

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: Prof. Dr. rer. nat. Eberhard Binder
Zweitbetreuer: Prof. Dr. rer. nat. Christian Höfert
Abgabedatum: 12.03.2020



Selbstständigkeitserklärung

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

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

.....
Ort, Datum

.....
Unterschrift

Inhaltsverzeichnis

1 Einleitung	5
2 Grundlagen	7
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
3 Realisierung Objekterkennung	19
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
4 Evaluierung	25
4.1 Evaluierungsmetriken	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
5 Entwicklung der Anwendung	37
5.1 Hardware	37
5.2 Software	38
6 Zusammenfassung	43

Abkürzungsverzeichnis

API (Application Programming Interface) Eine Programmierschnittstelle, mit der über einen definierten Satz an Regeln auf die Funktionalitäten eines anderen Programms oder eine Datenbank zugegriffen werden kann. 18, 22, 23, 35, 41

CNN (Convolutional Neural Network) Neuronales Netz, welches in sog. *Convolutional Layern* eine mathematische Faltung des Inputbilds mit einer *Filter-Matrix* durchführt. 5, 7, 13–15, 17, 21–23, 27, 34, 43

CPU (Central Processing Unit) Zentrale Recheneinheit eines Computers. 17

CSI (Camera Serial Interface) Kameraschnittstelle des Raspberry Pi's, die ein Flachbandkabel verwendet. 37

Downsampling Reduktion einer Anordnung von zeitdiskreten Werten. In der Bildverarbeitung wird durch eine Verringerung der Bildpunkte die Auflösung reduziert. 14

Faltung Mathematische Rechenoperation, bei der zwei Signale (z.b. in Form von Matrizen) übereinander verschoben und an jeder Stelle miteinander multipliziert werden. Anschließend werden die Produktterme aufsummiert. 13, 14, 16, 47

Feature Map Eine zweidimensionale Matrix, die durch die Faltung eines Inputbilds mit einer Filtermatrix entsteht. 13, 21

Fps (Frames per Second) Frequenz, die angibt wie viele Bilder pro Sekunde verarbeitet werden können. 35, 43

Framework Programmgerüst, dass die Entwicklung einer Anwendung durch vorgefertigte Strukturen unterstützt, selbst aber kein vollständiges Programm darstellt. 12, 13, 17, 22, 43

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

Inferenz Die Anwendung eines trainierten Machine-Learning-Modells für neue Inputdaten. 1, 7, 17, 19, 21, 23, 24

Künstliche Neuronale Netze Eine Form des Machine Learnings, bei der eine Vielzahl künstlicher Neuronen miteinander verbunden sind. Indem die Verbindungen unterschiedlich stark gewichtet sind, lassen sich für gegebene Eingaben die richtigen Ausgaben finden. 7, 8

Machine Learning Ein Teilgebiet der Künstlichen Intelligenz, dass sich mit selbstlernenden Algorithmen befasst. Damit können ohne explizite Programmierung Zusammenhänge in größeren Datenmengen erkannt werden. 7, 8

Overfitting Überanpassung eines Machine Learning Modells an die Trainingsdaten, wodurch keine Generalisierung für neue Daten mehr stattfinden kann. 11, 12, 15, 27, 28, 32

RPN (Region Proposial Network) Ein CNN-basiertes Modell, dass im *Sliding-Window*-Verfahren aus den Feature Maps räumliche Vorschläge für Objekte in einem Inputbild generiert. 21, 32

SCP (Secure Copy Protocoll) Protokoll, welches über eine SSH Verbindung die verschlüsselte Datenübertragung zwischen zwei Geräten ermöglicht. 41

SMTP (Simple Mail Transfer Protokoll) Protokoll zum Senden von E-Mails. 41

SoC (System on Chip) Komplexes System, bestehend aus einer CPU, einer GPU und ggf. weiteren Komponenten die gemeinsam auf einem Chip verbaut sind. 17

SSH (Secure Shell Protocoll) Netzwerkprotokoll zur Herstellung einer verschlüsselten Netzwerkverbindung zu einem entfernten Gerät. 40, 41

Thread Bezeichnet einen Ausführungsstrang eines Computerprogramms und ist damit Bestandteil eines Prozesses. 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 Bei der Verrechnung zweier ungleich großer Matrizen werden fehlende Werte mit Nullen aufgefüllt, um die Ausgangsgröße für das Ergebnis beizubehalten. 14

Kapitel 1

Einleitung

Wildkameras werden vielfach von Jägern genutzt, kommen zu Forschungszwecken zum Einsatz und werden auch zunehmend von Privatpersonen verwendet, die ihr eigenes Grundstück überwachen möchten.

Da das Aufnehmen der Bilder automatisch über einen Bewegungsmelder erfolgt, unabhängig davon, wodurch der Bewegungsmelder ausgelöst wird, kann es unter Umständen zu einer großen Menge an unwichtigen Daten kommen. Diese benötigen Speicherplatz auf dem Gerät oder verbrauchen bei automatischem Senden hohes Datenvolumen für eine mobile Netzwerkverbindung. Außerdem bringen sie einen großen Auswertungsaufwand mit sich, da nichtrelevante Daten ausgefiltert werden müssen.

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 von Füchsen im eigenen Garten oder zum Aufspüren von Wölfen und Bären in bestimmten Waldgebieten. Auch der Artbestand seltener oder aussterbender Tierarten könnte so leichter erfasst werden. Ziel der vorliegenden Arbeit ist es, ein autonomes Kamerasytem zu entwickeln, das mithilfe von Deep-Learning-Algorithmen verschiedene Wildtierarten erkennen und klassifizieren kann. Für die Umsetzung einer solchen Bilderkennungsaufgabe sind insbesondere Deep-Learning-Algorithmen geeignet, die ein Teilgebiet der künstlichen Intelligenz sind. Die Anwendung soll auf dem Einplatinencomputer *Raspberry Pi 4* ausgeführt werden und die 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 der Tiere auch bei Dunkelheit möglich sein.

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.

Die Arbeit gliedert sich zunächst in ein Grundlagenkapitel, in dem die Funktionsweise von Künstlichen Neuronalen Netzen für die Bilderkennung behandelt wird. Anschließend geht es um die Umsetzung und Auswertung des Trainings geeigneter Deep-Learning-Modelle. Der letzte Teil beschreibt die Entwicklung der Anwendung, in welcher die Inferenz eines fertig trainiertes Modells für den *Neural Compute Stick 2* implementiert wird.

Kapitel 2

Grundlagen

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

2.1 Machine Learning

Beim Machine Learning, einem zentralen Begriff in der Künstlichen Intelligenz, geht es um die Erstellung von Algorithmen, die Zusammenhänge in großen Datenmengen erkennen und daraus Regeln ableiten 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 Ausgabedaten abzuleiten. Dadurch unterscheidet sich das Vorgehen wesentlich von der Programmierung eines klassischen Programms, bei dem die Regeln vorab definiert werden müssen, wie in Abbildung 2.1 veranschaulicht ist:

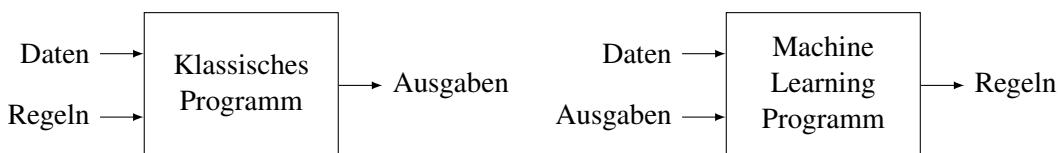


Abbildung 2.1: Vergleich herkömmliche und Machine-Learning-Programmierung

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

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 Rückmeldung zu seinen Aktionen bekommt und dadurch sein Vorgehen optimiert.

In dieser Bachelorarbeit wurde das *Supervised Learning* verwendet, da sich damit die Aufgabe zur Erkennung und Klassifizierung von Objekten am besten realisieren lässt.

2.1.1 Künstliche Neuronale Netze

Für die Verarbeitung komplexer Inputdaten, wie beispielsweise Bilder, bei denen die einzelnen Pixelwerte die Eingaben und der Inhalt des Bildes die gesuchte Ausgabe darstellen, eignen sich in besonderer Weise

die Künstlichen Neuronalen Netzen. Diese sind eine Teilgebiet des Machine Learnings und bestehen aus einer Vielzahl an programmatisch erzeugten künstlichen Neuronen, die in Schichten angeordnet miteinander verbunden sind.

Durch unterschiedlich starke Gewichtungen der einzelnen Verbindungen, welche als Gewichte bezeichnet werden, können zu gegebenen Eingaben die richtigen Ausgaben gefunden werden, wie in Abbildung 2.2 schematisch dargestellt ist.

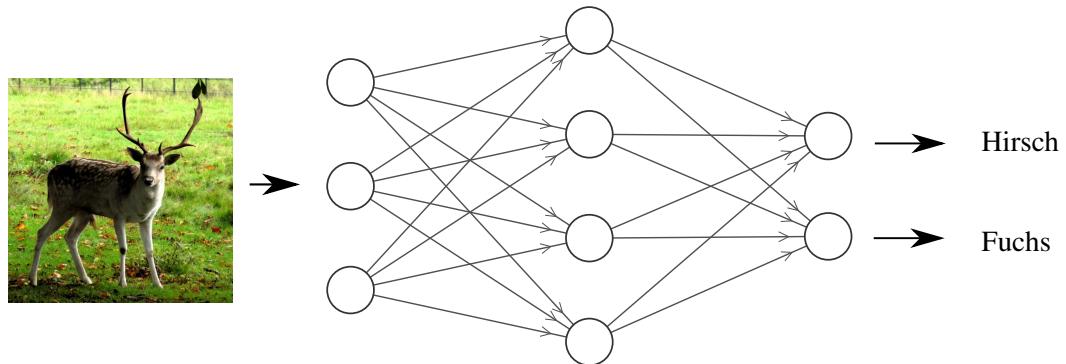


Abbildung 2.2: Vereinfachte Darstellung eines Künstlichen Neuronalen Netzes

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

- *Forward Pass* Für die Inputdaten anhand aktueller Gewichte eine Vorhersage für die Ausgabe treffen
- *Fehlerbestimmung* Abweichung der gemachten Vorhersage zum tatsächlichen Wert (Labels) über eine Fehlerfunktion berechnen
- *Backpropagation* Minimierung der Abweichung durch Anpassung der Gewichte (Optimierer)

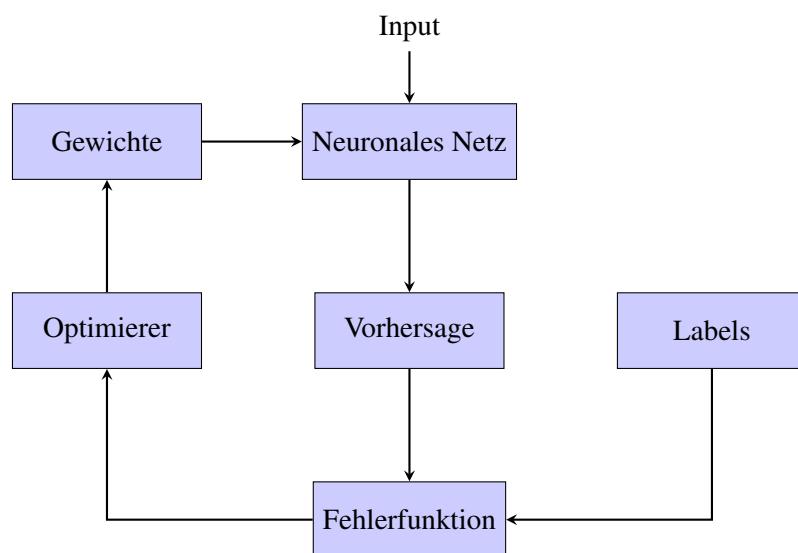


Abbildung 2.3: Schematischer Trainingsablauf eines Neuronalen Netzes

Durch mehrfaches Durchlaufen dieser Schritte kann die Fehlerfunktion soweit minimiert werden, dass das Modell auch für neue, unbekannte Inputdaten richtige 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 hindurch gereicht, um in der Ausgabeschicht 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, auf einen bestimmten Bereich skaliert.

