for Stochastic Optimization*. arXiv. 2015. – URL https://arxiv.org/abs/1412.6980

[Koushik 2016] Koushik, J.: *Understanding Convolutional Neural Networks*. arXiv. 2016. – URL https://arxiv.org/abs/1605.09081

[Krizhevsky u. a. 2012] Krizhevsky, A. ; Sutskever, I. ; Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *NeurIPS* (2012)

[Kutylowski 2017] Kutylowski, J.: *DeepL*. 2017. – URL https://www.deepl.com. – Zugriffsdatum: 04.12.2021

[LeCun u. a. 1989] LeCun, Y. ; Boser, B. ; Denker, J. S. ; Henderson, D. ; Howard, R. E. ; Hubbard, W. ; Jackel, L. D.: *Backpropagation applied to handwritten zip code recognition*, Massachisetts Institute of Technology, 1989

[LLC 1990] LLC, ImageMagick S.: *ImageMagick*. 1990. – URL https://imagemagick.org. – Zugriffsdatum: 07.07.2021

[Lu u. a. 2014] Lu, C. ; Lin, D. ; Jia, J. ; Tang, C.-K.: *Two-Class Weather Classification*. IEEE. 2014. – URL https://ieeexplore.ieee.org/document/6909870

[Lu u. a. 2016] Lu, C. ; Lin, D. ; Jia, J. ; Tang, C.-K.: *Two-Class Weather Classification*. IEEE. 2016. – URL https://ieeexplore.ieee.org/document/7784804

[Manliguez 2016] Manliguez, C.: *Generalized Confusion Matrix for Multiple Classes*. ResearchGate. 2016. – URL http://dx.doi.org/10.13140/RG.2.2.31150.51523

[Montavon u. a. 2012] Montavon, G. ; Orr, G. B. ; Müller, K.-R.: *Neural Networks: Tricks of the Trade*. Springer Link, 2012

[Morales u. a. 2019] Morales, A. ; Fierrez, J. ; Sánchez, J. S. ; Ribeiro, B.: *Pattern Recognition and Image Analysis*, Springer, 2019. – URL https://doi.org/10.1007/978-3-030-31321-0

[More 2016] More, A.: *Survey of resampling techniques for improving classification performance in unbalanced datasets*. arXiv. 2016. – URL https://arxiv.org/abs/1608.06048

[Naqa u. a. 2015] Naqa, I. E. ; Li, R. ; Murphy, M. J.: *Machine Learning in Radiation Oncology*. Springer, 2015 (S. 3)

[Nguyen u. a. 2021] Nguyen, T. T. ; Nguyen, Q. V. H. ; Nguyen, C. M. ; Nguyen, D. ; Nguyen, D. T. ; Nahavandi, S.: *Deep Learning for Deepfakes Creation and Detection: A Survey*. arXiv. 2021. – URL https://arxiv.org/abs/1909.11573

[Pérez und Granger 2015] Pérez, F. ; Granger, B.: *Project Jupyter*. 2015. – URL https://jupyter.org. – Zugriffsdatum: 07.07.2021

[Simonyan und Zisserman 2015] Simonyan, K. ; Zisserman, A.: *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv. 2015. – URL https://arxiv.org/abs/1409.1556

[Srivastava u. a. 2014] Srivastava, N. ; Hinton, G. ; Krizhevsky, A. ; Sutskever, I. ; Salakhutdinov, R.: *Dropout: A simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research 15, 2014

[Team 2015] Team, Google B.: *TensorFlow*. 2015. – URL https://www.tensorflow.org. – Zugriffsdatum: 28.05.2021

[Wang 2019] Wang, Phillip: *This Person Does Not Exist*. 2019. – URL https://thispersondoesnotexist.com. – Zugriffsdatum: 02.12.2021

[Yosinski u. a. 2014] Yosinski, J. ; Clune, J. ; Bengio, Y. ; Lipson, H.: *How transferable are features in deep neural networks?* arXiv. 2014. – URL https://arxiv.org/abs/1411.1792

[Zhan u. a. 2017] Zhan, Y. ; Wang, J. ; et.al.: *Distinguishing Cloud and Snow in Satellite Images via Deep Convolutional Network*. IEEE. 2017. – URL https://ieeexplore.ieee.org/document/8013916

[Zhang u. a. 2016] Zhang, Z. ; Ma, H. ; Fu, H. ; Zhang, C.: Scene-free multi-class weather classification on single images. In: *ELSEVIER* (2016), S. 9

[Zhao u. a. 2019] Zhao, B. ; Li, X. ; Lu, X. ; Wang, Z.: *A CNN-RNN Architecture for Multi-Label Weather Recognition*. arXiv. 2019. – URL https://arxiv.org/abs/1904.10709

