

# 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: TT.MM.JJJJ





# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
<b>2 Grundlagen</b>	<b>5</b>
2.1 Machine Learning . . . . .	5
2.2 Computer Vision . . . . .	8
2.3 Hardware/NCS2 . . . . .	10
<b>3 Anforderungen und Analyse</b>	<b>13</b>
3.1 Ziel der Arbeit . . . . .	13
3.2 Related Work . . . . .	13
<b>4 Realisierung Objekt Erkennung</b>	<b>15</b>
4.1 Datensatz . . . . .	15
4.2 Training . . . . .	16
<b>5 Evaluierung</b>	<b>19</b>
5.1 Evaluierungs Metriken . . . . .	19
5.2 Vergleich der Modelle . . . . .	20
5.3 Optimierungen: Faster r-CNN . . . . .	23
5.4 Inferenz zeit . . . . .	25
<b>6 Entwicklung der Anwendung</b>	<b>27</b>
6.1 Aufbau . . . . .	27
6.2 Raspberry Pi Kamera . . . . .	29
6.3 Server-Client-Connection . . . . .	29
6.4 Anwendung gesamt . . . . .	29
<b>7 Test und Validierung</b>	<b>31</b>
<b>8 Zusammenfassung und Ausblick</b>	<b>33</b>
<b>A Beispiel für ein Kapitel im Anhang</b>	<b>37</b>
A.1 Bsp für ein Abschnitt im Anhang . . . . .	37



# **Kapitel 1**

## **Einleitung**

Im Rahmen der Bachelor Arbeit wurde ein Überwachungssystem zur Wildtiererkennung, entwickelt, welches auf einem Raspberry Pi läuft und den Nutzer bei Erkennung bestimmter Tiere automatisch benachrichtigt, sowie das Bild an einen Server sendet.

Die Erkennung der Tiere erfolgte mithilfe Neuronaler Netze, wodurch es möglich ist die Überwachung gezielt nur auf bestimmte, relevante Tiere anzuwenden und so den Datenverkehr gering zu halten.

Die Inferenz der Neuronalen Netze wurde dabei auf einer separaten Hardware, dem Neural Compute Stick 2 von Intel ausgeführt.

Des weiteren wurde eine Infrarotfähige Kamera verwendet, damit das System auch in der Nacht einsetzbar ist.

gliederung



# Kapitel 2

## Grundlagen

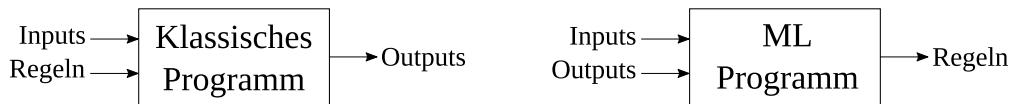
Das vorliegende Kapitel beschreibt zunächst die Grundlagen des Machine Learnings, mit Neuronalen Netzen allgemein und anschließend im Einsatz des maschinellen Sehens.

Des Weiteren wird die für die Inferenz, verwendete Hardware, der Neural Compute Stick 2 vorgestellt.

### 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, ohne explizit darauf programmiert worden zu sein.

Eine Form davon ist das *Supervised Learning*, bei dem das Programm neben den Input Daten auch die Zugehörigen Ausgaben erhält und daraus dann die Regeln für Zusammenhänge ableiten soll. Dadurch unterscheidet sich das Vorgehen wesentlich zur klassischen Programmierung, bei der die Regeln vorab definiert werden müssen.



Das Ableiten der Regeln erfolgt beim Machine Learning in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei werden die Zusammenhänge zwischen In- und Output Daten als mathematische Funktion betrachtet, welche numerisch angenähert wird.

Bei einem linearen Zusammenhang, handelt es sich um eine Regression und bei einem kategorischen um eine Klassifizierung.