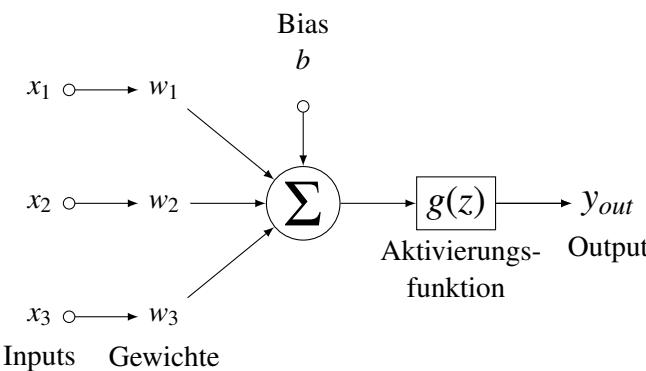


Abbildung 2.4: Berechnungen an einem einzelnen Neuron

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 (die sogenannten *Hidden Layer*) meistens um die in Abbildung 2.5 dargestellte Funktion *ReLU*, welche positive Werte beibehält und negative Werte auf 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 binäre Klassifikation die in Abbildung 2.6 dargestellte *Sigmoidfunktion* verwendet, welche die Werte s-förmig auf einen Bereich zwischen 0 und 1 skaliert.

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

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

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

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

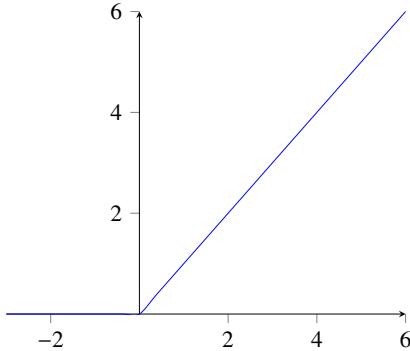


Abbildung 2.5: ReLU

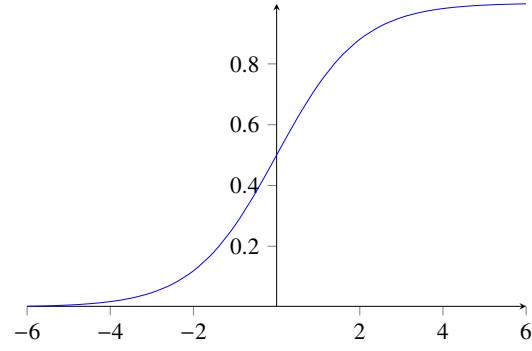


Abbildung 2.6: Sigmoidfunktion

Fehlerbestimmung

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

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

Backpropagation

Die Anpassung der Gewichte zur Minimierung der Loss-Funktion kann durch Berechnung des Gradienten der Loss-Funktion erfolgen.

Dafür wird die Loss-Funktion L für jede Schicht partiell mithilfe der Kettenregel nach den Gewichten w der jeweiligen Schicht abgeleitet, siehe Gleichung 2.4.

Mit den ermittelten Gradienten werden dann die Gewichte mit einer Schrittweite η angepasst, siehe Gleichung 2.5.

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

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

2.1.2 Validierung und Overfitting

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

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

Indem beide Loss-Werte als Funktionen 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 Loss-Wert der Trainingsdaten verringert und der Wert der Testdaten gleich bleibt oder sich wie in Abbildung 2.7 dargestellt, wieder verschlechtert.

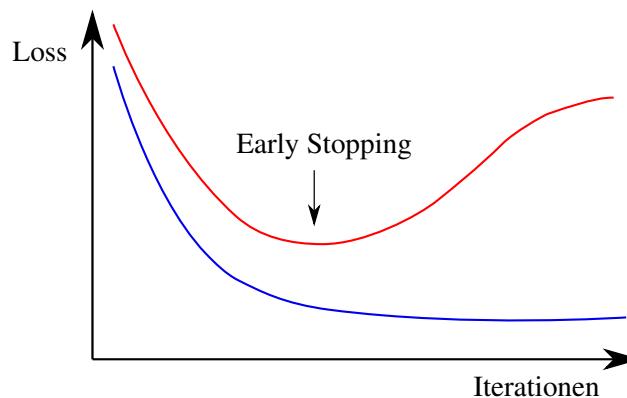


Abbildung 2.7: Overfitting dargestellt anhand der Loss-Kurven

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

Durch die große Anzahl an Parametern 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 Funktion, die sich mit unterschiedlich hohem Grad an gegebene Datenpunkte annähern soll.

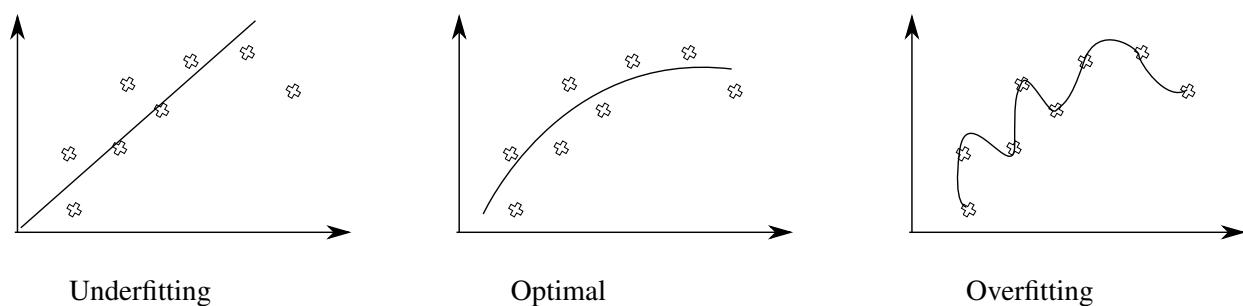


Abbildung 2.8: Annäherung unterschiedlich komplexer Funktionen an gegebene Datenpunkte

Das Auftreten von *Overfitting* kann entweder durch Verwendung einer größeren Anzahl von Trainingsdaten vermieden werden oder mit einer der folgenden Techniken:

Augmentierung

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

Regularisierung

Bei der *Regularisierung* wird der Loss-Funktion, als weiterer Term, eine Aufsummierung aller Gewichte hinzugefügt.

Die Minimierung der Loss-Funktion hat dann den Effekt, dass die Gewichte möglichst kleine Werte behalten, wodurch das Modell weniger Möglichkeiten zur Überanpassung hat.

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

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

Dropout

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

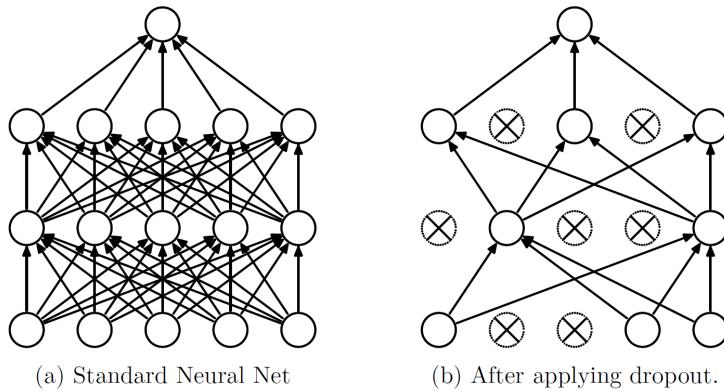


Abbildung 2.9: Funktionsprinzip Dropout [1]

Early Stopping

Beim *Early Stopping* wird das Training abgebrochen, bevor *Overfitting* stattfinden kann. Das ist, wie in Abbildung 2.7 markiert, durch das Minimum der Loss-Kurve 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, aufbauend auf vorimplementierten Programmkomponenten, eigene Modelle zu konstruieren.

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

Für diese Bachelorarbeit wurde *TensorFlow* verwendet, ein von Google stammendes Framework, welches aufgrund seiner hohen Flexibilität häufig in der Forschung verwendet wird.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) erweitern die in Abschnitt 2.1.1 beschriebenen Künstlichen Neuronalen Netze um zusätzliche Schichten, die vor der eigentlichen Klassifikation ausgeführt werden und Merkmale aus den Inputdaten herausholen. Diese Schichten erhalten die Inputdaten als zweidimensionale Matrix und führen darauf eine mathematische Faltung aus. CNNs kommen größtenteils in der Bilderkennung zum Einsatz, ein weiteres Anwendungsgebiete ist z.B. die Spracherkennung.

Anstatt dass alle Neuronen zweier benachbarter Schichten durch gewichtete Parameter miteinander verbunden sind, stellen kleinere, sogenannte Filtermatrizen, 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.

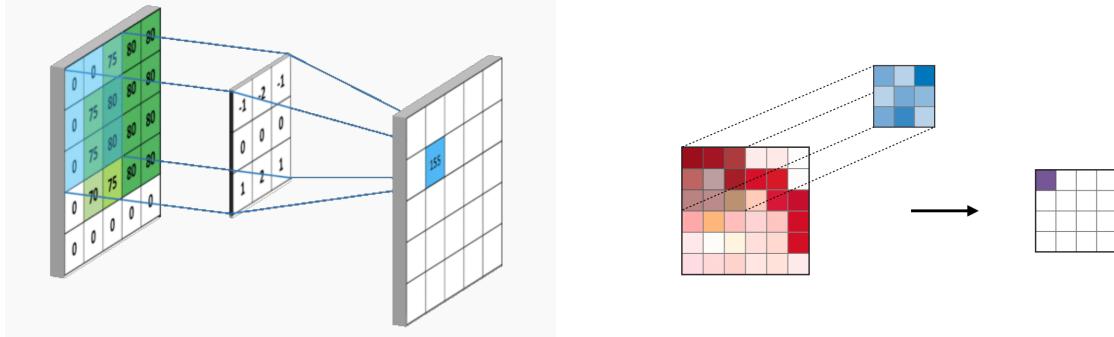


Abbildung 2.10: Faltung des Inputs mit einer Filtermatrix [2] [3]

Jedes Faltungsergebnis entspricht einem Wert in der nächsten Schicht, welche die Korrelation von Filter Matrix und Inputbild angibt. So werden in den Filtern definierte Muster, wie beispielsweise vertikale Linien, aus dem Inputbild herausholten und in den Folgeschichten in den sogenannten *Feature Maps* abgebildet.

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

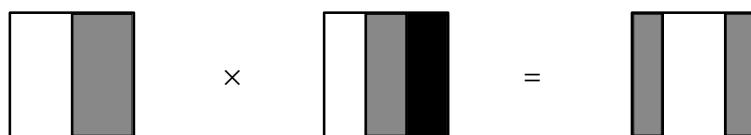


Abbildung 2.11: Faltung des Inputs mit einer Filtermatrix zur Erkennung vertikaler Linien

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 4×4 Matrix. Soll die Ausgangsgröße (hier 6×6) beibehalten werden, kann *Zero Padding* verwendet werden. Durch die Faltung entsteht eine räumliche Invarianz für das zu erkennende Objekt im Inputbild.

Den *Convolutional Layern* folgen meist *Pooling Layer* zum Downsampling, und eine ReLU Aktivierungsfunktion. Eine solche Abfolge wird als *Convolutional-Block* bezeichnet. Beim Pooling wird eine bestimmte Anzahl von Werten zusammengefasst, indem entweder das Maximum oder der Mittelwert dieser Werte verwendet wird. Durch Hintereinanderschaltung mehrerer solcher *Convolutional-Blöcke* lassen sich mit jeder weiteren Schicht auch komplexere Muster aus dem Inputbild herausextrahieren.

Die Features des letzten *Convolutional Layers* werden dann einem *Fully Connected Layer* zur Klassifikation übergeben. In Abbildung 2.12 ist die beschriebene Struktur anhand des ersten, von Yann LeCun entworfenen CNN, dargestellt.

