

# Hochschule Reutlingen

## Reutlingen University

– Studiengang Mechatronik Bachelor –

Bachelor–Thesis

## Entwicklung eines autonomen Systems zur Bilderkennung mithilfe Neuronaler Netze auf dedizierter Hardware

Manuel Barkey  
Pestalozzistraße 29  
72762 Reutlingen

Matrikelnummer : 762537

Betreuer: Eberhard Binder  
Zweitbetreuer: Christian Höfert  
Abgabedatum: 12.03.2020





## **Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst angefertigt habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Sämtliche Textausschnitte, Zitate oder Grafiken anderer Verfasser sind als solche gekennzeichnet.

Reutlingen, den 10. März 2020

.....

## **Danksagungen**

...



# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>5</b>
<b>2 Grundlagen</b>	<b>7</b>
2.1 Machine Learning . . . . .	7
2.1.1 Künstliche Neuronale Netze . . . . .	7
2.1.2 Validierung und Overfitting . . . . .	11
2.1.3 Machine Learning Frameworks . . . . .	12
2.2 Convolutional Neural Networks . . . . .	13
2.2.1 Architekturen . . . . .	14
2.2.2 Objekterkennung . . . . .	16
2.3 Neural Compute Stick 2 . . . . .	17
2.3.1 OpenVino Toolkit . . . . .	17
<b>3 Realisierung Objekterkennung</b>	<b>19</b>
3.1 Datensatz . . . . .	19
3.1.1 Augmentierung . . . . .	20
3.2 Object detection Modelle . . . . .	21
3.3 Training . . . . .	22
3.3.1 Tensorflow Object Detection Api . . . . .	23
3.4 Inferenz . . . . .	23
<b>4 Evaluierung</b>	<b>25</b>
4.1 Evaluierungs Metriken . . . . .	25
4.2 Vergleich der Modelle . . . . .	27
4.2.1 Evaluierung . . . . .	27
4.2.2 Test Inferenz . . . . .	28
4.3 Optimierungen: Faster R-CNN . . . . .	30
4.3.1 Verschiedene Augmentierungen . . . . .	30
4.3.2 Verschiedene Regularisierungen . . . . .	32
4.4 Inferenzzeit . . . . .	34
4.4.1 Synchrone- und Asynchrone Inferenz . . . . .	34
4.4.2 Vergleich der Modelle . . . . .	35
<b>5 Entwicklung der Anwendung</b>	<b>37</b>
5.1 Hardware . . . . .	37
5.2 Software . . . . .	38
<b>6 Zusammenfassung und Ausblick</b>	<b>43</b>



# Abkürzungsverzeichnis

**API (Application Programming Interface)** Eine Application Programming Interface ist ein bestimmter Satz an Regeln über die ein Programm funktionalitäten anderer Software verwenden kann.. 18, 22, 41

**CNN (Convolutional Neural Network)** Neuronales Netz, welches math. Faltung Verwendet, meist in der Bilderkennung verwendet.. 5, 7, 13–15, 17, 21, 23, 27, 34, 43

**CPU (Central Processing Unit)** zentrale Recheneinheit eines Computers. 17

**CSI (Camera Serial Interface)** Kamerataschnittstelle des Raspberry Pi's welche ein Flachbandkabel verwendet. 37

**Downsampling** In der Bildverarbeitung: kleiner skallieren eines Bildes. 14

**Feature Map** eine zweidimensionale Matrix, die durch die Faltung eines Bild mit einer Filtermatrix entsteht. 13, 21

**Fps (Frames per Second)** Bilder, die pro Sekunde verarbeitet werden können. 35

**Framework** Programmgerüst, bestehend aus mehreren Programmen, Schnittstellen, Tools, und weiterem, zur erleichterten Software Entwicklung. 12, 13, 17, 22, 43

**GPU (Graphics Processing Unit)** Prozessor, der auf die Berechnungen von Grafiken spezialisiert ist. 17, 22

**Künstliche Neuronale Netze** Eine Form des Machine Learning, bei der eine Vielzahl an künstlichen Neuronen miteinander verbunden sind, um aus gegebenen Eingaben die richtige Ausgabe zu erhalten. 7, 13

**Machine Learning** Ein Bereich der künstlichen Intelligenz, der sich mit selbstlernenden Algorithmen beschäftigt. 7, 8, 12, 47

**Overfitting** Überanpassung eines Machine Learning Modells an die Trainingsdaten, wodurch keine Generalisierung mehr erreicht wird. 11, 12, 15, 27, 28, 32

**SCP (Secure Copy Protocol)** zum senden von Daten über SSH. 41

**SMTP (Smart Mail Transfer Protokoll)** zum senden von Emails. 41

**SoC (System on Chip)** komplexes System, bestehend aus CPU, GPU und weiterem auf einem Chip verbaut.. 17

**SSH (Secure Shell Protocol)** Kommunikationsprotokoll. 40, 41

**Thred** wie Prozess. 35

**VM (Virtual Machine)** Virtuelle Nachbildung einer Rechnerarchitektur, welche innerhalb eines lauffähigen Rechnersystems ausgeführt wird. 22

**VPU (Vision Processing Unit)** Mikroprozessor für Bildverarbeitungsaufgaben häufig in KI-Beschleunigern eingesetzt. 17

**Zero Padding** Auffüllen mit nullen. 14

# Kapitel 1

## Einleitung

Wildkameras kommen in der Jagt oder zu biologischen Forschungszwecken zum Einsatz. Da das Aufnehmen der Bilder automatisch über einen Bewegungsmelder erfolgt, kann es unter Umständen zu einer großen Menge an unwichtigen Daten kommen. Diese benötigen Speicherplatz auf dem Gerät, oder, bei automatischem Senden, Datenvolumen für eine Mobile Netzwerkverbindung und bringen außerdem einen großen Auswertungsaufwand mit sich.

Ein System welches genau erkennt, um welche Tiere es sich bei den gemachten Aufnahmen handelt, kann wesentlich effizienter und gezielter für eine bestimmte Anwendung eingesetzt werden. Beispielsweise zur Überwachung des eigenen Gartens vor Füchsen, oder zum Aufspüren von Wölfen und Bären in bestimmten Waldgebieten. Auch der Artbestand seltener oder aussterbender Tierarten könnte so leichter erfasst werden. Für die Umsetzung einer solchen Bilderkennungsaufgabe werden *Deep Learning* Algorithmen verwendet, welche ein Teilgebiet der künstlichen Intelligenz sind. Für die Bilderkennung werden zumeist Convolutional Neural Networks (CNNs) verwendet. Durch die Fortschritte, die in diesem Bereich in den letzten Jahren gemacht wurden, sowie durch die Verfügbarkeit leistungsfähiger und zugleich kostengünstiger Hardware, ist die Realisierung einer solchen Anwendung auch für den Privatgebrauch und ohne Verwendung eines Großrechners möglich geworden.

Ziel der vorliegenden Arbeit war es, ein autonomes Kamerasystem zu entwickeln, welches mithilfe von Deep Learning Algorithmen, verschiedene Wildtierarten erkennen und klassifizieren kann. Dieses soll auf einem Raspberry Pi 4 laufen und Bilder von erkannten Tieren automatisch an den Nutzer senden. Die Inferenz der Neuronalen Netze wird dabei auf dem KI Beschleuniger Neural Compute Stick 2 von Intel ausgeführt. Durch verwendung einer infrarotfähigen Kamera soll die Erkennung des Systems auch bei Nacht stattfinden können.

Damit gliedert sich die Arbeit zunächst in ein Grundlagen-Kapitel, welches die Funktionsweise von künstlichen Neuronalen Netzen für die Bilderkennung behandelt. Anschließend wird es um die Umsetzung und Auswertung des Trainings geeigneter Deep Learning Modelle gehen.

Der letzte Teil beschreibt de Entwicklung der Anwendung, in welcher die Inferenz eines fertig trainiertes Modell, für den Neural Compute Stick, implementiert wird.



# Kapitel 2

## Grundlagen

In vorliegendem Kapitel werden zunächst die Grundlagen des Machine Learnings, mit Künstlichen Neuronalen Netzen, beschrieben. Der zweite Abschnitt wird die, in der Bilderkennung eingesetzten, CNNs näher erläutern. Im letzten Abschnitt wird die, für die Inferenz, verwendete Hardware, der *Neural Compute Stick 2* von *Intel* beschrieben werden.

### 2.1 Machine Learning

Beim Machine Learning, welches ein Teilgebiet der Computerwissenschaften ist, geht es um die Erstellung von Algorithmen, die Zusammenhänge in großen Datenmengen erkennen können, ohne explizit darauf programmiert worden zu sein. Eine Form davon ist das *Supervised Learning*, bei dem das Programm neben den Inputdaten auch die Zugehörigen Ausgaben, in Form von Labels, erhält, um daraus Regeln für die Zusammenhänge zwischen Ein- und Ausgabe Daten abzuleiten. Dadurch unterscheidet sich das Vorgehen wesentlich zur Programmierung eines klassischen Programms, bei dem die Regeln vorab definiert werden müssen, wie Abbildung 2.1 veranschaulicht.

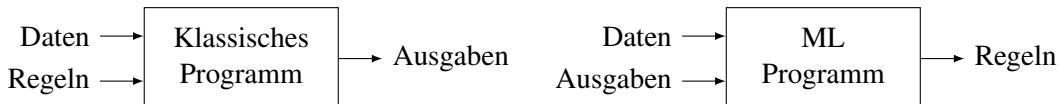


Abbildung 2.1: Vergleich herkömmliche- und Machine Learning- Programmierung

Das Ableiten der Regeln erfolgt beim Machine Learning in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei werden die Zusammenhänge zwischen Ein- und Ausgabe Daten als mathematische Funktion, die numerisch an die richtigen Werte angehähert wird, betrachtet.

Handelt es sich um einen linearen Zusammenhang der Daten, spricht man von einer *Regression*, wohingegen bei einer Kategorisierung von diskreten Werten von einer *Klassifikation* gesprochen wird.

Weitere Formen neben dem *Supervised Learning* sind das *Unsupervised Learning*, bei dem das Programm keine Labels erhält, sondern diese durch das Training selber finden soll, oder das *Reinforcement Learning*, bei dem das Programm, durch Interaktion mit der Umwelt, bestimmte Verhaltensmuster lernt. Da in der Bachelor Arbeit nur mit dem *Supervised Learning* gearbeitet wurde, wird auf diese Techniken im folgenden nicht näher eingegangen.

#### 2.1.1 Künstliche Neuronale Netze

Für komplexe Inputdaten, wie beispielsweise Bilder, bei denen die einzelnen Pixelwerte die Eingaben und der Inhalt des Bildes die gesuchte Ausgabe darstellen, werden meistens Künstlichen Neuronalen Netzen

verwendet. Diese sind eine Form des Machine Learnings und bestehen aus einer vielzahl an, programmatisch erzeugten künstlicher Neuronen, die in Schichten angeordnet, miteinander verbunden sind. Durch unterschiedlich starke Gewichtungen der einzelnen Verbindungen, welche auch als Gewichte bezeichnet werden, können, wie in Abbildung 2.2 schematisch dargestellt ist, zu gegebene Eingaben die richtigen Ausgaben gefunden werden.

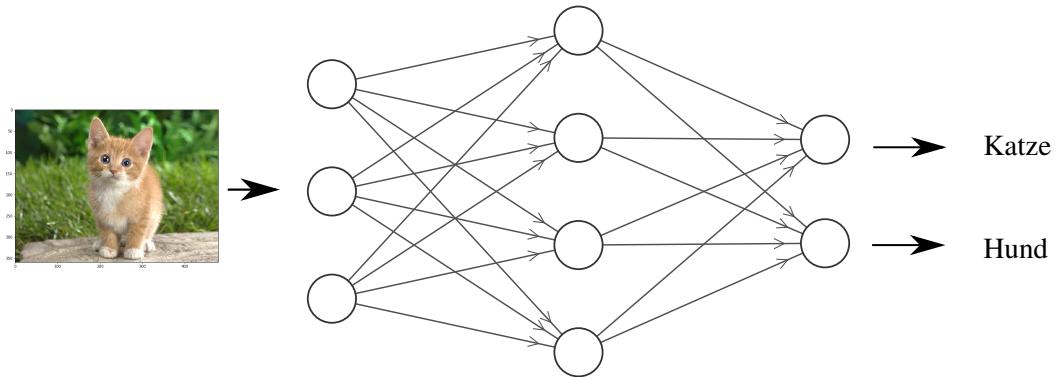


Abbildung 2.2: Vereinfachte Darstellung eines Künstlichen Neuronalen Netzes

Die richtigen Einstellungen der Gewichte erfolgt dabei in dem iterativen Trainingsprozess, welcher aus folgenden drei schritten besteht und in Abbildung 2.3 schematisch dargestellt ist.

- *Forward Pass* Für die Inputdaten anhand, aktueller Gewichte, eine Schätzung für die Ausgabe treffen
- *Fehlerbestimmung* Abweichung der gemachten Schätzung, zum tatsächlichen Wert, berechnen
- *Backpropagation* Minimierung der Abweichung, durch Veränderung der Gewichte

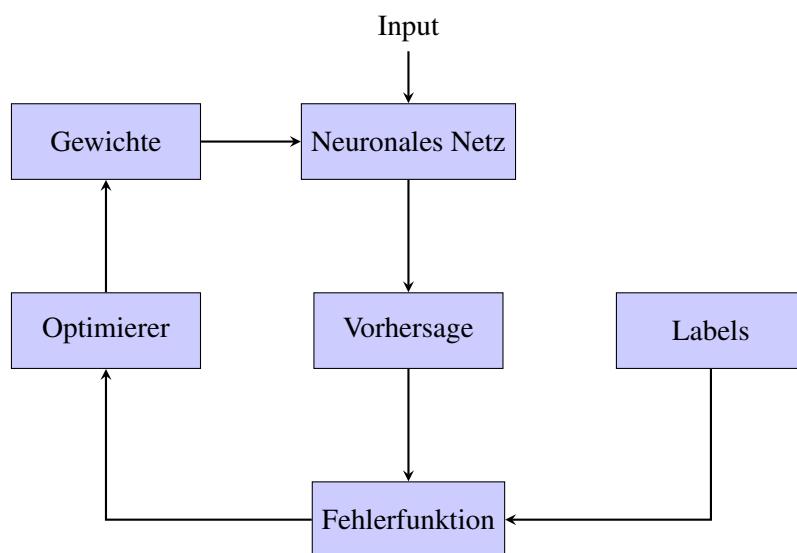


Abbildung 2.3: Schematischer Trainingsablauf eines Neuronalen Netzes

Durch mehrfaches Durchlaufen dieser Schritte, kann die Fehlerfunktion soweit minimiert werden, dass das Modell auch für neue, ungewohnte Input Daten richtigen Vorhersagen treffen kann. Die Funktionsweisen der drei Schritte werden im folgenden näher erläutert.

### Forward Pass

Im *Forward Pass* werden die Eingabewerte, welche an der ersten Schicht aus Neuronen anliegen, durch alle Schichten hindurchgereicht, um in der Ausgabeschicht Schicht das gesuchte Ergebnis zu liefern. Dabei erhält jedes Neuron, die mit den Parametern  $w_i$  gewichteten, Ausgabewerte aller Neuronen der vorherigen Schicht und summiert diese zusammen mit einem konstanten Bias Wert  $b$ , als Offset, auf.

Mithilfe einer Aktivierungsfunktion wird der Wert, wie in Abbildung 2.4 dargestellt ist, auf einen bestimmten Bereich skaliert.

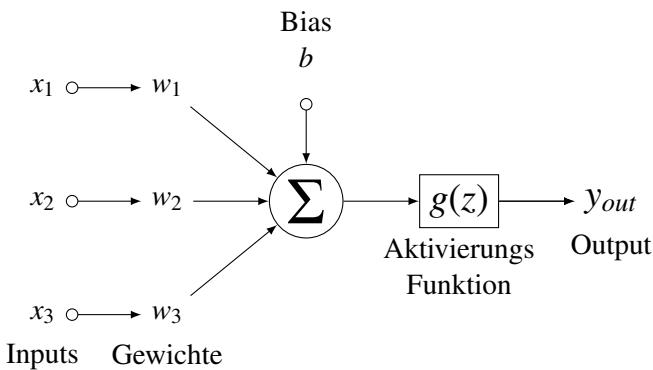


Abbildung 2.4: Berechnungen an einem einzelnen Neuron

Um diesen Vorgang für eine gesamte Schicht, bestehend aus einer Vielzahl an Neuronen, zu berechnen, werden die Schichten als Vektoren  $x$  und die Gewichte als Matrizen  $W$  dargestellt.

Durch Bilden der Matrixmultiplikation zwischen  $x$  und  $W$ , wie Gleichung 2.1 zeigt, erhält man den *Forward Pass* von einer Schicht zur nächsten.

$$z = W^T x + b \quad (2.1)$$

Der resultierende Vektor  $z$  wird dann elementweise einer nichtlinearen Aktivierungsfunktion  $g(z)$  übergeben. Bei dieser handelt es sich, für die mittleren Schichten, den sogenannten *Hidden Layers*, meist um die in Abbildung 2.5 dargestellte Funktion *ReLU*, welche positive Werte beibehält und negative Werte zu 0 setzt. In der letzten Schicht, welche die Wahrscheinlichkeiten für mögliche Ausgaben enthält, wird eine Aktivierungsfunktion verwendet, die den Wert zwischen 0 und 1 skaliert. Dabei wird für eine binären Klassifikation die in Abbildung 2.6 dargestellte Funktion *Sigmoid* verwendet, welche die Werte S-Förmig zwischen 0 und 1 skaliert.

