Erheben einer Verkehrsstatistik durch Klassifizierung von Verkehrsobjekten unter Verwendung eines neuronalen Netzes

Studienarbeit - T3201

des Studiengangs Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Florian Drinkler, Luca Stanger

11. Juni 2021

Bearbeitungszeitraum	06.10.2020 - 11.06.2021
Matrikelnummer, Kurs	6653948, 7474265, TINF-18B
Ausbildungsfirma	Balluff GmbH, camos GmbH, Stuttgart
Betreuer	Sebastian Trost, Telefónica Germany

Sperrvermerk

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung der Ausbildungsstätte vorliegt.

Stuttgart, 11. Juni 2021

Florian Drinkler, Luca Stanger

Erklärung

Wir versicheren hiermit, dass wir unsere Studienarbeit - T3201 mit dem Thema: *Erheben einer Verkehrsstatistik durch Klassifizierung von Verkehrsobjekten unter Verwendung eines neuronalen Netzes* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versicheren zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Stuttgart, 11. Juni 2021

Florian Drinkler, Luca Stanger

Abstract

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
CIFAR	Canadian Institute For Advanced Research
COCO	Common Objects in Context
CNN	Convolutional Neural Network
FLOP	Floating Point Operation Per Second
GPU	Graphical Processing Unit
HTML	Hypertext Markup Language
IOT	Internet of Things
MAP	Mean Average Precision
MOG	Mixture of Gaussians
MSE	Mean Squared Error
PDF	Portable Document Format
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
SVM	Support Vector Machine
TPU	Tensor Processing Unit
OvR	One-vs.-Rest
OvO	One-vs.-One
XLA	Accelerated Linear Algebra

Abbildungsverzeichnis

Listings

1 Einleitung

In der heutigen Zeit werden Algorithmen immer häufiger eingesetzt, um verschiedenen Personengruppen die Auswertung von Daten leichter zu machen. Maschinelle Lernverfahren bieten mit der Zeit immer fortgeschrittenere Möglichkeiten, unterschiedlichste Alltagssituationen zu analysieren. Im Zusammenhang mit dem Thema Verkehrsanalyse bietet das maschinelle Lernen vielfältige Möglichkeiten, den öffentlichen Raum zu verbessern. Hauptbestandteil dieser Arbeit soll es sein, eine Verkehrsstatistik unter Verwendung eines Klassifikationsalgorithmus zu erstellen. Anleitend hierzu wird ein eigenes Neuronales Netz trainiert, um dem Leser die Grundlagen des Themas näher zu bringen. Die daraus gewonnenen Erkenntnisse werden in dieser Arbeit wiedergegeben. Aus den resultierenden Ergebnissen wird eine einfache Ableitung über das Verkehrsaufkommen erreicht.

1.1 Motivation und Problemstellung

1.2 Zielsetzung

1.3 Methodik und Aufbau der Arbeit

2 Grundlagen und Stand der Forschung

2.1 Implementierungsumgebung Jupyter

Jupyter Notebooks ist eine von der non-profit Organisation Project Jupyter entwickelte Open-Source Lösung zur interaktiven Arbeit mit Dutzenden Programmiersprachen [Jup21b]. Der Name Jupyter leitet sich dabei von den drei primären Programmiersprachen Julia, Python und R ab. Jupyter Notebooks ist sprachunabhängig und unterstützt, unter Verwendung des IPython Kernel, die Programmiersprachen Julia, R, Haskell, Ruby und Python [Jup21a]. Darüber hinaus werden unterschiedlichste Export Möglichkeiten wie Hypertext Markup Language (HTML), Portable Document Format (PDF) und LaTeX unterstützt. Die in diesem Projekt verwendete Variante von Jupyter Notebooks ist Google Colab, eine speziell für die Python-Entwicklung entworfene Umgebung. Colab Notebooks führen Code auf Cloud-Servern aus und bieten somit unabhängige Vorteile gehosteter Hardware, wie Graphical Processing Units (GPUs) und Tensor Processing Units (TPUs) [Col21].

2.2 Maschinelle Lernverfahren

Maschinelle Lernverfahren lassen sich in drei Bereiche unterteilen. Sejnowski beschreibt in seinem Buch *Unsupervised Learning - Foundations of Neural Computation* das Unüberwachte Lernen *(unsupervised learning)* als maschinelles Lernverfahren, das ohne zuvor bekannte Werte oder Belohnungen, Abweichungen vom strukturlosen Rauschen erkennt [Sej99]. Ferner wird von Duda et al. die automatische Segmentierung (Clustering) und die Komprimierung von Daten zur Dimensionsreduktion erwähnt, die zum fortwährenden Erfolg der Lernmethode beitragen [DH+73, S. 51 f.; CC08, S. 51 f.]. Als eine weitere maschinelle Lernmethode führt Cord et al. das Überwachte Lernen *(supervised learning)* an. Das Überwachte Lernen beinhaltet das Lernen einer Abbildung zwischen einem Satz von Eingangsvariablen X und einer Ausgangsvariablen Y, sowie die Anwendung dieser Abbildung zur Vorhersage der Ausgabe für ungesehene Daten [CC08, S. 21 ff.]. Cord et al. nennt 2008 als verbreitestes Modell die Support Vector Machine (SVM), die ihre Stärken besonders in der Verarbeitung von Multimedialen Daten besitzt. Im Paradigma

des überwachten Lernens besteht das Ziel darin, eine Funktion $f : X \longrightarrow Y$ aus einem Beispiel- oder Trainingssatz A_n abzuleiten [CC08, S. 22]. Sei hierzu $x_i \in X$ und $y_i \in Y$:

$$A_n = ((x_1, y_1), \ldots, (x_n, y_n)) \in (X \times Y)^n. \tag{2.1}$$

Ein weiterer Fundamentaler Bestandteil des überwachten Lernens ist der Begriff des Verlusts zur Messung der Übereinstimmung zwischen der Vorhersage $f(x)$ und der gewünschten Ausgabe y. Hierfür wird eine Verlustfunktion $L : Y \times Y \longrightarrow \mathbb{R}^+$ zur Evaluierung des Fehlers eingeführt. Die Wahl der Verlustfunktion $L(f(x), y)$ hängt von dem zu lösenden Lernproblem ab [CC08, S. 22].

Zu unterscheiden sind hierbei drei verschiedenen Arten - *Binäre*, *Multi-Class* sowie *Multi-Label* Klassifikation. Für erstere ist es das Ziel, die Ausgabe eines messbaren zufälligen Klassifikators, parametrisiert durch $f : X \to [0, 1]$, der zwischen positiven ($Y = 1$) und negativen ($Y = 0$) Instanzen unterscheidet, zu erzeugen [MW18, S. 2 f.]. Ein zufälliger Klassifikator sagt jedes $x \in X$ mit der Wahrscheinlichkeit $f(x)$ als positiv voraus; die Qualität eines solchen Klassifikators wird durch ein statistisches Risiko $R(\cdot; D) : [0, 1]^X \to \mathbb{R}_+$ bewertet [MW18, S. 3]. Die standardmäßig gewählte Verlustfunktion $L(f(x), y)$ für Binäre Klassifikation ist die Binäre Kreuzentropie (*binary cross entropy*), definiert als

$$CE = -\sum_{i=1}^{C'=2} t_i \log(s_i) = -t_1 \log(s_1) - (1 - t_1 \log(1 - s_1). \tag{2.2}$$

Es wird von zwei Klassen C_1 und C_2 ausgegangen. $t_1 \in [0, 1]$ und s_1 sind die Grundwahrheit und der Wert für C_1; $t_2 = 1 - t_1$ sowie $s_2 = 1 - t_2$ für C_2 [RK14].

Sollte der vorgegebene Raum $[0, 1]$ für die Klassifizierung nicht ausreichen, kann die *Multi-Class Classification* eingesetzt werden. Hierfür wird die Binäre Klassifikation durch eine Auswahl verschiedener Strategien an das vorgegebene Lernproblem angepasst. Eine hierbei verwendete Methode ist die Transformation der Problemstellung in den binären Raum, welche mit Hilfe der One-vs.-Rest (OvR) oder One-vs.-One (OvO) Strategie umgesetzt werden kann. Die OvR-Strategie beinhaltet dabei das Training eines einzelnen Klassifikators pro Klasse, wobei die Proben dieser Klasse als positive Proben und alle anderen Proben als negative Proben gelten. Diese Strategie erfordert, dass die Basisklassifikatoren einen realwertigen Konfidenzwert für ihre Entscheidung erzeugen und nicht nur ein Klassenetikett; diskrete Klassenetiketten allein können zu Mehrdeutigkeiten führen, bei denen mehrere Klassen für eine einzelne Probe vorhergesagt werden [Bis06, S. 182]. Bei der OvO-Strategie hingegen werden $K(K-1)/2$ binäre Klassifikatoren für ein K-Wege-Mehrklassenproblem trainiert. Jeder Klassifikator K erhält die Proben eines Klassenpaares

aus der ursprünglichen Trainingsmenge und muss lernen, diese beiden Klassen zu unterscheiden [Bis06, S. 339]. Wie OvR leidet auch OvO unter Mehrdeutigkeiten, da einige Regionen des Eingaberaums die gleiche Anzahl von Stimmen erhalten können [Bis06, S. 183]. Neben dem Einsatz der Transformation in den binären Raum, kann ein binärer Klassifikator auch zur Lösung von Mehrklassen Problemen erweitert werden. Aufgrund der Bedeutsamkeit dieser Anpassungstechnik in Kontext dieser Arbeit, wird hierauf in Kapitel 2.3.2 tiefer eingegangen.

Optimierungsverfahren für maschinelle Lernmethoden

Damit eine Vielzahl von optimalen Lösungen für verschiedene Fragestellungen ermittelt werden kann, kommen Optimierungsverfahren zum Einsatz. Der Grundgedanke dieser ist es, die Parameter iterativ so zu verändern, dass eine Kostenfunktion minimiert wird.

Gradientenverfahren

Das Gradientenverfahren ist anwendbar, um eine differenzierbare Funktion $f : \mathbb{R}^n \rightarrow \mathbb{R}$ mit einem reellen Wert zu erhalten, die minimiert werden soll:

$$\min_{x \in \mathbb{R}^n} f(x). \tag{2.3}$$

Das Verfahren berechnet den lokalen Gradienten der Fehlerfunktion, entsprechend dem Parametervektor θ und bewegt sich in Richtung eines abnehmenden Gradienten. Sobald die Steigung Null annimmt, ist ein Minimum erreicht. Zu Beginn wird θ mit Zufallszahlen initialisiert, welche anschließend in kleinen Schritten verbessert werden um die Kostenfunktion[1] zu senken. Dieser Schritt wird wiederholt, bis der Algorithmus bei einem lokalen Minimum konvergiert [Gér17, S. 112 f.] (siehe Abbildung 2.1).

Bei der iterativen Suche eines Minimums ist dabei die Schrittgröße zu beachten. Diese wird durch die zuvor definierte Lernrate ermittelt, die als Hyperparameter der Gleichung anzusehen ist. Ist die Lernrate zu klein, steigen die Schritte des Algorithmus, was ihn langsam und ineffizient werden lässt. Eine zu große Lernrate kann hingegen zur Divergenz des Algorithmus führen [Gér17, S. 114].

[1] Als Kostenfunktion kann z.B. der Mean Squared Error (MSE) eingesetzt werden.

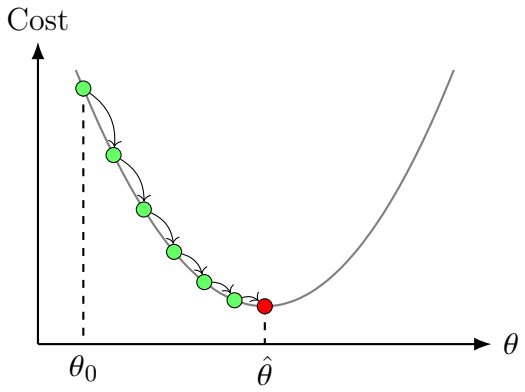


Abbildung 2.1: Gradientenverfahren mit zufälligem Startvektor θ_0 und lokalem Minimum $\hat{\theta}$

2.3 Datenstromorientierte Programmierung mit TensorFlow

TensorFlow ist eine Open-Source-Bibliothek für maschinelles Lernen, die von Google seit 2017 angeboten wird. Sie wurde vom Google Brain Team entwickelt und ermöglicht dank vieler Application Programming Interfaces (APIs) Deep-Learning, um Aufgaben effizient zu lösen. Seit Ende 2019 ist TensorFlow 2 mit Keras-Integration verfügbar (siehe Kapitel **??**). Der Kern von TensorFlow ist in der Programmiersprache C++ implementiert und ermöglicht die Ausführung von Operationen auf unterschiedlichen Hardware-Basen (**CPU!**, GPU und TPU) und Betriebssystemen. Parallel dazu wurde zur Verbesserung der Ausführungszeiten und der Speicheroptimierung Accelerated Linear Algebra (XLA) entwickelt, ein Compiler, der spezifische mathematische Funktionen für TensorFlow optimiert und diese unabhängig von der Hardware ausführen kann [DN19, S. 139 f.].

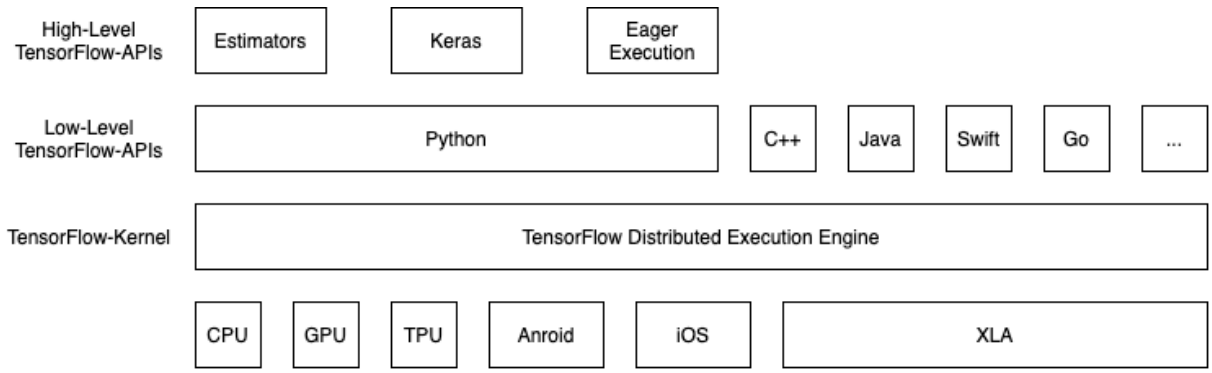
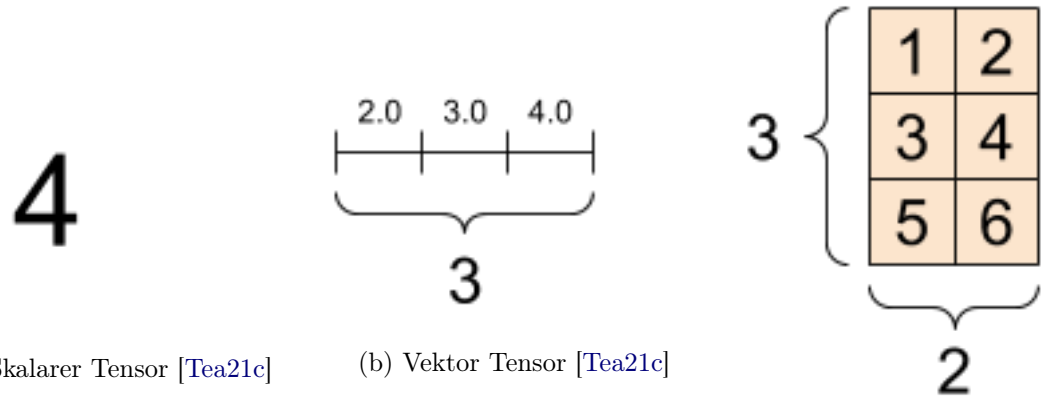


Abbildung 2.2: API-Struktur von TensorFlow 2 (grafisch adaptiert von [DN19, S. 140][Tea17])

2.3.1 Tensoren

Grundlegender Bestandteil TensorFlows sind die schon im Namen enthaltenen Tensoren. Sie werden primär zur Datenspeicherung in neuronalen Netzen eingesetzt. Ein Tensor wird in TensorFlow als multidimensionales Array dargestellt.



(a) Skalarer Tensor [Tea21c] (b) Vektor Tensor [Tea21c] (c) Matrix Tensor [Tea21c]

Abbildung 2.3: Darstellung Skalar, Vektor und Matrix Tensor

Tensoren werden als unveränderliche Variablen gespeichert, das heißt ein Tensor kann niemals aktualisiert werden, sondern muss neu erstellt werden um Anpassungen vorzunehmen. Die Dimensionalität eines Tensors kann vom Benutzer angepasst werden. Dabei kann die Dimenionalität frei definiert werden, wie in Abbildung 2.3a, 2.3b und 2.3c zu sehen. Darüber hinaus ist die multidimensionale Darstellung mit mehreren Achsen möglich (siehe Abbildung 2.4).