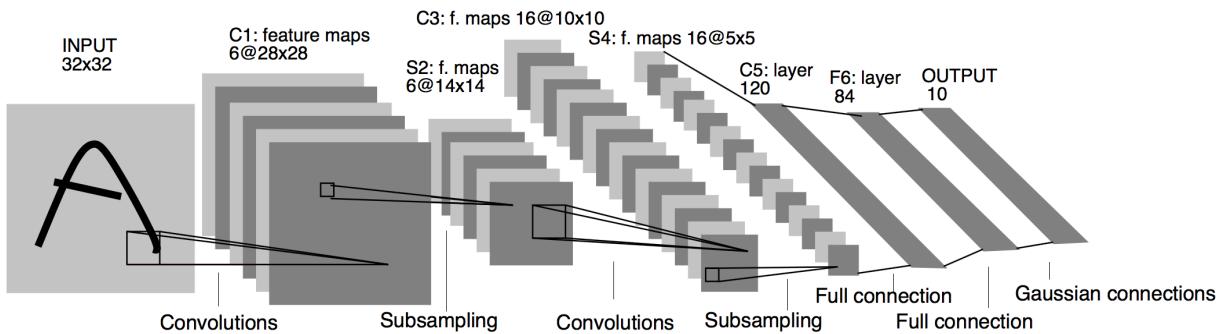


Abbildung 2.12: LeNet-5 Architektur [4]

Ein wesentlicher Vorteil gegenüber einem reinen *Feedforward Network* ist, dass durch die Verwendung von Filtermatrizen weniger Parameter verwendet werden müssen und somit ein geringerer Rechenaufwand notwendig ist. Die Werte der Filtermatrizen, welche die zu extrahierenden Muster repräsentieren, werden über die *Backpropagation* gelernt.

Da die Merkmale insbesondere in den vorderen *Convolutional Layern* für die meisten Klassen sehr ähnlich sind, werden häufig Modelle mit vortrainierten Filtern verwendet, was als *Transfer Learning* bezeichnet wird. Dadurch müssen die Gewichte nur noch geringfügig an den eigenen Datensatz angepasst werden.

2.2.1 Architekturen

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

Gemessen und verglichen werden die Ergebnisse häufig im Rahmen eines jährlich stattfindenden Software-Wettbewerbs, der *Large Scale Visual Recognition Challenge (ILSVRC)* [5].

Namhafte Modelle, welche den Wettbewerb 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 Filtergröße (3×3) bessere Ergebnisse erzielt.
- **GoogleLeNet**, (2014), auch bekannt als *Inception*, 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 Inputwert addiert wird.

Im Folgenden werden die in der Bachelorarbeit verwendeten CNN-Modelle genauer beschrieben.

GoogleLeNet (Inception)

Die Entwicklung der CNN Architekturen hat gezeigt, dass sich durch Hinzufügen weiterer Schichten, sowie durch die Verwendung einer größeren Anzahl von 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 1×1 , 3×3 und 5×5 , welche am Ende des Moduls über eine *Filter concatenation* wieder zusammengeführt werden. Zur Dimensionsreduktion werden, wie in Abbildung 2.13 dargestellt, diesen Filtern noch 1×1 Filter vorgeschaltet. Durch die *Inception-Module* erzielt das Modell mit deutlich weniger Parametern die gleichen Ergebnisse wie ein Modell ohne die Module mit entsprechend mehr Parametern. Ein weiterer Vorteil liegt darin, 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 5×5 Filter jeweils durch zwei 3×3 Filter ersetzt, was in Abbildung 2.14 dargestellt ist.

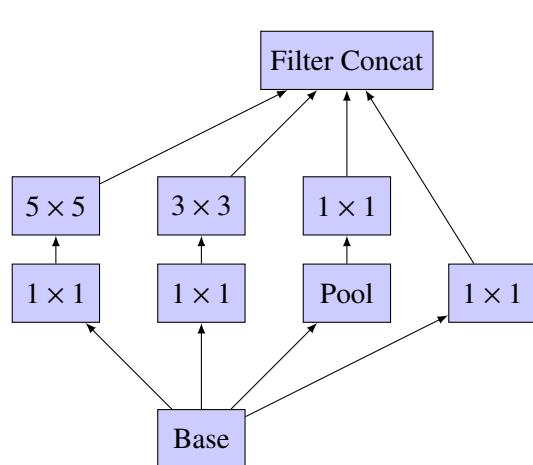


Abbildung 2.13: Inception-Module (Version 1)

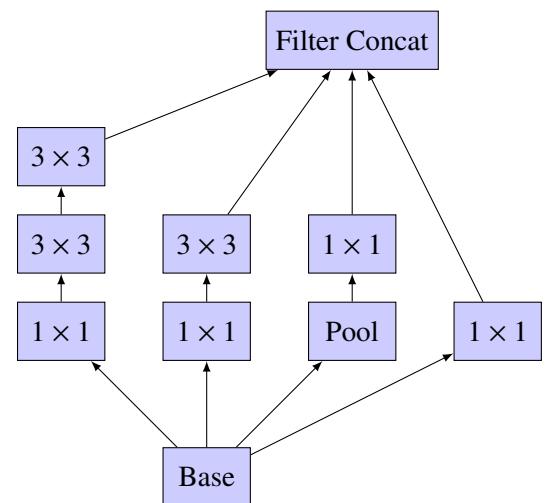


Abbildung 2.14: Inception-Module (Version 2)

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* durch sogenannten *Depthwise Separable Convolutions* ersetzt, welche die Faltung in zwei separaten Schichten ausführt. Zuerst wird eine *Depthwise Convolution* auf die drei Farbkanäle getrennt ausgeführt. Anschließend führt eine *pointwise convolution* mit einem 1×1 Filter diese wieder zusammen.

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

Zuerst wird eine 1×1 Convolution mit ReLU Aktivierungsfunktion ausgeführt, gefolgt von der *Depthwise Convolution* und zum Schluss eine weitere 1×1 Convolution mit linearer Aktivierungsfunktion.

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

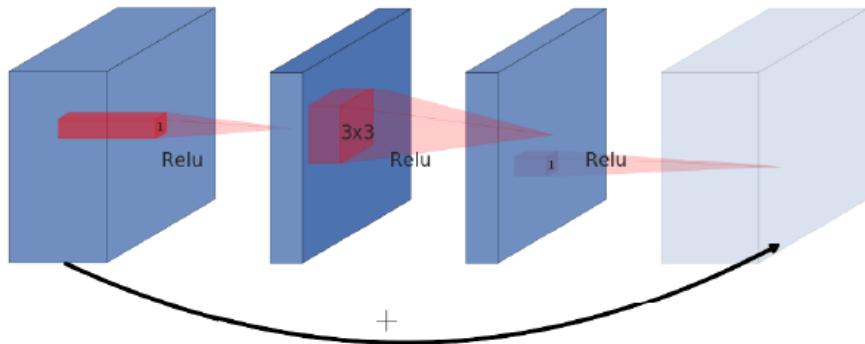


Abbildung 2.15: Residual block des MobileNetV2 [14]

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 vohersagen kann. Im rechten Bild (Objekterkennung) soll das Modell, in Form von Bounding-Boxen, zusätzlich eine Lokalisierung der Objekte vornehmen.



Abbildung 2.16: Unterschied: Klassifikation und Objekterkennung [15]

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 Koordinaten können mittels eines Regressionsverfahrens gelernt werden, wobei die geschätzten an die richtigen Koordinaten angenähert 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 Inputbild besteht, die mit hoher Wahrscheinlichkeit ein Objekt enthalten.

2.3 Neural Compute Stick 2

Da das Training und die Inferenz von Deep-Learning-Algorithmen sehr rechenintensiv sind, 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 Einplatinencomputern wie dem *Raspberry Pi* ausgeführt werden, sind dementsprechend durch die Rechenleistung der Hardware limitiert.

Eine Möglichkeit, diese Einschränkung zu umgehen, besteht darin die Bilddaten für die Verarbeitung an eine Cloud zu senden, wo sie von einem leistungsstärkeren Rechner inferiert und anschließend wieder zurückgesendet werden.

Sollen die Daten, wie dies beim *Edge Computing* der Fall ist, auf dem Anwendungsgerät direkt verarbeitet werden, gibt es speziell für die Inferenz von Deep-Learning-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 aus der Neural Compute Engine zur beschleunigten Berechnung Neuronaler Netze, einem Bildbeschleuniger, 16 SHAVE Prozessoren, einem Bildsignalprozessor sowie einem RISC CPU Core. [16]



Abbildung 2.17: Neural Compute Stick 2
[17]

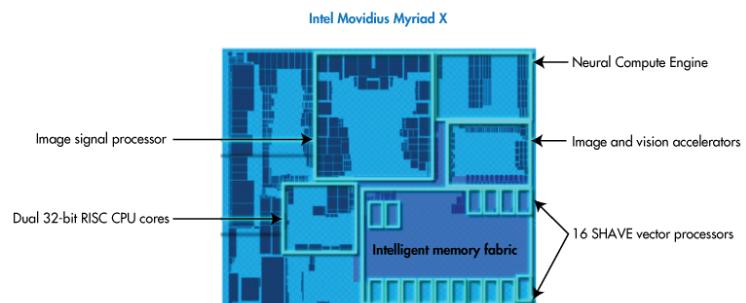


Abbildung 2.18: Movidius Myriad X [18]

2.3.1 OpenVino Toolkit

Zur Ausführung der Inferenz eines trainierten Deep-Learning-Modells auf dem NCS2 wird das Toolkit *OpenVino* von Intel verwendet. Dabei handelt es sich um eine Software Plattform zur Optimierung und Inferenz von CNN-basierten Modellen auf verschiedener Intel-Hardware.

Für die Modelle wird dabei ein eigenes Dateiformat 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, die in den Frameworks *TensorFlow*, *Caffe*, *ONNX*, *Kaldi*, oder *MXNET* trainiert wurden, in das IR-Format konvertiert werden.

Um diese Modelle 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 kann.

In Abbildung 2.19 ist der Workflow mit *Openvino* dargestellt, der das Training eines Deep-Learning-Modells mit der Implementierung einer Nutzeranwendung verbindet

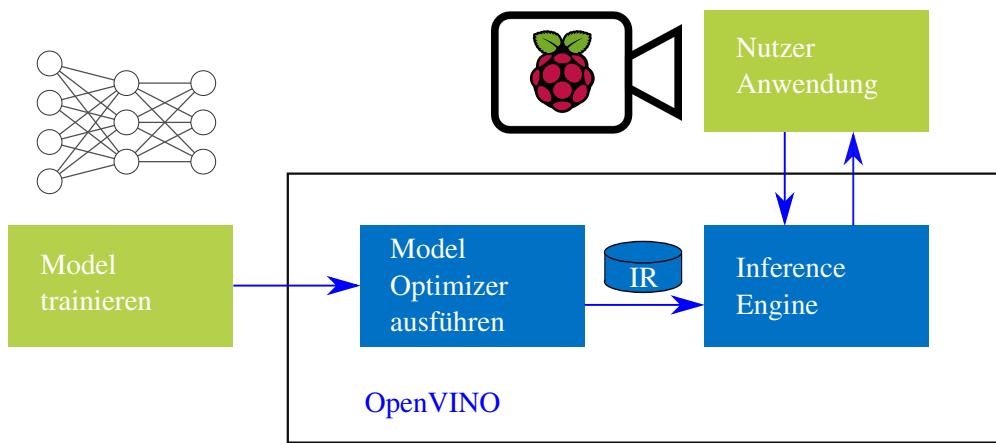


Abbildung 2.19: OpenVINO Workflow orientiert an [19]

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 Modelle 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 *Open Images* aus dem Internet heruntergeladen werden.

Für die Bachelorarbeit wurden aus dem Open Source Datensatz *Open Images* von Google [20], 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, in ein Trainings-, ein Validierungs- und ein Testset aufgeteilt, mit einem Verhältnis von 80%, 10%, 10%.

Bei den verschiedenen Klassen variierte die Anzahl der Bilder zwischen 200 und 2000 Stück, wodurch eine Verteilung der Klassen, wie in dem in Abbildung 3.1 dargestelltem Histogramm, zustandekam.