Für eine kategoriale Klassifikation, mit mehr als zwei möglichen Ausgabewerten, wird die in Gleichung 2.2 dargestellte Funktion *Softmax* verwendet, welche eine Wahrscheinlichkeitsverteilung über alle Werte der Ausgabeschicht generiert.

$$g(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad (2.2)$$

$$g(z) = \max\{0, z\}$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

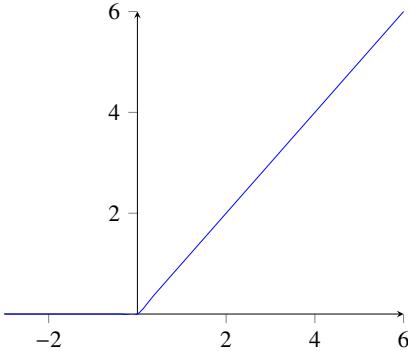


Abbildung 2.5: ReLU Funktion

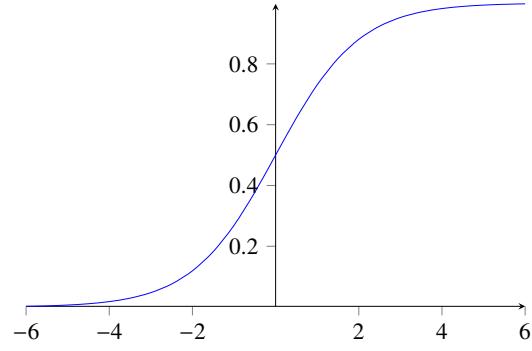


Abbildung 2.6: Sigmoid Funktion

### Fehlerbestimmung

Die Abweichung des geschätzten Wertes, welcher an den Neuronen der letzten Schicht anliegt, zum tatsächlichen Wert, wird mithilfe einer geeigneten Fehler- oder auch Lossfunktion ermittelt. Ein Regressionsmodell verwendet hierbei oft den absoluten oder quadratischen Abstand der beiden Werte, wohingegen für Klassifikationsmodelle meist der Logarithmus verwendet wird. In Gleichung 2.3 ist die logarithmische Lossfunktion *Cross entropy*, für eine Binäre Klassifikation dargestellt. Durch den Logarithmus wird der Loss um so größer, je weiter die Schätzung  $\hat{y}$  vom tatsächlichen Wert  $y$  abweicht.

$$L = \hat{y} \log(y) + (1 - \hat{y}) \log(1 - y) \quad (2.3)$$

### Backpropagation

Die Anpassung der Gewichte, zur Minimierung der Lossfunktion, kann durch Berechnung des Gradienten dieser, erfolgen.

Dafür wird die Fehlerfunktion  $L$ , für jede Schicht, partiell nach den Gewichten  $w$  der Schicht abgeleitet, was, wie in Gleichung 2.4 dargestellt, mithilfe der Kettenregel geschieht. Mit dem ermittelten Gradienten werden dann die Gewichte nach Gleichung 2.5 mit einer Schrittweite  $\eta$  angepasst.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial w} \quad (2.4)$$

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \quad (2.5)$$

## 2.1.2 Validierung und Overfitting

Um überprüfen zu können, ob und wie gut, ein Modell die Zusammenhänge in den Trainingsdaten generalisiert hat, also auch für neue Daten anwendbar ist, wird der Datensatz in einen Trainings- und einen Testdatensatz aufgeteilt.

Die Abweichung wird während des Trainings für beide Datensätze berechnet, die Korrektur der Gewichte, mittels Backpropagation, erfolgt jedoch nur anhand der Trainingsdaten.

Indem beide Lossfunktionen als Funktion, in Abhängigkeit der Iterationen, geplottet werden, kann eine Überanpassung des Modells an die Trainingsdaten, festgestellt werden.

Dabei handelt es sich um das sogenannte *Overfitting*, was daran zu erkennen ist, dass sich nur noch der Wert des Trainingsloss verringert und der der Testdaten gleich bleibt oder sich wie in Abbildung 2.7 dargestellt, wieder verschlechtert.

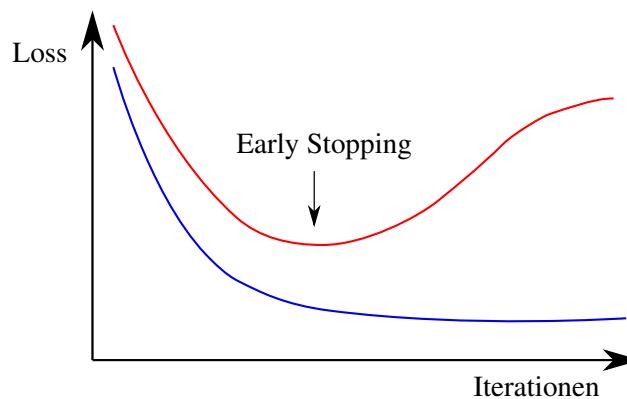


Abbildung 2.7: Overfitting, anhand der Losskurven

Gründe für das Overfitting können sein, dass zu wenig Trainingsdaten verwendet wurden, oder ein, für den Anwendungsfall, zu komplexes Modell gewählt wurde.

Durch die Überparametrisierung eines zu komplexen Modells hat dieses die Möglichkeit sich an jeden Trainingsdatenpunkt einzeln anzupassen, sodass keine generalisierbaren Aussagen für neue Datenpunkte mehr getroffen werden können.

Der andere Extremfall ist das *Underfitting*, bei dem das Modell, aufgrund zu weniger Parameter, nicht die Möglichkeit hat, sich an die Trainingsdaten anzunähern.

Die Plots in Abbildung 2.8 veranschaulichen die drei Fälle anhand einer polynomiauen Funktionen, die sich an gegebene Datenpunkte, mit unterschiedlich hohem Grad, annähern soll.

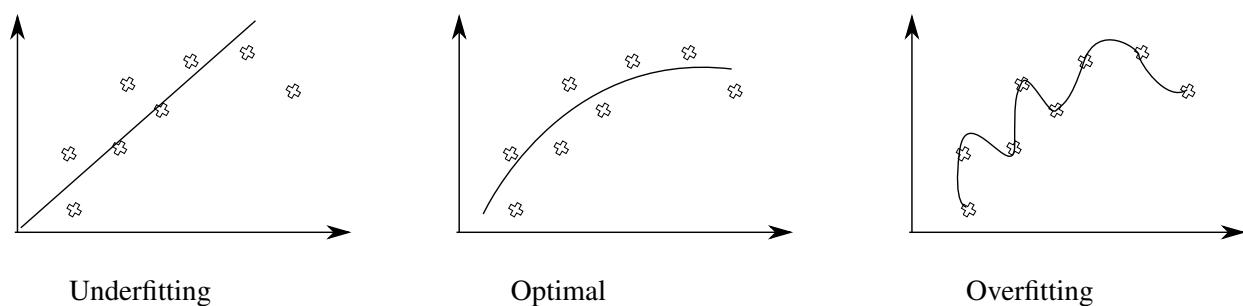


Abbildung 2.8: Annäherung unterschiedlich komplexer Modelle an die gleichen Datenpunkte

Das Auftreten von Overfitting kann entweder durch Verwendung einer größeren Anzahl an Trainingsdaten, oder mit einer der folgenden Techniken, vermieden werden.

### Augmentierung

*Augmentierung* der Daten ist eine effektive Technik gegen Overfitting, bei der künstlich, aus den vorhandenen Daten, mehr Daten generiert werden. Im Fall der Bilderkennung werden hierbei die Inputbilder leicht abgeändert, indem z.B. geometrische Transformationen oder Manipulationen der Pixelwerte vorgenommen werden.

### Regularisierung der Parameter

Bei der *Regularisierung* wird der Lossfunktion, als weiterer Term, eine Aufsummierung aller Gewichte hinzugefügt.

Die Minimierung der Lossfunktion hat dann auf die Gewichte den Effekt, dass diese möglichst kleine Werte behalten, wodurch das Modell weniger Möglichkeit zur Überanpassung hat.

Dabei wird zwischen der *L1 Regularisierung*, mit einer absoluten, und der in Gleichung 2.6 dargestellten *L2 Regularisierung*, mit einer quadratischen Aufsummierung der Gewichte unterschieden.

$$J = L + \lambda \sum_i w_i^2 \quad (2.6)$$

### Dropout

*Dropout* ist eine Technik, bei der in einigen Schichten mit einer bestimmten Wahrscheinlichkeit Werte von Neuronen zu 0 gesetzt werden. Dadurch wird das Modell gezwungen alternative Gewichtsanpassungen zu finden.

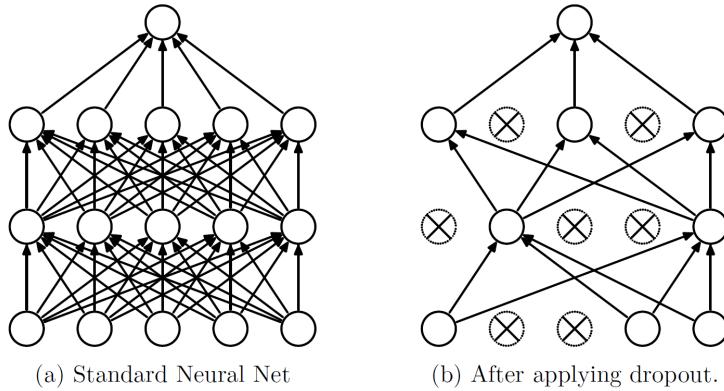


Abbildung 2.9: Prinzip Dropout [1]

### Early Stopping

Beim *Early Stopping* wird das Training, bevor Overfitting stattfinden kann, abgebrochen. Das ist, wie in Abbildung 2.7 markiert, durch das Minimum der Losskurve der Testdaten definiert.

### 2.1.3 Machine Learning Frameworks

Machine Learning Algorithmen beinhalten eine Vielzahl an komplexen Berechnungsschritten und Parametern. Um diese nicht jedesmal von Grund auf neu implementieren zu müssen, bieten Frameworks eine einfache Möglichkeit die Modelle zu konstruieren.

Einige der bekannten Open Source Frameworks sind Tensorflow, Caffe, Torch, Kaldi oder Scikit-Learn.

Für die Bachelor Arbeit wurde Tensorflow verwendet, ein von Google stammendes Framework, welches aufgrund seiner hohen Flexibilität besonders in der Forschung oft verwendet wird.

## 2.2 Convolutional Neural Networks

CNNs erweitern die in Abschnitt 2.1.1 beschriebenen Künstlichen Neuronalen Netzen um zusätzliche Schichten, die vor der eigentlichen Klassifikation ausgeführt werden und Merkmale aus den Input Daten herausextrahieren. Diese Schichten erhalten die Input Daten als zweidimensionale Matrix und führen darauf mathematische Faltungsoperationen aus. CNNs kommen größtenteils in der Bilderkennung zum Einsatz, weitere Anwendungsgebiete sind z.B. die Spracherkennung.

Anstatt das alle Neuronen zweier benachbarter Schichten durch gewichtete Parameter miteinander verbunden sind, stellen kleinere, sogenannte Filter Matrizen, die Parameter dar. Diese werden zeilenweise über das Inputbild geschoben, wobei an jeder Stelle eine mathematische Faltung mit dem überlappten Bereich des Inputs durchgeführt wird. In Abbildung 2.10 ist dieser Vorgang zur Veranschaulichung dargestellt.

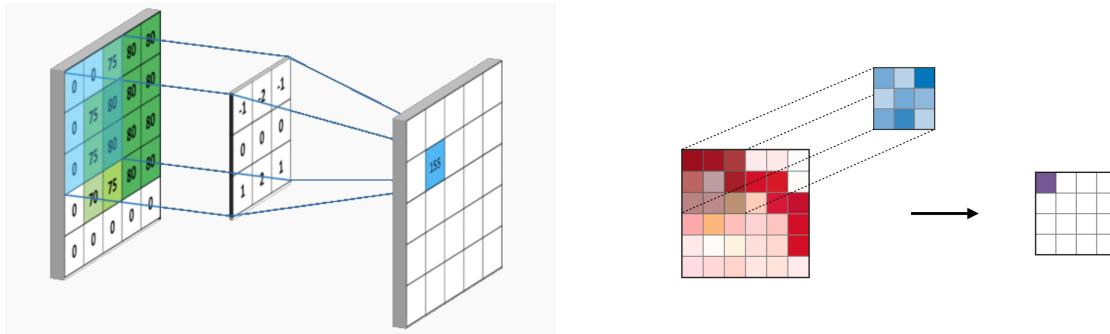


Abbildung 2.10: Faltung des Inputs mit einer Filter Matrix, Quellen: [2] und [3]

Jedes Faltungsergebnis, ergibt einen Wert für die nächste Schicht, die dadurch ein Korrelationsverhältnis zwischen Filter Matrix und Input Bild erhält. So werden, in den Filtern definierte Muster, wie beispielsweise vertikale Linien, aus dem Inputbild herausextrahiert und in den Folgeschichten, als sogenannte Feature Maps abgebildet.

$$\begin{pmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{pmatrix} \quad (2.7)$$

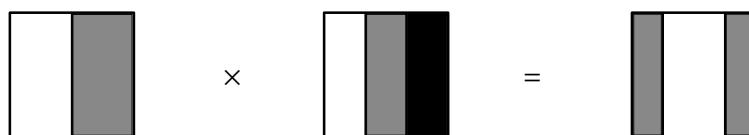


Abbildung 2.11: Faltung mit Filtermatrix zur Erkennung vertikaler Linien

In Gleichung 2.7 ist beispielhaft die Faltung eines Inputbildes mit einem Filter, zur Erkennung vertikaler Linien, dargestellt. Da pro Zeile vier Faltungen angewendet werden, entsteht eine  $4 \times 4$  Matrix. Soll die Ausgangsgröße (hier  $6 \times 6$ ) beibehalten werden, kann *Zero Padding* verwendet werden. Durch die Faltung entsteht eine räumliche Invarianz für das zu erkennende Objekt im Input Bild.

Den *Convolutional Layern* folgen meist *Pooling Layer*, zum Downsampling, und eine ReLU-Aktivierungsfunktion. Beim Pooling wird eine bestimmte Anzahl an Werten zusammengefasst, indem entweder das Maximum oder der Mittelewert dieser Werte verwendet werden.

Durch hintereinanderschaltung mehrerer solcher *Convolutional Blöcke*, können in jeder Schicht immer komplexere Muster aus dem Inputbild herausextrahiert werden.

Die Features des Letzten *Convolutional Layers* werden dann einem *Fully Connected Layer* zur Klassifikation übergeben, wie in Abbildung 2.12 anhand des ersten, von Yann LeCun entworfenen, CNNs zu erkennen ist.

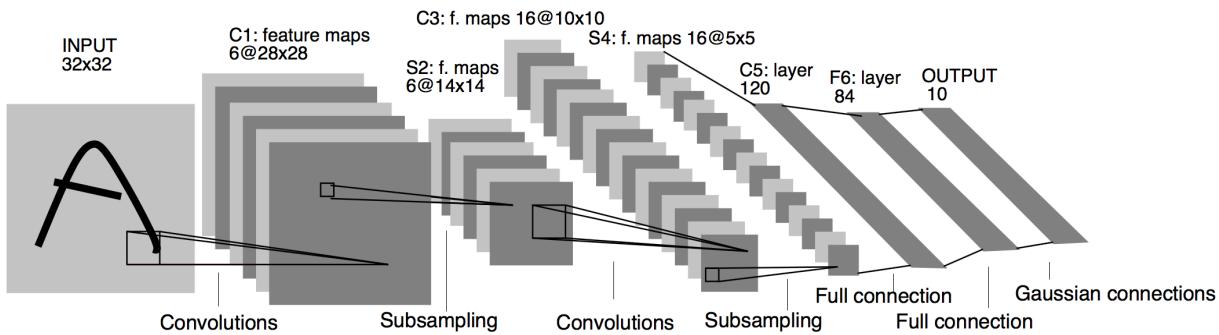


Abbildung 2.12: LeNet-5 Architektur [4]

Ein wesentlicher Vorteil gegenüber einem reinen *Feedforward Network* ist, dass durch die gemeinsame Nutzung von Parametern, durch die Filter, ein geringerer Rechenaufwand entsteht.

Die Werte der Filter Matrizen, welche die zu extrahierenden Muster darstellen, werden über die *Backpropagation* eingelernt.

Da die Merkmale, insbesondere in den vorderen *Convolutional Layern*, für die meisten Klassen sehr ähnlich sind, werden häufig Modelle mit vortrainierten Filtern verwendet.

Durch das sogenannte *Transfer Learning* müssen die Gewichte dann nur noch leicht, für den eigenen Datensatz, angepasst werden.