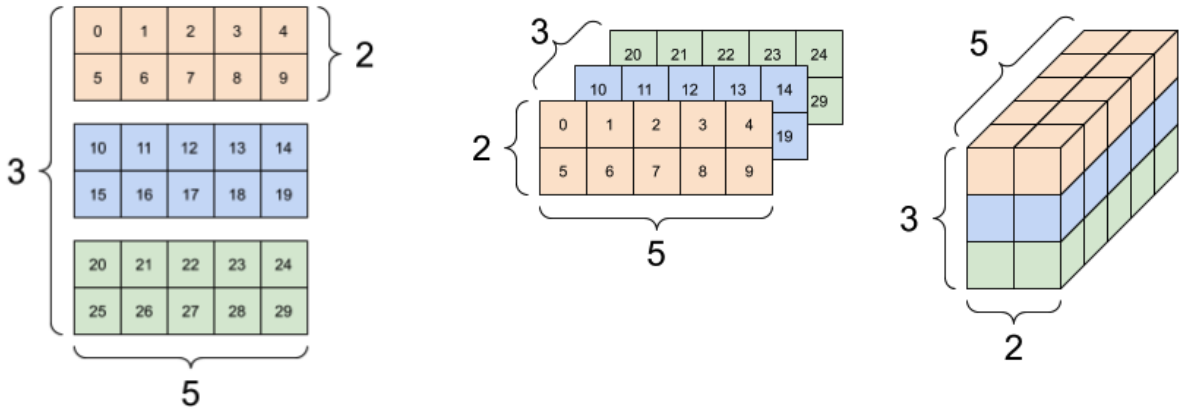


Abbildung 2.4: Unterschiedliche Darstellung eines 3-Achsen Tensors der Form [3, 2, 5] [Tea21c]

2.3.2 Künstliche neuronale Netze

Die Ursprünge der künstlichen neuronalen Netze lassen sich auf McCulloch et al. im Jahre 1943 zurückführen [MP43]. Eine von Donald O. Hebb 1949 formulierte Lernregel stellt seither in ihrer allgemeinen Form die Grundlage der künstlichen neuronalen Lernverfahren dar [Mai97]. Ein künstliches neuronales Netz besteht aus einer Eingabeschicht von Neuronen, 1..n versteckter Schichten und einer letzten Schicht von Ausgangsneuronen. Ein einzelnes Neuron nimmt überlicherweise mehrere Werte $x_1, ..., x_n$ und einen Bias-Term w_0 als Eingabe und berechnet daraus die Ausgabe $y = h(z)$.

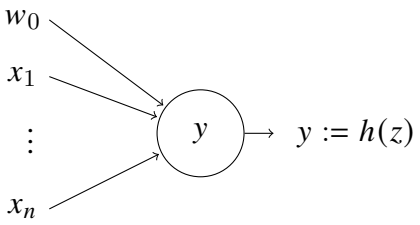


Abbildung 2.5: Einzelnes Neuron mit dessen Eingangsvariablen. Die Aktivierungsfunktion ist beschrieben als h und wird auf die tatsächlichen Eingabe z angewandt. $x_1, ..., x_n$ repräsentieren die Eingabe von anderen Neuronen innerhalb des Netzes. w_0 wird Bias genannt und repräsentiert ein externes Gewicht [Stu14].

Die Ausgabe h_i des Neurons i in der versteckten Schicht wird beschrieben durch

$$h_i = \varphi(\sum_{j=1}^{N} V_{ij} x_j + \theta_i^{hid}) \tag{2.4}$$

wo $\varphi(\cdot)$ die Aktivierungsfunktion, N die Anzahl der Eingangsneuronen, V_{ij} die Gewichte, x_j die Eingabe zum Neuron und θ_i^{hid} der Schwellenwertterm der versteckten Neuronen ist [Wan03, S. 81–100; NMS95, S. 195–201]. Die Intention der Aktivierungsfunktion $\varphi(\cdot)$ neben der Einführung von Nichtlinearität in das neuronale Netz ist, den Wert eines Neurons zu begrenzen, damit das neuronale Netz nicht durch divergierende Neuronen gelähmt wird. Eine gängige Aktivierungsfunktion ist die Sigmoid Funktion $\sigma(\cdot)$, wie definiert in 2.5.

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{2.5}$$

Weitere sigmoide Aktivierungsfunktionen sind der Arkustangens (arctan) und Tangens Hyperbolikus (tanh) [NMS95, S. 195–201]. Sie haben ein ähnliches Ansprechverhalten auf die Eingangswerte wie die Sigmoidfunktion, unterscheiden sich aber in den Ausgangsbereichen. Darüber hinaus gibt es noch nicht-sigmoide Funktionen wie die Rectified Linear Unit (ReLU), die ebenfalls häufig als Aktivierungsfunktion in neuronalen Netzen eingesetzt werden (siehe Abbildung 2.7).

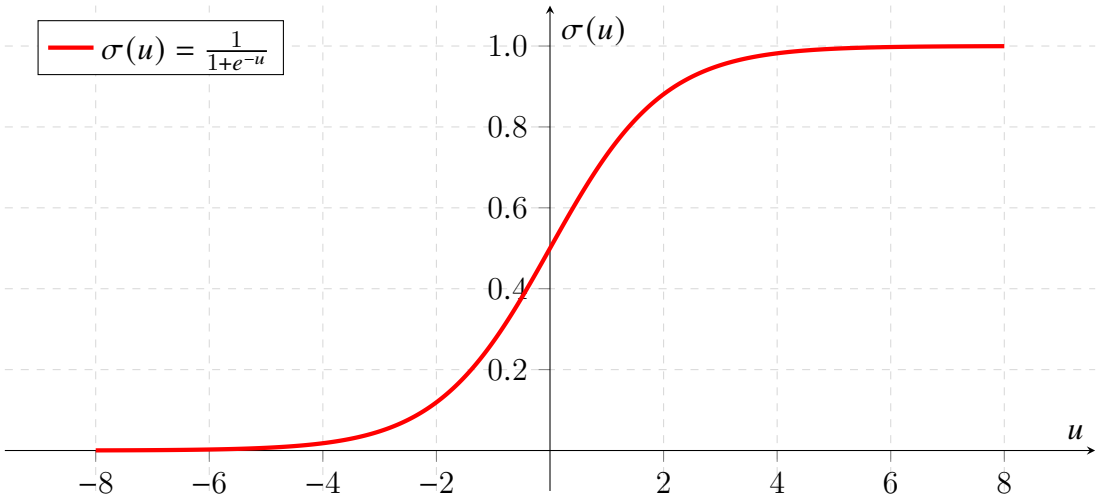


Abbildung 2.6: Darstellung der Sigmoid Aktivierungsfunktion

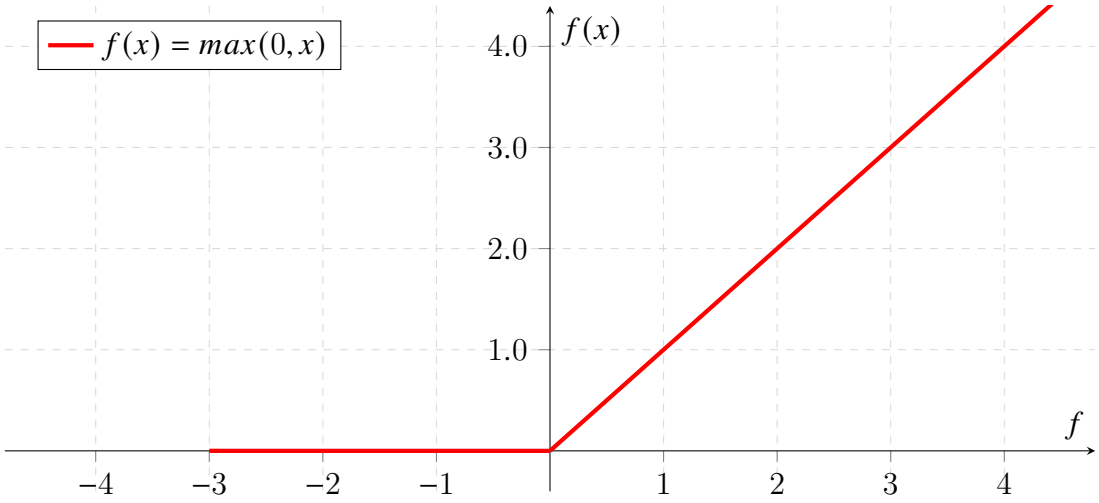


Abbildung 2.7: Darstellung der ReLU Aktivierungsfunktion

2.3.3 Convolutional Neural Networks

Seitdem AlexNet im Jahre 2012 eine Auszeichnung beim jährlichen Wettbewerb der Benchmark-Datenbank ImageNet erzielte [Ima12], hat sich der Forschungsfokus auf das Themengebiet *Deep Learning* zubewegt [KSH12; Ras+16; Rus+15]. Zuvor waren SVMs der prävaliernde Ansatz zur Erkennung von Mustern.

CNNs werden seit 1995 in der digitalen Bildverarbeitung eingesetzt und sind fester Bestandteil des *Deep Learnings*. Es werden Faltmatrizen der Größe 3x3, 5x5, 7x7 bzw. 9x9 eingesetzt, um Bereiche der Eingabematrix sukzessiv zu analysieren. Die dabei verwendeten *convolutional operations* (Faltoperationen) erzeugen rezeptive Felder, die eine Merkmalskarte (*feature map*) des CNN generieren [Rus+15]. Die rezeptiven Felder korrespondieren mit einer Region aus dem Originalbild [Yan20].

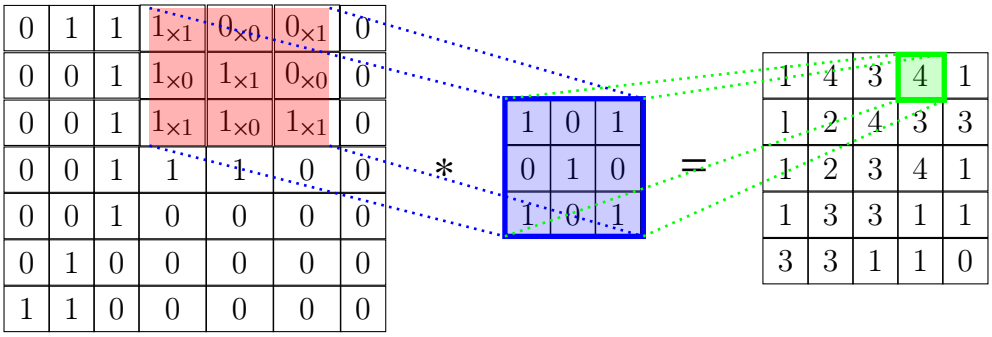


Abbildung 2.8: Erzeugen einer Merkmalskarte durch schrittweise Faltung

In der Mathematik wird die Faltung als eine Operation auf zwei Funktionen f, g beschrieben, die eine dritte Funktion $f * g$ erzeugt. Die dritte Funktion beschreibt, wie die Form von f durch g verändert oder gefiltert wird. Für eine Position $z_{i,j}$ in der Ausgabe gilt

$$z_{i,j} = b + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i+u,j+v} \cdot w_{u,v} \tag{2.6}$$

worin $z_{i,j}$ die Position innerhalb der Matrix z beschreibt und b der Bias ist [Kar20, S. 6]. Betrachtet man nun die Position $z_{i,j}$ in der Ausgabe eines Layers gilt

$$z_{i,j} = b + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i',j'} \cdot w_{u,v} \qquad \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \tag{2.7}$$

worin s_h der vertikale und s_w der horizontale Stride sind [Kar20, S. 13 f.]. Der Stride ist eine Komponente des CNN, abgestimmt auf die Kompression des Eingabedatensatzes. Weitergehend wird er als Parameter des CNN-Filters bezeichnet, der die Bewegung über die Eingabematrix bestimmt.

Die darauffolgende *Volume Convolution* erweitert die Gleichung um einen Parameter k , der die Anzahl der Farbräume in die Gleichung einbezieht [Kar20, S. 25; Gér17, S. 365]. Es gilt

$$z_{i,j,k'} = b_{k'} + \sum_{c=1}^{k} \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} x_{i',j',c} \cdot w_{u,v,c,k'} \qquad \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \tag{2.8}$$

Die Anzahl der Parameter eines CNN ist unabhängig von der Eingabe, jedoch abhängig von der Größe des Filters [Kar20, S. 28]. Allgemein gilt daher

$$\#\ \text{Params}_{conv} = (f_w \cdot f_h \cdot k^{l-1} + 1) \cdot k^l \tag{2.9}$$

2.4 Regionsbasierte Convolutional Neural Networks

Ein erweiterter Ansatz der CNNs sind die von Girshick et al. im Artikel „Rich feature hierarchies for accurate object detection and semantic segmentation“ vorgestellten Region-based Convolutional Neural Networks (R-CNNs) [Gir+13]. Der Artikel stellt ein fundamentales Konzept für alle modernen Objekterkennungen vor: Die Kombination von Regionsvorschlägen und CNNs. Das zu lösende Problem sind lokalisierbare Objekte in einem Bild. Girshick et al. erwähnt zunächst die Verwendung einer Brute-Force Methode, die das Bild mit unterschiedlichsten Rechtecken überdeckt und anschließend versucht, diese einzeln zu klassifizieren. Das damit einhergehende Problem sind eine Unmenge an zu klassifizierender Bilder, die die Performanz deutlich schmälern. Elfouly vereinfacht das hier erwähnte Konzept wie folgend:

> ¨ Region proposals are just smaller parts of the original image, that we think could contain the objects we are searching for. [Elf19] ¨

2.4.1 Sliding Window Algorithmus

Der Sliding Window Algorithmus ist die zuvor erwähnte Methode, das Eingabebild mit einem verschiebbaren Kasten auszuwählen. Die dabei erzeugten Bildausschnitte werden jeweils unter Verwendung des Objekterkennungsmodells klassifiziert. Eine solche Methode muss alle Stellen im Bild abdecken, sowie verschiedene Maßstäbe der selektierten Bildausschnitte einbeziehen. Grund hierfür ist, dass Modelle zur Objekterkennung in der Regel für einen bestimmten Maßstab trainiert werden. Dies führt dazu, dass Zehntausende von Bildfeldern klassifiziert werden müssen. Der Sliding-Window-Ansatz ist gut für Objekte mit festem Seitenverhältnis, wie z. B. Gesichter oder Fußgänger. Bilder sind 2D-Projektionen von 3D-Objekten. Objektmerkmale wie das Seitenverhältnis und die Form variieren je nach Aufnahmewinkel erheblich. Der Sliding-Window-Ansatz ist sehr rechenintensiv, wenn nach mehreren Seitenverhältnissen gesucht wird [Cha17].

2.4.2 Region Proposal Algorithmus

Das zuvor erwähnte Problem lässt sich mit Algorithmen zum Vorschlagen von Regionen lösen. Diese Methoden nehmen ein Bild als Eingabe und geben anschließend Begrenzungsrahmen aus, die allen Bereichen in einem Bild entsprechen, bei denen es sich höchstwahrscheinlich um Objekte handelt. Diese Regionsvorschläge können verrauscht sein, sich überlappen und das Objekt nicht perfekt enthalten. Jedoch wird unter diesen Regionsvorschlägen einen Vorschlag geben, der dem tatsächlichen Objekt im Bild sehr

nahe kommt. Die somit erzeugten Vorschläge können mit einem Objekterkennungsmodell klassifiziert werden. Die Regionsvorschläge mit hohen Wahrscheinlichkeitswerten sind Standorte des Objekts [Cha17].

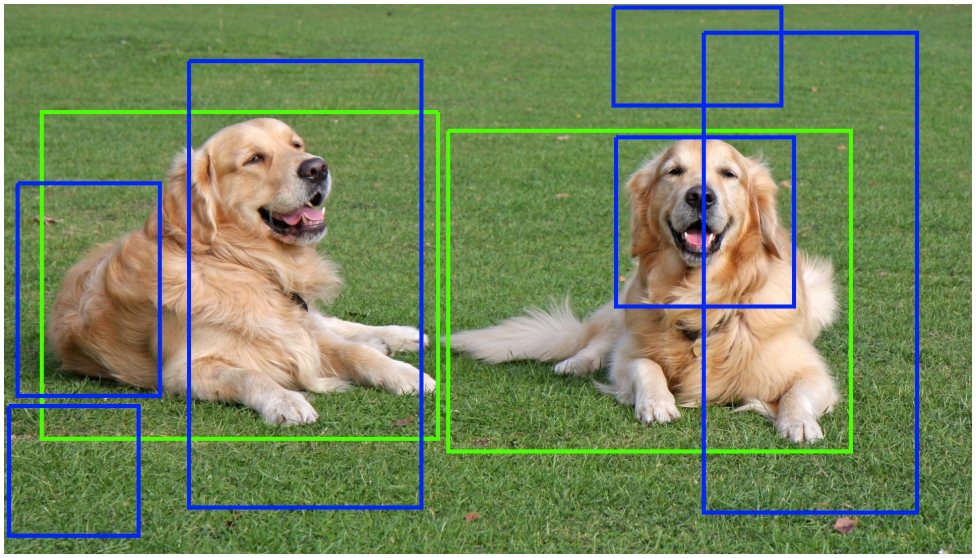


Abbildung 2.9: Regionsvorschläge - Blaue Boxen stellen False-Positives und Grüne Boxen True-Positives dar [Cha17].