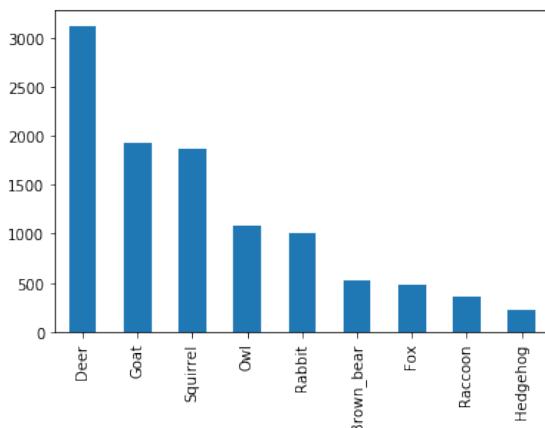


Abbildung 3.1: Verteilung der Klassen

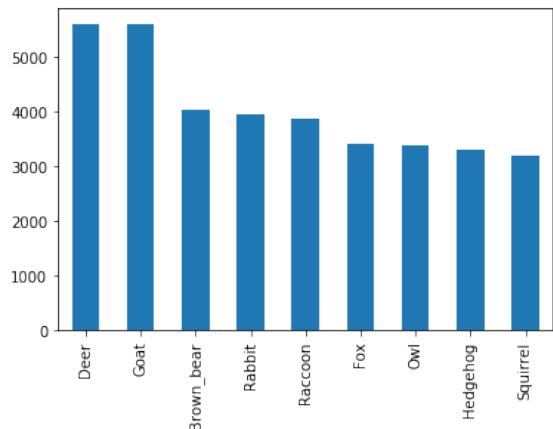


Abbildung 3.2: Verteilung der Klassen nach Augmentierung der Daten

Um diese quantitativen Unterschiede auszugleichen, wurden die Daten, wie im nächsten Abschnitt genau-

er beschrieben wird, so augmentiert, dass für jede Klasse 3000 Bilder vorhanden waren, was zu einer in Abbildung 3.2 dargestellten Verteilung führte.

Da sich häufig mehrere Tiere derselben 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 erfolgt, indem geometrische Transformationen oder Manipulationen an den Pixelwerten auf die Bilder angewendet werden. Die Anzahl der Bilder wird durch dieses Verfahren künstlich vermehrt. Neben dem Ausgleich der Klassenbalance ist dies auch eine sehr effektive Technik, um Overfitting zu verhindern.

Die Augmentierung des *Open Images* Datensatzes wurde mithilfe eines Python-Scripts, in welchem die Library *imgaug* [21] verwendet wurde, durchgeführt. Dabei wurde für jedes zu augmentierende Bild eine geometrische und eine pixelbezogene Transformation angewendet, die 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:

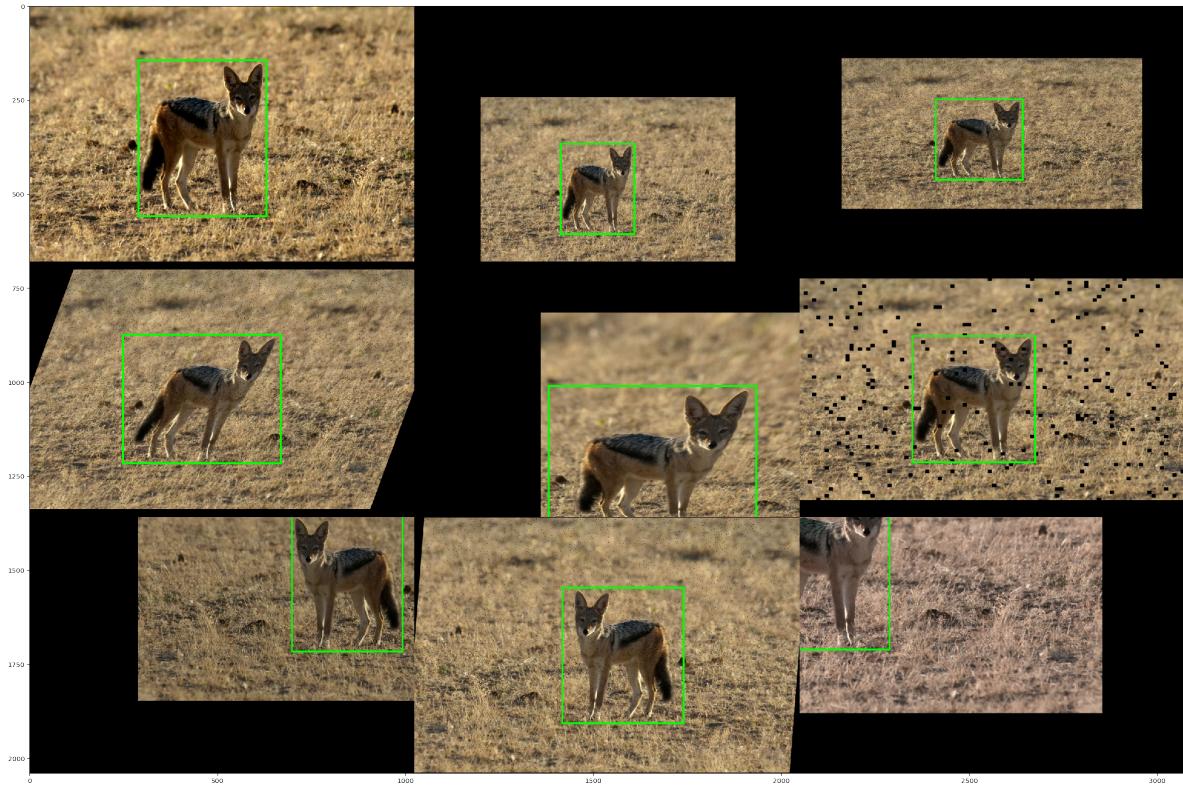


Abbildung 3.3: Beispielhafte Augmentierungsergebnisse für Bilder der Klasse *Fuchs*

3.2 Object detection Modelle

Object detection Modelle lassen sich in einstufige- und zweistufige Detektoren einteilen, siehe [22] für eine ausführliche Beschreibung.

Zweistufige Detektoren generieren in der ersten Stufe eine Auswahl an räumlichen Vorschlägen für das Inputbild, in dem 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 des Vorhandenseins eines Objekts klassifiziert.

Für diese Bachelorarbeit wurden Modelle beider Ansätze verwendet und hinsichtlich Genauigkeit und Inferenzenzeit miteinander verglichen. Im Folgenden werden die beiden verwendeten Modelle näher erläutert.

Faster R-CNN

Das Faster R-CNN [23] ist ein Modell zur Objekterkennung, welches ein zweistufiges Verfahren verwendet und in Abbildung 3.5 dargestellt ist. Die Vorschlagsgenerierung erfolgt in einem *Region Proposial Network (RPN)*, welches auf einem *fully convolutional network* basiert. Über die generierten *Feature Maps* werden im *Sliding-Window*-Verfahren vordefinierte *Anker Boxen* konvolviert. Der daraus resultierende Feature-Vektor wird einem binären Klassifikator (*cls layer*), der angibt, ob sich ein Objekt in dem Vorschlag befindet, sowie einem Bounding-Box-Regressor (*reg layer*) zur Lokalisierung, übergeben.

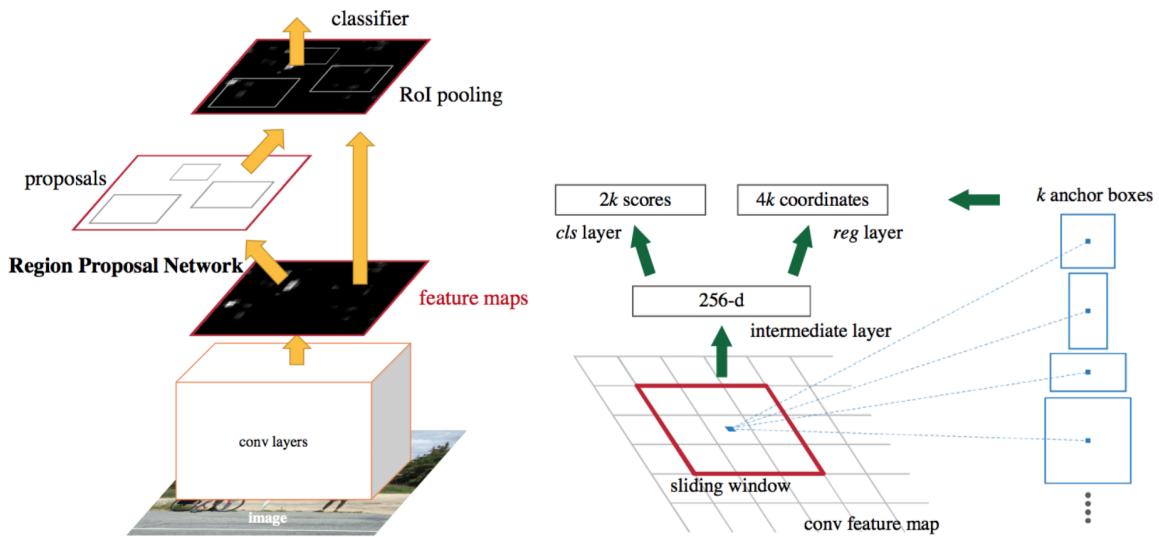


Abbildung 3.4: Faster R-CNN Architektur [23]

SSD: Single Shot MultiBox Detector

Der Single Shot MultiBox Detector (SSD) [24] verwendet ein einstufiges Verfahren zur Objekterkennung, bei dem das Inputbild 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 übergeben, der Boxen, die der selben Klasse angehören zusammenführt.[25]

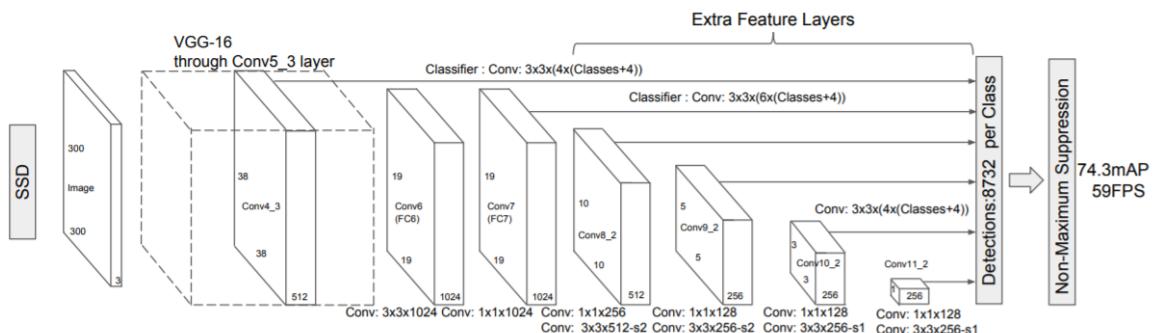


Abbildung 3.5: SSD Architektur [24]

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 Objekterkennung 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* [26] durchgeführt, die eine für Deep Learning geeignete, GPU kostenlos zur Verfügung stellt.

3.3.1 TensorFlow Object Detection API

Die *TensorFlow Object Detection API* ist unter den Research Modellen des offiziellen TensorFlow Repositories auf *GitHub* zu finden [27] und enthält Implementierungen einiger gängiger Object-detection-Modelle mit verschiedenen vortrainierten Basis-CNNs.

Der für diese Bachelorarbeit 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 von der TensorFlow API verwendete binäre Dateiformat *TFRecords* umgewandelt werden. Dieses ist eine serialisierte Darstellung der Bilder und Labelfiles als *Protocol Buffer*, welche einen schnellen 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 Modellkonfiguration festgestellt und korrigiert werden, indem z.B. andere Augmentierungs-techniken verwendet oder *Hyperparameter* des Modells umgestellt wurden.

In Abbildung 3.6 ist der so entstandene Trainingsworkflow dargestellt.

Die Ergebnisse der trainierten Modelle werden im nächsten Kapitel diskutiert.

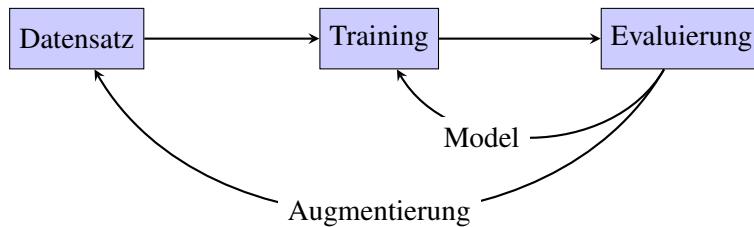


Abbildung 3.6: Trainingsworkflow

3.4 Inferenz

Zur Ausführung der Inferenz eines fertig trainierten Modells 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 *Inference Engine* zur Inferenz gelesen werden.