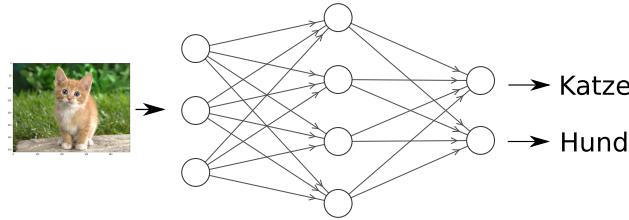
Weitere Formen neben dem *Supervised Learning* sind das *Unsupervised Learning*, bei dem das Programm keine Labels erhält, sondern diese durch Clustering Verfahren selber finden soll, oder das *Reinforcement Learning*, bei dem das Programm mit der Umwelt interagieren soll.

Diese Techniken wurden in der Bachelorarbeit nicht verwendet und werden daher nicht näher erläutert.

#### 2.1.1 Künstliche Neuronale Netze

Für komplexe Input Daten, wie beispielsweise Bilder, bei denen die einzelnen Pixelwerte als Inputs und der Inhalt des Bildes als Output dienen, werden in der Regel künstliche neuronale Netze verwendet. Diese sind eine Form des Machine Learnings und bestehen aus einer Vielzahl an miteinander verbundener Neuronen. Durch unterschiedlich starke Gewichtungen der einzelnen Verbindungen, auch Gewichte genannt, können für unterschiedliche Input Daten die entsprechenden Outputs gefunden werden.

Die richtige Einstellung der Gewichte, welche zunächst zufällig initialisiert werden, erfolgt dabei im Trainingsprozess, welcher in ?? schematisch dargestellt ist aus folgenden drei Schritten besteht.



- *Forward Pass* anhand aktueller Gewichte vorhersage aus den Inputs treffen
- *Fehlerbestimmung* Abweichung zum tatsächlichen werten berechnen
- *Backpropagation* minimierung der Fehlerfunktion durch Anpassung der Gewichte

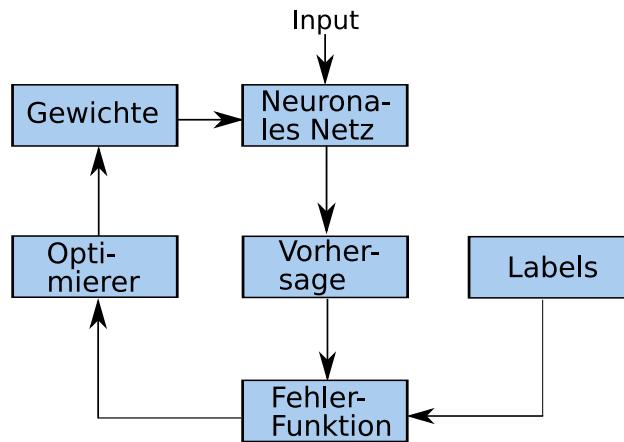


Abbildung 2.1: Trainingsablauf NN

Durch mehrfaches durchlaufen der Schritte wird die Fehlerfunktion soweit minimiert, sodass das Modell auch für neue Input Daten die richtigen Aussagen treffen kann.

### Forward Pass

Im *Forward Pass* werden die Inputs durch alle Schichten hindurch gereicht, um in der letzten Schicht den gewünschten Output zu liefern. Dabei erhält jedes Neuron die mit  $w_i$  gewichteten Ausgabewerte aller Neuronen der vorherigen Schicht und summiert diese zusammen mit einem konstanten Bias Wert  $b$  auf. Mithilfe einer Aktivierungsfunktion wird der Wert auf einen bestimmten Bereich skaliert ??