Algorithmen mit Regionsvorschlägen identifizieren voraussichtliche Objekte in einem Bild durch Segmentierung. Bei der Segmentierung werden benachbarte Regionen gruppiert, die einander ähnlich sind, anhand einiger Kriterien wie Farbe oder Textur. Im Gegensatz zum Sliding-Window-Ansatz, bei dem das Objekt an allen Pixelpositionen und in allen Maßstäben gesucht wird, gruppiert der Region-Proposal-Algorithmus die Pixel in eine kleinere Anzahl von Segmenten. Daher ist die endgültige Anzahl der generierten Vorschläge um ein Vielfaches geringer als beim Sliding-Window-Ansatz. Dadurch wird die Anzahl der zu klassifizierenden Bildbereiche reduziert. Diese generierten Regionsvorschläge haben unterschiedliche Maßstäbe und Seitenverhältnisse.

2.4.3 Selective Search Algorithmus

Selective Search ist ein Regionsvorschlag-Algorithmus, der bei der Objekterkennung verwendet wird. Er ist so konzipiert, dass er schnell ist und eine sehr hohe Wiedererkennung aufweist. Er basiert auf der Berechnung einer hierarchischen Gruppierung ähnlicher Regionen auf der Grundlage der Kompatibilität von Farbe, Textur, Größe und Form [Cha17; Uij+13]. Selective Search beginnt mit einer Übersegmentierung des Bildes auf Basis der Intensität der Pixel unter Verwendung einer graphbasierten Segmentierungsmethode von Felzenszwalb et al. [FH98]. Selective Search nimmt die Übersegmentierungen als initialen Input und iteriert anschließend über die Bildinkremente.

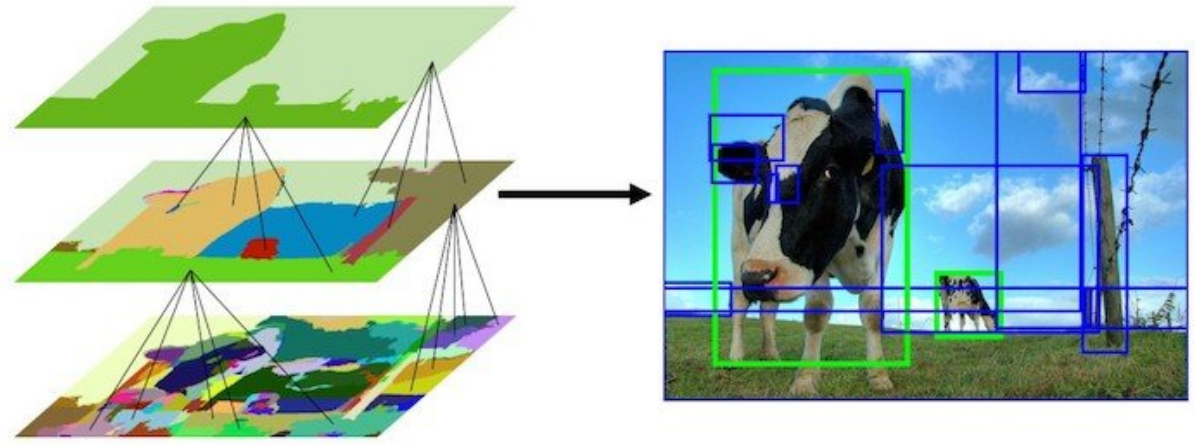


Abbildung 2.10: Hierarchischer Segmentierungsprozess nach Uijlings et al.[Cha17; Uij+13]

2.4.4 Berechnung der Segmentierung in der Theorie

Um aus den zuvor berechneten Segmenten eine Ähnlichkeit herauszufinden, werden vier unterschiedliche Berechnungsmethoden verwendet.

Farbmerkmale

Für jeden Kanal des Bildes wird ein Farbhistogramm mit 25 Bins berechnet und die Histogramme für alle Kanäle werden zu einem Farbdeskriptor verkettet, so dass sich ein $25 * 3 = 75$-dimensionaler Farbdeskriptor ergibt. Die Farbähnlichkeit zweier Regionen basiert auf der Histogrammschnittmenge und kann wie folgt berechnet werden [Cha17]

$$s_{color}(r_i, r_j) = \sum_{k=1}^{n} \min(c_i^k, c_j^k). \tag{2.10}$$

c_i^k ist der Histogrammwert für das k^{te} Bin im Farbdeskriptor.

Texturmerkmale

Texturmerkmale werden durch Extraktion von Gauß-Ableitungen bei 8 Orientierungen für jeden Kanal berechnet. Für jede Ausrichtung und für jeden Farbkanal wird ein 10-Bin-Histogramm berechnet, was zu einem 10x8x3 = 240-dimensionalen Merkmalsdeskriptor führt. Die Texturähnlichkeit zweier Regionen wird ebenfalls mithilfe von Histogrammschnittpunkten berechnet [Cha17].

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} \min(t_i^k, t_j^k). \tag{2.11}$$

t_i^k ist der Histogrammwert für das k^{te} Bin im Texturdeskriptor.

Größenmerkmale

Die Größenähnlichkeit regt kleinere Regionen dazu an, frühzeitig zu verschmelzen. Sie sorgt dafür, dass an allen Stellen des Bildes Regionsvorschläge in allen Maßstäben gebildet werden. Wenn dieses Ähnlichkeitsmaß nicht berücksichtigt wird, verschlingt eine einzelne Region nach und nach alle kleineren benachbarten Regionen und daher werden nur an dieser Stelle Regionsvorschläge in mehreren Maßstäben erzeugt [Cha17]. Die Größenähnlichkeit ist definiert als:

$$s_{size}(r_i, r_j) = \frac{\text{size}(r_i) + \text{size}(r_j)}{\text{size}(im)}. \tag{2.12}$$

wo size(im) die Größe des Bildes in Pixeln ist.

Formkompatibilität

Die Formkompatibilität misst, wie gut zwei Regionen r_i und r_j ineinander passen. Wenn r_i in r_j passt, sollen diese zusammengeführt werden. Tritt der Fall ein, dass die beiden Regionen sich nicht berühren, werden sie nicht zusammengeführt [Cha17].

Die Formkompatibilität ist definiert als:

$$s_{fill}(r_i, r_j) = 1 - \frac{\text{size}(BB_{ij}) - \text{size}(r_i) - \text{size}(r_j)}{\text{size}(im)}. \tag{2.13}$$

wo size(BB_{ij}) eine Bounding Box um r_i und r_j ist.

Finale Kompatibilität

Die finale Kompatibilität zweier Regionen des Bildes ist als eine lineare Kombination der zuvor genannten Merkmale definiert [Cha17].

$$s(r_i, r_j) = a_1 s_{color}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{size}(r_i, r_j) + a_4 s_{fill}(r_i, r_j). \tag{2.14}$$

wo r_i und r_j zwei Regionen oder Segmente des Bildes sind und $a_i \in 0, 1$ die Verwendung des Ähnlichkeitsmaßes ist.

2.5 Verwandte Arbeiten

Der Einsatz maschineller Lernmethoden zur Erkennung von Objekten im Straßenverkehr ist weit verbreitet. Es existieren bereits mehrere Arbeiten, die sich mit der intelligenten Erkennung von Objekten im Straßenverkehr befassen. Ziel ist dabei, Rückschlüsse zu unterschiedlichsten Situationen zu erlangen, zum Beispiel der Verkehrsoptimierung innerhalb von Großstätten oder der Erkennung von Geschwindigkeitsüberschreitungen.

Die Arbeit [E+05] beschreibt eine Möglichkeit, die Überwachung von Verkehrsflüssen zur Analyse des Straßenverkehrs, mit Hilfe von Bildverarbeitungs- und Mustererkennungsmethoden umzusetzen. Die dabei verwendeten Methoden ergeben die Funktionalität des Systems zur Überwachung der Straße sowie der Messung von Geschwindigkeiten. Zusätzlich werden die Nummernschilder der Fahrzeuge erfasst.

In der Arbeit [WB20] werden parametrische sowie nicht-parametrische Methoden zur Vorhersage von Verkehrsaufkommen als wichtiger Teil des intelligenten Verkehrssystems (ITS) analysiert. Dabei wird die Berücksichtigung nichtlinearer Merkmale in maschinellen Lernmethoden analysiert und der Implementierungsaufwand in Bezug auf den Zeitaufwand für Training und Vorhersage gegenübergestellt. Darüber hinaus wurde eine Optimierungsmethode entwickelt um die Skalierbarkeit bestimmter Modelle zu verbessern.

Eine weitere Arbeit [Ata+19] nimmt das dramatische Wachstum der Bevölkerung in Großstädten als Motivation zur Entwicklung effizienter Verkehrssysteme. Der darin erwähnte dynamische Verkehrsfluss erfordert einen Mechanismus zur Vorhersage von Verkehrsstaus mit Hilfe künstlicher Neuronaler Netze. Der entwickelte Mechanismus soll die entstandenen Staus kontrollieren bzw. minimieren und zu einer Beruhigung des Straßenverkehrs führen. Die Autoren verwenden einen Backpropagation-Algorithmus zum Trainieren des Neuronalen Netzes und der daraus hervorkommenden Lösung zum Treffen intelligenter Entscheidungen in der Verkehrssteuerung.

In der Arbeit [DN17] werden Internet of Things (IOT)-Sensoren innerhalb eines Smart-City Szenarios eingesetzt, um Daten für eine Analyse des Verkehrsflusses unter Verwendung neuronaler Netze zu sammeln. Die dabei generierte Verkehrsstauvorhersage wird für die Analyse des Verkehrs sowie der Vorhersage von Staus auf bestimmten Strecken eingesetzt.

In Abgrenzung zu den genannten Arbeiten wird in dieser Arbeit ein R-CNN dazu eingesetzt, Verkehrsobjekte zu erkennen, ihre Geschwindigkeit zu messen und zur Erzeugung einer aussagekräftigen Statistik beizutragen.

3 Analyse der Datenströme

3.1 Anforderungen an die Analyse

3.2 Datenaufbereitung

3.2.1 Datenerhebung und Integration

3.2.2 Datenberechnung

3.2.3 Datenaggregation

3.2.4 Datenbereinigung

4 Entwicklung des Modells

Als ersten Schritt zur Erkennung von Objekten im Straßenverkehr wird der Entwurf eines CNN angeführt. Maßgeblichen Einfluss auf die Entscheidung der Architektur des CNN sowie der Vorgehensweise hat die Arbeit *Image Classification using a Convolutional Neural Network* [SD20]. Das Modell wird als beispielhafte Implementierung einer Multi-Class Classification entworfen und bietet somit die Möglichkeit einer theoretischen Transformation in eine Multi-Label Classification. Als Datensatz wird der Canadian Institute For Advanced Research (CIFAR)-10 Datensatz verwendet. Der CIFAR-10-Datensatz ist eine Sammlung von Bildern, die üblicherweise zum Trainieren von Algorithmen für maschinelles Lernen und Computer Vision verwendet werden. Bestehend aus 60000 32x32 Farbbildern, unterteilt in 10 Klassen, ist es einer der am häufigsten verwendeten Datensätze für die Forschung zum maschinellen Lernen [Ea17; Ham18]. Das Test-Batch enthält 1000 zufällig ausgewählte Bilder jeder Klasse, wohingegen der Train-Batch die restlichen 5000 Bilder einer jeden Klasse enthält [Kri09b; Kri09a].

4.1 Vorverarbeitung der Daten

Goodfellow et al. erwähnen in ihrem Buch *Deep Learning (Adaptive Computation and Machine Learning series)* die Problematik des Overfittings von Neuronalen Netzen.

> The central challenge in machine learning is that we must perform well on new, previously unseen inputs — not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization [GBC16, S. 110].

Um dem Overfitting Effekt des CNN vorzubeugen, wird der gewählte Datensatz zunächst einer Reihe von Augmentationsalgorithmen unterzogen. Diese Schritte sind notwendig für die Sicherstellung eines generalisierten Ansatzes zur Erkennung neuer Test-Splits. Dabei lernt das Modell von bekannten Beispielen und kann somit verallgemeinert auf neue Beispiele in der Zukunft eingesetzt werden.

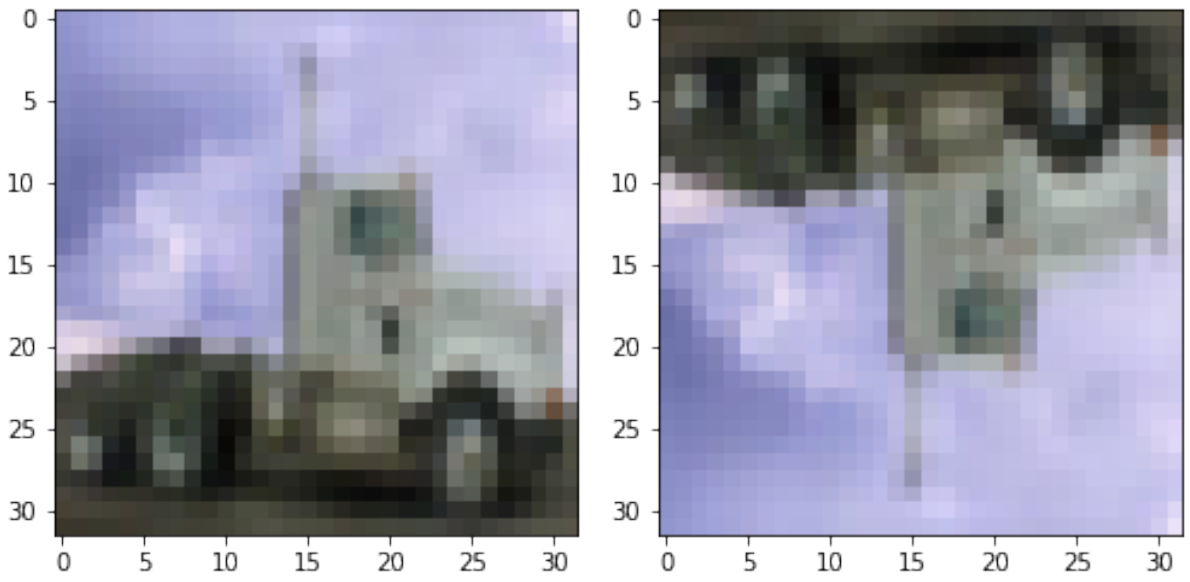
4.1.1 Ausrichtung des Bildtensors

Der erste Schritt der Augmentation ist die zufällige Veränderung der Ausrichtung des Bildes. TensorFlow bietet passend hierfür zwei Funktionen an, die eine Spiegelung an horizontaler bzw. vertikaler Achse vornehmen.

```python
def flip_c(image: tf.Tensor) -> tf.Tensor:
  # Randomize alignment (left/right)
  image = tf.image.random_flip_left_right(image)
  # Randomize alignment (top/down)
  image = tf.image.random_flip_up_down(image)
  # Return the randomized image
  return image
```

Listing 4.1: Python Funktion zum zufälligen Ändern der Ausrichtung

Die Funktion *random_flip_left_right* gibt mit einer Wahrscheinlichkeit von 1 zu 2 den Inhalt des Bildes gespiegelt entlang der zweiten Dimension, also der Breite, aus. Andernfalls wird das Bild so ausgegeben, wie es ist. Wenn ein Stapel von Bildern übergeben wird, wird jedes Bild unabhängig von den anderen Bildern zufällig gespiegelt [Tea21f]. Die Funktion *random_flip_up_down* gibt mit derselben Wahrscheinlichkeit ein gespiegeltes Bild entlang der ersten Dimension zurück [Tea21g].