Inference Engine

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 angegeben.

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*). Diese stellen die Dimensionen der Ein- und Ausgabeschicht des Modells darstellen. Das zu inferierende Bild, dass 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 Kamerastream, werden die Schritte *preprocess*, *Inferenz* und *process Output* in einer Schleife wiederholt.

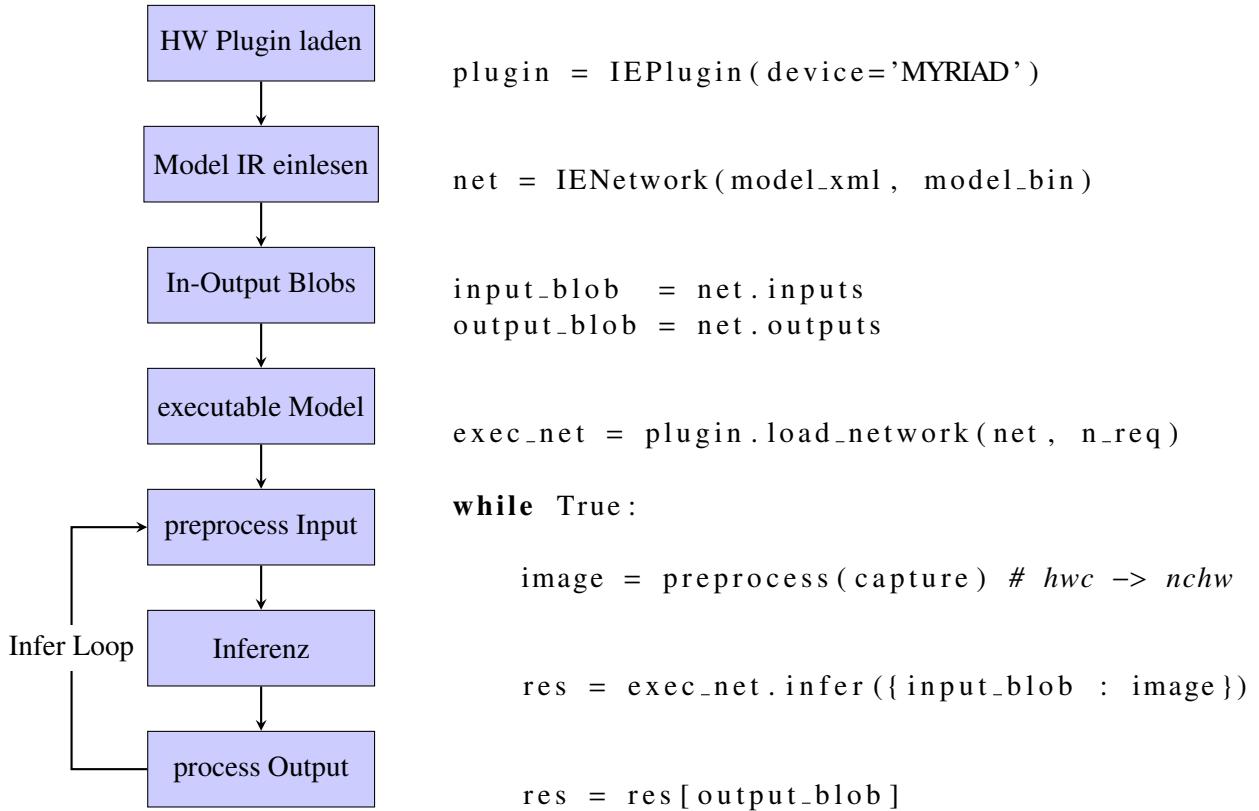


Abbildung 3.7: Programmablauf der InferenceEngine

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

Für Object-detection-Modelle enthält das Ergebnis Datenstrukturen, die den Index, die zugehörige Wahrscheinlichkeit sowie die 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 (z.B. bei 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 derer die Evaluierung erfolgte.

Im zweiten Abschnitt werden die beiden verwendeten Object detection Modelle Single Shot Detector (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 hinsichtlich der Inferenzzeit miteinander verglichen.

4.1 Evaluierungsmetriken

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

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

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

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

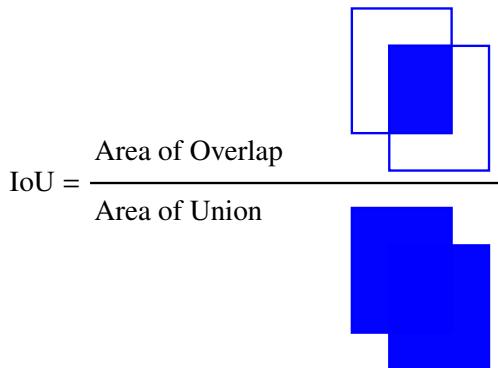


Abbildung 4.1: Intersection over Union

		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*, der 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* abgebildet, ergibt sich eine abnehmende Kurve, dessen Flächeninhalt, wie in Gleichung 4.3 dargestellt, die durchschnittliche *Precision* für diese Klasse darstellt.

Wird die durchschnittliche *Precision* 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 *Open Images* Datensatzes.

Die Berechnung der Evaluierungsergebnisse 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 sowohl für den originalen, als auch für den augmentierten Datensatz durchgeführt.

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

Tabelle 4.1: Trainingsergebnisse: 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 auf diese Weise auftretendes 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 Modellstruktur zurückzuführen sein kann.

Je mehr Parameter einem Modell zur 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 der Loss-Wert nach ca. 100k Iterationen wieder zu, wohingegen der Loss beim Training mit augmentierten Datensatz den Wert weitestgehend beibehält.

Early Stopping war ein weiterer Ansatz, das Overfitting beim Faster-R-CNN-Modell zu verhindern. Dafür wurde das Training vor der Wiederzunahme des Loss-Wertes 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.

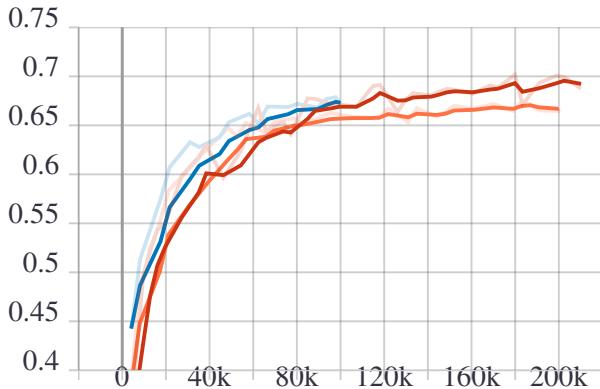


Abbildung 4.3: Trainingsverläufe: Faster R-CNN, mAP

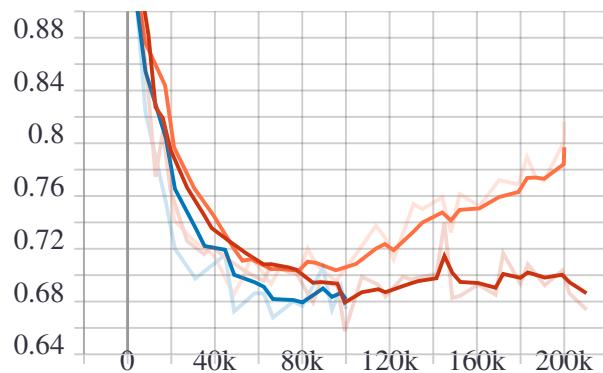


Abbildung 4.4: Trainingsverläufe: Faster R-CNN, Loss

● Ohne ● Early Stopping ● Augmentierung

Daraus ließe 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 *Open Images* Datensatzes verwendet. Da diese jedoch sehr ähnlich zu den Trainingsdaten sind, wurden auch Bilder aus anderen Quellen inferiert. Als weiterer Testdatensatz wurden, zum einen Teile des *iWildCam 2019 Datasets* [28], und zum anderen eigene Aufnahmen von Tieren verwendet.

Dadurch lässt sich die Robustheit des Modells gegenüber anderer Datensatzausprägungen feststellen, wie beispielsweise die Qualität der Bilder, Beleuchtung, oder geographische Lage. Durch ein entsprechend 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.

Open Images Testset

Die Inferenzergebnisse des *Open Images* 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 entfernt, 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.

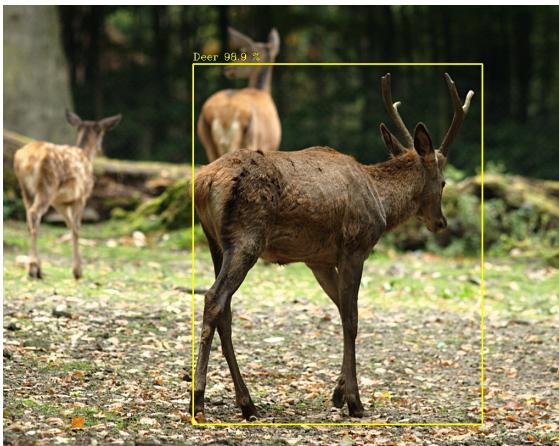


Abbildung 4.5: Inferenzergebnis: SSD

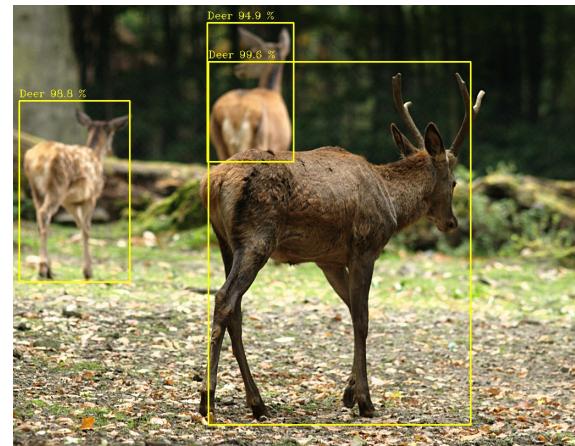


Abbildung 4.6: Inferenzergebnis: Faster R-CNN

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 entfernt und damit kleiner abgebildet waren, besser von den Faster R-CNN Modellen erkannt wurden.