## 2.2.1 Architkturen

Nachdem 1998 das erste CNN (Abbildung 2.12) von Yann LeCun in [4] vorgestellt wurde, gab es eine Vielzahl an Weiterentwicklungen, welche genauere und effizientere Modelle hervorbrachten.

Gemessen und verglichen werden die Ergebnisse häufig an der *Large Scale Visual Recognition Challenge (ILSVRC)* [5].

Namhafte Modelle, welche die Challenge in den letzten Jahren gewinnen konnten sind unter [6] zu finden und im Folgenden aufgelistet.

- **AlexNet**, (2012), von Alex Krizhevsky [7] besitzt eine ähnliche Struktur wie LeCuns *LeNet*, ist jedoch tiefer und besitzt mehrere *Convolutional Layer* am Stück hintereinander, wodurch die Genauigkeit erhöht wurde.
- **ZF Net**, (2013), von Matthew Zeiler and Rob Fergus, [8] konnte das *AlexNet* durch eine Vergrößerung der mittleren *Convolutional Layer* und eine Verkleinerung der Filter in den vorderen Schichten weiter

optimieren.

- **VGGNet**, (2014), von Karen Simonyan and Andrew Zisserman [9]. Dieses Modell zeigte, dass ein tieferes Netz (16 bis 19 *Convolutional Layer*) mit reduzierter Filter Größe ( $3 \times 3$ ) bessere Ergebnisse erzielt.
- **GoogleLeNet**, auch bekannt als *Inception*, (2014), von Szegedy et al [10], konnte mit den *Inception Modulen*, welche im Folgenden genauer erläutert werden, die Zahl der Parameter, und dadurch den Rechenaufwand, deutlich verringern.
- **ResNet**, (2015) von Kaiming He et al [11], enthält als Erweiterung die *Residual Blöcke*, in denen auf das Ergebnis eines Blocks zusätzlich der unveränderte Input Wert addiert wird.

### GoogleLeNet (Inception)

Die Entwicklung der CNN Architekturen hat gezeigt, dass sich durch Hinzufügen weiterer Schichten, sowie der Verwendung einer größeren Anzahl an Neuronen je Schicht, die Genauigkeit verbessern lässt. Das bringt jedoch auch die Nachteile eines größeren Rechenaufwands sowie der erhöhten Gefahr des Overfittings mit sich.

Das in [10] vorgestellte *GoogleLeNet*, hat mit den in Abbildung 2.13 dargestellten *Inception Modulen*, einen neuen, effizienteren, Ansatz gefunden, die Komplexität und damit die Genauigkeit eines CNNs zu erhöhen. Die Module bestehen aus parallel ausgeführten *Convolutional Layern* mit den unterschiedlichen Filtergrößen  $1 \times 1$ ,  $3 \times 3$  und  $5 \times 5$  welche am Ende des Moduls über eine *Filter concatenation* wieder zusammengeführt werden. Zur Dimensionsreduktion werden, wie in Abbildung 2.13 dargestellt, diesen Filtern noch  $1 \times 1$  Filter vorgeschaltet. Durch die Inception Module kommt das Modell, für gleiche Ergebnisse, mit deutlich weniger Parametern aus, als ein Modell ohne die Module. Ein weiterer Vorteil ist, dass durch die unterschiedlichen Filtergrößen, Merkmale unterschiedlicher Skalierungen besser gefunden werden können. Um die Effizienz weiter zu Steigern, wurden in der zweiten Version des *GoogleLeNet*, beschrieben in [12], neben anderen Verbesserungen, die  $5 \times 5$  Filter, jeweils durch zwei  $3 \times 3$  Filter, ersetzt, was in Abbildung 2.14 dargestellt ist.

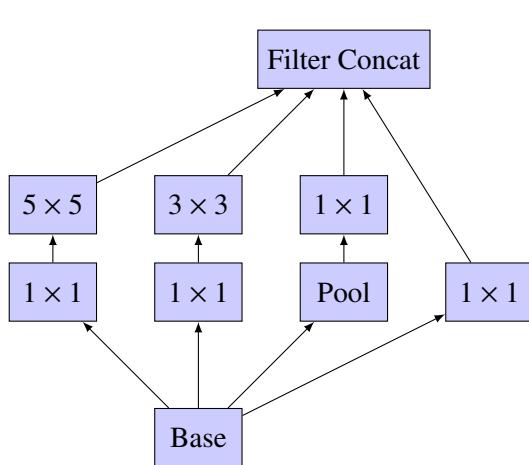


Abbildung 2.13: Inception Module der 1. Version

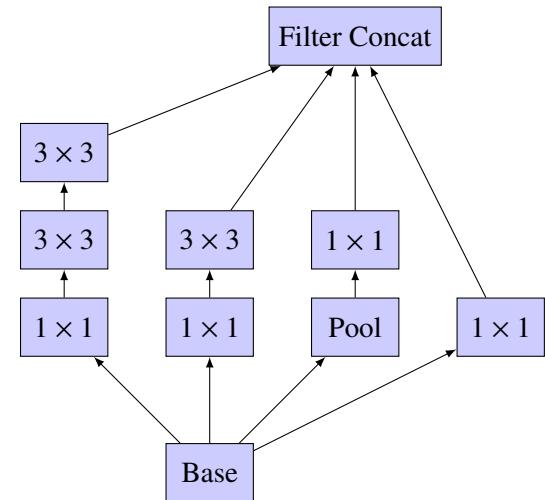


Abbildung 2.14: Inception Module der 2. Version

## Mobilenet

Das MobileNet [13] wurde mit dem Ziel geschaffen, durch eine geringere Komplexität, für Mobile Endgeräte oder Embedded-Anwendungen geeignet zu sein.

Dafür wurden die üblichen *Convolutional Layer* mit sogenannten *Depthwise Separable Convolutions* ersetzt, welche die Faltung in zwei separaten Schichten ausführt. Zuerst wird eine *Depthwise Convolutions* auf die drei Farbkanäle getrennt ausgeführt. Anschließend führt eine *pointwise convolution* mit  $1 \times 1$  Filter diese wieder zusammen.

In der zweiten Version des MobileNet [14], wurden die *Depth-wise Separable Convolutions* wie folgt abgeändert:

Zuerst wird eine  $1 \times 1$  Convolution mit ReLU Aktivierungsfunktion ausgeführt, anschließend die *Depthwise Convolutions*, gefolgt von einer weiteren  $1 \times 1$  mit linearer Aktivierungsfunktion.

Des Weiteren soll wie beim ResNet eine *residual connection*, welche Ein- mit Ausgabewert eines Blocks verbindet, den Gradientenfluss unterstützen, wie in Abbildung 2.15 dargestellt ist.

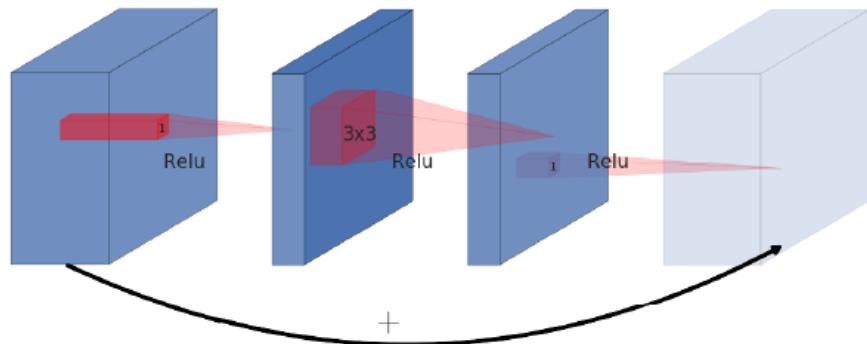


Abbildung 2.15: Residual block des MobileNetV2 [15]

### 2.2.2 Objekterkennung

Neben der Information, was sich auf einem Bild befindet, soll bei der Objekterkennung zusätzlich herausgefunden werden, wo sich das erkannte Objekt in dem Bild befindet. In Abbildung 2.16 wird der Unterschied veranschaulicht. Im linken Bild (Klassifikation) reicht es aus, dass das Modell das Vorhandensein einer Katze im Bild, mit einer bestimmten Wahrscheinlichkeit, schätzen kann, im rechten Bild (Objekterkennung), soll das Modell, in Form von *Bounding Boxen*, auch eine Lokalisierung vornehmen.



Abbildung 2.16: Unterschied: Klassifikation - Objekterkennung, Quelle: [16]

Dafür wird die CNN Architektur so erweitert, dass dem Modell für das Training neben den Klassen-Labels auch die Koordinaten der *Bounding Boxen*, welche das Objekt umrahmen, mit übergeben werden. Diese können dann, mittels Regressionsverfahren, durch Annäherung der geschätzten an die richtigen Koordinaten, gelernt werden.

Bei den Verfahren zur Objekterkennung gibt es verschiedene Ansätze, die alle ein Basis-CNN, zur *Feature Extraction* verwenden. Die Lokalisierung findet über eine Vorschlagsgenerierung statt, welche aus Regionen im Input Bild besteht, die am wahrscheinlichsten ein Objekt enthalten.

## 2.3 Neural Compute Stick 2

Da das Training und die Inferenz von Deep Learning Algorithmen sehr rechenintensiv ist, werden entsprechend leistungsfähige Prozessoren benötigt. Dabei ist die Ausführung auf einer Graphics Processing Unit (GPU) meist effizienter als auf einer Central Processing Unit (CPU).

Anwendungen die auf eingebetteten Systemen oder einplatinen Computern wie dem *Raspberry Pi* laufen, kommen dabei schnell an die Grenzen.

Eine Möglichkeit, dieses Problem zu umgehen, ist es, die Bilddaten für die Verrechnung an eine Cloud zu senden, wo sie von einem leistungsstärkeren Rechner inferiert und dann wieder zurückgesendet werden.

Sollen die Daten, wie es beim *Edge Computing* der Fall ist, auf dem Anwendungsgerät direkt verarbeitet werden, gibt es speziell für die Inferenz von Deep Learing Algorithmen geeignete Hardware. Durch Fokus auf hohe Parallelität, anstatt schneller Taktrate bei den Berechnungen, können solche Prozessoren Deep Learning spezifische Rechenoperationen, wie z.B. die Matrixmultiplikation, besonders effizient ausführen. Die Inferenzbeschleunigende Hardware kann dabei entweder als eigenständiges *System on Chip* System wie z.B. der *Nvidia Jetson TX2* agieren, oder in Verbindung mit einem *Host Pc*, wie der, in der Arbeit verwendete, *Neural Compute Stick 2* (NCS2) von *Intel*.

Der in Abbildung 2.17 gezeigte NCS2 verwendet für die Inferenz eine *Movidius Myriad X* Vision Processing Unit (VPU), welche in Abbildung 2.18 schematisch dargestellt ist.

Diese besteht, wie in [17] genauer beschrieben ist, aus der Neural Compute Engine, zur beschleunigten Berechnung Neuronaler Netze, einem Bildbeschleuniger, 16 SHAVE Prozessoren, einem Bildsignalprozessor sowie einem RISC CPU Core.



Abbildung 2.17: NCS2, [18]

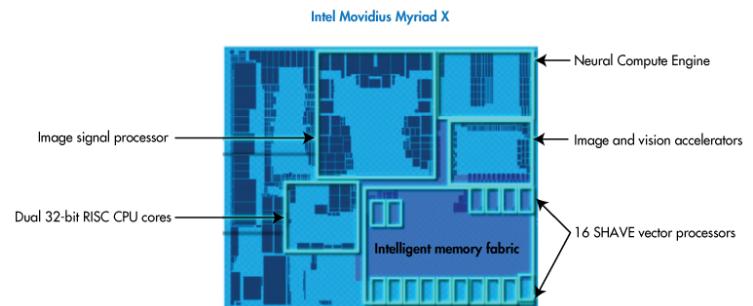


Abbildung 2.18: Myriad Chip, [19]

### 2.3.1 OpenVino Toolkit

Zur Ausführung der Inferenz, eines trainierten Deep Learing Modells, auf dem NCS2, wird das Toolkit *OpenVino* von Intel verwendet. Dieses ist eine Software Plattform zur Otimierung und Inferenz von CNN-basierten Modellen auf verschiedener Intel Hardware.

Dabei wird ein eigenes Dateiformat für die Modelle verwendet, die *Intermediate Representation* (IR), welche die Struktur des Modells in einer Xml-Datei und die trainierten Gewichte in einer Binärdatei definiert.

Mit dem *Model Optimizer* des Toolkits, können Modelle, welche in den den Frameworks *TensorFlow*, *Caffe*, *ONNX*, *Kaldi*, oder *MXNET* trainiert wurden, in das IR-Format konvertiert werden.

Um diese dann auf die entsprechende Hardware zu laden und anwendbar zu machen, wird die auch in *OpenVino* enthaltene *InferenceEngine* verwendet. Diese bietet eine Application Programming Interface (API), mit der aus der Anwendung heraus, in den Programmiersprachen C++ oder Python, auf die Funktionen der *InferenceEngine* zugegriffen werden können.

In Abbildung 2.19 ist der Workflow mit *Openvino*, welcher das Training eines Deep Learning Modells mit der Implementierung einer Nutzer Anwendung verbindet, dargestellt.

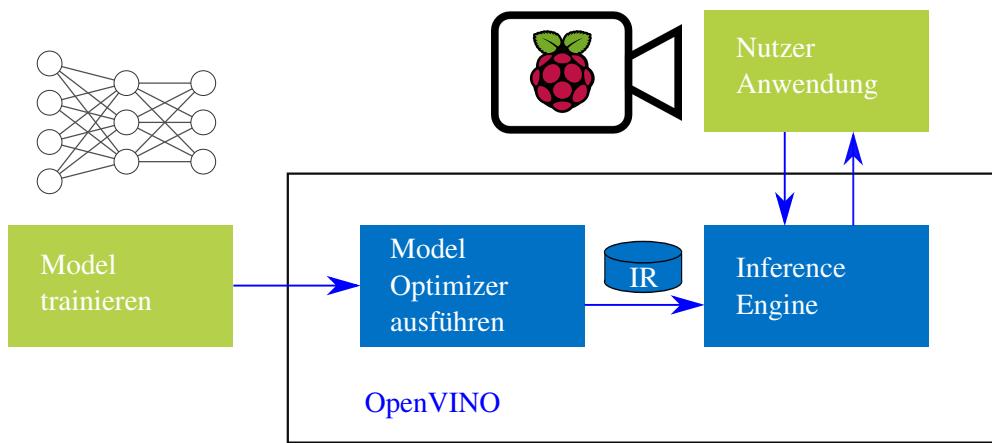


Abbildung 2.19: OpenVino Workflow, angelent an [20]

# Kapitel 3

## Realisierung Objekterkennung

In diesem Kapitel werden zunächst die Beschaffung und Aufbereitung der Trainingsdaten beschrieben. Anschließend wird es um das Training geeigneter Deep Learning Modelle in *Tensorflow* sowie um die Inferenz dieser in *OpenVino* gehen.

### 3.1 Datensatz

Für das Training eines Deep Learning Modells werden eine Große Menge an Trainingsdaten benötigt. Handelt es sich um ein Modell zur Objekterkennung, müssen die Labels neben der Klasse, auch die Koordinaten, der *Bounding Boxen* enthalten.

Die Trainingsdaten können entweder selber erstellt, oder aus frei zugänglichen Datensätzen wie z.B. *ImageNet*, *COCO*, oder *OpenImages* aus dem Internet heruntergeladen werden.

Für die Bachelor Arbeit wurden aus dem Open Source Datensatz *OpenImages* von Google [21], welches 600 gelabelte Klassen enthält, die 9 Klassen *Brown bear*, *Deer*, *Fox*, *Goat*, *Hedgehog*, *Owl*, *Rabbit*, *Raccoon* und *Squirrel* heruntergeladen und für das Training verwendet.

Für die Evaluierung des Trainings wurde der Datensatz, mit einem Verhältnis von 80%, 10%, 10%, in ein Trainings-, ein Validierungs- und ein Testset aufgeteilt.

Je Klasse variierte die Anzahl an Bildern zwischen 200 und 2000 Stück, wodurch eine Verteilung der Klassen, wie in Abbildung 3.1 dargestelltem Histogramm, zustande kam.

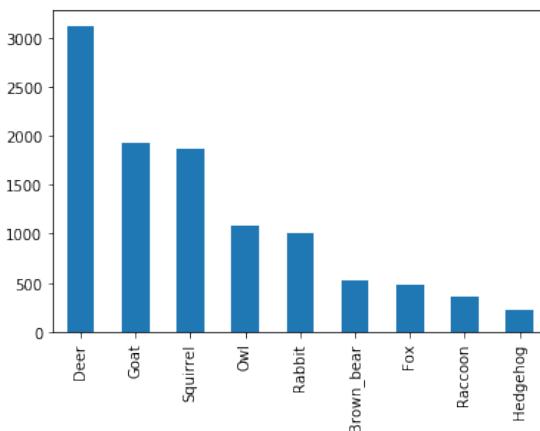


Abbildung 3.1: Ohne Augmentierung

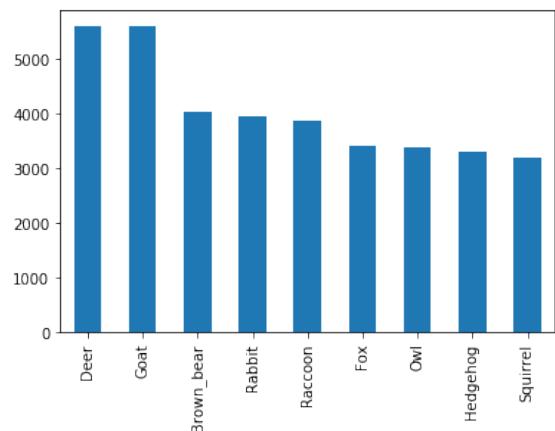


Abbildung 3.2: Ohne Augmentierung

Um diese auszugleichen, wurden die Daten, wie im nächsten Abschnitt genauer beschrieben wird, so augmentiert, dass für jede Klasse 3000 Bilder vorhanden waren, was zu einer in in Abbildung 3.2 dargestellten

Verteilung führte.