(a) Initiales Bild

(b) Bild nach Flip-Augmentation

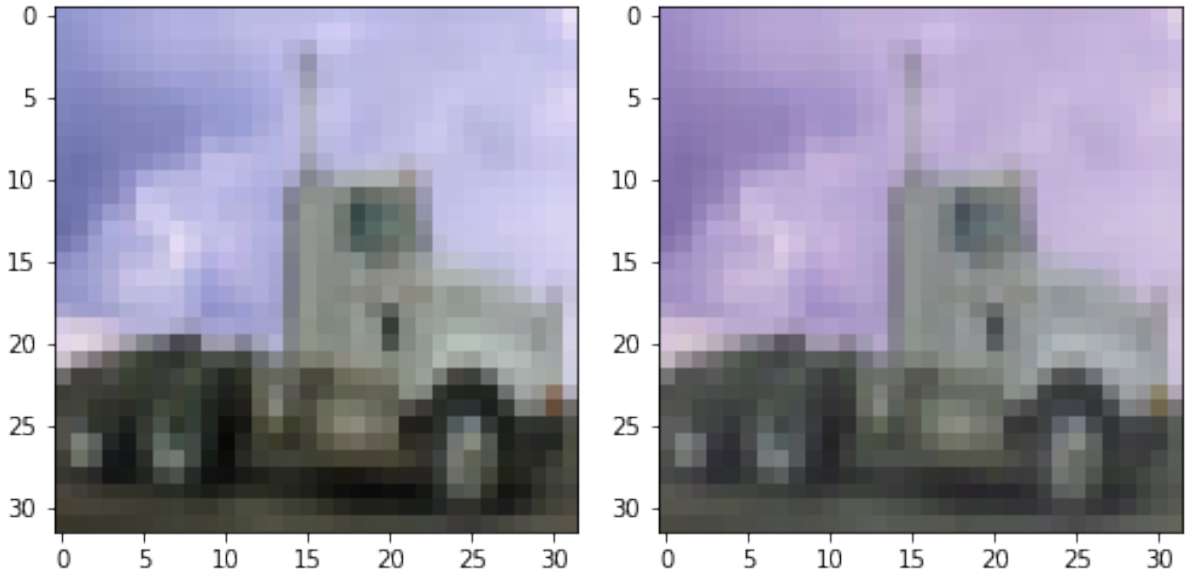
Abbildung 4.1: Vergleich desselben Bildes nach Randomisierung der Ausrichtung

4.1.2 Farbanpassung des Bildtensors

Als weiterer Bestandteil des Augmentationsprozesses werden zufällige Werte für Farbton, Sättigung, Helligkeit und Kontrast gewählt. Der Farbton wird anhand eines übergebenen Deltas im Interval $[0, 0.08]$ modifiziert [Tea21h]. Die Sättigung des Bildes wird im Interval $[0.7, 1.3]$ angepasst [Tea21i]. Ein ebenso wichtiger Bestandteil der Farbaugmentation ist die Anpassung der Helligkeit, welche in diesem Fall mit Hilfe des Intervals $[0, 0.05]$ angepasst wird [Tea21d]. Abschließend wird eine Modifikation des Kontrastes im Interval $[0.8, 1]$ durchgeführt [Tea21e].

```python
def color_c(image: tf.Tensor) -> tf.Tensor:
  # Randomize hue
  image = tf.image.random_hue(image, max_delta=0.08)
  # Randomize saturation
  image = tf.image.random_saturation(image, lower=0.7, upper=1.3)
  # Randomize brightness
  image = tf.image.random_brightness(image, 0.05)
  # Randomize contrast
  image = tf.image.random_contrast(image, lower=0.8, upper=1)
  # Return the randomized image
  return image
```

Listing 4.2: Python Funktion zum zufälligen Ändern des Farbwertes



(a) Initiales Bild

(b) Bild nach Farbaugmentation

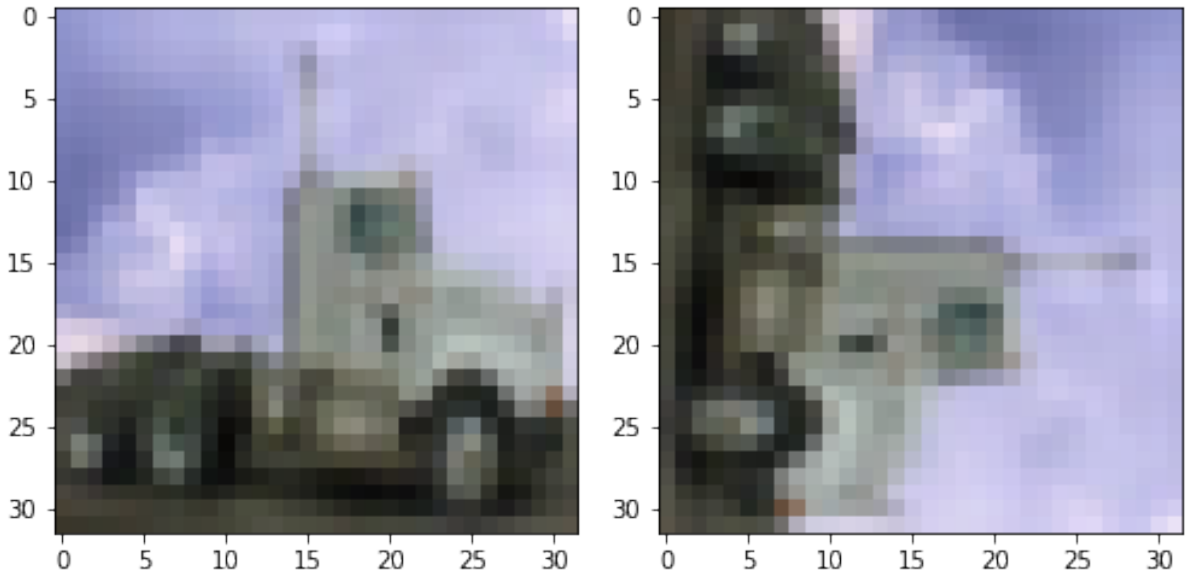
Abbildung 4.2: Vergleich desselben Bildes nach Randomisierung der Farbwerte

4.1.3 Rotation des Bildtensors

Zur weiteren Unterbindung des Overfittings wird ein Verfahren zur Rotation des Bildtensors eingesetzt. TensorFlow bietet hierzu in der zugehörigen Bildbibliothek eine Funktion zur 90-Grad Rotation an [Tea21k]. Über einen Parameter lässt sich die Anzahl der Rotationsschritte bestimmen, woraus eine zufällige Rotation entsteht. Die dynamische Erzeugung von Rotationsschritten wird mittels der Funktion *tf.random.uniform* generiert, welche einen Zufallswert aus einer Gleichverteilung ausgibt [Tea21j].

```
1  def rotation_c(image: tf.Tensor) -> tf.Tensor:
2    # Rotate 0, 90, 180, 270 degrees
3    # Return the randomized image
4    return tf.image.rot90(
5        image,
6        tf.random.uniform(shape=[], minval=0, maxval=4, dtype=tf.int32)
7    )
```

Listing 4.3: Python Funktion zum zufälligen Rotieren des Bildtensors



(a) Initiales Bild

(b) Bild nach Augmentation der Rotation

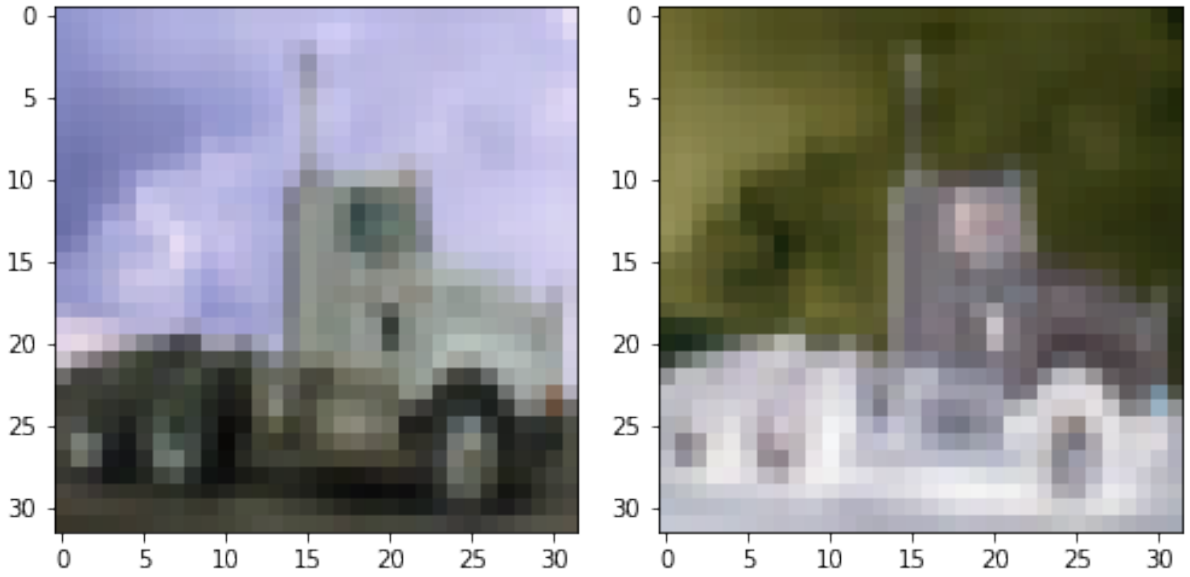
Abbildung 4.3: Vergleich desselben Bildes nach Randomisierung der Rotation

4.1.4 Inversion des Bildtensors

Um einen noch größeren Grad der variablen Gestaltung einzelner Bildtensoren zu erreichen, wird von dem übergebenen Tensor, mit einer Wahrscheinlichkeit von 1 zu 2, die Inverse gebildet und zurückgegeben.

```
1  def inversion_c(image: tf.Tensor) -> tf.Tensor:
2    # Invert the submitted tensor
3    random = tf.random.uniform(shape=[], minval=0, maxval=1)
4    if random > 0.5:
5        image = tf.math.multiply(image, -1)
6        image = tf.math.add(image, 1)
7    # Return the randomized image
8    return image
```

Listing 4.4: Python Funktion zum zufälligen Rotieren des Bildtensors



(a) Initiales Bild

(b) Bild nach Inversion des Farbraums

Abbildung 4.4: Vergleich desselben Bildes nach Inversion des Farbraums

4.1.5 Finale Augmentation des Bildtensors

Nach der Definition aller Augmentationsfunktionen können diese auf den eigentlichen Bildtensor angewandt werden.

```
def augment_c(image, label):
  # Randomize alignment
  image = flip_c(image)
  # Randomize color (saturation/hue/brightness/contrast)
  image = color_c(image)
  # Randomize rotation
  image = rotation_c(image)
  # Randomize inversion
  image = inversion_c(image)
  # Return the fully randomized image
  return image, label
```

Listing 4.5: Finaler Augmentationsschritt

Dieser Schritt fasst die zuvor definierten Schritte zusammen und gibt die modifizierte Version des Bildtensors zurück, wie in Abbildung 4.5 zu sehen ist. Die Schwierigkeit des Erkennens durch das menschliche Auge ist gewollt, da hierdurch die Fähigkeit einer Generalisierung des Netzwerks geschmälert wird.

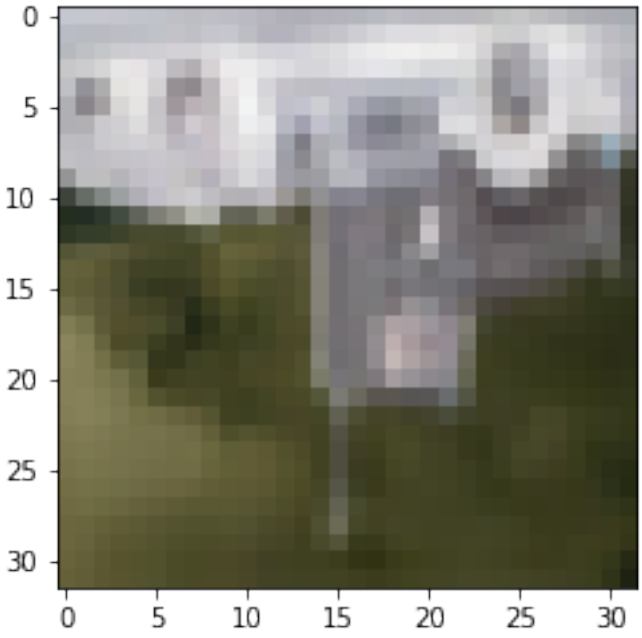


Abbildung 4.5: Darstellung des Bildtensors nach vollständiger Augmentation

4.2 Entwurf eines Netzwerks zur Klassifikation von Objekten

Künstliche neuronale Netze haben zwei Haupt-Hyperparameter, die die Architektur oder Topologie des Netzes steuern: die Anzahl der Schichten und die Anzahl der Knoten in jeder versteckten Schicht. Diese Werte müssen während des Architekturentwurfs definiert werden und sind zur Trainingszeit des Netzwerks nicht veränderbar. Der zuverlässigste Weg der Hyperparametersuche ist ein systematisches Experimentieren mit iterativer Validierung [Bro18]. Es ist gängige Praxis mit einem Grundmodell anzufangen und dieses Schritt-für-Schritt anzupassen. Als Grundmodell wird das Sequentielle Modell eingesetzt, kombiniert mit einem Convolutional2D-Layer, welcher die räumliche Faltung des Bildtensors realisiert (siehe Kapitel 2.3.3). Mit Hilfe des Flatten-Layers wird die Dimensionalität aus dem Netzwerk genommen, woraus ein Layer mit $32x32x32 = 32768$ Neuronen entsteht. Die Ausgabe des Netzwerks wird durch einen Dense-Layer realisiert, der die 10 Ausgabeneuronen anhand der Softmax Aktivierungsfunktion bewertet (siehe Kapitel 2.3.2). Abbildung 4.6 zeigt das Modell in der Architekturansicht.

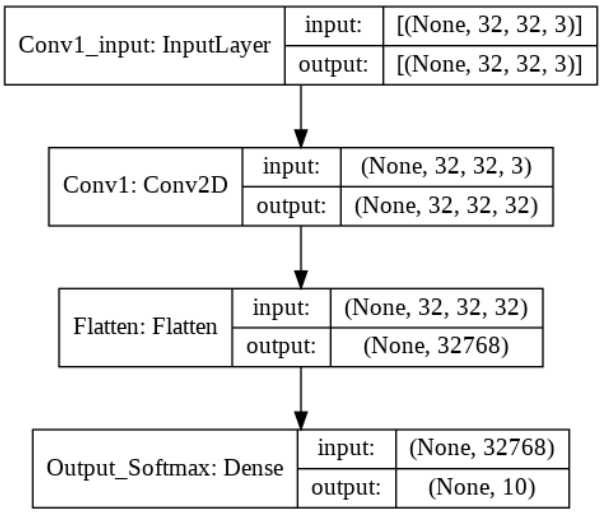


Abbildung 4.6: Architektur des Grundmodells

```
Train loss: 0.693916380405426
Train accuracy: 0.7575250267982483
----------------------------------
Validation loss: 1.8844605684280396
Validation accuracy: 0.5077000260353088
----------------------------------
Test loss: 1.929477334022522
Test accuracy: 0.5001999735832214
```

Abbildung 4.7: Verlust- und Genauigkeitskennzahlen des Grundmodells

Das zuvor gezeigte Modell erzielt auf den Trainings-, Test- und Validierungsdatensätzen die in Abbildung 4.7 aufgezeigten Verlust- und Genauigkeitskennzahlen. Die erhobenen

Metriken zeigen die Dringlichkeit einer Anpassung des Modells auf. Die erzielte Genauigkeit des Trainings ist mit einem Score von 0.75 zunächst in Ordnung, jedoch zeigen die Validierungs- und Test Genauigkeit eine starke Abweichung zu diesem Wert auf. Damit eine Verbesserung der einzelnen Werte erzielt werden kann, wird im nächsten Schritt die Architektur analysiert. Hierfür wird die Anzahl der Schichten evaluiert und Anpassungen von Hyperparametern vorgenommen.

4.2.1 Evaluierung der Anzahl an Convolutional Layers

Um die Anzahl an Convolutional Layers mit der besten Genauigkeit ermitteln zu können, erfolgt eine Evaluierung vier unterschiedlicher Modellarchitekturen. Deotte führt die Grundidee in der Publikation *How to choose CNN Architecture* ein [Deo18]. Die vier Modelle werden wie folgt klassifiziert:

- 2432 - **[32C5-P2]** - 256 - 10
- 2432 - **[32C5-P2]** - **[48C5-P2]** - 256 - 10
- 2432 - **[32C5-P2]** - **[48C5-P2]** - **[64C5-P2]** - 256 - 10
- 2432 - **[32C5-P2]** - **[48C5-P2]** - **[64C5-P2]** - **[80C5-P2]** - 256 - 10

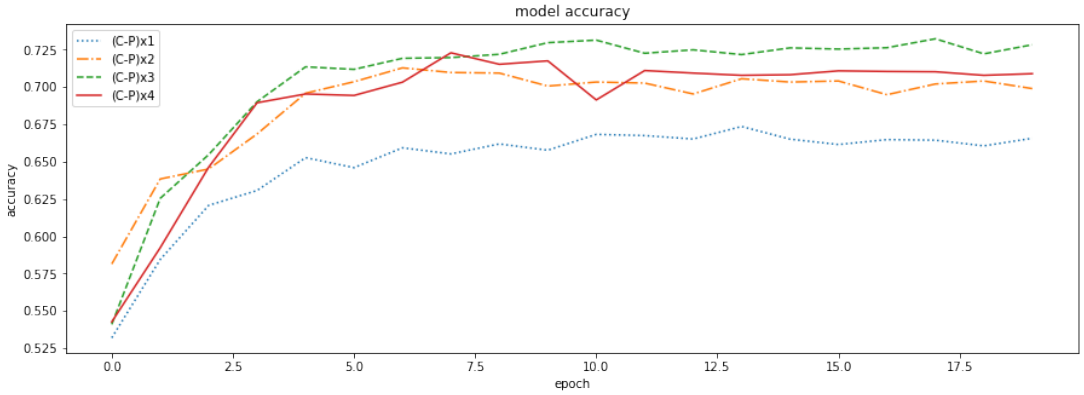
Die Notation lässt sich folgendermaßen erklären: 2432 - Anzahl der Eingabeneuronen; [ΔC5-P2] - Convolutional Layer mit Δ Filtern und anschließendem MaxPooling Layer der Größe 2, Kernel-Größe von 5 und einem Stride von 2; 256 - Fully-connected Dense Layer mit 256 Einheiten; 10 - Ausgabeneuronen zur Klassifikation der Labels.