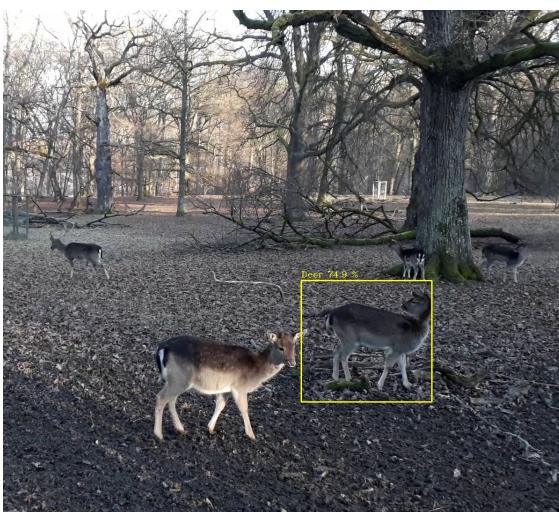


Abbildung 4.7: Inferenzergebnis: SSD-Mobilnet

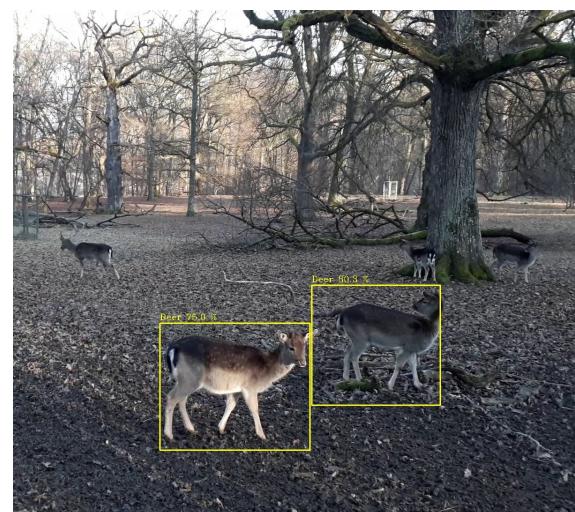


Abbildung 4.8: Inferenzergebnis: SSD-Inception

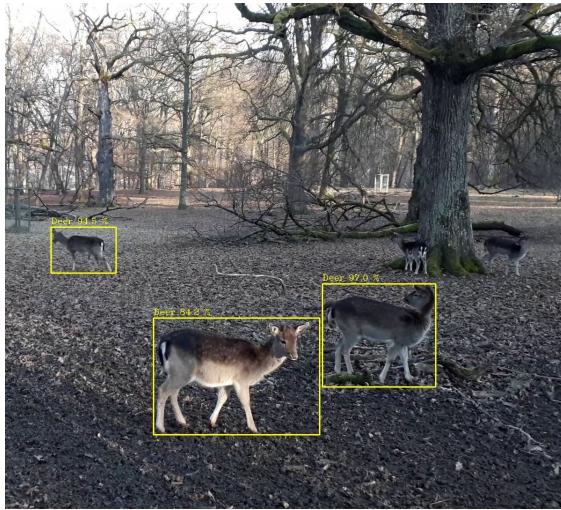


Abbildung 4.9: Inferenzergebnis: Faster R-CNN mit Early Stopping

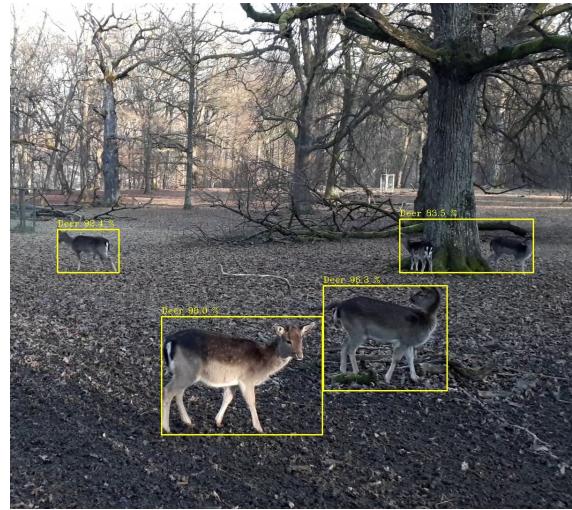


Abbildung 4.10: Inferenzergebnis: Faster R-CNN mit Augmentierung

iWildCam Datensatz

Für die Test-Inferenz wurden die Klassen aus dem *iWildCam 2019 Dataset* heruntergeladen, die sich mit den für das Training verwendeten Klassen überschnitten. Bei den Bildern des Datensatzes handelt es sich um Aufnahmen von Wildtierkameras 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 Infrarotkamera aufgenommen und daher in Graustufen sind.

Da die Inferenzergebnisse hier bei allen vier Modelvariationen nicht so gut wie bei den anderen Datensätzen waren, wurde versucht, eine Verbesserung der Inferenzergebnisse durch weitere Optimierungen zu erreichen.

4.3 Optimierungen: Faster R-CNN

Als Ausgangslage zur Verbesserung der Ergebnisse diente das Faster R-CNN mit augmentiertem Datensatz, das bei den im vorherigen Abschnitt beschriebenen Evaluierungen die besten Resultate erzielte hatte.

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 bestand darin, 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 folgenden zusätzlichen 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.

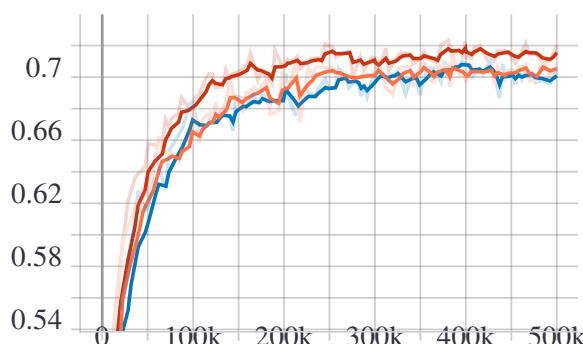


Abbildung 4.11: Trainingsverläufe: mAP

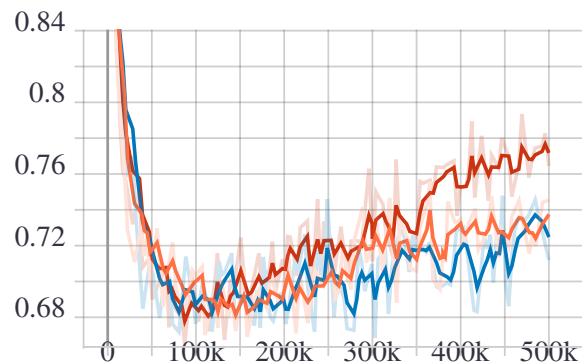


Abbildung 4.12: Trainingsverläufe: Loss

● 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.

Wie zu erwarten war, fällt das Overfitting, bei den weniger augmentierten Datensätzen stärker aus, die jedoch andererseits 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: Inferenzergebnis für 3000 Bilder



Abbildung 4.14: Inferenzergebnis für 4000 Bilder



Abbildung 4.15: Inferenzergebnis für 50% Augmentierung

Die Abbildungen 4.13 bis 4.15 zeigen beispielhaft die Inferenzergebnisse für Bilder des *iWildCam* Datensatzes, bei dem sich dieses Mal 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 zustandekommende Overfitting zu vermeiden, wurde nun zusätzlich die *L2 Regularisierung* angewendet. Diese soll, wie in dem Grundlagenkapitel (Abschnitt 2.1.2) beschrieben wurde, durch Anhängen einer Aufsummierung der Gewichte an die Loss-Funktion eine Überanpassung des Modells an die Trainingsdaten reduzieren.

In der Konfigurationsdatei des Faster R-CNN kann dies durch Setzen eines bestimmten Parameters sowohl für die erste Stufe des Modells, dem 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 der Abbildungen 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 geringen Verschlechterung des mAPs einher, wie anhand der in Abbildung 4.16 dargestellten Verläufe zu erkennen ist.

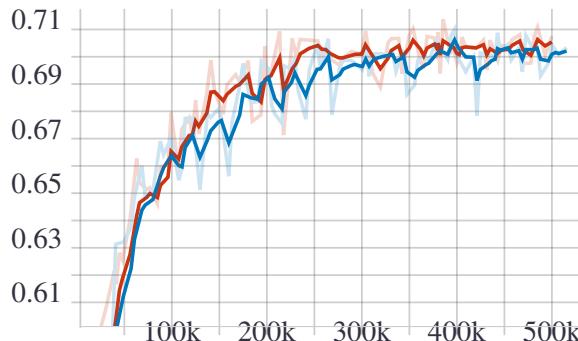


Abbildung 4.16: Trainingsverläufe: mAP

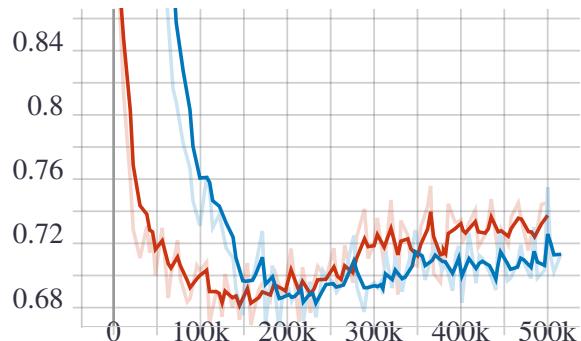


Abbildung 4.17: Trainingsverläufe: Gesamt-Loss

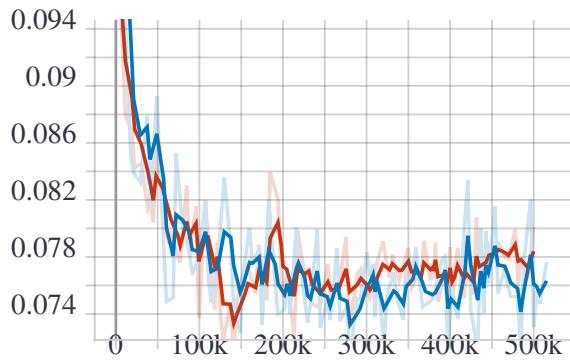


Abbildung 4.18: Trainingsverläufe:
Klassifikations-Loss

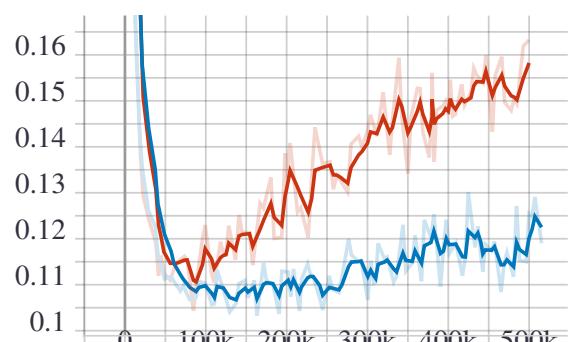


Abbildung 4.19: Trainingsverläufe: RPN Loss

● nur Augmentierung

● Augmentierung + L2 Regularisierung

Zur Feststellung der Auswirkung der unterschiedlichen Ergebnisse wurde auch hier wieder die Testinfenz angewendet, was beispielhaft für Bilder der eigenen Aufnahmen in den Abbildungen 4.20 und 4.21 dargestellt ist.

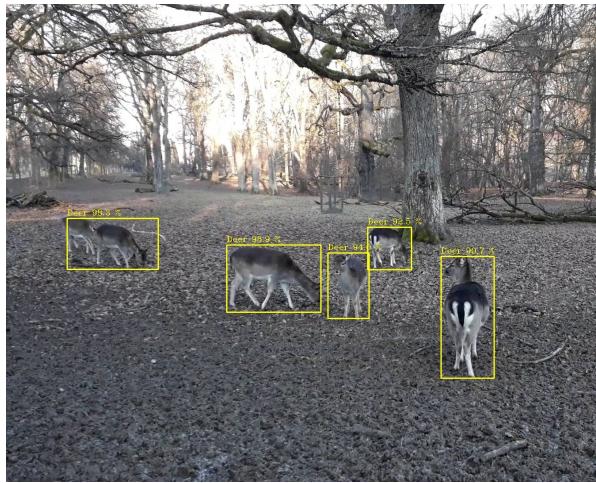


Abbildung 4.20: Inferenzergebnis für reine Augmentierung

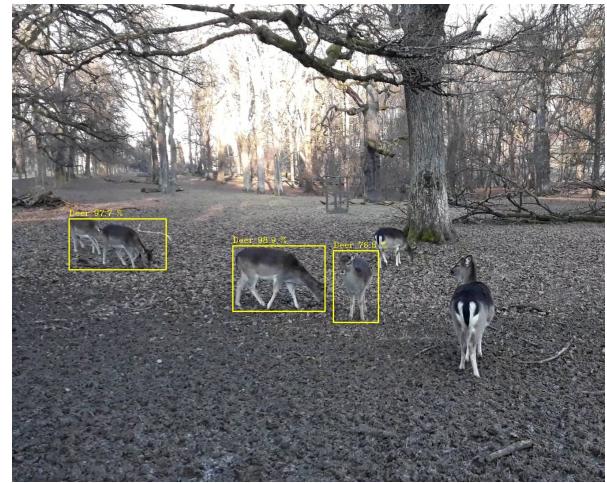


Abbildung 4.21: Inferenzergebnis für Augmentierung und L2 Regularisierung

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

Weitere angewendete Regularisierungen waren *Dropout* sowie L2 mit $\lambda = 0,02$. Diese führten jedoch zu keiner nennenswerten Veränderung der Trainingsergebnisse 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: Trainingsergebnisse: Verschiedene Regularisierungstechniken

Aus den Ergebnissen lässt 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 Trainingsiteratio-

nen 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 Modell für die Inferenz benötigt, ein weiteres Kriterium für die Auswahl des in der Anwendung zu verwendenden Modells. Einer der Faktoren, der die Inferenzzeit beeinflusst, ist die Art Hardware, auf der die Inferenz stattfindet, sowie die zur Implementierung verwendete *Library*.