Da sich häufig mehrere Tiere der selben Klasse auf einem Bild befinden, weicht, wie in den Histogrammen zu erkennen ist, die Anzahl der Bilddateien (3000) von der Anzahl der Klassen ab.

### 3.1.1 Augmentierung

Das Augmentieren von Bilddaten für Deep Learning Modelle ist, neben dem Ausgleichen der Klassenimbalance, eine sehr effektive Technik, Overfitting zu verhindern. Indem geometrische Transformationen oder Manipulationen an den Pixelwerten auf die Bilder angewendet werden, können diese künstlich vermehrt werden.

Die Augmentierung des *OpenImages* Datensatzes wurde mithilfe eines Python Scripts, in welchem die Library *imgaug* [22] verwendet wurde, durchgeführt. Dabei wurde, je zu augmentierendem Bild, eine geometrische und eine pixelbezogene Transformation angewendet, die dabei zufällig aus einer Auswahl an Augmentern ausgewählt wurde.

In folgendem Codeausschnitt, des Python Scripts, sind die verwendeten Augmentierungstechniken dargestellt.

```
import imgaug.augmenters as iaa

color_augmenters = [
    iaa.Dropout(p=(0, 0.1)),
    iaa.CoarseDropout((0.01, 0.05), size_percent=0.1),
    iaa.Multiply((0.5, 1.3), per_channel=(0.2)),
    iaa.GaussianBlur(sigma=(0, 5)),
    iaa.AdditiveGaussianNoise(scale=((0, 0.2*255))),
    iaa.ContrastNormalization((0.5, 1.5)),
    iaa.Grayscale(alpha=((0.1, 1))),
    iaa.ElasticTransformation(alpha=(0, 5.0), sigma=0.25),
    iaa.PerspectiveTransform(scale=(0.15)),
    iaa.MultiplyHueAndSaturation((0.7))
]

geometric_augmenters = [
    iaa.Affine(scale=((0.6, 1.2))),
    iaa.Affine(translate_percent=(-0.3, 0.3)),
    iaa.Affine(shear=(-25, 25)),
    iaa.Affine(translate_percent={"x": (-0.3, 0.3), "y": (-0.2, 0.2)}),
    iaa.Fliplr(1),
    iaa.Affine(scale={"x": (0.6, 1.4), "y": (0.6, 1.4)})
]
```

Das Ergebnis von einigen zufällig angewendeten Augmentierungen auf ein Bild der Klasse Fuchs, ist in Abbildung 3.3 dargestellt.

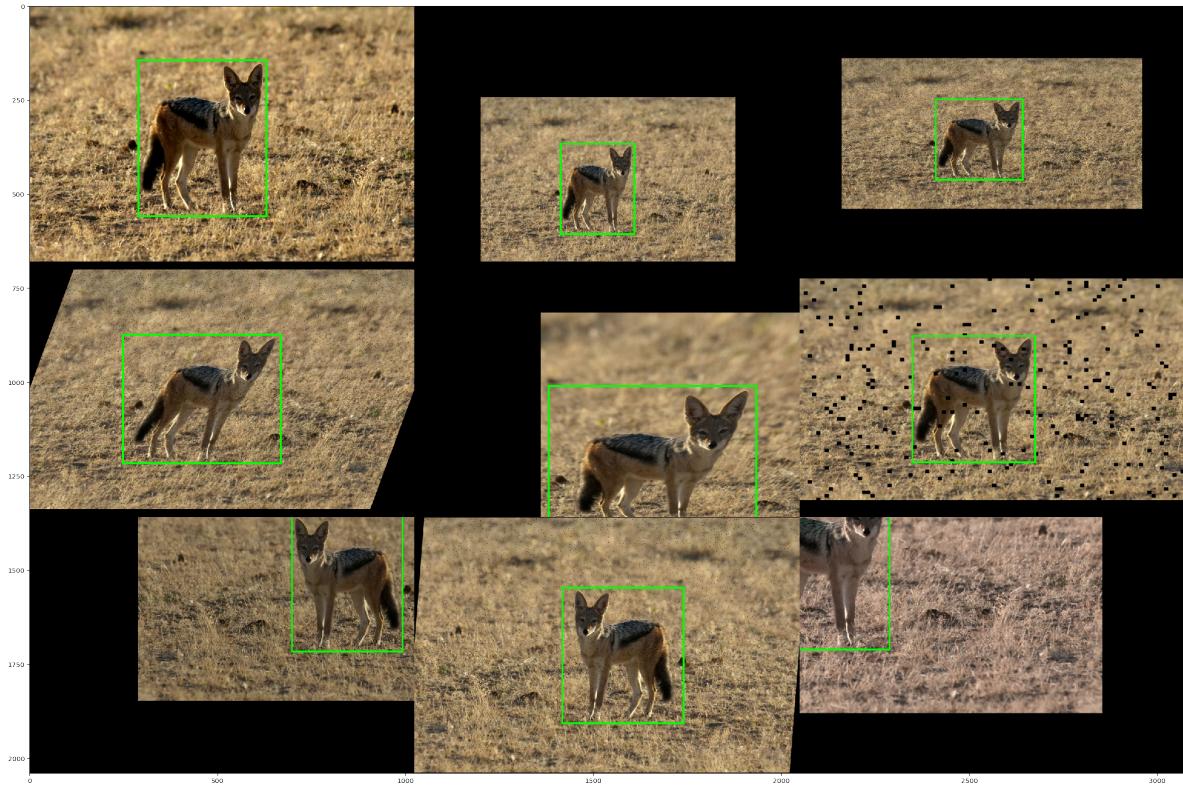


Abbildung 3.3: Anwendung von Augmentierungstechniken

## 3.2 Object detection Modelle

Object detection Modelle lassen sich, wie in [23] beschrieben wird, in Einstufige- und Zweistufige Detektoren einteilen.

Zweistufige Detektoren generieren in der ersten Stufe eine Auswahl an räumlichen Vorschlägen für das Inputbild, in welchen Objekte enthalten sein können. In der zweiten Stufe, werden die Vorschläge zur *Feature Extraction* einem CNN übergeben, welches neben einem Klassifikator auch einen *Regressor* für die *Bounding Box* Koordinaten besitzt.

Einstufige Verfahren verwenden kein separates Netz zur Vorschlagsgenerierung. Stattdessen wird das gesamte Bild als potentielle Region für Objekte betrachtet, indem dieses Gitterartig unterteilt wird. Jeder Teil, der eine mögliche Region darstellt, wird dann hinsichtlich Vorhandensein eines Objekt, klassifiziert.

Für die Bachelor Arbeit wurden Modelle beider Ansätze verwendet und hinsichtlich Genauigkeit und Inferenzzeit miteinander verglichen.

Im Folgenden werden die beiden verwendete Modelle näher erläutert.

### Faster R-CNN

Das *Faster R-CNN* [24], dargestellt in Abbildung 3.5, ist ein Modell zur Objekterkennung, welches ein zweistufiges Verfahren verwendet.

Die Vorschlagsgenerierung erfolgt in einem *Region Proposial Networks* (RPN), welches auf einem *fully convolutional network* basiert, welches *Feature Maps* generiert.

Über die *Feature Maps* werden im *Sliding-Window* Verfahren vordefinierte *Anker Boxen* konvolviert. Der daraus resultierende Feature Vektor wird einem binären Klassifikator (*cls layer*), welcher angibt ob sich ein Objekt in dem Vorschlag befindet, sowie einem Bounding Box Regresor (*reg layer*) zur Lokalisierung, übergeben.

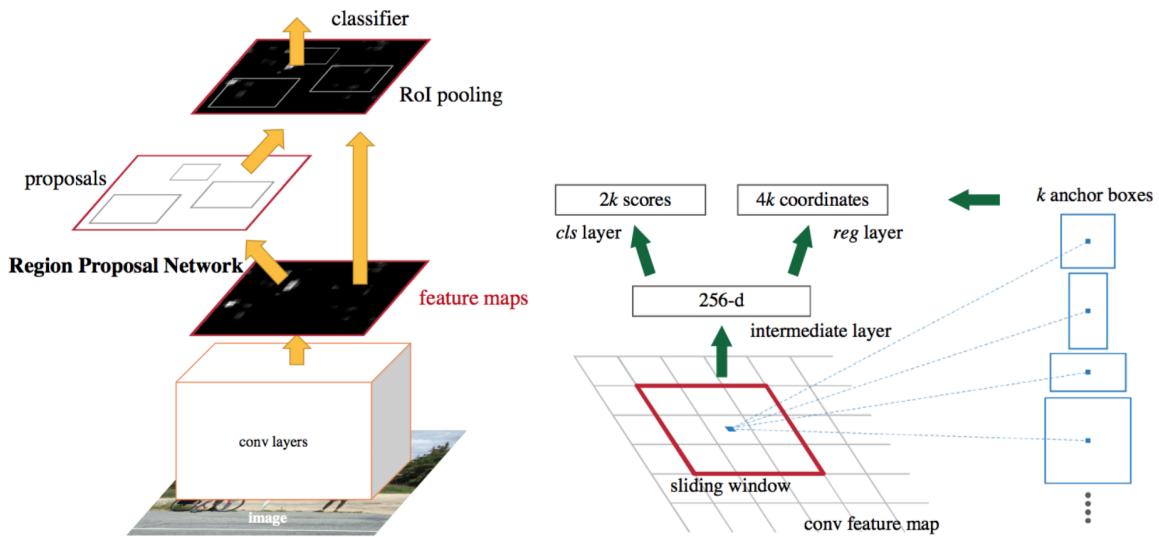


Abbildung 3.4: Faster R-CNN Architektur, [25]

### SSD: Single Shot MultiBox Detector

Der *Single Shot MultiBox Detector* (SSD) [26], verwendet ein einstufiges Verfahren zur Objekterkennung, bei dem das Input Bild gitterartig unterteilt wird. In jeder Zelle des Gitters werden *default Anker Boxen* unterschiedlicher Skalierungen definiert.

Indem an das Basis CNN weitere *Extra Feature Layer* verschiedener Größen angehängt werden, kann dieses, für jede default-Box, eine Klassifikation, in Form eines *confidence scores*, sowie eine Lokalisierung, in Form eines *Offsets* zur default-Box, vornehmen.

Diese werden zur finalen Detektion einem *non-maximum suppression* Layer [27] übergeben, welcher alle, zu einer Klasse gehörenden Boxen, in einer Box zusammenführt.

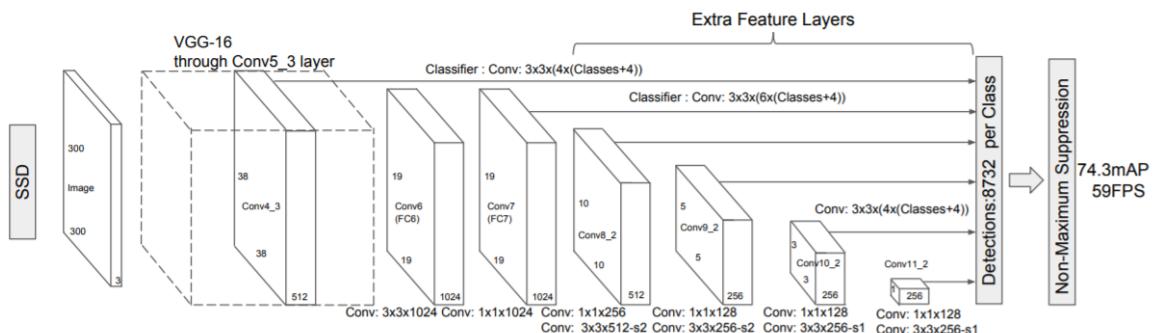


Abbildung 3.5: SSD Architektur, [28]

### 3.3 Training

Das Training der Deep Learning Modelle erfolgte in dem Framework *Tensorflow*, welches auch von *OpenVino*, für den *Neural Compute Stick*, unterstützt wird. Dabei wurde eine speziell für die Object detection entwickelte API von *Tensorflow* verwendet.

Um unabhängig von der Leistungsfähigkeit der GPU des Rechners zu sein, wurde das Training in der Cloud-basierten Virtual Machine (VM) *Google Colab* [29] durchgeführt, welche kostenlos eine, für Deep Learning geeignete, GPU zur Verfügung stellt.

### 3.3.1 Tensorflow Object Detection Api

Die *Tensorflow Object Detection Api* ist unter den *Research Modellen* des offiziellen Tensorflow Repositorys auf *GitHub* zu finden [30], und enthält Implementierungen einiger gängiger Object detection Modelle, mit verschiedenen vortrainierten Basis-CNNs.

Der, für die Bachelor Arbeit Verwendete, *Single Shot Detector* (SSD), wurde zum einen mit dem *MobilenetV2* und zum anderen mit dem *InceptionV2* als Basis CNN trainiert. Für das *Faster R-CNN* wurde, aufgrund der Verfügbarkeit, nur mit dem *InceptionV2* trainiert.

Um die Modelle trainieren zu können, mussten zunächst die Trainingsdaten in das binäre Dateiformat *TFRecords* umgewandelt werden, welches die Tensorflow Api verwendet. Dieses ist eine serialisierte Darstellung der Bilder und Labelfiles als *Protocol Buffer*, welche einen schnelleren Zugriff auf die Daten ermöglichen. Parameter für das Modell konnten vor dem Training in einer Konfigurationsdatei festgelegt werden.

Diese wurde dann, zusammen mit den *TFRecord* Dateien, dem Konsolen-Kommando, mit dem das Training gestartet wurde, übergeben.

Während des Trainings wurden in regelmäßigen Abständen die trainierten Gewichte abgespeichert.

Mithilfe des Evaluierungstools *Tensorboard* konnte der Trainingsfortschritt, anhand bestimmter Metriken, angezeigt und ausgewertet werden.

So konnten schon während des Trainings fehlerhafte Einstellungen der Datensatz- und Modelkonfiguration festgestellt und korrigiert werden, indem z.B. andere Augmentierungstechniken verwendet, oder *Hyperparameter* des Models umgestellt wurden.

In Abbildung 3.6 ist der Trainingsworkflow dargestellt welcher dabei zustande kam.

Die Ergebnisse der trainierten Modelle werden im nächsten Kapitel diskutiert.

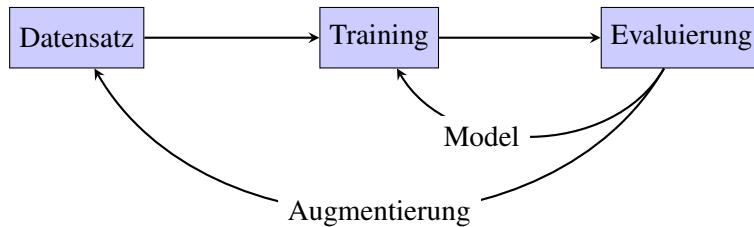


Abbildung 3.6: Trainingsworkflow

## 3.4 Inferenz

Die Anwendung, eines fertig trainierten Modells, für neue Inputdaten, wird als *Inferenz* bezeichnet. Zur Ausführung dieser, auf dem *Neural Compute Stick 2*, wird das Toolkit *OpenVino* von *Intel* verwendet. Dafür musste zunächst der trainierte *Tensorflow Graph* exportiert, d.h. die aktuellen Werte der Gewichte eingefroren werden.

Anschließend konnte mit dem *Model Optimizer* das Modell in die *Intermediate Representation* (IR) konvertiert werden. Diese besteht aus einer .xml- und einer .bin-Datei und kann von der *Inferecne Egine* zur Inferenz gelesen werden.

### Inferecne Egine

Um die Inferenz eines Modells im IR-Format auf dem NCS2 ausführen zu können, werden in der *Inference Engine*, die in Abbildung 3.7 schematisch dargestellten Schritte, durchgeführt. Daneben ist jeweils die entsprechende Codezeile in Python dargestellt.

Zunächst wird das Zielgerät, auf dem die Inferenz ausgeführt werden soll, spezifiziert (*HW Plugin laden*). Anschließend wird das Modell anhand der IR Dateien definiert (*Model IR einlesen*) woraus sich die *In- und Outputblobs* generieren lassen, (*In- und Outputblob*), welche die Dimensionen der Ein- und Ausgabe Schicht des Modells darstellen.

Das zu inferierende Bild, welches als Matrix aus Pixelwerten vorliegt, muss dann in das *Input Blob* Format gebracht werden (*process Input*).