Die Verwendung von genau vier Modellen in dieser Evaluierung lässt sich auf die Eingabegröße des Bildes zurückführen. Das Eingabebild besitzt die Größe $32x32$. Nach der Ersten Faltung ist die Größe auf $16x16$ reduziert. Die Zweite Faltung verringert dies auf $8x8$, nach der Dritten besitzt das Bild lediglich eine Größe von $4x4$. Kompiliert werden die Modelle mit Hilfe des Adam-Optimizers [KB17] und der Sparse-Categorical-Crossentropy Verlustfunktion [Tea21a].

Aus den in Abbildung 4.8a und 4.8b zu entnehmenden Metriken lässt sich die beste Genauigkeit bei dem Modell **(C-Px3)** erkennen, was auf die Verwendung von drei Convolutional Layers schließen lässt. Die gewonnene Erkenntnis kann somit im nächsten Schritt übernommen und als neues Grundmodell verwendet werden.

```
CNN (C-P)x1: Epochs=20, Train accuracy=0.98410, Validation accuracy=0.67360
CNN (C-P)x2: Epochs=20, Train accuracy=0.99825, Validation accuracy=0.71290
CNN (C-P)x3: Epochs=20, Train accuracy=0.99353, Validation accuracy=0.73240
CNN (C-P)x4: Epochs=20, Train accuracy=0.99280, Validation accuracy=0.72300
```

(a) Genauigkeitsmetriken unterschiedlicher CNN Architekturen



(b) Grafische Darstellung der Genauigkeitsmetriken

Abbildung 4.8: Entscheidungsfindung CNN Architektur

4.2.2 Evaluierung der Feature-Map Größe

Die Feature-Map gibt an, wie hoch die Dimensionalität des Ausgangsraumes ist (d.h. die Anzahl der Ausgangsfilter in der Faltung) [Tea21b]. Für eine aussagekräftige Statistik der Feature-Map Größe wird das zuvor neu definierte Grundmodell weitergehend modifiziert. Dabei werden folgende Netzwerke evaluiert:

- 2432 - [**8**C5-P2] - [**16**C5-P2] - [**24**C5-P2] - 256 - 10
- 2432 - [**16**C5-P2] - [**32**C5-P2] - [**48**C5-P2] - 256 - 10
- 2432 - [**24**C5-P2] - [**48**C5-P2] - [**72**C5-P2] - 256 - 10
- 2432 - [**32**C5-P2] - [**64**C5-P2] - [**96**C5-P2] - 256 - 10
- 2432 - [**40**C5-P2] - [**80**C5-P2] - [**120**C5-P2] - 256 - 10
- 2432 - [**48**C5-P2] - [**96**C5-P2] - [**144**C5-P2] - 256 - 10

Geprüft wird hierbei jeweils die steigende Anzahl an Filtern, definiert durch

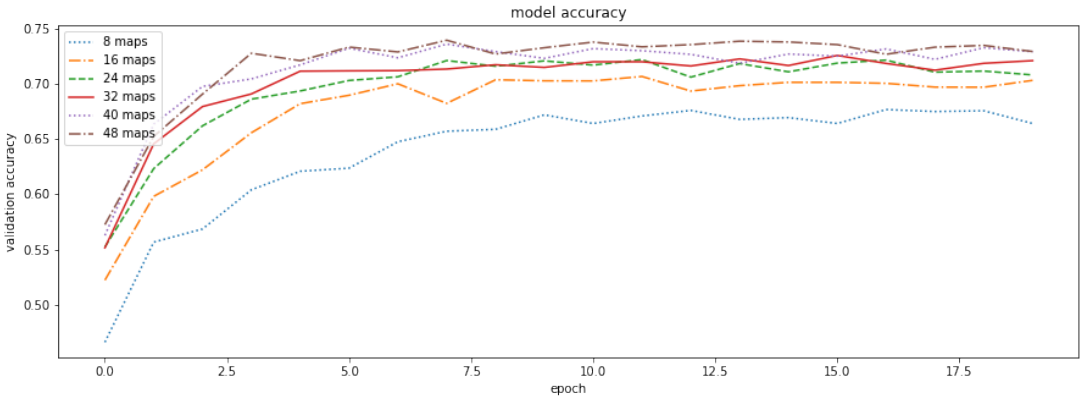
$$\text{size(featuremap)} = \sum_{n=0}^{k} n * (8 * x) + (8 * x) \tag{4.1}$$

worin $k \in [1, 2, 3]$ die Tiefe des Convolutional Layers und $x \in A; A = \{y \mid 1 \leqslant y \leqslant 6\}$ ist. Anhand der in Abbildung 4.9a und 4.9b ersichtlichen Metriken, erzielt ein Netzwerk mit steigender Anzahl an Filtern eine höhere Genauigkeit auf dem Validierungssatz. Da die

Komplexität und Berechnungszeit jedoch ebenfalls mit ansteigen, wird sich für das CNN mit 32 Maps entschieden. Dies bildet einen guten Kompromiss zwischen Genauigkeit und Performance. Die Änderung der Filtergrößen wird in das Grundmodell übernommen.

```
CNN 8 maps: Epochs=20, Train accuracy=0.85250, Validation accuracy=0.67650
CNN 16 maps: Epochs=20, Train accuracy=0.98397, Validation accuracy=0.70640
CNN 24 maps: Epochs=20, Train accuracy=0.99322, Validation accuracy=0.72170
CNN 32 maps: Epochs=20, Train accuracy=0.99540, Validation accuracy=0.72540
CNN 40 maps: Epochs=20, Train accuracy=0.99690, Validation accuracy=0.73570
CNN 48 maps: Epochs=20, Train accuracy=0.99690, Validation accuracy=0.73920
```

(a) Genauigkeitsmetriken unterschiedlicher Feature-Map Größen



(b) Grafische Darstellung der Genauigkeitsmetriken

Abbildung 4.9: Entscheidungsfindung Feature-Map Größe

4.2.3 Bestimmung der Größe des Fully-Connected Layers

In den vorherigen Schritten wurden Anpassungen ausschließlich an den Convolutional Layern vorgenommen. Im nächsten Schritt wird der Fully-Connected Dense Layer evaluiert, um eine Aussage über die in diesem Layer lokalisierten Neuronen treffen zu können. Um den Matrix-Output der Convolutional- und Pooling-Layer in einen Dense Layer speisen zu können, muss dieser zunächst ausgerollt werden *(flatten)*. Die Positionsmerkmale gehen bei dieser Modifikation zwar verloren, jedoch werden die ortsunabhängigen Objektinformationen behalten, wodurch eine Übergabe in einen Output Layer mit passender Anzahl von Neuronen ermöglicht werden kann. Aktiviert werden die Neuronen im Dense Layer durch die ReLU Funktion (siehe Kapitel 2.3.2). Zur ausführlichen Evaluierung werden erneut mehrere Netzwerke erstellt und auf die erzielte Leistung überprüft.

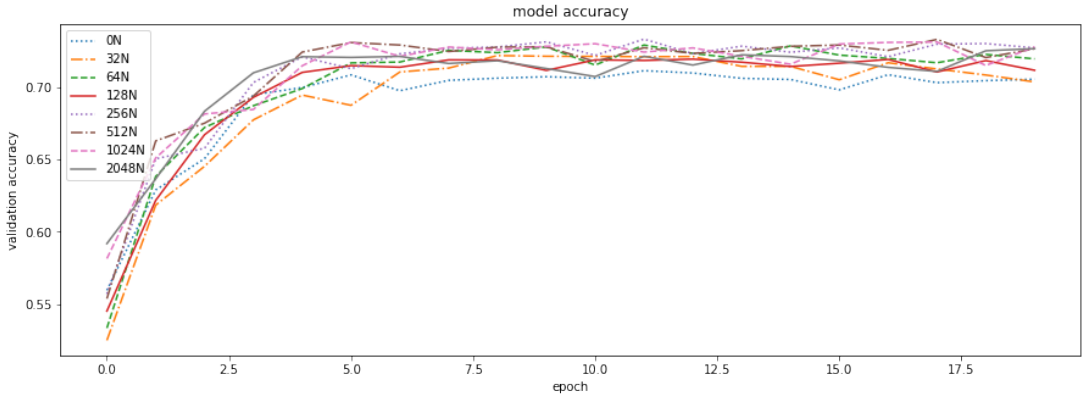
- 2432 - $[32C5 - P2]$ - $[64C5 - P2]$ - $[96C5 - P2]$ - **0** - 10
- 2432 - $[32C5 - P2]$ - $[64C5 - P2]$ - $[96C5 - P2]$ - **32** - 10
- 2432 - $[32C5 - P2]$ - $[64C5 - P2]$ - $[96C5 - P2]$ - **64** - 10

- 2432 - [32$C5-P2$] - [64$C5-P2$] - [96$C5-P2$] - **128** - 10
- 2432 - [32$C5-P2$] - [64$C5-P2$] - [96$C5-P2$] - **256** - 10
- 2432 - [32$C5-P2$] - [64$C5-P2$] - [96$C5-P2$] - **512** - 10
- 2432 - [32$C5-P2$] - [64$C5-P2$] - [96$C5-P2$] - **1024** - 10
- 2432 - [32$C5-P2$] - [64$C5-P2$] - [96$C5-P2$] - **2048** - 10

Abbildung 4.10a und 4.10b geben Aufschluss darüber, welche Performance das Modell mit unterschiedlich großen Fully-Connteced Layern erzielt. Die Verwendung von 256 Neuronen erzielt die höchste Validierungsgenauigkeit, weshalb dieser Wert für die weitere Berechnung übernommen wird.

```
CNN 0N: Epochs=20, Train accuracy=0.99510, Validation accuracy=0.71170
CNN 32N: Epochs=20, Train accuracy=0.98453, Validation accuracy=0.72220
CNN 64N: Epochs=20, Train accuracy=0.99245, Validation accuracy=0.72950
CNN 128N: Epochs=20, Train accuracy=0.99423, Validation accuracy=0.71970
CNN 256N: Epochs=20, Train accuracy=0.99545, Validation accuracy=0.73360
CNN 512N: Epochs=20, Train accuracy=0.99735, Validation accuracy=0.73340
CNN 1024N: Epochs=20, Train accuracy=0.99677, Validation accuracy=0.73170
CNN 2048N: Epochs=20, Train accuracy=0.99990, Validation accuracy=0.72700
```

(a) Genauigkeitsmetriken unterschiedlicher Fully-Connected Layer Größen



(b) Grafische Darstellung der Genauigkeitsmetriken

Abbildung 4.10: Entscheidungsfindung Fully-Connected Layer Größe

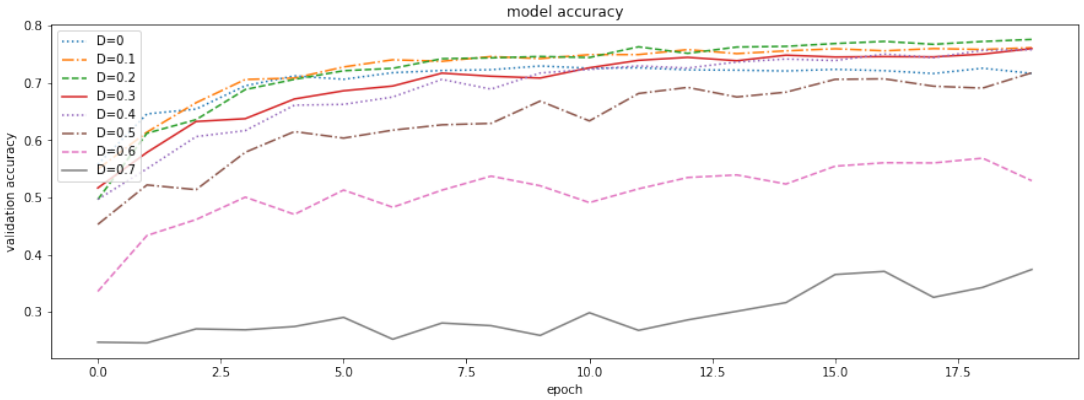
4.2.4 Ermittlung des Dropout Thresholds

Dropout wird als Regularisierungsmethode eingesetzt um die Gefahr von Overfitting zu verringern. Hierfür wird eine vorher spezifizierte Anzahl von Neuronen in jedem Layer des Netzwerks ausgeschaltet (engl. dropout) und für die kommenden Berechnungsschritte ignoriert [Sri+14, S. 1929–1958]. Um den besten Dropout Wert zu ermitteln werden sieben Netzwerke trainiert, die jeweils ein Inkrement von 10% besitzen. Als Nebeneffekt des Dropouts ist in Abbildung 4.11a und 4.11b eine Schmälerung der Trainingsgenauigkeit

zu beobachten. Da hierbei jedoch Overfitting reduziert wird, kann davon ausgegangen werden, dass die zuvor erzielten Genauigkeiten nur erzielt wurden, da das Netzwerk zu viele Informationen über die Bilddaten kannte. Währenddessen ist ein Anstieg der Validierungsgenauigkeit zu erkennen, was auf einen Erfolg der Overfitting Reduktion schließen lässt. Der Dropout Wert von 20% wird anschließend für das finale Modell verwendet.

```
CNN D=0: Epochs=20, Train accuracy=0.99427, Validation accuracy=0.72950
CNN D=0.1: Epochs=20, Train accuracy=0.94488, Validation accuracy=0.76140
CNN D=0.2: Epochs=20, Train accuracy=0.85413, Validation accuracy=0.77590
CNN D=0.3: Epochs=20, Train accuracy=0.77763, Validation accuracy=0.75990
CNN D=0.4: Epochs=20, Train accuracy=0.73362, Validation accuracy=0.75800
CNN D=0.5: Epochs=20, Train accuracy=0.67710, Validation accuracy=0.71770
CNN D=0.6: Epochs=20, Train accuracy=0.58495, Validation accuracy=0.56860
CNN D=0.7: Epochs=20, Train accuracy=0.48417, Validation accuracy=0.37420
```

(a) Genauigkeitsmetriken unterschiedlicher Dropout Layern



(b) Grafische Darstellung der Genauigkeitsmetriken

Abbildung 4.11: Entscheidungsfindung Dropout Threshold

4.3 Definition des finalen Modells

Durch die in den vorherigen Kapiteln gewonnenen Erkenntnisse kann folglich ein finales Modell erstellt und mit dem initialen Modell verglichen werden. Das Modell setzt sich zusammen aus

- 2432 - [$32C5 - P2$] - [$64C5 - P2$] - [$96C5 - P2$] - 256 - 10

mit einem errechneten Dropout von 20%.