Die Hardware war mit dem *Neural Compute Stick 2* festgelegt, als *Library* kamen 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 weiterer Faktor ist die Komplexität des CNNs, sowie die für die Objekterkennung verwendete Modellarchitektur. Üblicherweise sind komplexere Modelle wie das Faster R-CNN zwar genauer, dafür 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. Im Folgenden Abschnitt wird zunächst die Funktionsweise der synchronen und asynchronen Inferenzausführung mit *OpenVino* erklärt.

4.4.1 Synchrone und asynchrone Inferenz

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

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

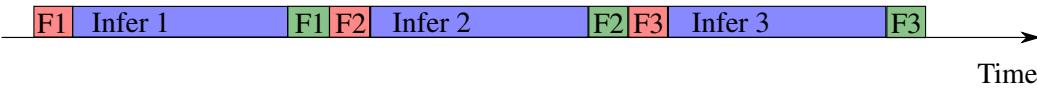


Abbildung 4.22: Zeitlicher Ablauf der synchronen Inferenz

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, die über einen bestimmten Funktionsaufruf die Inferenz in einem separaten Thread startet.

Indem vor Erhalt und Verarbeitung eines aktuellen Inferenzergebnisses der Inferenz-Request für den nächsten Durchlauf aufgegeben wird, wie im Algorithmus 2 als Pseudocode dargestellt, 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

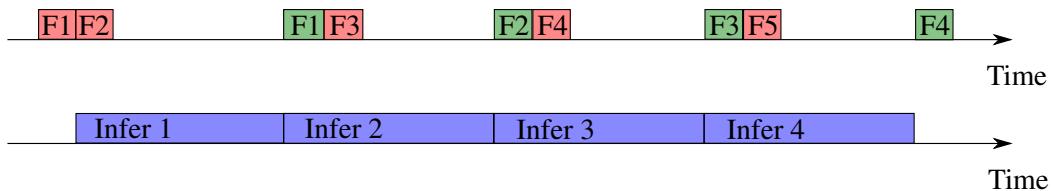


Abbildung 4.23: Zeitlicher Ablauf der asynchronen Inferenz

Die hier mit *Current* und *Next* bezeichneten Inferenz-Requests stehen für die Indizes der jeweiligen Requests und können beliebig 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 Anzahl der Bilder die pro Sekunde inferiert werden können (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 drei 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 Infrarotfilters 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 Infrarotspektrums (850nm) auf die Linse treffen und verarbeitet werden können.

Zudem verfügt die Kamera über zwei Infrarot LEDs, sodass auch Aufnahmen in völliger Dunkelheit gemacht werden können. Der Vorteil dieser Infrarot-LEDs gegenüber normalen LEDs liegt darin, dass die Tiere von keiner sichtbaren Lichtquelle gestört oder verscheucht werden.

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

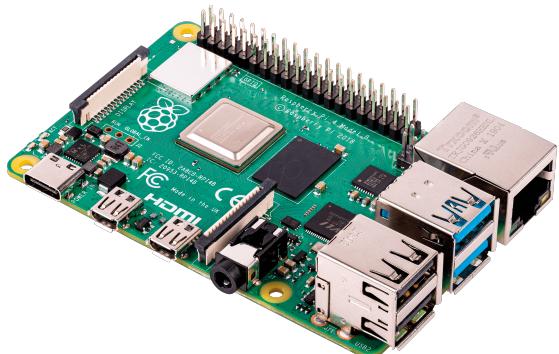


Abbildung 5.1: Raspberry Pi 4



Abbildung 5.2: Raspberry Pi Kameramodul mit Infrarot-LEDs

Des Weiteren wurde für eine mobile Internetverbindung der *Huawei E3531 SurfStick* und zur Stromversorgung eine Powerbank verwendet.

5.2 Software

Die Implementierung der Applikation für den *Raspberry Pi* wurde mit Python vorgenommen. Dabei sind die Funktionalitäten zur Objekterkennung in dem Script *detection.py* und die Funktionalitäten 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 verwendet werden, welche in *detection.py* und *connection.py* enthalten sind.

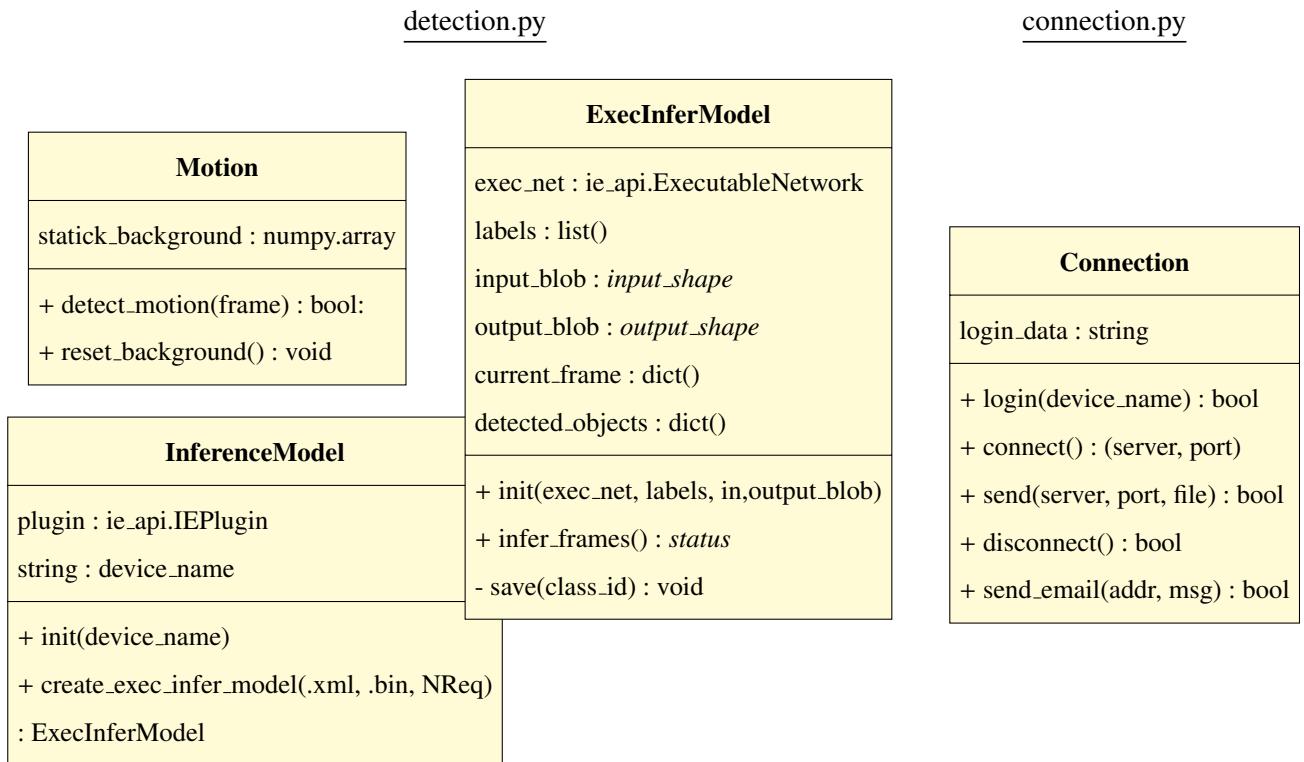


Abbildung 5.3: Klassendiagramm der Anwendung

Die Klasse *Motion* dient der 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 von der Annahme ausgegangen, dass zur Laufzeit der Anwendung, nicht durchgehend inferiert werden muss, sich also zeitweise keine Tiere und damit auch keine Bewegungen vor der Kamera befinden.

Um Bewegungen feststellen zu können, wurde mithilfe der Library *OpenCV* ein Bewegungsmelder implementiert. Dieser speichert zu Beginn des Kamera-Streams einen Referenz-Frame ab, mit welchem alle weiteren Frames verglichen werden.

Übersteigt der Abstand der einzelnen Pixelwerte im Graustufenbereich einen bestimmten Threshold, wird dies als Bewegung gewertet. Indem die Frames, die der Kamera-Stream permanent liefert, zunächst auf Bewegung überprüft werden, lässt sich unnötiges inferieren vermeiden, was Zeit und Energie spart. Frames, die

Bewegung enthalten und aufgrund der langsamen Inferenzzeit des Faster R-CNN nicht sofort inferiert werden können, werden in einem Buffer zwischengespeichert und in Phasen inferiert, zu denen keine Bewegung stattfindet.

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 ablaufen kann. Der Gesamtlauf der Applikation ist in Abbildung 5.4 schematisch als Flussdiagramm dargestellt.

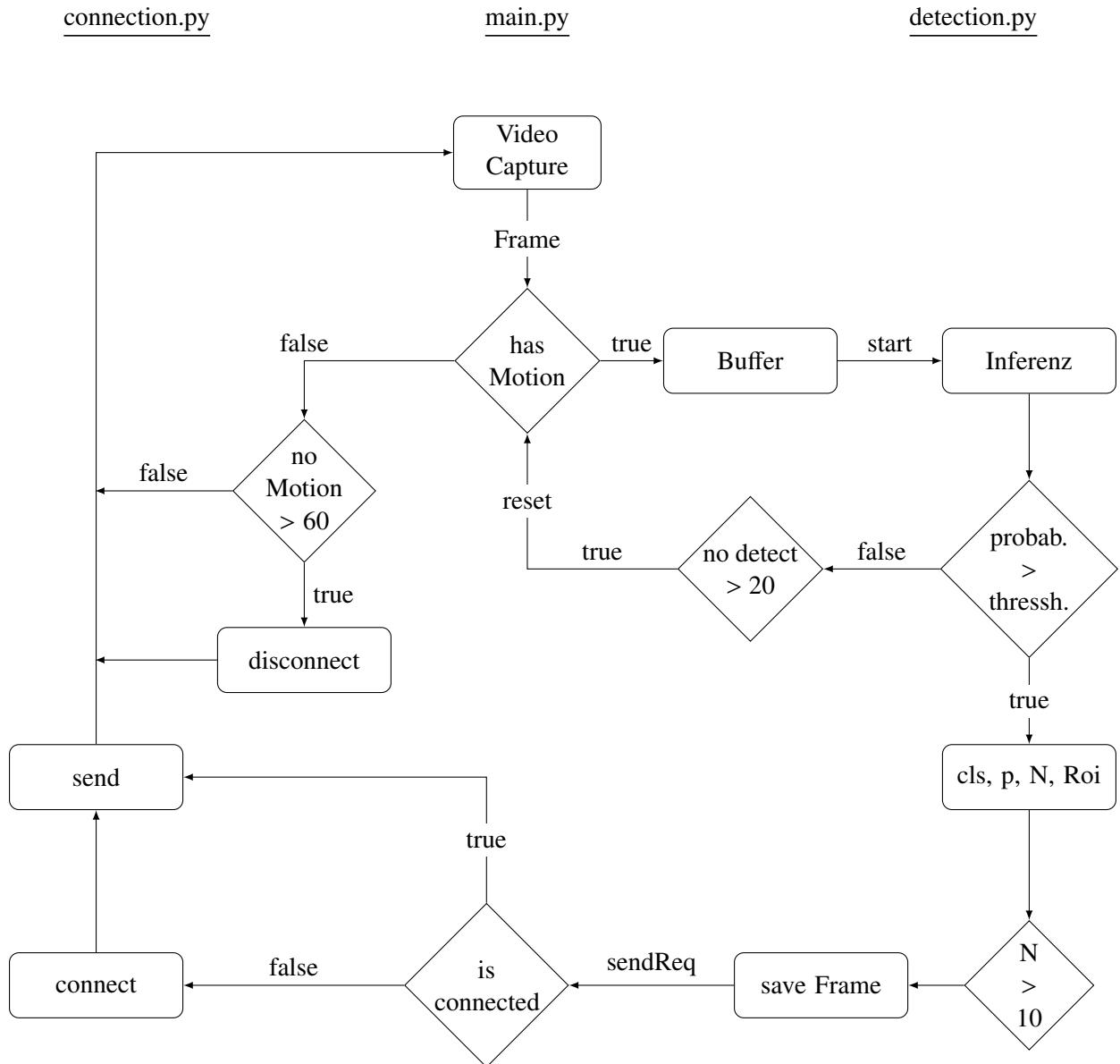


Abbildung 5.4: Schematischer Ablauf des Applikationscode

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 der Bounding-Box-Koordinaten (Roi, (Region of

Interest)) abgespeichert.