Nachdem das Bild inferiert wurde (*Inferenz*), kann es, zusammen mit den Inferenzergebnissen, weiterverarbeitet werden (*process output*).

Handelt es sich bei den Inputs um einen fortlaufenden Video- oder Kamera Stream, werden die Schritte *preprocess*, *Inferenz* und *process Output* in einer Schleife wiederholt.

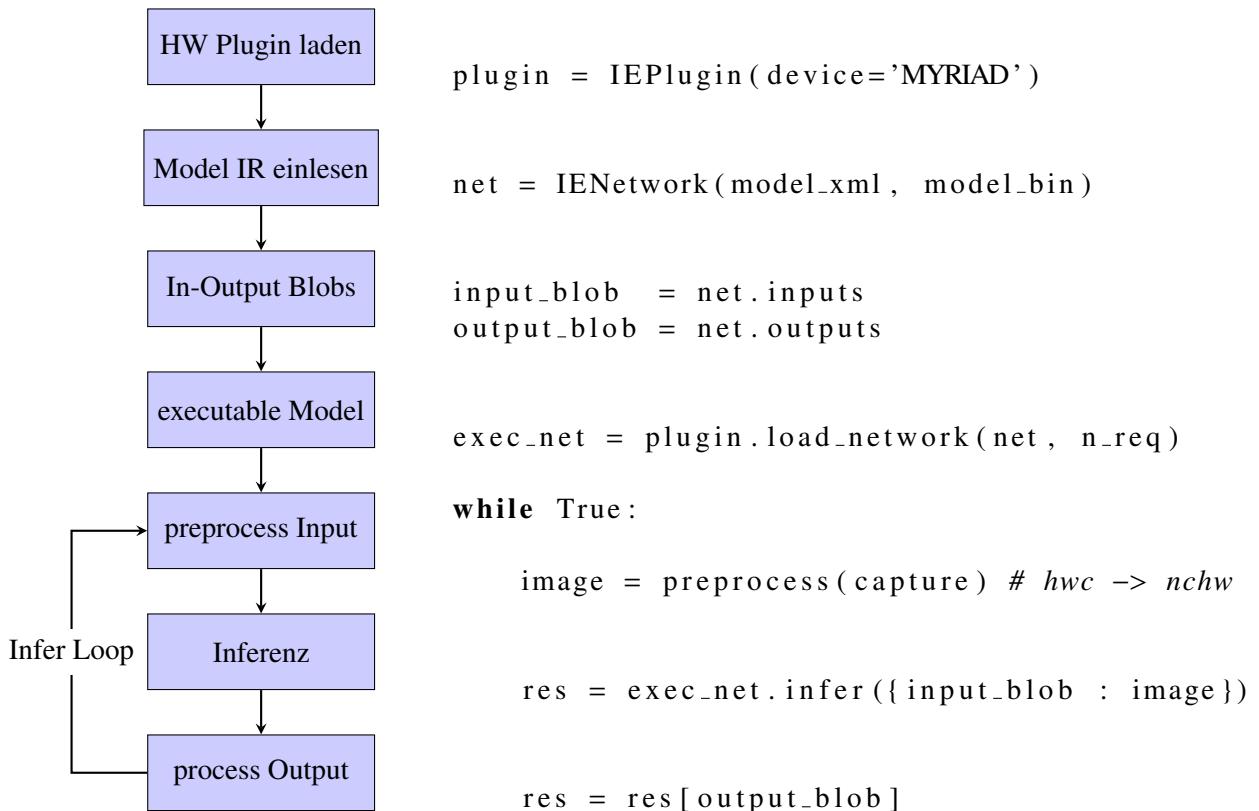


Abbildung 3.7: Ablauf der InferenceEngine

Die Form des Inferenzergebnisses hängt von der Art des verwendeten Deep Learning Modells ab, welche z.B. *Image Classification*, *Object detection*, oder *Instance Segmentation* sein können.

Für Object detection Modelle enthält das Ergebnis Datenstrukturen, welche den Index, zugehörige Wahrscheinlichkeit, sowie Bounding Box Koordinaten der geschätzten Objekte im Bild enthalten.

Indem für alle Schätzungen, die in einem Bild gemacht wurden, ein Threshold für die Wahrscheinlichkeit festlegt wird (zb. 70%), können die sinnvollen Ergebnisse herausgefiltert werden.

Zur veranschaulichung können die Koordinaten dann als Bounding Box in das inferierte Bild gezeichnet werden.

# Kapitel 4

## Evaluierung

In diesem Kapitel wird die Auswertung der Ergebnisse der trainierten Modelle beschrieben.

Dafür werden im ersten Abschnitt zunächst die Metriken erklärt, anhand denen die Evaluierung erfolgte.

Im zweiten Abschnitt werden die beiden verwendeten Object detection Modelle *SSD* und *Faster R-CNN*, hinsichtlich dieser Metriken, sowie anhand von Inferenzergebnissen verglichen.

Der dritte Abschnitt beschreibt Methoden, mit denen die Ergebnisse des *Faster R-CNN* optimiert werden konnten.

Im vierten Abschnitt werden die Modelle dann noch hinsichtlich der Inferenzzeit miteinander verglichen.

### 4.1 Evaluierungs Metriken

Zur Messung der Genauigkeit der Object detection Modelle, wurde die *Mean Average Precision* (mAP) verwendet. Diese bezieht sowohl Klassifikations-, als auch Lokalisierungs Genauigkeit mit ein und lässt sich aus den folgenden Werten berechnen.

- *True Positive (TP)*: Das Model hat richtig das Vorhandensein eines Objekts geschätzt
- *True Negative (TN)*: Das Model hat richtig die Abwesenheit eines Objekts geschätzt
- *False Positive (FP)*: Das Model hat fälschlicherweise das Vorhandensein eines Objekts geschätzt
- *False Negative (FN)*: Das Model hat fälschlicherweise die Abwesenheit eines Objekts geschätzt

Die Festlegung, für *True Positive* Werte wird dabei über die, in Abbildung 4.1 dargestellte, *Intersection over Union* (IoU) ermittelt.

Diese ist durch den Überlappungsgrad der, im Label definierten, *Ground Truth* Bounding Box und der geschätzten Bounding Box bezogen auf den Gesamtbereich, den die beiden Boxen einschließen, definiert.

Ist dieser größer, als ein definierter Threshold, welcher häufig bei 50% liegt, gilt die Schätzung als *True Positive*, andernfalls als *False Positive*.

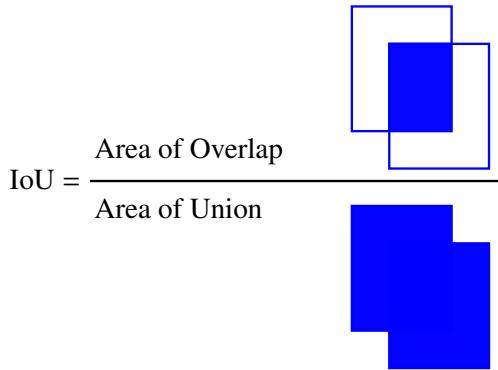


Abbildung 4.1: Intersection over Union

		Geschätzter Wert	
		p	n
Tatsächlicher Wert	p'	True Positive	False Negative
	n'	False Positive	True Negative

Abbildung 4.2: Confusion Matrix

Anhand dieser, in der *Confusion Matrix* (Abbildung 4.2) dargestellten Werte, lassen sich die Metriken *Precision* und *Recall* berechnen.

Der *Recall* ist dabei durch das Verhältnis der richtig gefundenen, zu allen sich im Bild befindenden Objekten definiert, was sich auch durch das, in Gleichung 4.1 gezeigte Verhältnis von True Positive und False Positive, darstellen lässt.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (4.1)$$

Im Gegensatz zum *Recall*, welcher die Trefferquote des Modells angibt, gibt die *Precision* die Genauigkeit an mit der die Objekte gefunden wurden. Definiert ist diese *Precision* durch das Verhältnis der richtigen Schätzungen bezogen, auf alle gemachten Schätzungen, was auch durch das, in Gleichung 4.2 dargestellte, Verhältnis von True Positives und False Positives ausgedrückt werden kann.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4.2)$$

Werden, für eine Klasse, alle *Precision* Werte über dem *Recall* aufgetragen, ergibt sich eine abnehmende Kurve, dessen Flächeninhalt, wie in Gleichung 4.3 dargestellt, die durchschnittliche *Precision* für diese Klasse darstellt.

Wird diese für alle Klassen gebildet und im Mittel genommen, erhält man die in Gleichung 4.4 dargestellte, *mean Average Precision* (mAP).

$$\text{Average Precision} = \sum \text{Precision}(\text{Recall}) \quad (4.3)$$

$$mAP = \frac{1}{N} \sum \text{Average Precision} \quad (4.4)$$

## 4.2 Vergleich der Modelle

In diesem Abschnitt geht es um die Auswertung der Evaluierungsergebnisse der beiden, für das Training verwendeten Object detection Architekturen *Single Shot Detector* (SSD) und *Faster R-CNN*.

### 4.2.1 Evaluierung

Die im Folgenden dargestellten Ergebnisse beziehen sich auf den Validierungsanteil des, für das Training verwendeten, *OpenImages* Datensatzes.

Die Berechnung dieser, anhand der in Abschnitt 4.1 erläuterten Metriken, sowie eine Visualisierte Darstellung des Trainingsverlaufs, erfolgte über das Evaluierungstool *Tensorboard*.

Das Training wurde, für alle drei Modelle zum Vergleich, jeweils einmal für den originalen und einmal für den augmentierten Datensatz durchgeführt.

Model	Optimierung	mAP	Loss
SSD + MobilenetV2	Ohne	0,62	3,56
	Augmentierung	0,61	3,50
SSD + InceptionV2	Ohne	0,65	3,86
	Augmentierung	0,62	3,71
Faster R-CNN +InceptionV2	Ohne	0,67	0,82
	Augmentierung	0,69	0,67
	Early Stopping	0,67	0,69

Tabelle 4.1: Trainingsergebnisse von SSD und Faster R-CNN

Anhand der in Tabelle 4.1 dargestellten Ergebnisse, ist zu erkennen, dass sich mit dem zweistufigen *Faster R-CNN* bessere Ergebnisse, als mit dem einstufigen *SSD* erzielen ließen. Der Unterschied ist besonders deutlich anhand des Loss-Wertes festzustellen.

Des Weiteren wurden bei den SSD Konfigurationen, mit dem *InceptionV2* als Basis CNN, bessere Ergebnisse erreicht, als mit dem *MobilenetV2*.

Bei allen Modellen war durch die Augmentierung des Datensatzes eine Verbesserung des Loss Wertes festzustellen, da dadurch Overfitting reduziert oder verhindert werden konnte.

Bei den *SSD* Architekturen führte die Augmentierung jedoch auch zu einer Verringerung des mAP Wertes, was auf die weniger komplexe Model Struktur zurückzuführen sein kann.

Je mehr Parameter einem Model zu Verfügung stehen, desto besser kann es sich an die Trainingsdaten anpassen, desto eher findet jedoch auch Overfitting statt. Dieser Zusammenhang hat sich deutlich bei dem *Faster R-CNN* Modell bemerkbar gemacht.

Der Plot in Abbildung 4.4 zeigt den Trainingsverlauf, der verschiedenen *Faster R-CNN* Trainingskonfigurationen, anhand der Loss-Kurve.

Für das Training mit dem originalen Datensatz nimmt diese nach ca. 100k Iterationen wieder zu, wohingegen der Loss, beim Training mit augmentierten Datensatz, den Wert weitestgehend beibehalten kann.

*Early Stopping* war ein weiterer Ansatz, das Overfitting beim *Faster R-CNN* Modell zu verhindern. Dafür wurde das Training, bevor der Loss-Wert wieder zunahm, abgebrochen.

Anhand der Loss-Kurve im Plot in Abbildung 4.4, ist zu erkennen, dass sich dadurch der gleiche Wert, wie durch die Augmentierung erreichen ließ. Jedoch konnte der mAP, wie im Plot in Abbildung 4.3 zu erkennen ist, durch das frühzeitige Stoppen des Trainings, seinen möglichen Endwert nicht erreichen.

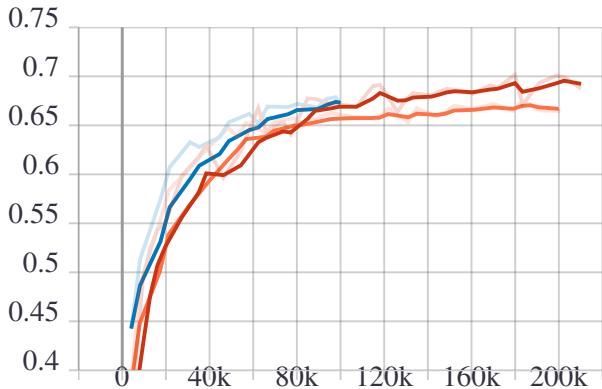


Abbildung 4.3: mAP

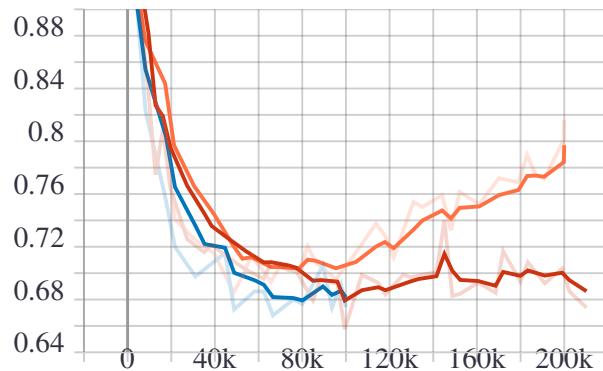


Abbildung 4.4: Loss

● Ohne     ● Early Stopping     ● Augmentierung

Daraus ließ sich schließen, dass die Augmentierung das bessere Vorgehen zur Vermeidung von Overfitting ist. Um herauszufinden, ob sich diese Annahme bestätigt und wie sehr sich die Unterschiedlichen Ergebnisse zwischen SSD und Faster R-CNN in der Praktischen Anwendung bemerkbar machen, wurde die Inferenz der Trainierten Modelle testweise auf verschiedene Bilder ausgeführt.

#### 4.2.2 Test Inferenz

Um die Inferenz ausführen zu können, wurden die trainierten Modelle, wie in Abschnitt 3.4 beschrieben, in die *Intermediate Representation* konvertiert und anschließend mithilfe der *Inference Engine* auf dem *Neural Compute Stick 2* inferiert.

Dafür wurden zunächst die Bilder aus dem Testset des *OpenImages* Datensatzes verwendet. Da diese jedoch sehr ähnlich zu den Trainingsdaten sind, wurden auch Bilder aus anderen Quellen inferiert.

Dadurch lässt sich die Robustheit des Modells, gegenüber anderer Datensatzausprägungen, die beispielsweise Qualität der Bilder, Beleuchtung, oder geographische Lage betreffen, feststellen. Durch ein dahingehend robusteres Modell, ist auch mit besseren Ergebnissen in der praktischen Anwendung des Modells zu rechnen, da sich dabei die Daten ebenfalls von den Trainingsdaten unterscheiden werden.

Als weiterer Testdatensatz wurden daher, zum einen Teile des *iWildCam 2019 Datasets* [31], und zum anderen eigene Aufnahmen von Tieren verwendet.

#### OpenImages Test Set

Die Inferenzergebnisse des *OpenImages* Testsets ergaben, dass in den meisten Fällen, sowohl mit dem SSD, als auch mit dem Faster R-CNN, die Tiere in den Bildern richtig erkannt werden konnten.

Waren die Tiere auf dem Bild jedoch weiter weg, oder in schlechterer Qualität abgebildet, waren die Ergebnisse beim Faster R-CNN deutlich besser, wie beispielhaft in den Abbildungen 4.5 und 4.6 zu erkennen ist.

Der Unterschied zwischen MobilenetV2 und InceptionV2 beim SSD sowie zwischen Early Stopping und Augmentierung beim Faster R-CNN machte sich kaum bemerkbar.

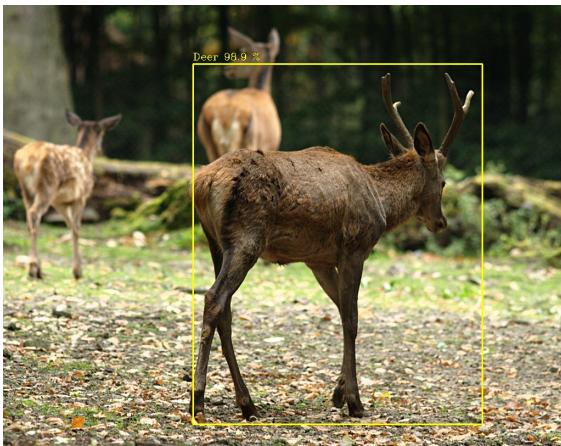


Abbildung 4.5: SSD

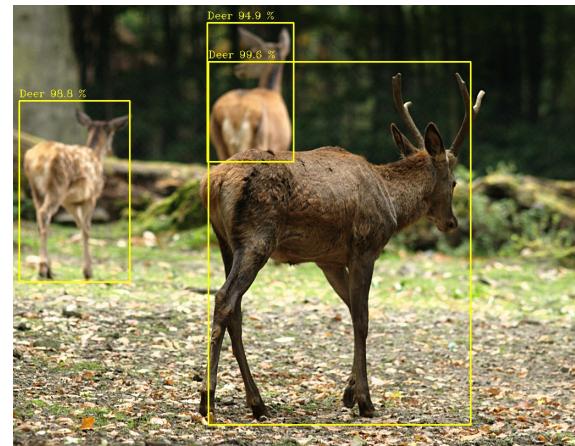


Abbildung 4.6: Faster R-CNN

### Eigene Aufnahmen

Bei der Inferenz, auf die eigenen Bilder, war ein deutlicher Unterschied der Modelle festzustellen.