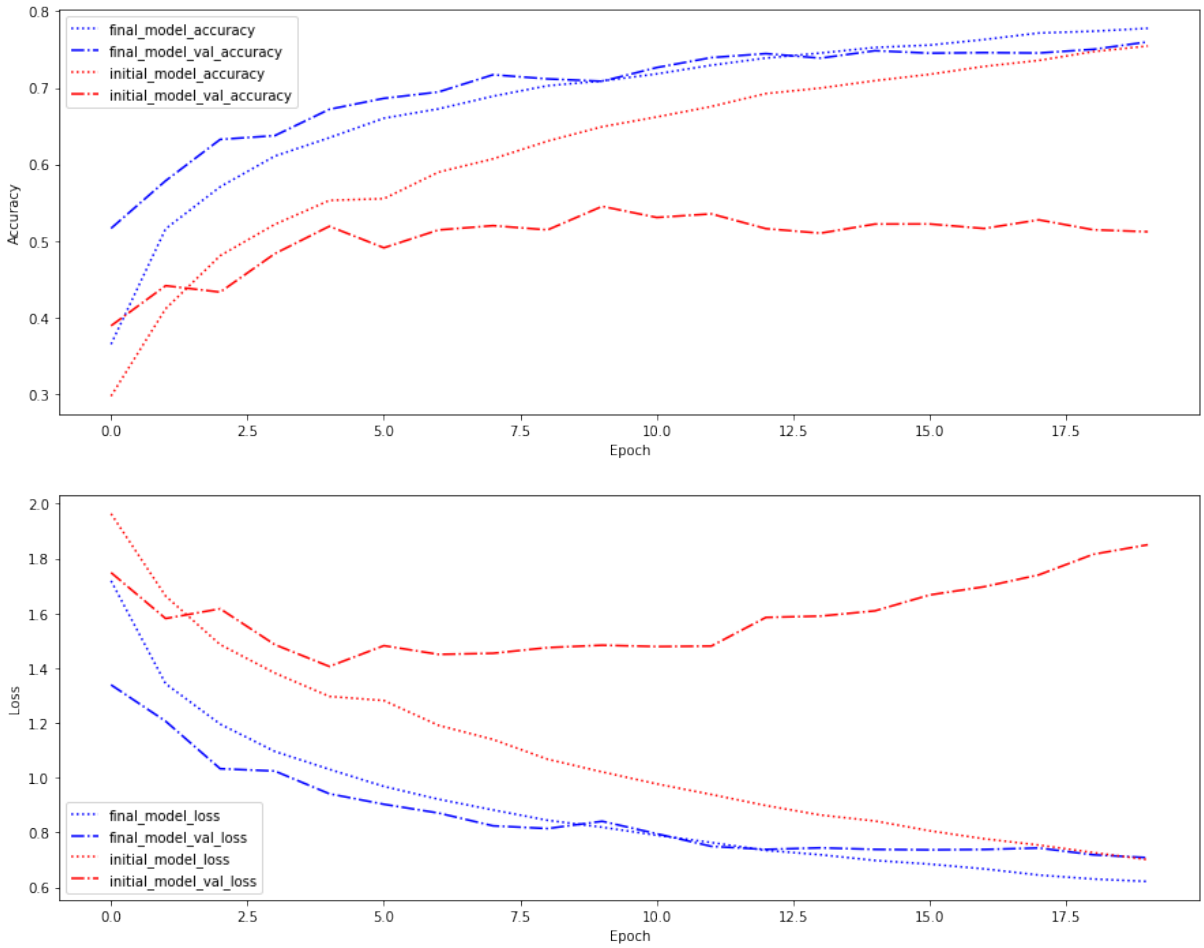


Abbildung 4.12: Vergleich Initiales und Finales Modell

Es ist zu erkennen, dass die initial errechnete Trainingsgenauigkeit eine viel geringere Abweichung zur Validierungsgenauigkeit aufzeigt. Selbiges ist bei der Berechnung des Verlusts zu erkennen, was auf einen Erfolg der Modelloptimierung schließen lässt. Abschließend ist zu erwähnen, dass dieses Modell einen Multi-class Anwendungsfall beschreibt. Für die ordnungsgemäße Verwendung ist die Transformation des Anwendungsfalles von Multi-class zu Multi-label benötigt. Die Erzeugung dieses Modells legt die theoretische Umsetzbarkeit hierfür dar. Die ausstehende Transformation ist in Kapitel 2.2 beschrieben.

5 Prototypische Implementierung

5.1 Bewegungserkennung

Das Ziel der Bewegungserkennung ist es, nur sich bewegende Fahrzeuge zu analysieren und zu erkennen. Dieses Vorgehen wird mit Hilfe der OpenCV Hintergrundsubtraktion ermöglicht.

Die Hintergrundsubtraktion (engl. *background subtraction*) ist eine gängige und weit verbreitete Vorgehensweise zur Erzeugung einer Vordergrundmaske unter Verwendung statischer Kameras. Wie der Name besagt, berechnet die Hintergrundsubtraktion die Vordergrundmaske durch die Subtraktion zwischen dem aktuellen Bild und einem Hintergrundmodell, das den statischen Teil der Szene enthält, oder allgemeiner all das, was auf Grund der Eigenschaften der beobachteten Szene als Hintergrund betrachtet werden kann.

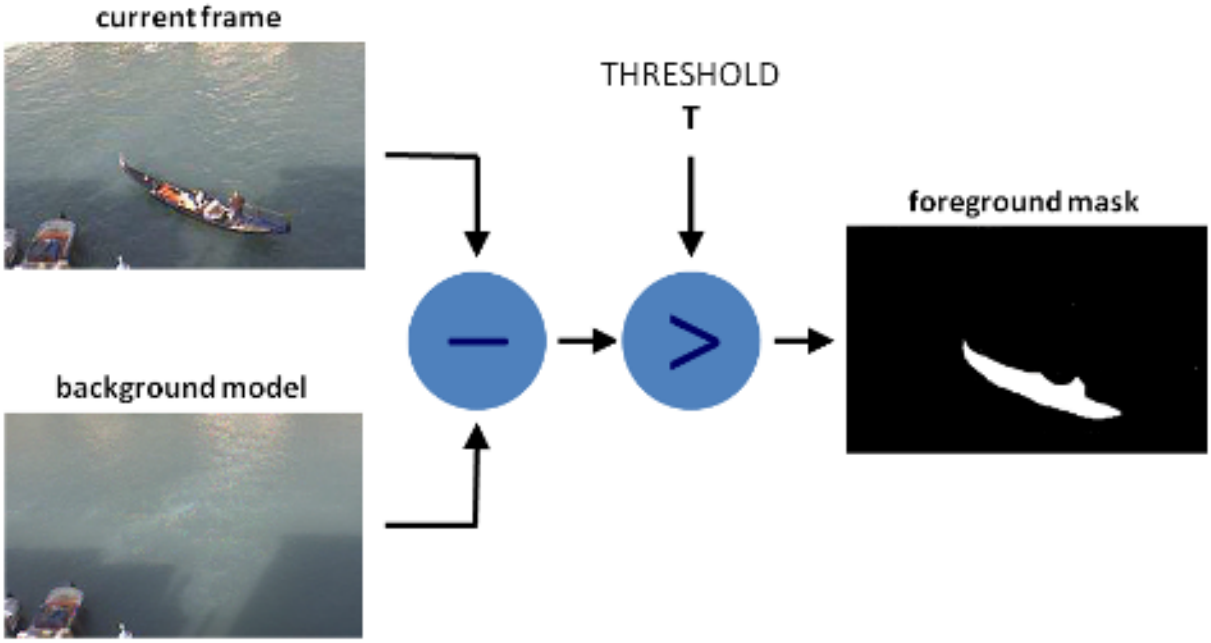


Abbildung 5.1: Vorgehen der Hintergrundsubtraktion [Ope21d]

Im ersten Schritt wird ein anfängliches Modell des Hintergrundes berechnet, während im nächsten Schritt dieses Modell aktualisiert wird, um sich an mögliche Änderugungen in der Szene anzupassen. Der Schwellenwert (engl. *threshold*) legt fest, was als Vordergrund anerkannt wird. Resultierend aus der Berechnung entsteht eine Vordergrundmaske.

In diesem Prototypen wird eine Hintergrundsubtraktion mit dem Mixture of Gaussians (MOG)2 Algorithmus durchgeführt. Dabei handelt es sich um einen Mixture-basierten Hintergrund-/Vordergrund Segmentierungsalgorithmus. Zivkovic et al. führt diesen Algorithmus in den Arbeiten „Improved adaptive Gaussian mixture model for background subtraction“ und „Efficient adaptive density estimation per image pixel for the task of background subtraction“ an [Ziv04; Zv06]. Eine wichtige Eigenschaft dieses Algorithmus ist die geeignete Anzahl von Gaußverteilungen für jeden Pixel. Er bietet somit eine bessere Anpassungsfähigkeit an wechselnde Szenen auf Grund von Beleuchtungsänderungen [Ope21a].

```
# Generate background subtraction with opencv built-in methods
fgbg = cv2.createBackgroundSubtractorMOG2(history=800, detectShadows=
    False, varThreshold=100)
```

Listing 5.1: Generation der Hintergrundsubtraktion

Abbildung 5.1 zeigt die Erstellung eines Hintergrundsubtraktionsobjekts für den Prototypen. Der Parameter *history* wird dazu verwendet, um den Wert der Historie begrenzen zu können. Je höher dieser Wert ist, desto länger ist der Schweif den ein Objekt mit sich zieht. Für diesen Prototypen wurde die standardmäßige Schattenerkennung mit dem Parameter *detectShadows* ausgeschaltet. Das Deaktivieren dieses Parameters führt zu einer deutlichen Leistungssteigerung des Algorithmus. *varThreshold* ist ein Schwellenwert für den quadrierten Mahalanobis-Abstand zwischen dem Pixel und dem Modell. Dieser wird dazu verwendet, um zu entscheiden, ob ein Pixel durch das Hintergrundmodell gut beschrieben ist [Ope21e]. Bevor die Hintergrundsubktration auf das Video angewandt wird, muss das Video durch andere Hilfsfunktionen bearbeitet werden. Der erste Schritt zur genaueren Vorhersage ist die Grauskalierung [Rod21].

Abbildung Grauskalierung

Eine Grauskalierung wird angewandt, da für die Bewegungserkennung keine Farbe benötigt wird und es zur Verbesserung der Anzeigequalität eines Bildes ohnehin zu einer Verminderung des Rauschens führt.

```
# Apply grayscaling on video input
gray_frame = cv2.cvtColor(src=frame, code=cv2.COLOR_BGR2GRAY)
```

Listing 5.2: Graustufen auf das Video anwenden

Mit Hilfe der in Abbildung 5.2 aufgeführten Funktion *cvtColor* wird eine Grauskalierung des aktuellen Bildes durchgeführt. Der Parameter *src* stellt das aktuelle Bild dar, auf welchem

die Farbkonvertierung angewendet werden soll. *code* übergibt eine OpenCV Konstante, die ein Color Space Conversion Code ist. Dieser Code gibt an, ob und was für eine Art der Farbtransformation durchgeführt werden soll. Die Konstante *COLOR_BGR2GRAY* stellt die Gewichtung der Veränderung von Farbkanälen dar (siehe Abbildung 5.2 [Ope21b]).

$$\text{RGB[A] to Gray} : Y \leftarrow 0.299 * R + 0.587 * G + 0.114 * B$$

Abbildung 5.2: Gewichtung der Farbkanäle

Durch den in Abbildung 5.2 aufgezeigten Vorgang wird die Multiplikation der einzelnen Farbkanäle dargestellt. Durch das multiplizieren der Kanäle mit der jeweiligen Konstante entsteht ein neuer RGB Wert Y. Dieser neue Wert kann im Anschluss auf das Bild angewandt werden, um eine Graustufierung zu ermöglichen [Ope21b]. Weitergehend wird auf das daraus entstandene Bild eine Gauß'sche Unschärfe eingesetzt.

Da auch bei zwei unterschiedlichen, in sehr kurzem Abstand aufgenommenen Bildern, ein Unterschied der Beleuchtung oder dem Sensor der Kamera festgestellt werden kann, ist die Verwendung dieser Unschärfeapplikation von Nöten.

```
1 # Application of Gaussian Blur
2 blur_frame = cv2.GaussianBlur(src=gray_frame, ksize=(3, 3), sigmaX=0)
```

Listing 5.3: Anwendung des Weichzeichners

Die Gauß'sche Unschärfe ist als ein NxN-Tap-Faltungsfilter definiert, der die Pixel innerhalb seines Footprints, basierend auf der Gauß-Funktion gewichtet. Die Unschärfefunktion ist definiert durch

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Abbildung 5.3: Gauß'sche Unschärfefunktion

Die Pixel des Filter-Footprints werden mit den Werten aus der Gauß-Funktion gewichtet und erzeugen so einen Unschärfe-Effekt. Die räumliche Darstellung des Gauß-Filters, zeitweise auch als ¨Glockenfläche¨ bezeichnet, zeigt, wie sehr die einzelnen Pixel des Footprints zur endgültigen Pixelfarbe beitragen [Ras10].

Dieser Prototyp setzt einen 3x3 Kernel ein. Ein Kernel ist in diesem Anwendungsfall ein quadratisches Array von Pixeln.

> ¨Jedes Pixel im Bild wird mit dem Gaußschen Kern multipliziert. Dazu wird das mittlere Pixel des Kernels auf dem Bildpixel platziert und die Werte im

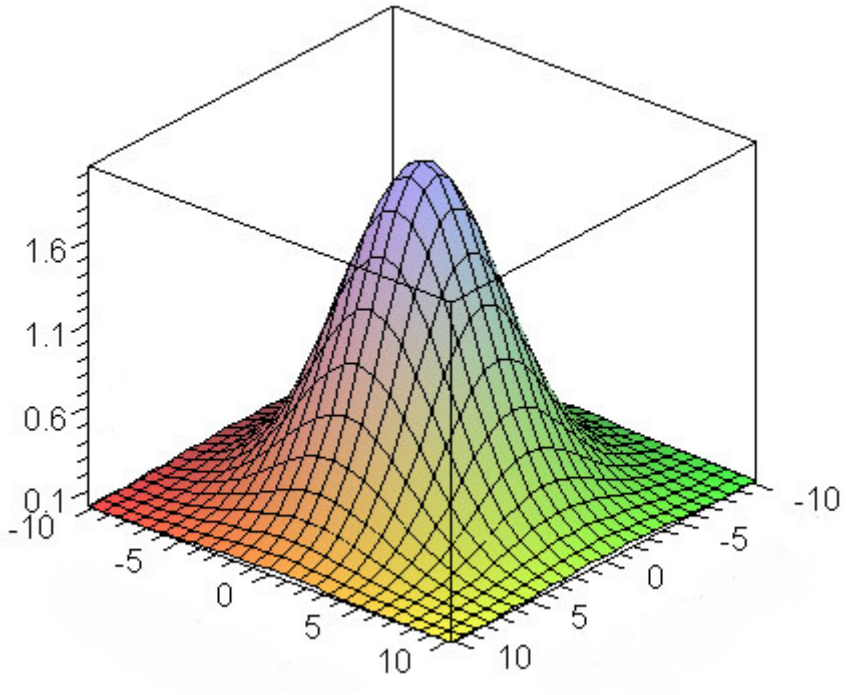


Abbildung 5.4: Grafische Darstellung der zweidimensionalen Gauß-Funktion[Ras10]

> Originalbild mit den Pixeln im Kernel multipliziert, die sich überlappen. Die aus diesen Multiplikationen resultierenden Werte werden addiert und dieses Ergebnis wird für den Wert am Zielpixel verwendet.¨[Ber21].

Je größer der Kernel ist, desto größer ist der Radius der Unschärfe und die Berechnungsdauer. Der σ-Wert der in Abbildung 5.3 aufzeigten Gauß'schen Formel bestimmt die Standardabweichung der X- und Y-Richtung. Bei einem Wert von 0 wird die Standardabweichung anhand der Kernelgröße berechnet [Ope21f]. Nach der Minimierung des Rauschens kann anschließend die Hintergrundsubtraktion angewandt werden.

```
1 # Application of background subtraction
2 fgmask = fgbg.apply(blur_frame)
```

Listing 5.4: Anwendung der Hintergrundsubtraktion

Das Ergebnis der Hintergrundsubtraktion ist eine Vordergrundmaske, wie in Abbildung 5.3 zu sehen. Die weiß markierten Stellen stellen den Vordergrund dar, wohingegen die schwarzen Stellen den Hintergrund hervorheben.