[Zhao u. a. 2017] Zhao, X. ; Wu, Y. ; Song, G. ; Li, Z. ; Zhang, Y. ; Fan, Y.: A deep learning model integrating FCNNs and CRFs for brain tumor segmentation. URL https://doi.org/10.1016/j.media.2017.10.002, 2017. – Forschungsbericht

Anhang

Anhang A: Thesis (PDF-Datei)
Anhang B: Bilddatensätze (verarbeitet)
Anhang C: Python-Code (Jupyter Notebook-Dateien)
Anhang D: Ergebnisse (Jupyter Notebook-Dateien)
Anhang E: Modell und Ergebnisse des VGG16-Modells auf Basis von muccam01
Anhang F: Vorverarbeitung der Bilddatensätze

Die Anhänge A bis E sind in elektronischer Form auf einer DVD abgelegt und können bei Prof. Dr.-Ing. Dipl.-Kfm. Jörg Dahlkemper oder Prof. Dr.-Ing. Andreas Meisel eingesehen werden.

Schritt 1/3: Datensätze (Original)

	Garten	muccam01	bp_snowcam (Tag)	bp_snowcam (Nacht)	Wasserkuppe (Tag)	Wasserkuppe (Nacht)
class000	6396	5227	8313	6429	-	-
class010	493	5448	1331	368	-	-
class020	51	814	884	296	-	-
class030	59	405	715	250	-	-
class040	364	568	669	221	-	-
class050	942	266	596	199	-	-
class060	676	266	689	218	-	-
class070	346	782	1033	440	-	-
class080	649	388	1473	631	-	-
class090	860	311	2106	772	-	-
class100	2381	266	2908	833	-	-
Summe	**13217**	**14741**	**20717**	**8207**	-	-
					-	-

class1_uncovered	6396	5227	8313	6429	739	687
class2_snow_remnants	493	5448	1331	368	49	35
class3_snow_patches	1416	2053	2864	966	52	59
class4_broken	2531	1747	5301	2061	214	197
class5_closed	2381	266	2908	833	485	542
Summe	**13217**	**14741**	**20717**	**10657**	**1539**	**1520**

Dominierende Klasse | Minderwertige Klasse

Schritt 2/3: Datensätze (Undersampling)

	Garten	muccam01	bp_snowcam (Tag)	bp_snowcam (Nacht)	Wasserkuppe (Tag)	Wasserkuppe (Nacht)
class000	250	1000	2000	766	-	-
class010	250	1000	1331	351	-	-
class020	51	814	884	275	-	-
class030	59	405	715	217	-	-
class040	250	568	669	202	-	-
class050	250	266	596	172	-	-
class060	250	266	689	181	-	-
class070	250	782	1033	413	-	-
class080	250	388	1473	600	-	-
class090	250	311	2000	719	-	-
class100	250	266	2000	766	-	-
Summe	**2360**	**6066**	**13390**	**4662**	-	-
train/val/test splitt	*80/10/10*	*70/15/15*	*70/15/15*	*70/15/15*	-	-

class1_uncovered	2532	900	3000	1500	400	200
class2_snow_remnants	493	900	1331	368	49	35
class3_snow_patches	1416	900	2384	966	52	59
class4_broken	2532	900	2755	1500	214	197
class5_closed	2381	266	2908	833	400	200
Summe	**9354**	**3866**	**12378**	**5167**	**1115**	**691**
train/val/test splitt	*80/10/10*	*70/15/15*	*60/20/20*	*70/15/15*	*80/10/10*	*80/10/10*

Downsampling | Minderwertige Klasse

Schritt 3/3: Datensätze (Upsampling)

	Garten	muccam01	bp_snowcam (Tag)	bp_snowcam (Nacht)	Wasserkuppe (Tag)	Wasserkuppe (Nacht)
class000	250	1000	2000	766	-	-
class010	250	1000	2000	766	-	-
class020	250	1000	2000	766	-	-
class030	250	1000	2000	766	-	-
class040	250	1000	2000	766	-	-
class050	250	1000	2000	766	-	-
class060	250	1000	2000	766	-	-
class070	250	1000	2000	766	-	-
class080	250	1000	2000	766	-	-
class090	250	1000	2000	766	-	-
class100	250	1000	2000	766	-	-
Summe	**2750**	**11000**	**22000**	**8426**	-	-
class1_uncovered	2532	900	3000	1500	400	200
class2_snow_remnants	2532	900	3000	1500	400	200
class3_snow_patches	2532	900	3000	1500	400	200
class4_broken	2532	900	3000	1500	400	200
class5_closed	2532	900	3000	1500	400	200
Summe	**12660**	**4500**	**15000**	**7500**	**2000**	**1000**

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original