In aufsteigender Reihenfolge lieferten das SSD mit MobilenetV2, das SSD mit InceptionV2, das Faster R-CNN mit Early Stopping und Faster R-CNN mit Augmentierten Daten wie in den Abbildungen 4.7 bis 4.10 zu sehen ist, bessere Ergebnisse.

Auch hier fiel auf, dass Tiere, die weiter weg und damit kleiner abgebildet sind, besser von den Faster R-CNN Modellen erkannt wurden.

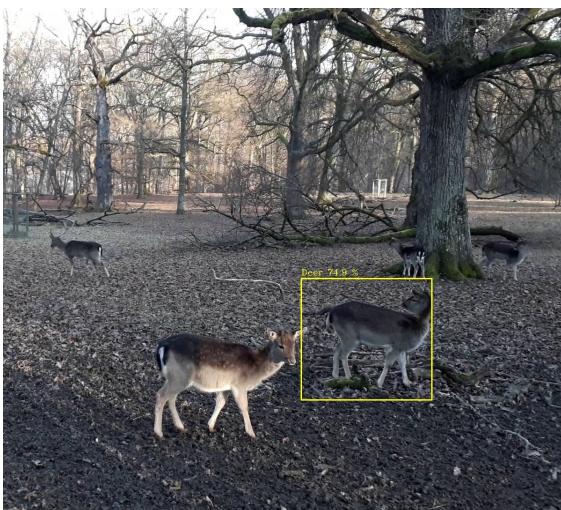


Abbildung 4.7: SSD Mobilnet

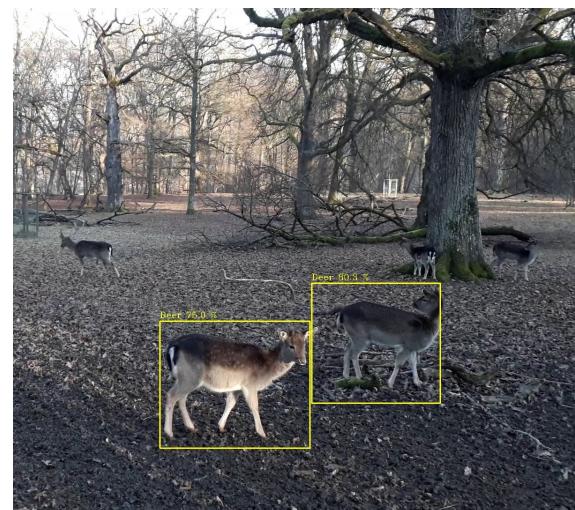


Abbildung 4.8: SSD Inception

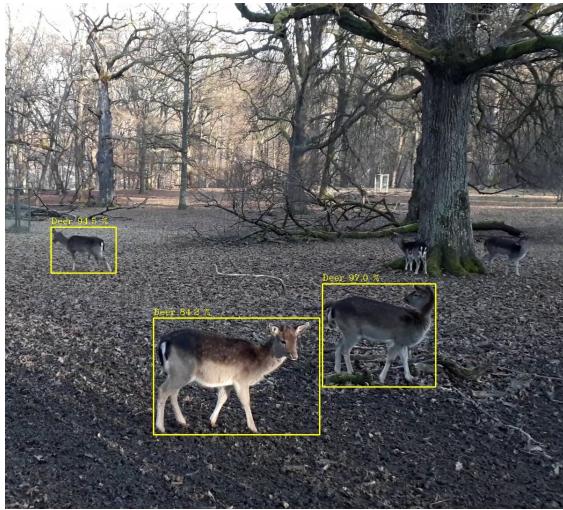


Abbildung 4.9: Faster R-CNN mit Early Stopping

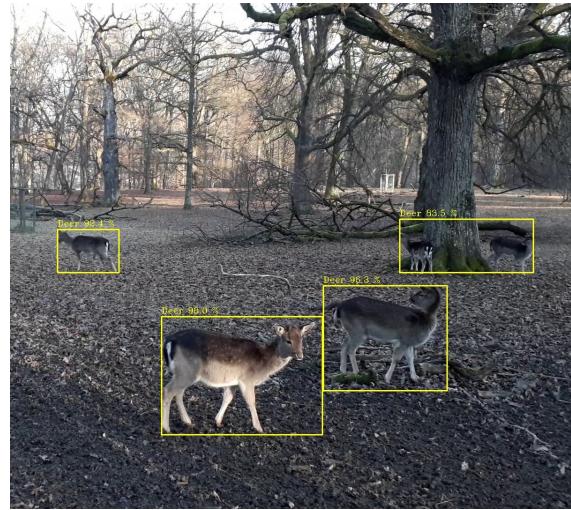


Abbildung 4.10: Faster R-CNN mit Augmentierung

### iWildCam Datensatz

Aus dem *iWildCam 2019 Dataset*, welches aus einer *Kaggle Competition* stammt, wurden die Klassen, welche sich mit den für das Training verwendeten überschnitten, für die Test Inferenz heruntergeladen. Bei den Bildern handelt es sich um Aufnahmen von Wildtier Kameras, aus dem Süd- und Nordwesten Amerikas, welche aus der *iNaturalist* und der *Microsoft DatAirSim* Datenbank stammen.

Darunter enthalten sind viele Nachtaufnahmen, welche teils schlecht beleuchtet und mit einer Infrarot Kamera aufgenommen und daher in Graustufen sind.

Da die Inferenzergebnisse hier bei allen vier Model Variationen nicht so gut wie bei den anderen Datensätzen waren, wurde versucht dieses, durch weitere Optimierungen, zu verbessern.

## 4.3 Optimierungen: Faster R-CNN

Als Ausgangslage zur Verbesserung der Ergebnisse diente das Faster R-CNN mit augmentiertem Datensatz, welches bei der Evaluierung im vorherigen Abschnitts die besten Resultate erzielte.

Die Auswertung erfolgt hier wieder zunächst anhand der Evaluierungsmetriken und der Trainingsverläufe aus Tensorboard und anschließend anhand der, auf Testbilder ausgeführten, Inferenzergebnisse.

### 4.3.1 Verschiedene Augmentierungen

Der erste Ansatz zur Verbesserung der Ergebnisse war es, das Faster R-CNN mit unterschiedlich starker Augmentierung der Daten, für insgesamt mehr Iterationen (500k statt 200k), zu trainieren.

Dabei wurde wieder das im Abschnitt 3.1.1 erläuterte Augmentierungsverfahren angewendet, mit zusätzlich folgenden Variationen:

1. Nur eine zufällige Augmentierung pro Bild, (anstelle von zwei)
2. 4000 Bilder pro Klasse generieren (anstelle von 3000)

Die Trainingsergebnisse für 500k Iterationen sind anhand der Trainingsverläufe des Loss- und mAP-Wertes in den Abbildungen 4.11 und 4.12 dargestellt.

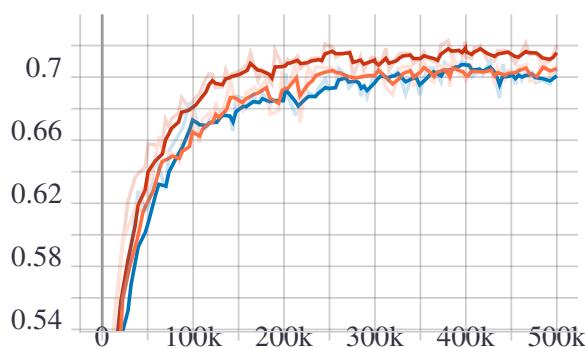


Abbildung 4.11: mAP

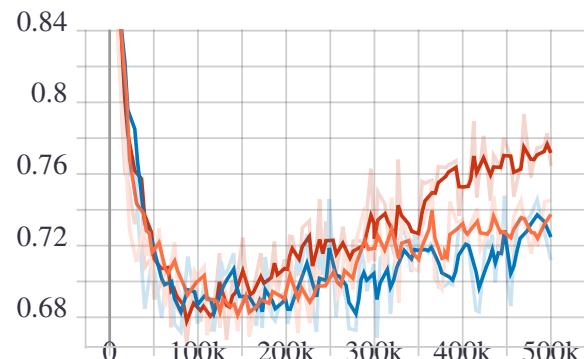


Abbildung 4.12: Loss

● nur eine Augmentierung je Bild      ● 3000 Bilder      ● 4000 Bilder

Aufgrund des länger durchgeführten Trainings ist bei allen Konfigurationen im, gegensatz zu den Ergebnissen des vorherigen Abschnitts, Overfitting festzustellen.

Dieses fällt, wie zu erwarten war, bei den weniger augmentierten Datensätzen stärker aus, welche auf der anderen Seite einen besseren mAP Wert erreichen konnten.

Ob sich, konkret für diesen Fall, ein höherer mAP oder besserer Loss Wert positiver auf das Ergebnis auswirkt, konnte wieder mithilfe der Inferenz auf Testbilder herausgefunden werden.



Abbildung 4.13: 3000 Samples

Abbildung 4.14: 4000 Samples

Abbildung 4.15: 50% Augment

Abbildung 4.13 bis 4.15 zeigen beispielhaft die Inferenzergebnisse für Bilder des *iWildCam* Datensatzes, bei dem sich diesesmal der Unterschied deutlicher bemerkbar machte. Bei den Modellen mit normaler oder mehr Augmentierung der Trainingsdaten war eine deutliche Verbesserung festzustellen, wohingegen das Modell, welches auf weniger stark augmentierte Daten trainiert wurde, die Tiere schlechter oder gar nicht erkannte.

### 4.3.2 Verschiedene Regularisierungen

Um das, trotz Augmentierung, zu stande kommende Overfitting zu vermeiden, wurde nun zusätzlich die *L2 Regularisierung* angewendet. Diese soll, wie in den Grundlagen (Abschnitt 2.1.2) beschrieben wurde, durch Anhängen einer Aufsummierung der Gewichte an die Loss Funktion, ein Überanpassen des Modells an die Trainingsdaten, reduzieren.

In der Konfigurationsdatei des Faster R-CNN kann dies, durch setzen eines bestimmten Parameters für, sowohl die erste Stufe des Modells, dem *Region Proposal Network* (RPN), als auch für die zweite Stufe, dem Klassifikationsmodell, separat eingestellt werden.

Ebenso lassen sich die beiden Loss-Kurven, aus denen sich beim Faster R-CNN der gesamte Loss zusammensetzt, separat anzeigen, was in den Verläufen in Abbildung 4.18 und 4.19 dargestellt ist.

Durch die getrennte Beobachtung der Loss-Kurven ließ sich feststellen, dass das Overfitting nur das RPN betrifft, weshalb der Parameter zur *L2 Regulierung* nur für die erste Stufe eingestellt wurde. Dafür wurde der Faktor  $\lambda = 0.01$  gesetzt.

Vergleicht man nach dem Training mit reguliertem Modell wieder die Loss-Kurven, ist deutlich zu erkennen, dass sich das Overfitting im RPN reduzierten ließ, wodurch sich auch der in Abbildung 4.17 dargestellte gesamt Loss verbesserte.

Die Verbesserung des Loss Wertes ging hier wieder mit einer leichten Verschlechterung des mAPs einher, wie in Abbildung 4.16 zu erkennen ist.

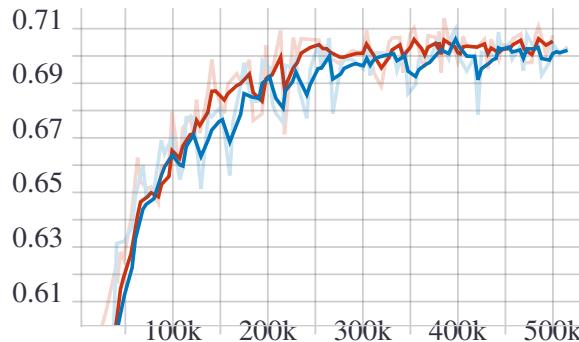


Abbildung 4.16: mAP

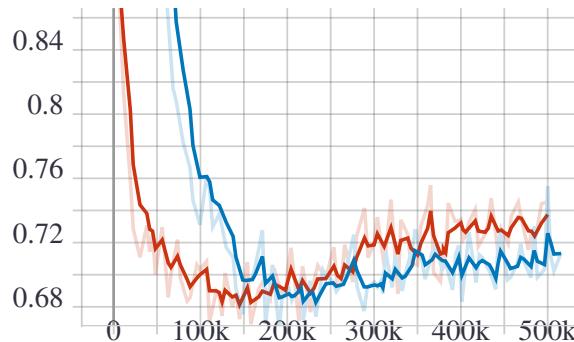


Abbildung 4.17: Total Loss

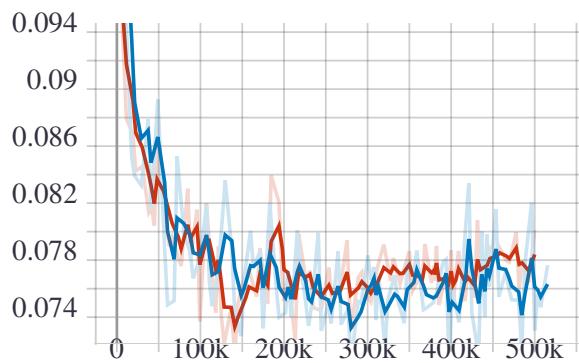


Abbildung 4.18: Klassifikations Loss

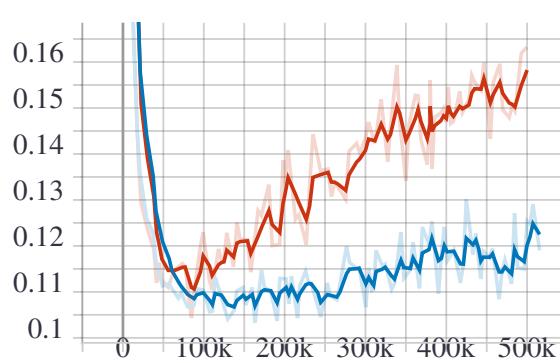


Abbildung 4.19: RPN Loss

● nur Augmentierung

● Augmentierung + L2 Regularisierung

Auch hier wurde, zur Feststellung der Auswirkung der unterschiedlichen Ergebnisse, wieder die Testinfe-

renz angewendet, was beispielhaft für Bilder der eigenen Aufnahmen in den Abbildungen 4.20 und 4.21 dargestellt ist.

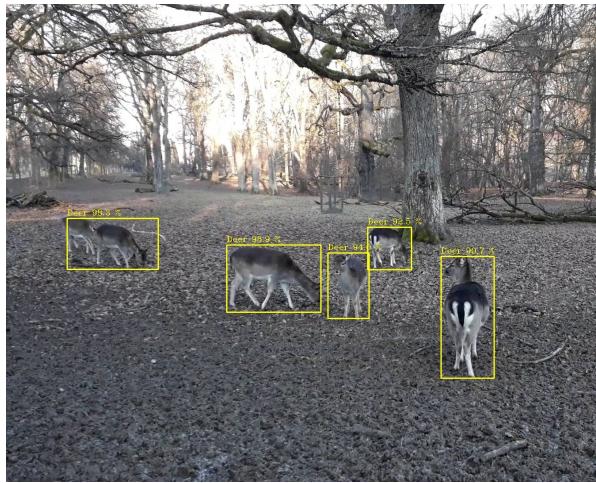


Abbildung 4.20: Augmentierung (normal)

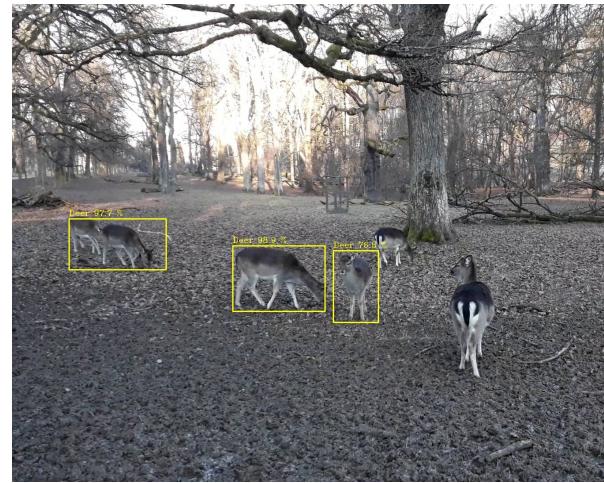


Abbildung 4.21: Augmentierung + L2

Hier ergaben die Inferenzergebnisse, dass eine L2 Regulierung der Modelle in den meisten Fällen keinen Einfluss auf das Ergebnis hat, falls doch, dieses sich tendenziell sogar verschlechterte.