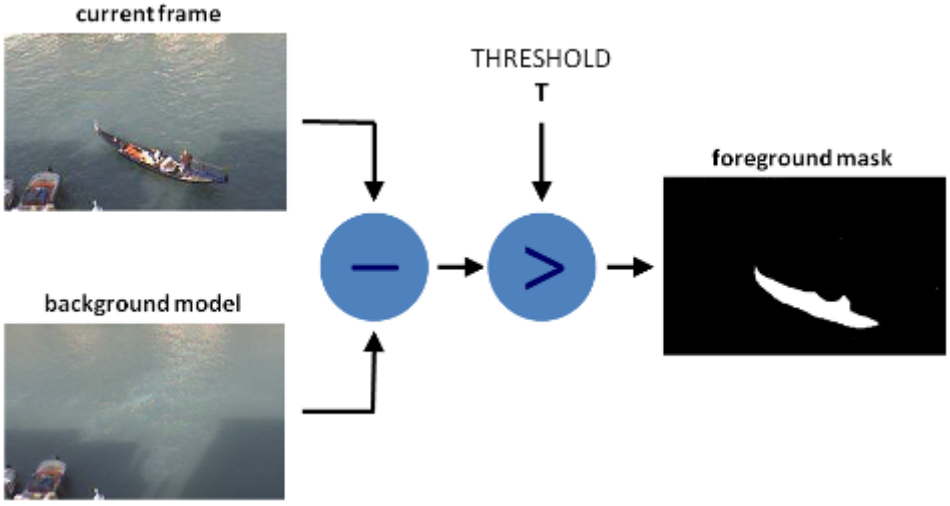


Abbildung 5.5: Erzeugung einer Vordergrundmaske

Nachdem die beweglichen Elemente aus dem Video herauskristallisiert werden können, kann der Video Input für die Objekterkennung vorbereitet werden. Um dieses Ziel erfüllen zu können, wird die Vordergrundsubtraktion eingesetzt.

```
# Application of foreground dilation
dilated_frame = cv2.dilate(src=fgmask, kernel=np.ones((3,3),np.uint8),
    iterations=3)
```

Listing 5.5: Dilation der Vordergrundmaske

Eine Erweiterung, auch Dilation genannt, ist Bestandteil der mathematischen Morphologie. Die Morphologie ist eine Untergruppe der Bildverarbeitung. Dabei befasst man sich mit binären Bildern bzw. Rastergrafiken, also Bildern mit nur einem oder zwei Farbwerten [Soi98]. Durch die Dilation der Vordergrundmaske werden kleine schwarze Flecken zwischen weißen Flächen verdeckt, sowie die weißen Flächen größer und zu einer Region zusammengefasst. In Abbildung **??** sind bereits größere Strukturen von Objekten zu erkennen, dennoch sind ungewollte, kleinere Flächen sichtbar. Diese werden im nächsten Schritt gefiltert und entfernt. Um die kleineren weißen Flecken zu entfernen werden zunächst alle Konturen der digitalen Vordergrundmaske gesucht, wie in Abbildung **??** dargestellt.

```
# Contour detection of foreground mask
contours, _ = cv2.findContours(image=dilated_frame, mode=cv2.
    RETR_EXTERNAL, method=cv2.CHAIN_APPROX_SIMPLE)
```

Listing 5.6: Finden aller Konturen in der Vordergrundmaske

Konturen können als eine Kurve erklärt werden, die alle kontinuierlichen Punkte entlang einer Begrenzung verbindet, die die gleiche Farbe oder Intensität besitzen [Ope21c]. Mit Hilfe der *findContours* Funktion wird eine Liste aller gefundenen Konturen zurückgegeben. Die Kontur ist wiederum eine Liste von Koordinatenpunkten. Der Modus *RETR_EXTERNAL* gibt nur die extremen Außenkonturen, die eine Schmälerung der gefundenen Knoten mit sich bringt, zurück. Die Methode *CHAIN_APPROX_SIMPLE* komprimiert horizontale, vertikale und diagonale Segmente und lässt deren Endpunkte zurück [Ope21g]. Um die Konturen auf das Video anzuwenden muss zunächst eine Maske des Videos erstellt werden (siehe Abbildung 5.7).

```
# Create mask
mask = np.zeros_like(frame)
```

Listing 5.7: Erstellung einer Maske des aktuellen Bildes

Unter Verwendung der *zeroes_like* Funktion des numpy Packages kann ein Array mit der Form und Größe des Bildes zurückgegeben werden. Dieses Array enthält nur Nullen und ist somit ein schwarzes Bild [Num21]. Die im Anschluss entstandenen Konturen können nun auf eine vorbestimmte Größe gefiltert werden. Hierfür wird die *contourArea* Funktion verwendet (siehe Abbildung 5.8). Mit Hilfe eines Grenzwerts unter 1200 können kleinere, nicht relevante Konturen gefiltert und Objekte wie Personen oder Fahrräder auf der Straße erkannt werden.

```
# Draw rectangles around contours
for contour in contours:

    # Calculate the contour area. Skips small countours
    if cv2.contourArea(contour) < 1200:
        continue

    # Returns corners of contour. Doesn't consider the rotation of the
        object
    (x, y, w, h) = cv2.boundingRect(contour)

    # draw filled contours in mask
    cv2.rectangle(img=mask, pt1=(x, y), pt2=(x+w, y+h), color
        =(255,255,255), thickness=-1)
```

Listing 5.8: Erstellen einer Konturenmaske

Nach der Filterung der Konturen können die geraden Begrenzungsrechtecke mit Hilfe der *boundingRect* Funktion ermittelt werden. Als gerades Begrenzungsrechteck ist ein Rechteck

gemeint, das die Rotation eines Objekts außer Acht lässt. Diese Rechtecke werden als weiße Rechtecke auf die zuvor ermittelte Maske aufgetragen. Der Parameter *thickness* füllt das Rechteck mit der angegebenen Farbe aus, sofern dieser den Wert -1 beinhaltet.

Abschließend wird eine bitweise Verrechnung der Maske mit den im Video befindlichen Frames vorgenommen. Durch den ∩-Operator werden lediglich die weißen Stellen des Bildes hervorgehoben (siehe Abbildung **??** und **??**). Um diese Operation anwenden zu können müssen beide Bilder die selbe Größe und Form besitzen (siehe Abbildung 5.9).

```
1 # Bitwise and operation
2 movement_frame = cv2.bitwise_and(frame, mask)
```

Listing 5.9: UND-Operation der Maske und des aktuellen Bildes

Abbildung **??** zeigt das Ergebnis der Bewegungserkennung. Es werden nur Objekte, die sich im Video bewegen und groß genug für die Erkennung sind, angezeigt. Dies kann anschließend für die Objekterkennung verwendet werden.

5.2 Objekterkennung

Um unterschiedliche Verkerhsteilnehmer wie Fahrräder, Autos und Motorräder wahrzunehmen, müssen diese Identifiziert und Erkannt werden. Unter Verwendung der Tensorflow Object Detection API wird die Objekterkennung in diesem Prototyp umgesetzt [Git21]. Die Tensorflow Object Detection API ist ein Open-Source Framework, welches auf Tensorflow aufbaut und das Konstruieren, Trainieren und Bereitstellen von Objekterkennungsmodellen erleichtert. Alle im Tensorflow Repository enthaltenen Modelle wurden anhand des Common Objects in Context (COCO) 2017 Datensatzes trainiert. COCO ist eine umfangreiche Objekterkennungs-, Segmentierungs- und Beschriftungsdatensatz Bibliothek mit über 330.000 Bildern und 91 Suchkategorien. Diese Modelle sind vor allem für Out-of-the-Box Inferenz nützlich, wenn die gewünschten Kategorien in den Datensätzen vorhanden sind. Objekte wie Fahrräder, Autos, Motorräder, LKWs und Busse sind bereits in den COCO Datensätzen enthalten. Dies führt zu dem Anlass, ein bereits vortrainiertes Modell zu verwenden. Jedes Modell wird mit Hilfe der Geschwindigkeit in Millisekunden und der COCO Mean Average Precision (MAP) gemessen. Die Geschwindigkeit gibt an, wie lange ein Modell zur Erkennung von Objekten benötigt. Der COCO-MAP ist eine weitverbreitete Metrik zur Messung der Genauigkeit eines Modells. Je höher der MAP Wert, desto genauer ist das Modell. Die Geschwindigkeit und der MAP stehen häufig in Relation zueinander. Auf Grund dieser Merkmale muss das zu wählende Modell mit

den Anforderungen des Projektes zusammenpassen. Für die Echtzeiterkennung wird ein schnelles Modell mit geringer Genauigkeit benötigt. Für diesen Prototypen wurde sich für das Modell mit dem höchsten COCO-MAP Wert entschieden. Diese Wahl bringt eine durchaus hohe Einbuße der Erkennungsgeschwindigkeit mit sich, welche jedoch auf Grund der Verzichtbarkeit einer Echtzeiterkennung nicht weiter relevant ist. Neben Boxen (Umrandung des Objekts) können Modelle ebenfalls Keypoints zurückgeben, die für die weitere Kompatibilität mit Prototypen relevant ist. Bei der Keypoint Erkennung werden Personen erkannt und gleichzeitig deren Keypoint lokalisiert. Keypoints sind räumliche Orte bzw. Punkte im Bild, die definieren, was interessant ist oder im Bild hervorsticht. Sie sind invariant gegenüber Bilddrehung, -Schrumpfung, -Verschiebung und -Verzerrung [Pap21].

Abbildung 5.6: Keypoint Erkennung [1]

Für diesen Prototypen wird das EfficientDet-D7 Modell verwendet, das mit einem COCO-MAP Wert von 51,2 im Tensorflow Model Zoo ideal dazu geeignet ist, um die markantesten Objekte in einem Bild zu lokalisieren [Tea21l]. EfficientDet ist die Objekterkennungs Variante des weitverbreiteten EfficientNet [TL19] und baut auf dessen Erfolg auf. EfficientNets stammen aus einer Familie von Modellen, die eine hohe Leistung bei Benchmark-Aufgaben erreichen, während sie für eine Reihe von Effizienzparametern wie Modellgröße und Floating Point Operation Per Seconds (FLOPs) kontrolliert werden. Das Netzwerk wird in einer Reihe von Modellgrößen d0-d7 ausgeliefert. FLOPs bemisst die Anzahl an Gleitkommaoperationen pro Sekunde und ist ein Maß für die Leistungsfähigkeit eines Modells [Hui21].

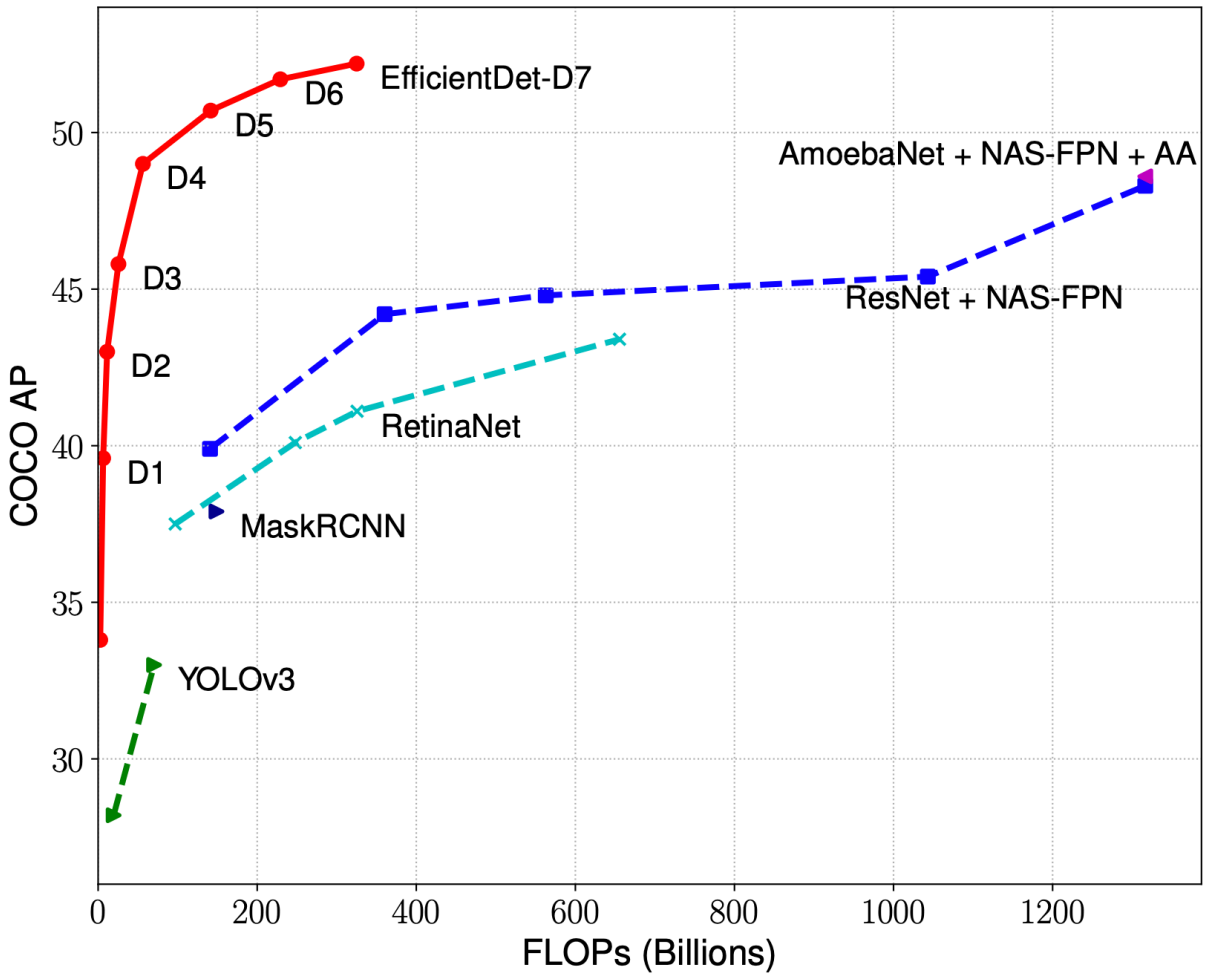


Abbildung 5.7: Modell FLOPs vs. COCO Genauigkeit [TPL20]

EfficientDet erreicht die beste Leistung, mit der geringsten Anzahl von Trainingsepochen, unter den Objekterkennungsmodellen. Dies ist insbesondere von Vorteil, wenn mit begrenzter Rechenleistung gearbeitet wird [Sol20]. Bevor das vortrainierte Modell verwendet werden kann, wird dieses als verpackte Datei heruntergeladen. Ebenfalls enthalten ist die zugehörige Beschreibungsdatei, welche eine Liste von Strings beinhaltet, die zur richtigen Beschriftung eines Objekts dient. Jede Beschriftung ist durch eine ID, die später zur Visualisierung verwendet wird, eindeutig erkennbar. Nach dem Herunterladen des Modells wird dieses extrahiert und mit der Beschreibungsdatei geladen [Vla21].

```
1 category_index = label_map_util.create_category_index_from_labelmap(
      use_display_name=True)
2 # Load pipeline config and build a detection model
3 configs = config_util.get_configs_from_pipeline_file(PATH_TO_CFG)
4 model_config = configs['model']
5 detection_model = model_builder.build(model_config=model_config,
      is_training=False)
```

Listing 5.10: Laden der Beschreibungsdatei mit anschließendem Erzeugen des Modells

Das Modell enthält zusätzlich eine Pipeline Konfigurationsdatei, welche zur Erzeugung des Modells verwendet wird. Eine Pipeline definiert Konfigurationen für ein Modell und automatisiert dadurch den Workflow des zu bauenden Modells. Pipelines für maschinelles Lernen bestehen aus mehreren aufeinanderfolgenden Schritten, die von der Datenextraktion und Vorverarbeitung bis hin zum Modelltraining und der Bereitstellung reichen [Val21].

```
# Restore checkpoint
ckpt = tf.compat.v2.train.Checkpoint(model=detection_model)
ckpt.restore(os.path.join(PATH_TO_CKPT, 'ckpt-0')).expect_partial()
```

Listing 5.11: Laden der Checkpoints

Checkpoints erfassen den genauen Wert aller von einem Modell verwendeten Parameter [Tea21m]. Sodurch kann ein Modell wiederhergestellt und verwendet werden. Um nun eine Objekterkennung durchführen zu können, erwartet das EfficientDet-D7 Modell ein Dreikanalbild variabler Größe. Die Eingabewerte benötigen einen Tensor mit einer Höhe von 1 und einer Breite von 3 und zugehörigen Werten zwischen 0 und 255 [Tea21l]. Auf Grund dieser Anforderung wird das Bild mit Hilfe der Numpy Funktion *expand_dims* auf die erwartete Größe skaliert und in einen Tensor umgewandelt. Anschließend kann die Objekterkennung ausgeführt werden.

```
# Expand dimensions since the model expects images to have shape: [1,
    None, None, 3]
image_np_expanded = np.expand_dims(frame, axis=0)
input_tensor = tf.convert_to_tensor(image_np_expanded, dtype=tf.float32)
```

Listing 5.12: Anpassung des Eingabetensors an das Modell

```
# Execute object detection
detections, predictions_dict, shapes = detect_fn(input_tensor)
```

Listing 5.13: Ausführung der Objekterkennung

Die Funktion *detect_fn* gibt alle notwendigen Angaben als Python Dictionary zurück. Für den Prototypen werden die *detection_boxes*, *detection_scores* und *detection_classes* benötigt. *Detection_boxes* ist ein Tensor welcher die Koordinaten für 100 minimal umgebene Rechtecke, sogenannte Bounding Boxes, beinhaltet. Anhand dieser Koordinaten kann im späteren Verlauf der Objekterekennung die Umrandung der Objekte gezeichnet werden.

```
'detection_boxes': <tf.Tensor: shape=(1, 100, 4), dtype=float32, numpy=
 array([[[0.4823483 , 0.33256257, 0.8408261 , 0.6648078 ],
         [0.39217225, 0.3756986 , 0.5196357 , 0.50812906],
         [0.482716  , 0.3359049 , 0.84076875, 0.6666065 ],
         [0.6159764 , 0.5550573 , 0.6771329 , 0.59031546],
         [0.39108938, 0.39384803, 0.43105263, 0.4248545 ]
```

Listing 5.14: Beispielausschnitt der detection boxes

Das zugehörige Objekt kann anhand des Indizes in der *detection_class* gefunden werden.

```
'detection_classes': <tf.Tensor: shape=(1, 100), dtype=int32, numpy=
 array([[ 2,  2,  7,  0,  0,  0,  2,  2,  7,  7,  2,  7,  2,  7,  2,  2,
          0, 26,  7,  5,  2,  7,  0,  2,  7,  0,  2, 30,  0,  0,  2,  2,
          0,  2,  5,  7,  2,  0, 36,  7,  7,  0,  2,  0,  9,  7, 13,  3,
         76,  7, 36, 61,  2,  2, 61,  7, 26, 61,  7,  2, 32, 13, 13,  0,
         17,  0, 32,  5,  5, 76, 27,  7,  7, 32, 26,  2,  7,  2,  9, 26,
         27,  0,  7, 13, 32, 33,  3,  2, 63, 43, 30, 13, 43,  1,  7, 13,
         30,  9,  2,  7]], dtype=int32)>,
```

Listing 5.15: Beispielausschnitt der detection classes

Jede Zahl in dem *detection_classes* Tensor ist eine ID der in Auflistung 5.10 geladenen Beschreibungsdatei. Ebenso kann anhand des Indizes die Genauigkeit der Erkennung in dem *detection_scores* Objekt abgefragt werden (siehe Auflistung 5.16).

```
'detection_scores': <tf.Tensor: shape=(1, 100), dtype=float32, numpy=
 array([[0.65809727, 0.49682707, 0.32241914, 0.191383  , 0.16080539,
```

Listing 5.16: Beispielausschnitt der detection scores

Das Dictionary *detections* enthält weitere, nicht benötigte Ausgabe Parameter, die für diesen Prototypen jedoch irrelevant sind [Tea21l]. Nicht alle Objekte die vom Modell erkannt werden, sind für die Objekterkennung notwendig. Daher werden die erkannten Objekte anhand der Klassen-ID gefiltert und aussortiert. Somit erhält man die Umrandungen, Klassen und Genauigkeiten eines einzelnen Bildes für die gewünschten Klassen. Die Visualisierung der Daten erfolgt in einem späteren Schritt.