Nach einer bestimmten Anzahl an Erkennungen des selben Objekts, wird dieses als lokale Bilddatei abgespeichert und ein *Send-Request* 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, wird diese nach einer bestimmten Zeit ohne Bewegung getrennt. Dadurch wird das Datenvolumen des mobilen Internets nicht zu schnell aufgebraucht.

Im Folgenden werden die Funktionsweise der Inferenz sowie der Verbindungsaufbau genauer erklärt.

Inferenz

Der im Abschnitt 4.4 beschriebene asynchrone Inferenzablauf wurde dahingehend angepasst, dass eine beliebige Anzahl an Inferenz-Requests verwendet werden kann und dass das Warten auf ein Inferenzergebnis nicht mehr blockierend ist. Dafür wurde der Timeout in der Wait-Funktion auf *0ms* gesetzt. Im 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 Routerkonfigurationen und Firewalleinstellungen zu sein, wurde mithilfe des Dienstes *remot3.it* [29] eine Cloud-basierte Remote-Verbindung hergestellt.

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

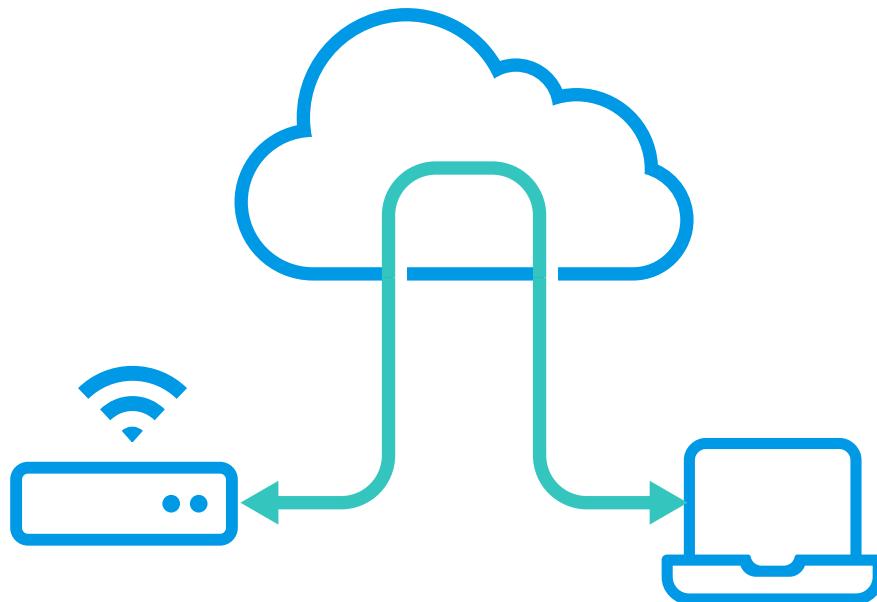


Abbildung 5.5: Funktionsprinzip der Proxy-Verbindung

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 Protocoll (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 Verbindungsau- 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.

Zusätzlich wurde eine Funktion implementiert, die den Nutzer bei einer Erkennung automatisch per E-Mail benachrichtigt. Dafür wurde das Smart Mail Transfer Protokoll (SMTP) verwendet.

Kapitel 6

Zusammenfassung

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. Auf diese Weise kann das System gezielter für eine bestimmte Anwendung eingesetzt werden. Zur Realisierung wurde neben einem *Raspberry Pi* sowie einer nachtsichtgeeigneten Kamera 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 *Open Images* heruntergeladen 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 Single Shot Detector (SSD) und Faster R-CNN mit verschiedenen Basis-CNNs und Parametereinstellungen 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 die Inferenz zusammen mit dem Anwendungscode für den *Raspberry Pi* zu implementieren, wofür mit *OpenVino* gearbeitet wurde.

Die Evaluierung der trainierten Modelle zeigte, dass eine erhöhte Genauigkeit mit einer langsameren Inferenzzeit einhergeht. Durch umfangreiche Testläufe mit variierenden Parametern konnten die optimalen Konfigurationen für die Anwendung erforscht und somit die Ergebnisse verbessert werden.

Daraus resultierend wurde für die Anwendung das Faster R-CNN Modell mit InceptionV2 als Basis-CNN gewählt. Dieses erreichte durch ein Training von 500k Iterationen auf den *Open Images* Datensatz einen mAP-Wert von 0.7 und einen Loss-Wert von 0.74. Der Datensatz wurde durch geometrische und pixelwertbezogene Veränderungen der Bilder augmentiert, wodurch für jede Klasse 3000 Bilder für das Training vorhanden waren.

Durch einen asynchronen Inferenzablauf mit drei Inferenz-Requests konnte die Inferenzzeit für das Faster R-CNN von 0,63 Fps auf 0,75 Fps erhöht werden. Indem ein Bewegungsmelder sowie ein Zwischenspeichern der Frames im Anwendungsablauf implementiert wurde, ließ sich die Performance weiter verbessern.

Literaturverzeichnis

- [1] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.
- [2] P. Matthew Stewart, “Simple Introduction to Convolutional Neural Networks, feb 27, 2019.”
<https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac>. Abrufdatum: 11.02.2020.
- [3] S. Amidi, “Stanford: CS 230 — Deep Learning, Convolutional Neural Networks cheatsheet.”
<https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>. Abrufdatum: 11.02.2020.
- [4] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” p. 46, 1998.
- [5] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
- [6] “Stanford CS: CS231n Convolutional Neural Networks for Visual Recognition.”
<http://cs231n.github.io/convolutional-networks/#layers>. Abrufdatum: 11.03.2020.
- [7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Communications of the ACM*, vol. 60, pp. 84–90, May 2017.
- [8] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” *arXiv:1311.2901 [cs]*, Nov. 2013.
- [9] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *arXiv:1409.1556 [cs]*, Apr. 2015.
- [10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” *arXiv:1409.4842 [cs]*, Sept. 2014.
- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *arXiv:1512.03385 [cs]*, Dec. 2015.
- [12] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” *arXiv:1512.00567 [cs]*, Dec. 2015.
- [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,” *arXiv:1704.04861 [cs]*, Apr. 2017.
- [14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *arXiv:1801.04381 [cs]*, Mar. 2019.

- [15] A. Ouaknine, “Review of Deep Learning Algorithms for Object Detection, medium - feb 5, 2018.”
<https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852>. Abrufdatum: 11.01.2020.
- [16] M. Häußermann, “Funktion und Effizienz von Hardware für Deep Neural Networks,” Oct. 2019. Masterarbeit, Reutlingen.
- [17] “ROBU.IN: Intel Movidius Neural Compute Stick 2.”
<https://robu.in/product/intel-movidius-neural-compute-stick-2/>. Abrufdatum: 11.03.2020.
- [18] W. G. Wong, “ElectronicDesign: Intel’s Myriad X Vision Chip Incorporates Neural Network, aug. 30, 2017.” <https://www.electronicdesign.com/industrial-automation/article/21805511/intels-myriad-x-vision-chip-incorporates-neural-network>. Abrufdatum: 06.01.2020.
- [19] “QNAP: OpenVINO™ Workflow Consolidation Tool.”
<https://www.qnap.com/solution/openvino/de-de/>. Abrufdatum: 11.03.2020.
- [20] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Mallocci, T. Duerig, and V. Ferrari, “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale,” *arXiv:1811.00982 [cs]*, Nov. 2018.
- [21] A. Jung, “imgaug.” <https://github.com/aleju/imgaug>. Abrufdatum: 11.03.2020.
- [22] X. Wu, D. Sahoo, and S. C. H. Hoi, “Recent Advances in Deep Learning for Object Detection,” *arXiv:1908.03673 [cs]*, Aug. 2019.
- [23] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *arXiv:1506.01497 [cs]*, Jan. 2016.
- [24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” *arXiv:1512.02325 [cs]*, vol. 9905, pp. 21–37, 2016.
- [25] J. Hosang, R. Benenson, and B. Schiele, “Learning non-maximum suppression,” *arXiv:1705.02950 [cs]*, May 2017.
- [26] “Google: Colaboratory.”
<https://colab.research.google.com/notebooks/intro.ipynb#recent=true>. Abrufdatum: 11.03.2020.
- [27] “TensorFlow: Object Detection Api.”
https://github.com/tensorflow/models/tree/master/research/object_detection. Abrufdatum: 11.03.2020.
- [28] S. Beery, D. Morris, and P. Perona, “The iWildCam 2019 Challenge Dataset,” *arXiv:1907.07617 [cs]*, July 2019.
- [29] “remot3.it.” <https://remote.it/>. Abrufdatum: 11.03.2020.

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	10
2.6	Sigmoidfunktion	10
2.7	Overfitting dargestellt anhand der Loss-Kurven	11
2.8	Annäherung unterschiedlich komplexer Funktionen an gegebene Datenpunkte	11
2.9	Funktionsprinzip Dropout [1]	12
2.10	Faltung des Inputs mit einer Filtermatrix [2] [3]	13
2.11	Faltung des Inputs mit einer Filtermatrix zur Erkennung vertikaler Linien	13
2.12	LeNet-5 Architektur [4]	14
2.13	Inception-Module (Version 1)	15
2.14	Inception-Module (Version 2)	15
2.15	Residual block des MobilenetV2 [14]	16
2.16	Unterschied: Klassifikation und Objekterkennung [15]	16
2.17	Neural Compute Stick 2 [17]	17
2.18	Movidius Myriad X [18]	17
2.19	OpenVino Workflow orientiert an [19]	18
3.1	Verteilung der Klassen	19
3.2	Verteilung der Klassen nach Augmentierung der Daten	19
3.3	Beispielhafte Augmentierungsergebnisse für Bilder der Klasse <i>Fuchs</i>	21
3.4	Faster R-CNN Architektur [23]	22
3.5	SSD Architektur [24]	22
3.6	Trainingsworkflow	23
3.7	Programmablauf der InferenceEngine	24
4.1	Intersection over Union	26
4.2	Confusion Matrix	26
4.3	Trainingsverläufe: Faster R-CNN, mAP	28
4.4	Trainingsverläufe: Faster R-CNN, Loss	28
4.5	Inferenzergebnis: SSD	29
4.6	Inferenzergebnis: Faster R-CNN	29
4.7	Inferenzergebnis: SSD-Mobilnet	29
4.8	Inferenzergebnis: SSD-Inception	29
4.9	Inferenzergebnis: Faster R-CNN mit Early Stopping	30
4.10	Inferenzergebnis: Faster R-CNN mit Augmentierung	30
4.11	Trainingsverläufe: mAP	31
4.12	Trainingsverläufe: Loss	31
4.13	Inferenz- ergebnis für 3000 Bilder	31
4.14	Inferenz- ergebnis für 4000 Bilder	31

4.15 Inferenz- ergebnis für 50% Augmentierung	31
4.16 Trainingsverläufe: mAP	32
4.17 Trainingsverläufe: Gesamt-Loss	32
4.18 Trainingsverläufe: Klassifikations-Loss	32
4.19 Trainingsverläufe: RPN Loss	32
4.20 Inferenzergebnis für reine Augmentierung	33
4.21 Inferenzergebnis für Augmentierung und L2 Regularisierung	33
4.22 Zeitlicher Ablauf der synchronen Inferenz	34
4.23 Zeitlicher Ablauf der asynchronen Inferenz	35
5.1 Raspberry Pi 4	37
5.2 Raspberry Pi Kameramodul mit Infrarot-LEDs	37
5.3 Klassendiagramm der Anwendung	38
5.4 Schematischer Ablauf des Applikationscode	39
5.5 Funktionsprinzip der Proxy-Verbindung	41

Tabellenverzeichnis

4.1	Trainingsergebnisse: SSD und Faster R-CNN	27
4.2	Trainingsergebnisse: Verschiedene Regularisierungstechniken	33
4.3	Vergleich von Inferenzzeiten der Modelle in FPS	36