Weitere angewendete Regularisierungen, die jedoch zu keiner nennenswerten Veränderung der Trainingsergebnisse führten, waren Dropout sowie L2 mit  $\lambda = 0,02$  und sind in Tabelle 4.2, zusammen mit den anderen Ergebnissen dargestellt.

	mAP	Loss
Augmentierung (50%)	0,72	0,76
+ L2 Reg. ( $\lambda = 0,01$ )	0,71	0,75
Augmentierung (normal)	0,7	0,74
+ Dropout	0,7	0,73
+ L2 Reg. ( $\lambda = 0,01$ )	0,7	0,69
+ L2 Reg. ( $\lambda = 0,02$ )	0,69	0,7
Augmentierung (4000 Samples)	0,7	0,71

Tabelle 4.2: Regularisierungen

Aus den Ergebnissen ließ sich schließen, dass die Art der Aufbereitung der Daten den größeren Einfluss auf die Ergebnisse haben und durch Anpassungen der Hyperparameter, wenn überhaupt, nur noch geringfügige Optimierungen betrieben werden können. So hat insgesamt das Faster R-CNN, mit 500k Trainingsiterationen auf Augmentierte Daten mit 3000 Bildern je Klasse, zu den besten Ergebnissen, bezüglich Genauigkeit, geführt.

## 4.4 Inferenzzeit

Neben der Genauigkeit, war die Ausführungszeit, welche ein Model für die Inferenz benötigt, ein weiteres Kriterium für die Auswahl des, in der Anwendung zu verwendenden Modells. Einer der Faktoren, welcher die Inferenzzeit beeinflusst, ist die Hardware, auf der die Inferenz stattfindet, sowie die zur Implementierung verwendeten Library.

Die Hardware war mit dem *Neural Compute Stick 2* festgelegt, als Library kommen dafür *OpenCV* oder *OpenVino* in Frage, wobei mit *OpenVino* die Möglichkeit zur asynchronen Inferenzausführung, sowie der Verwendung mehrerer, parallel ausgeführter, Inferenz Requests besteht, wodurch sich die Inferenzzeit optimieren lässt.

Ein weiter Faktor ist die Komplexität des CNNs sowie die für die Objekterkennung verwendete Modell Architektur. Üblicherweise sind Komplexere Modelle, wie das Faster R-CNN, zwar genauer, jedoch auch langsamer.

Um den Effekt, den die drei unterschiedlichen, für das Training verwendeten, Varianten SSD mit MobileNetV2, SSD mit InceptionV2 und Faster R-CNN mit InceptionV2 auf die Inferenzzeit haben zu untersuchen, wurden diese durch Messen der Inferenzzeit verglichen.

Dabei wurde die Asynchrone Inferenz mit unterschiedlicher Anzahl an Inferenz Requests verwendet, weshalb in diesem Abschnitt zunächst die Funktionsweise der synchronen- und asynchronen Inferenzausführung mit OpenVino erklärt wird.

### 4.4.1 Synchrone- und Asynchrone Inferenz

Wird die Inferenz im synchronen Modus ausgeführt, kann immer nur entweder inferiert, oder das Vor- und Nachverarbeiten, der Bilder stattfinden.

Vorverarbeitung der Bilder beinhaltet dabei zum Beispiel die Umwandlung des, von der Kamera gelieferten Bildformats, in das für das jeweilige Model richtige Input Format. Die Nachverarbeitung bezieht sich auf das Verwenden der Inferenz Ergebnisse in der Anwendung.

Die Implementierung der Inferenz in *OpenVino* erfolgt dementsprechend sequentiell, wie im Algorithmus 1 als Pseudocode dargestellt ist.

Anhand des zeitlichen Ablaufs, dargestellt in Abbildung 4.22, sind die Abschnitte, in denen keine Inferenz stattfinden kann, zu erkennen.

---

#### Algorithm 1 Synchrone Inferenz

---

```

while true do
    capture FRAME
    preprocess CURRENT InferRequest
    start CURRENT InferRequest
    wait for CURRENT InferRequest
    process CURRENT result
end while

```

Preprocessing

Process Output

Infer Frame



Abbildung 4.22

Da die Inferenz auf dem *Myriad Chip* des NCS2 und nicht auf dem ausführenden PC bzw. *Raspberry Pi* läuft, kann diese ungehindert, parallel zum restlichen Programmablauf, erfolgen.

In *OpenVino* wird dieser Ablauf mithilfe der *Asynchronen Api* erreicht, welche, über einen bestimmten Funktionsaufruf, die Inferenz in einem separaten Thred startet.

Indem vor Erhalt und Verarbeitung eines aktuellen Inferenz Ergebnisses der Inferenz Request für den nächsten Durchlauf aufgegeben wird, wie in Algorithmus 2 als Pseudocode dargestellt ist, kann der in Abbildung 4.23 dargestellte Zeitliche Ablauf erreicht werden.

---

**Algorithm 2** Asynchrone Inferenz

---

```

while true do
    capture FRAME
    preprocess NEXT InferRequest
    start NEXT InferRequest
    wait for CURRENT InferRequest
    process CURRENT result
    swap CURRENT and NEXT InferRequest
end while

```

---

Preprocessing

Process Output

Infer Frame

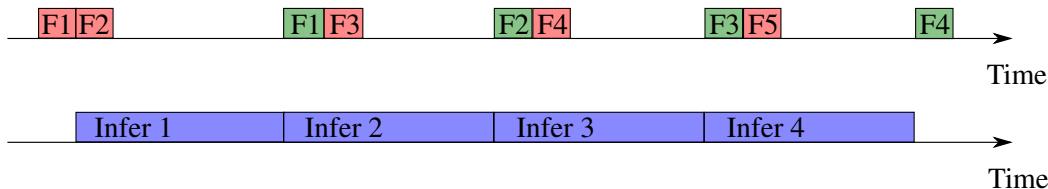


Abbildung 4.23

Die hier mit *Current* und *Next* bezeichneten Inferenz Requests stehen für die Indizes der jeweiligen Requests und können belieb erweitert werden. Dadurch wird erreicht, dass die Inferenz auf mehreren Threads parallel ausgeführt wird.

#### 4.4.2 Vergleich der Modelle

Mithilfe eines Python Scripts, in welchem die Asynchrone Inferenz für eine variabel einstellbare Anzahl an Inferenz Requests implementiert wurde, konnte für die drei Modelle die durchschnittliche Frames per Second (Fps) der Inferenz ermittelt werden.

Diese wurden auf dem *Raspberry Pi* mit dem *Neural Compute Stick* ausgeführt und lieferten die in Tabelle 4.3 dargestellten Ergebnisse.

Model	Asynchrone Inferenz Requests			
	1	2	3	4
SSD MobilenetV2	19,5	35,2	40,6	40,3
SSD InceptionV2	15,6	27,7	31,1	31,7
Faster R-CNN Incept.	0,63	0,67	0,75	0,74

Tabelle 4.3: Vergleich von Inferenzzeiten der Modelle in FPS

Die asynchrone Inferenzausführung führte bei allen Modellen, für bis zu 3 inferenz Requests, zu besseren Ergebnissen. Ein deutlicher Unterschied der Inferenzzeit war zwischen SSD und Faster R-CNN Architekturen festzustellen.

Da für die Anwendung zur Wildtiererkennung keine Realtime Performance erforderlich ist, wurde durch geschickte Implementierung der Inferenz in der Applikation, trotz langsamerer Inferenzzeit, das Faster R-CNN verwendet.

# Kapitel 5

## Entwicklung der Anwendung

Dieses Kapitel beschreibt die Realisierung der Anwendung, als autonomes Kamerasystem zur Wildtiererkennung.

Zunächst werden dabei die verwendeten Hardwarekomponenten erläutert.

Im zweiten Abschnitt wird die Implementierung der Inferenz, für eines der trainierten Modelle, sowie einer geeigneten Kommunikationsmöglichkeit zur Übertragung der Daten beschrieben.

### 5.1 Hardware

Der Aufbau der Anwendung besteht aus einem, in Abbildung 5.1 dargestellten *Raspberry Pi 4*, auf dem der Programmcode ausgeführt wird, sowie dem *Neural Compute Stick 2* für die Inferenz, welcher über eine USB Schnittstelle mit dem Raspberry Pi verbunden wird.

Zur Aufnahme der Bilder wurde das in Abbildung 5.2 dargestellte *Pi Kamera Modul*, mit einem *5MP OV5647 Sensor* der Marke *Longrunner* verwendet. Dieses ermöglicht, durch mechanisches zu und abschalten eines Infrarot Filters vor die Linse, zwischen Tag- und Nachtsicht zu wechseln. Der dafür verwendete Magnetschalter wird automatisch über einen Helligkeitssensor getriggert. Im Infrarotmodus befindet sich der Filter nicht vor der Linse, sodass neben den elektromagnetischen Wellen des Sichtbaren Lichts, auch die des langwelligeren des Infrarot Spektrums (850nm) auf die Linse treffen und verarbeitet werden können.

Zudem verfügt die Kamera über zwei Infrarot LEDs, sodass auch Aufnahmen, in bis zu 3m Entfernung, in völliger Dunkelheit gemacht werden können. Diese haben den Vorteil gegenüber normalen LEDs, dass die Tiere von keiner Sichtbaren Lichtquelle gestört oder verscheucht werden.

Verbunden wird das Kamera Modul über die Camera Serial Interface (CSI) Schnittstelle des Raspberry Pi's.

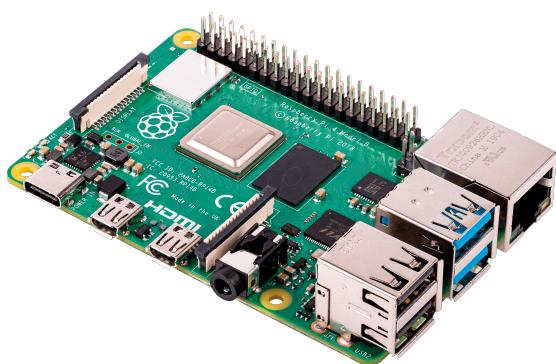


Abbildung 5.1: Raspberry Pi 4



Abbildung 5.2: Longrunner Kamera Modul

Desweiteren wurde für eine mobile Internetverbindung der *Huawei E3531 SurfStick* und zu Stromversorgung eine Powerbank verwendet.

## 5.2 Software

Die Implementierung der Applikation für den *Raspberry Pi* wurde in Python vorgenommen. Dabei sind die Funktionalitäten zur Objekterkennung in dem Script *detection.py* und die, zur Herstellung einer Verbindung und Senden der Daten, in dem *connection.py* Script definiert.

Der Kamera-Inputstream ist in einem *main.py* Script implementiert, von dem aus auch die in Abbildung 5.3 dargestellten Klassen, welche in *detection.py* und *connection.py* enthalten sind, verwendet werden.

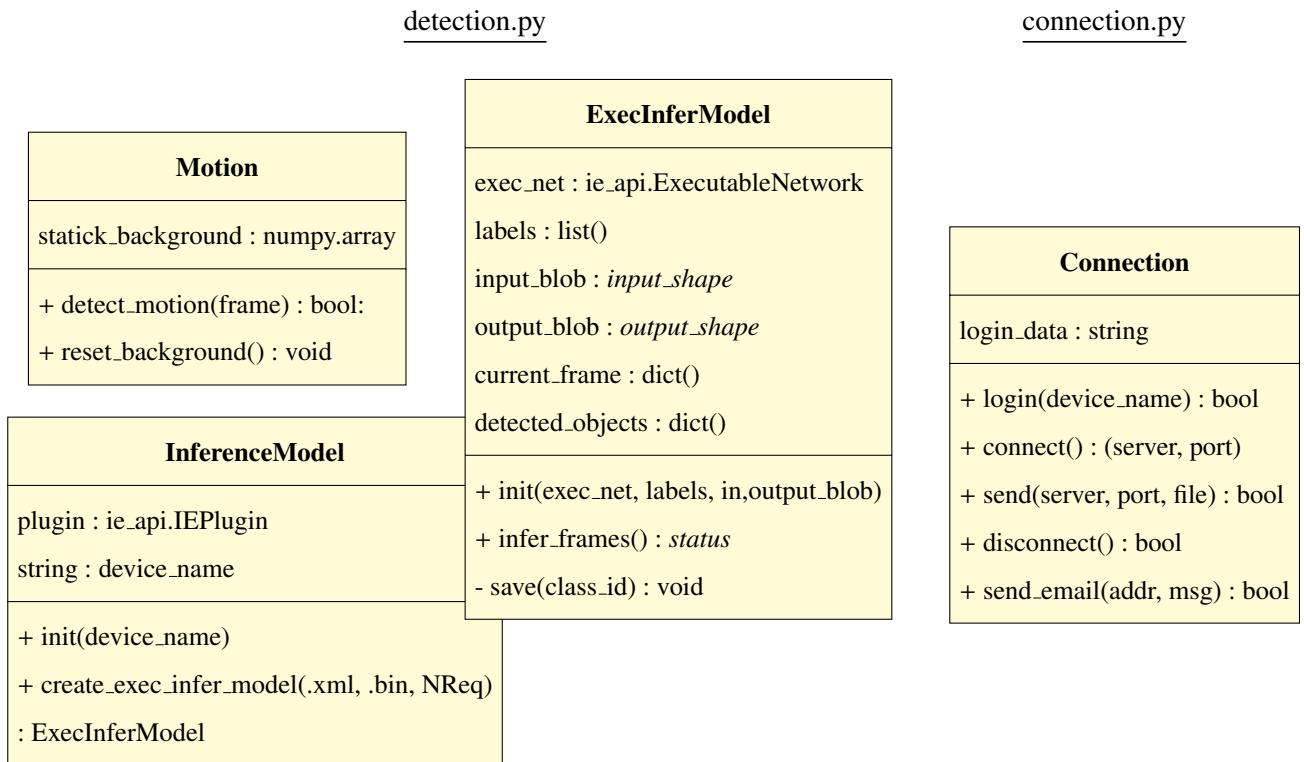


Abbildung 5.3: Klassendiagramm der Anwendung

Die Klasse *Motion* dient zur Erkennung von Bewegungen im Kamera-Inputstream, *InferenceModel* und *ExecInferModel* realisieren die Inferenz eines trainierten Modells und die Klasse *Connection* dient dem Aufbau einer Verbindung zu einem anderen Gerät, sowie dem Senden der erkannten Bilder darüber.

Durch geeigneten Implementierung des Applikationsablaufes, sollte eine Möglichkeit gefunden werden, trotz der langsamen Inferenzzeit, mit dem Faster R-CNN alle relevanten Frames, also die, in denen Tiere zu vermuten sind, inferieren zu können. Dafür wurde die Annahme gemacht, dass zur Laufzeit der Anwendung, nicht durchgehend inferiert werden muss, sich also Zeitweise keine Tiere und damit auch keine Bewegung vor der Kamera befinden.

Um Bewegungen feststellen zu können, wurde, mithilfe der Library *OpenCV* ein Bewegungsmelder implementiert. Dieser speichert zu Begin des Kamera-Streams ein Referenz Frame ab, mit dem alle weiteren Frames verglichen werden. Beträgt der Abstand, der einzelnen Pixelwerte im Graustufenbereich mehr, als ein bestimmter Threshold, wird dies als Bewegung gewertet. Indem nun die Frames, welche der Kamera-Stream permanent liefert, zunächst auf Bewegung überprüft werden, lässt sich unnötiges inferieren vermeiden, was Zeit und Energie kostet. Frames, die Bewegung enthalten, und aufgrund der langsamen

Inferenzzeit des Faster R-CNN nicht sofort inferiert werden können, werden in einem Buffer zwischen gespeichert und in Phasen, zu denen keine Bewegung stattfindet, inferiert.

Dafür musste der in Abschnitt 4.4 beschriebene asynchrone Inferenzablauf dahingehend angepasst werden, dass kein blockierendes warten auf ein Inferenzergebnis stattfindet, wodurch die Inferenz komplett zeitasynchron zu den Input-Frames abläuft. Der Gesamtlauf der Applikation ist in Abbildung 5.4 schematisch als Flussdiagramm dargestellt.