5.3 Geschwindigkeitserkennung

5.4 Fahrtrichtungserkennung

6 Evaluation des Prototypen

6.1 Metriken zur Bewertung der Klassifikation

6.2 Optimierung des neuronalen Netzes

6.3 Evaluierung der Ergebnisse

7 Abschluss

7.1 Fazit

7.2 Ausblick

Literatur

[Ata+19] A Ata et al. „Modelling smart road traffic congestion control system using machine learning techniques“. In: *Neural Network World* 29.2 (2019), S. 99–110.

[Ber21] Bert. *Wie wird der Gauß'sche Weichzeichner implementiert?* 2021. URL: https://qastack.com.de/computergraphics/39/how-is-gaussian-blur-implemented. (abgerufen am 22.05.2021).

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Berlin, Heidelberg: Springer, 2006. ISBN: 978-0-387-31073-2.

[Bro18] Jason Brownlee. *How to Configure the Number of Layers and Nodes in a Neural Network*. 2018. URL: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/.

[CC08] Matthieu Cord und Pádraig Cunningham. *Machine Learning Techniques for Multimedia - Case Studies on Organization and Retrieval*. Berlin Heidelberg: Springer Science und Business Media, 2008. ISBN: 978-3-540-75171-7.

[Cha17] Vaibhaw Singh Chandel. *Selective Search for Object Detection*. 2017. URL: https://learnopencv.com/selective-search-for-object-detection-cpp-python/. (abgerufen am 06.05.2021).

[Col21] Google Colab. *Willkommen bei Colaboratory - Colaboratory*. 2021. URL: https://colab.research.google.com/notebooks/intro.ipynb#scrollTo=OwuxHmxllTwN. (abgerufen am 01.02.2021).

[Deo18] Chris Deotte. *How to choose CNN Architecture*. 2018. URL: https://www.kaggle.com/cdeotte/how-to-choose-cnn-architecture-mnist#1.-How-many-convolution-subsambling-pairs?. (abgerufen am 10.05.2021).

[DH+73] Richard O Duda, Peter E Hart et al. *Pattern classification and scene analysis*. Bd. 3. Wiley New York, 1973.

[DN17] Suguna Devi und T Neetha. „Machine Learning based traffic congestion prediction in a IoT based Smart City“. In: *Int. Res. J. Eng. Technol* 4 (2017), S. 3442–3445.

[DN19] Matthieu Deru und Alassane Ndiaye. *Deep Learning mit TensorFlow, Keras und TensorFlow.js*. Bonn: Rheinwerk, 2019. ISBN: 978-3-836-26509-6.

[E+05] Atkočiūnas E. et al. „Image Processing in Road Traffic Analysis“. In: *Nonlinear Analysis: Modelling and Control* 10.4 (Okt. 2005), S. 315–332. DOI: 10.15388/NA.2005.10.4.15112. URL: https://www.journals.vu.lt/nonlinear-analysis/article/view/15112.

[Ea17] Peter Eckersley und Yomna Nasser et al. *AI Progress Measurement*. 2017. URL: https://www.eff.org/ai/metrics.

[Elf19] Sharif Elfouly. *R-CNN (Object Detection)*. 2019. URL: https://medium.com/@selfouly/r-cnn-3a9beddfd55a. (abgerufen am 06.05.2021).

[FH98] Pedro F Felzenszwalb und Daniel P Huttenlocher. „E ciently computing a good segmentation“. In: *DARPA Image Understanding Workshop*. Citeseer. 1998.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning (Adaptive Computation and Machine Learning series)*. Cambridge: MIT Press, 2016. ISBN: 978-0-262-03561-3.

[Gér17] Aurélien Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow : Konzepte, Tools und Techniken für intelligente Systeme* -. Sebastopol: O'Reilly, 2017. ISBN: 978-3-960-09061-8.

[Gir+13] Ross B. Girshick et al. „Rich feature hierarchies for accurate object detection and semantic segmentation“. In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[Git21] Github. *Tensorflow/Models - GitHub*. 2021. URL: https://github.com/tensorflow/models/tree/master/research/object_detection. (abgerufen am 01.06.2021).

[Ham18] Ben Hamner. *Popular Datasets Over Time*. 2018. URL: https://www.kaggle.com/benhamner/popular-datasets-over-time/code.

[Hui21] Jonathan Hui. *Object detection - speed and accuracy comparison*. 2021. URL: https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359. (abgerufen am 02.06.2021).

[Ima12] ImageNet. *ImageNet Large Scale Visual Recognition Competition*. 2012. URL: http://www.image-net.org/challenges/LSVRC/2012/results.html. (abgerufen am 01.02.2021).

[Jup21a] Project Jupyter. *Jupyter Kernels - jupyter/jupyter*. 2021. URL: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels. (abgerufen am 01.02.2021).

[Jup21b] Project Jupyter. *Project Jupyter - Home*. 2021. URL: https://jupyter.org/. (abgerufen am 01.02.2021).

[Kar20] Michael Dr. Karl. *Grundlagen Maschineller Lernverfahren - Convolutional Neural Networks*. DHBW Stuttgart. 2020.

[KB17] Diederik P. Kingma und Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[Kri09a] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[Kri09b] Alex Krizhevsky. *The CIFAR-10 Dataset*. 2009. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[KSH12] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems*. Hrsg. von F. Pereira et al. Bd. 25. Curran Associates, Inc., 2012, S. 1097–1105. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[Mai97] Klaus Mainzer. „Komplexität neuronaler Netze“. In: *Gehirn, Computer, Komplexität*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, S. 143–161. ISBN: 978-3-642-60524-6. DOI: 10.1007/978-3-642-60524-6_9. URL: https://doi.org/10.1007/978-3-642-60524-6_9.

[MP43] Warren McCulloch und Walter Pitts. „The Linear theory of Neuron Networks: The Static Problem“. In: *Bulletin of Mathematical Biology* 4.4 (1943), S. 169–175.

[MW18] Aditya Krishna Menon und Robert C Williamson. „The cost of fairness in binary classification“. In: *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*. Hrsg. von Sorelle A. Friedler und Christo Wilson. Bd. 81. Proceedings of Machine Learning Research. New York, NY, USA: PMLR, 23–24 Feb 2018, S. 107–118. URL: http://proceedings.mlr.press/v81/menon18a.html.

[NMS95] International Workshop on Artificial Neural Networks, Jose Mira und Francisco Sandoval. *From Natural to Artificial Neural Computation - International Workshop on Artificial Neural Networks, Malaga-Torremolinos, Spain, June 7-9, 1995 : Proceedings*. Berlin Heidelberg: Springer Science und Business Media, 1995. ISBN: 978-3-540-59497-0.

[Num21] Numpy. *numpy.zeroeslike - NumPy Manual*. 2021. URL: https://numpy.org/doc/stable/reference/generated/numpy.zeros_like.html. (abgerufen am 22.05.2021).

[Ope21a] OpenCV. *OpenCV: Background Subtraction*. 2021. URL: https://docs.opencv.org/3.4/de/df4/tutorial_js_bg_subtraction.html. (abgerufen am 22.05.2021).

[Ope21b] OpenCV. *OpenCV: Color conversions*. 2021. URL: https://docs.opencv.org/master/de/d25/imgproc_color_conversions.html#color_convert_rgb_gray. (abgerufen am 22.05.2021).

[Ope21c] OpenCV. *OpenCV: Contours : Getting Started*. 2021. URL: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. (abgerufen am 22.05.2021).

[Ope21d] OpenCV. *OpenCV: How to Use Background Subtraction Methods*. 2021. URL: https://docs.opencv.org/master/d1/dc5/tutorial_background_subtraction.html. (abgerufen am 22.05.2021).

[Ope21e] OpenCV. *OpenCV: Motion Analysis*. 2021. URL: https://docs.opencv.org/3.4/de/de1/group__video__motion.html#ga2beb2dee7a073809ccec60f145b6b29c. (abgerufen am 22.05.2021).

[Ope21f] OpenCV. *OpenCV: Smoothing Images*. 2021. URL: https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html. (abgerufen am 22.05.2021).

[Ope21g] OpenCV. *OpenCV: Structural Analysis and Shape Descriptors*. 2021. URL: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html. (abgerufen am 22.05.2021).

[Pap21] PapersWithCode. *Keypoint Detection - Papers with Code*. 2021. URL: https://paperswithcode.com/task/keypoint-detection. (abgerufen am 02.06.2021).

[Ras+16] Mohammad Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. 2016. arXiv: 1603.05279 [cs.CV].

[Ras10] Rastergrid. *Efficient Gaussian blur with linear sampling*. 2010. URL: https://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/. (abgerufen am 22.05.2021).

[RK14] R.Y. Rubinstein und D.P. Kroese. *The Cross-Entropy Method*. Springer, 2014. ISBN: 9781475743227. URL: https://books.google.de/books?id=7hj4sgEACAAJ.

[Rod21] Dr. Raul Rodriguez. *Computer Vision: Write Your Motion Detection Code Using OpenCV*. 2021. URL: https://analyticsindiamag.com/computer-vision-write-your-motion-detection-code-using-opencv/. (abgerufen am 22.05.2021).

[Rus+15] Olga Russakovsky et al. *ImageNet Large Scale Visual Recognition Challenge*. 2015. arXiv: 1409.0575 [cs.CV].

[SD20] Luca Stanger und Florian Drinkler. *Image Classification using a Convolutional Neural Network*. 2020. URL: https://github.com/lucastanger/rock_paper_scissors.

[Sej99] Howard Hughes Medical Institute Computational Neurobiology Laboratory Terrence J Sejnowski. *Unsupervised Learning - Foundations of Neural Computation*. Cambridge: MIT Press, 1999. ISBN: 978-0-262-58168-4.

[Soi98] Pierre Soille. „Grundlagen“. In: *Morphologische Bildverarbeitung: Grundlagen, Methoden, Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, S. 15–49. ISBN: 978-3-642-72190-8. DOI: 10.1007/978-3-642-72190-8_2. URL: https://doi.org/10.1007/978-3-642-72190-8_2.

[Sol20] Jacob Solawetz. *Training EfficientDet Object Detection Model with a Custom Dataset*. 2020. URL: https://towardsdatascience.com/training-efficientdet-object-detection-model-with-a-custom-dataset-25fb0f190555. (abgerufen am 02.06.2021).

[Sri+14] Nitish Srivastava et al. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15.56 (2014), S. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Stu14] David Stutz. „Introduction to Neural Networks“. In: *RWTH Aachen University* (März 2014).

[Tea17] The TensorFlow Team. *Google Developers Blog: Introduction to TensorFlow Datasets and Estimators*. 2017. URL: https://developers.googleblog.com/2017/09/introducing-tensorflow-datasets.html. (abgerufen am 09.03.2021).

[Tea21a] The Tensor Flow Team. *SparseCategoricalCrossentropy - TensorFlow Core v2.5.0*. Mai 2021. URL: https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy.

[Tea21b] The Tensor Flow Team. *tf.keras.layers.Conv2D - TensorFlow Core v2.5.0*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D.

[Tea21c] The TensorFlow Team. *Introduction to Tensors - TensorFlow Core*. 2021. URL: https://www.tensorflow.org/guide/tensor. (abgerufen am 06.05.2021).

[Tea21d] The TensorFlow Team. *TensorFlow - Random Brightness*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_brightness.

[Tea21e] The TensorFlow Team. *TensorFlow - Random Contrast*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_contrast.

[Tea21f] The TensorFlow Team. *TensorFlow - Random Flip Left Right Function*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_flip_left_right.

[Tea21g] The TensorFlow Team. *TensorFlow - Random Flip Up Down Function*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_flip_up_down.

[Tea21h] The TensorFlow Team. *TensorFlow - Random Hue*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_hue.

[Tea21i] The TensorFlow Team. *TensorFlow - Random Saturation*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/random_saturation.

[Tea21j] The TensorFlow Team. *TensorFlow - Random Uniform*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/random/uniform.

[Tea21k] The TensorFlow Team. *TensorFlow - Rotate 90 Degrees*. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/image/rot90.

[Tea21l] The TensorFlow Team. *TensorFlow Hub - EfficientDet*. 2021. URL: https://tfhub.dev/tensorflow/efficientdet/d7/1. (abgerufen am 02.06.2021).

[Tea21m] The TensorFlow Team. *Trainingskontrollpunkte - TensorFlow Core*. 2021. URL: https://www.tensorflow.org/guide/checkpoint. (abgerufen am 02.06.2021).

[TL19] Mingxing Tan und Quoc Le. „Efficientnet: Rethinking model scaling for convolutional neural networks“. In: *International Conference on Machine Learning*. PMLR. 2019, S. 6105–6114.

[TPL20] Mingxing Tan, Ruoming Pang und Quoc V Le. „Efficientdet: Scalable and efficient object detection“. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, S. 10781–10790.

[Uij+13] J. R. R. Uijlings et al. „Selective Search for Object Recognition“. In: *International Journal of Computer Vision* 104.2 (2013), S. 154–171. URL: https://ivi.fnwi.uva.nl/isis/publications/2013/UijlingsIJCV2013.

[Val21] Valohai. *What is a Machine Learning Pipeline?* 2021. URL: https://valohai.com/machine-learning-pipeline/. (abgerufen am 02.06.2021).

[Vla21] Lyudmil Vladimirov. *Object Detection From TF2 Saved Model*. 2021. URL: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/auto_examples/plot_object_detection_saved_model.html. (abgerufen am 02.06.2021).

[Wan03] Sun-Chong Wang. „Artificial Neural Network“. In: *Interdisciplinary Computing in Java Programming*. Boston, MA: Springer US, 2003, S. 81–100. ISBN: 978-1-4615-0377-4. DOI: 10.1007/978-1-4615-0377-4_5. URL: https://doi.org/10.1007/978-1-4615-0377-4_5.

[WB20] J. Wang und A. Boukerche. „The Scalability Analysis of Machine Learning Based Models in Road Traffic Flow Prediction“. In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. 2020, S. 1–6. DOI: 10.1109/ICC40277.2020.9148964.

[Yan20] Wei Qi Yan. *Computational Methods for Deep Learning - Theoretic, Practice and Applications*. Singapore: Springer Nature, 2020. ISBN: 978-3-030-61081-4.

[Ziv04] Z. Zivkovic. „Improved adaptive Gaussian mixture model for background subtraction“. In: *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.* Bd. 2. 2004, 28–31 Vol.2. DOI: 10.1109/ICPR.2004.1333992.

[Zv06] Zoran Zivkovic und Ferdinand van der Heijden. „Efficient adaptive density estimation per image pixel for the task of background subtraction“. In: *Pattern Recognition Letters* 27.7 (2006), S. 773–780. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2005.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0167865505003521.

Glossar

Adam

Erweiterung des RMSProp Optimizers.

AlexNet

Ein von Alexander Krizhevsky entworfenes CNN.

Bias

Unabhängiges Gewicht eines neuronalen Netzes.

Bin

Klasse in einem Histogram mit konstanter oder variabler Breite.

Bounding Box

Begrenzungsrahmen für die Formkompatibilität.

Hyperparameter

Wert zur Steuerung des Lernprozesses.

Kernel

Ein Programm, das den Code des Anwenders ausführt und introspektiert.

Multi-Class Classification

Klassifizierungsmodell zur Erkennung einzelner Klassen in einem Bild.

Multi-Label Classification

Klassifizierungsmodell zur Erkennung multipler Labels in einem Bild.

Numpy

Programmierbibliothek in Python für die einfache Handhabung von Vektoren, Matrizen und mehrdimensionalen Arrays.