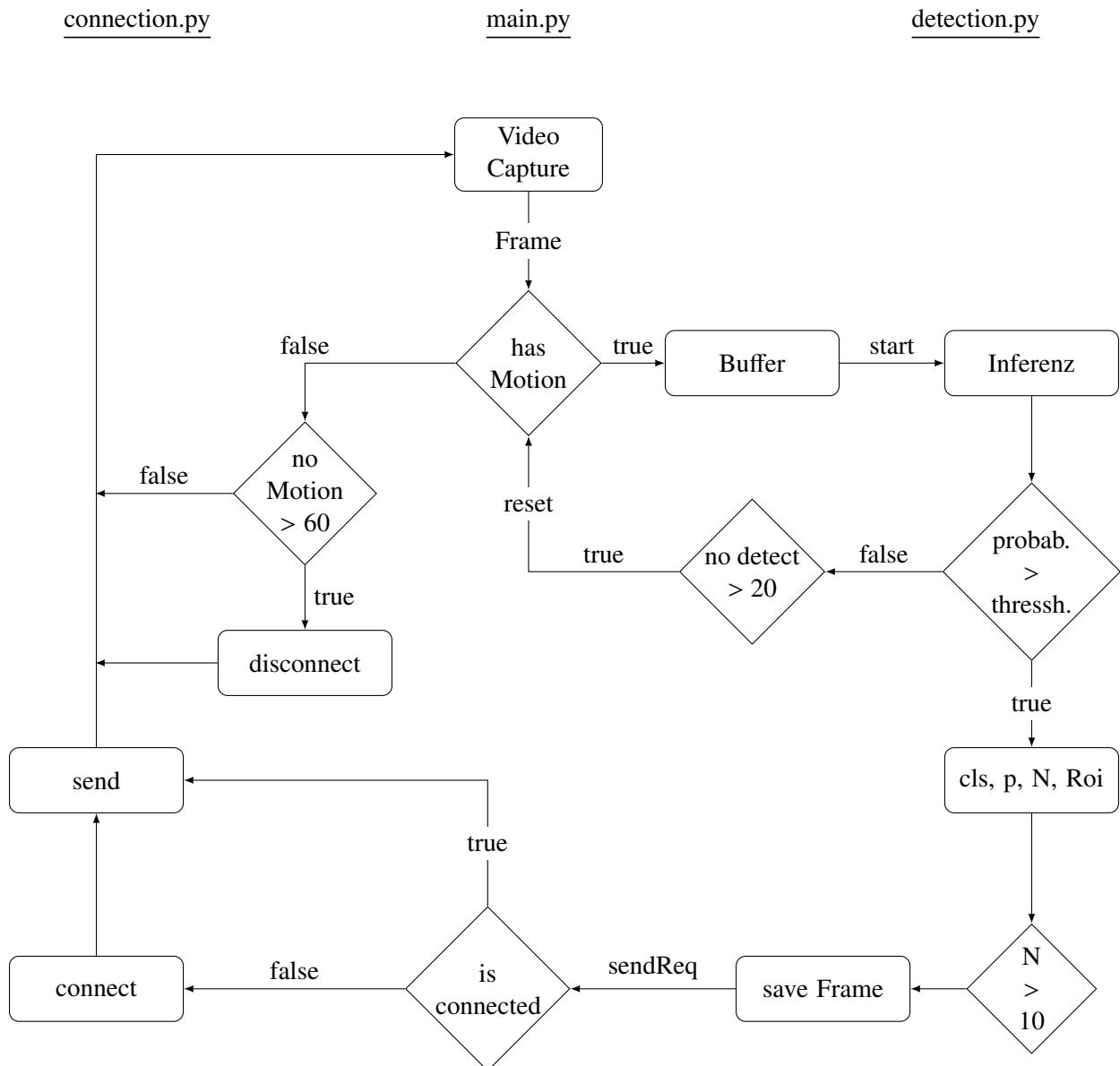


Abbildung 5.4: Schematischer Ablauf des Applikationscode

Wird in inferierten Frames mehrfach nichts erkannt, wird das Referenz-Frame des Bewegungsmelders durch ein aktuelles Frame ersetzt.

Erkannte Objekte werden in einer Datenstruktur (Python Dictionary) zusammen mit Klassenname (cls), Wahrscheinlichkeit (p) Anzahl an Erkennungen (N) sowie die Bounding Box Koordinaten (Roi, (Region of Interest)) abgespeichert.

Nach einer bestimmte Anzahl an Erkennungen des selben Objekts, wird dieses als lokale Bilddatei abgespeichert und ein Send-Requst an das Main Script zurückgegeben.

Dieses prüft dann ob eine Verbindung zu einem anderen Gerät besteht, stellt diese gegebenenfalls her, und sendet die lokal abgespeicherten Bilder.

Um nicht permanent die Verbindung zu einem Pc aufrecht erhalten zu müssen, was das Datenvolumen des mobilen Internets schneller aufbrauchen würde, wird diese nach einer bestimmte zeit ohne Bewegung getrennt.

Im folgenden werden die Funktionsweise der Inferenz sowie der Verbindungsauflauf genauer erklärt.

## Inferenz

Der im Abschnitt 4.4 beschriebene asynchronen Inferenzablauf wurde dahingehend angepasst, dass eine beliebige Anzahl an Inferenz-Requests verwendet werden kann und dass das Warten auf ein Inferenz Ergebnis nicht mehr blockierend ist. Dafür wurde der Timeout in der Wait-Funktion auf *0ms* gesetzt. In Algorithmus 3 ist der Inferenzablauf als Pseudocode dargestellt.

---

### Algorithm 3 Asynchrone Inferenz, ohne Blockierung

---

```

while true do
    capture FRAMES
    for all InferRequests do
        if wait for InferRequest is 0 then
            Result ← InferRequest.output
        end if
        if Buffer not empty then
            preprocess InferRequest
            start InferRequest
        end if
        if Result not NULL then
            process Result
        end if
    end for
end while
```

---

## Connection

Um die Bilder mit erkannten Tieren an ein anderes Gerät z.B. einen Pc senden zu können, musste eine Verbindung hergestellt werden, die auch über verschiedene Netzwerke hinweg funktioniert.

Um unabhängig von Router Konfigurationen und Firewall Einstellungen zu sein, wurde mithilfe des Dienstes *remot3.it* [32] eine Cloud-basierte Remote Verbindung hergestellt.

Mit dieser war es möglich eine Remote-Proxy Secure Shell Protocoll (SSH) Verbindung, über das Internet zu einem anderen Gerät herzustellen.

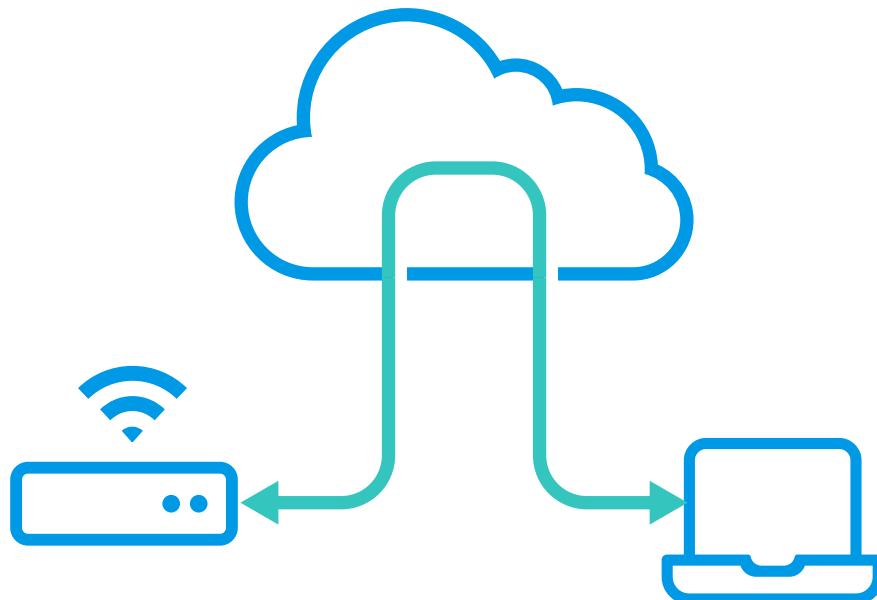


Abbildung 5.5: Prinzip Proxy Verbindung

Da die Daten vom *Raspberry Pi* aus automatisch gesendet werden sollen, wurde der *Pc*, an den sie Daten gesendet werden, als *Remote-Gerät* implementiert.

Gesendet wurden die Daten über das *Secure Copy Protocol* (SCP) welches *SSH* verwendet.

Dieses lässt sich über folgendes Kommando, welches im *connection.py* Script ausgeführt wird, bedienen:

```
$ scp -P port file.jpg user@proxyadresse /zielpfad/file.jpg
```

Server und Port werden dabei von *remote.it* generiert, *file.jpg* ist das zu sendende Bild und *user* der Nutzernname des Geräts, an welches gesendet wird. Um das Einloggen sowie den Verbindungs Auf- und Abbau über *remote.it* zu einem Gerät automatisieren zu können, bietet *remote.it* eine API mit der über Post- und Get-Requests die Befehle dafür programmatisch aufgerufen werden können.

Um den Nutzer bei einer Erkennung automatisch per Email zu benachrichtigen, wurde eine Funktion implementiert, welche das Smart Mail Transfer Protokoll (SMTP) verwendet.



## Kapitel 6

# Zusammenfassung und Ausblick

Ziel der Arbeit war, es ein autonomes Kamerasystem zur Wildtierkennung, mithilfe Neuronaler Netze, zu entwickeln. Dafür wurden vortrainierte, CNN basierte, Deep Learning Modelle zur Objekterkennung auf einen geeigneten Datensatz trainiert. Dadurch sollte es möglich sein, nicht nur die Anwesenheit eines Tieres zu erkennen, sondern auch eine Klassifizierung der Tierart vorzunehmen, wodurch das System gezielter für eine bestimmte Anwendung eingesetzt werden kann. Zur Realisierung, wurde neben einem *Raspberry Pi*, sowie einer nachtsichtgeeigneten Kamera, für die Inferenz der *Neural Compute Stick 2* von *Intel* verwendet, um die Verarbeitung der Daten auf dem Gerät ausführen zu können.

Für das Training wurde ein Datensatz, bestehend aus 9 Wildtierklassen verwendet, welcher aus *OpenImages* herunter geladen werden konnte. Anschließend wurden die Daten für das Training aufbereitet und zur Evaluierung, in verschiedene Sets aufgeteilt. Das Training wurde dann mithilfe des Frameworks *Tensorflow* durchgeführt, wobei die Modelle *SSD* und *Faster R-CNN* mit verschiedenen Basis CNNs und Parameter-Einstellungen verwendet wurden. Durch anschließende Evaluierung, konnte festgestellt werden, welches Modell sich, bezogen auf Genauigkeit und Geschwindigkeit, am besten für die Anwendung eignet. Der letzte Schritt war es die Inferenz, zusammen mit dem Anwendungscode für den *Raspberry Pi* zu implementieren, wofür mit *OpenVino* gearbeitet wurde.

Die Evaluierung der Modelle zeigte, dass eine erhöhte Genauigkeit, mit einer langsameren Inferenzzeit einhergeht. Verbessert werden konnte die Genauigkeit zum einen durch eine Augmentierung der Daten was eine größere Robustheit gegenüber anderer Datensätze mitsich brachte, und zum anderen durch verwenden des Faster R-CNN, mit welchem auch Tiere weiter weg erkannt werden konnten. Die Performance der Inferenz konnte durch asynchrone Inferenzausführung und verwenden eines bewegungsmelders sowie zwischenspeichern der Frames verbessert werden.

Auffällig war der hohe Energieverbrauch, der durch den *Neural Compute Stick*, die Kamera mit Infrarot LEDs sowie dem Internetstick für den *Raspberry Pi* zustande kam.



# Literaturverzeichnis

- [1] R. Maksutov, “Deep study of a not very deep neural network. Part 5: Dropout and Noise.”
- [2] M. S. Researcher, PhD, “Simple Introduction to Convolutional Neural Networks.”
- [3] S. Amidi and A. Amidi, “Super VIP Cheatsheet: Deep Learning,” p. 13.
- [4] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” p. 46.
- [5] “ImageNet Large Scale Visual Recognition Competition (ILSVRC).”
- [6] “CS231n Convolutional Neural Networks for Visual Recognition.” <http://cs231n.github.io/convolutional-networks/#layers>.
- [7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” vol. 60, no. 6, pp. 84–90.
- [8] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,”
- [9] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,”
- [10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,”
- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,”
- [12] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,”
- [13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,”
- [14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,”
- [15] “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” <https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5>.
- [16] A. Ouaknine, “Review of Deep Learning Algorithms for Object Detection.”
- [17] M. Häußermann, “Funktion und Effizienz von Hardware für Deep Neural Networks.”
- [18] “Intel Movidius Neural Compute Stick 2.” <https://robu.in/product/intel-movidius-neural-compute-stick-2/>.
- [19] “Intel’s Myriad X Vision Chip Incorporates Neural Network — Electronic Design.”
- [20] “OpenVINO™ Workflow Consolidation Tool.” <https://www.qnap.com/solution/openvino/de-de/>.

- [21] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, T. Duerig, and V. Ferrari, “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale,”
- [22] A. B. Jung, K. Wada, J. Crall, S. Tanaka, J. Graving, C. Reinders, S. Yadav, J. Banerjee, G. Vecsei, A. Kraft, Z. Rui, J. Borovec, C. Vallentin, S. Zhydenko, K. Pfeiffer, B. Cook, I. Fernández, F.-M. De Rainville, C.-H. Weng, A. Ayala-Acevedo, R. Meudec, M. Laporte, *et al.*, “imgaug.” <https://github.com/aleju/imgaug>, 2020. Online; accessed 01-Feb-2020.
- [23] X. Wu, D. Sahoo, and S. C. H. Hoi, “Recent Advances in Deep Learning for Object Detection,”
- [24] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,”
- [25] “Object Detection for Dummies Part 3: R-CNN Family.”
- [26] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” vol. 9905, pp. 21–37.
- [27] J. Hosang, R. Benenson, and B. Schiele, “Learning non-maximum suppression,” in *CVPR*, 2017.
- [28] “SSD : Single Shot Detector for object detection using MultiBox.”
- [29] “Google Colaboratory.” <https://colab.research.google.com/notebooks/intro.ipynb#recent=true>.
- [30] “TensorFlow Object Detection Api.” [https://github.com/tensorflow/models/tree/master/research/object\\_detection](https://github.com/tensorflow/models/tree/master/research/object_detection).
- [31] S. Beery, D. Morris, and P. Perona, “The iwildcam 2019 challenge dataset,” *arXiv preprint arXiv:1907.07617*, 2019.
- [32] “remot3.it.” <https://remote.it/>.

# Abbildungsverzeichnis

2.1	Vergleich herkömmliche- und Machine Learning- Programmierung . . . . .	7
2.2	Vereinfachte Darstellung eines Künstlichen Neuronalen Netzes . . . . .	8
2.3	Schematischer Trainingsablauf eines Neuronalen Netzes . . . . .	8
2.4	Berechnungen an einem einzelnen Neuron . . . . .	9
2.5	ReLU Funktion . . . . .	10
2.6	Sigmoid Funktion . . . . .	10
2.7	Overfitting, anhand der Losskurven . . . . .	11
2.8	Annäherung unterschiedlich komplexer Modelle an die gleichen Datenpunkte . . . . .	11
2.9	Prinzip Dropout [1] . . . . .	12
2.10	Faltung des Inputs mit einer Filter Matrix, Quellen: [2] und [3] . . . . .	13
2.11	Faltung mit Filtermatrix zur Erkennung vertikaler Linien . . . . .	13
2.12	LeNet-5 Architektur [4] . . . . .	14
2.13	Inception Module der 1. Version . . . . .	15
2.14	Inception Module der 2. Version . . . . .	15
2.15	Residual block des MobilenetV2 [15] . . . . .	16
2.16	Unterschied: Klassifikation - Objekterkennung, Quelle: [16] . . . . .	16
2.17	NCS2, [18] . . . . .	17
2.18	Myriad Chip, [19] . . . . .	17
2.19	OpenVino Workflow, angelent an [20] . . . . .	18
3.1	Ohne Augmentierung . . . . .	19
3.2	Ohne Augmentierung . . . . .	19
3.3	Anwendung von Augmentierungstechniken . . . . .	21
3.4	Faster R-CNN Architektur, [25] . . . . .	22
3.5	SSD Architektur, [28] . . . . .	22
3.6	Trainingsworkflow . . . . .	23
3.7	Ablauf der InferenceEngine . . . . .	24
4.1	Intersection over Union . . . . .	26
4.2	Confusion Matrix . . . . .	26
4.3	mAP . . . . .	28
4.4	Loss . . . . .	28
4.5	SSD . . . . .	29
4.6	Faster R-CNN . . . . .	29
4.7	SSD Mobilnet . . . . .	29
4.8	SSD Inception . . . . .	29
4.9	Faster R-CNN mit Early Stopping . . . . .	30
4.10	Faster R-CNN mit Augmentierung . . . . .	30
4.11	mAP . . . . .	31
4.12	Loss . . . . .	31
4.13	3000 Samples . . . . .	31
4.14	4000 Samples . . . . .	31

4.15	50% Augment . . . . .	31
4.16	mAP . . . . .	32
4.17	Total Loss . . . . .	32
4.18	Klassifikations Loss . . . . .	32
4.19	RPN Loss . . . . .	32
4.20	Augmentierung (normal) . . . . .	33
4.21	Augmentierung + L2 . . . . .	33
4.22	. . . . .	34
4.23	. . . . .	35
5.1	Raspberry Pi 4 . . . . .	37
5.2	Longrunner Kamera Modul . . . . .	37
5.3	Klassendiagramm der Anwendung . . . . .	38
5.4	Schematischer Ablauf des Applikationscode . . . . .	39
5.5	Prinzip Proxy Verbindung . . . . .	41

# Tabellenverzeichnis

4.1	Trainingsergebnisse von SSD und Faster R-CNN . . . . .	27
4.2	Regularisierungen . . . . .	33
4.3	Vergleich von Inferenzzeiten der Modelle in FPS . . . . .	35