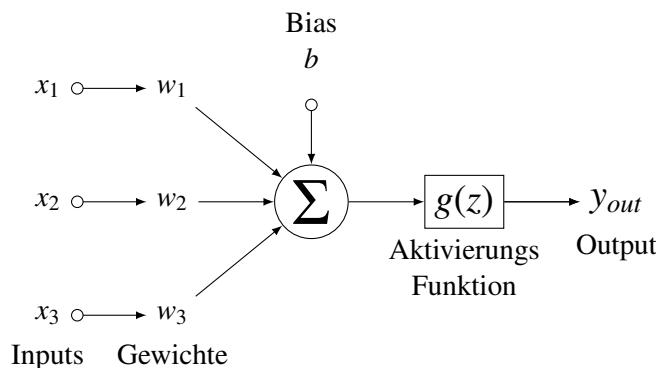


Abbildung 2.2: Einzelnes Perzeptron

Um den Forward Pass für eine gesammte Schicht, bestehend aus einer Vielzahl an Neuronen, zu berechnen, werden die Schichten als Vektoren und die Gewichte als Matrizen dargestellt.

Die Matrixmultiplikation aus dem Vektor der vorherigen Schicht  $x$  der Gewichtsmatrix  $W$  ergibt die Werte des Vektors der aktuellen Schicht,

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

welcher anschließend elementweise einer nichtlinearen Aktivierungsfunktion  $g(z)$  übergeben wird. Für mittlere Schichten wird dabei oft die in 2.2 dargestellte *ReLU* verwendet eine Funktion die negative Werte zu 0 setzt.

$$g(z) = \max\{0, z\} \quad (2.2)$$

Da die Ausgabe meist einen Wahrscheinlichkeites Wert zwischen 0 und 1 Um für die Outputs einen Wahrscheinlichkeits Wert zwischen 0 und 1 zu erhalten, wird an der Ausgabeschicht für eine binäre Klassifikation die Sigmoid Funktion (2.3) verwendet, welche S-Förmig zwischne 0 und 1 verläuft.

$$g(z) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

Für eine Kategorische Ausgabe mit mehr als zwei Werten wird die Softmax (2.4) Funktion verwendet, welche eine Wahrscheinlichkeits Verteilung über alle Ausgabe Neuronen generiert.

$$g(z) = \frac{e^z}{\sum e^x} \quad (2.4)$$

### Fehlerbestimmung

Die Abweichung der Schätzung, welche an den Neuronen der letzten Schicht vorliegen, zu den tatsächlichen Werten, den Labels, wird mithilfe einer geeigneten Fehlerfunktion bestimmt. Für die lineare Regression wird dabei z.B. der absolute oder quadratischen Abstand verwendet, für ein Klassifizierungs Modell ist jedoch eine logarithmische Fehlerberechnung mit der Cross Entropy Funktion, in 2.5 dargestellt effektiver.

$$L = \hat{y} \log(y) + (1 - \hat{y}) \log(1 - y) \quad (2.5)$$

Durch den Logarithmus wird der Loss um so größer, je weiter die Schätzung  $y$  vom tatsächlichen Wert  $\hat{y}$  abweicht.

### Backpropagation

Durch Berechnung des Gradienten der Fehlerfunktion kann ermittelt werden in welche Richtung die Gewichte angepasst werden müssen, sodass diese sich im nächsten Durchgang minimiert.

Dafür wird die die Fehlerfunktion  $L$  für jede Schicht partiell nach den Gewichten  $w$  abgeleitet, was wie in gl. 2.6 dargestellt mithilfe der Kettenregel über die Aktivierungsfunktion  $z$  geschieht.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial w} \quad (2.6)$$

Mit dem ermittelten Gradienten werden dann die Gewichte nach Gleichung 2.7 angepasst.

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \quad (2.7)$$

wobei die *Lernrate*  $\eta$  die Schrittweite ist, mit der die Anpassungen vorgenommen werden.

## 2.1.2 Validierung und Overfitting

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

Während des Trainings wird für beide Sätze der Fehler berechnet, die Korrektur der Gewichte mittels Backpropagation erfolgt jedoch nur anhand der Trainingsdaten.

Entsteht eine Abweichung der beiden Fehlerfunktion wie in [plot overfitting] dargestellt, findet eine Überanpassung an die trainingsdate *Overfitting* statt.

Gründe dafür sind zu wenig Trainingsdaten oder ein zu komplexes Modell, welches sich ähnlich einer polynomialen Funktion mit hohem Grad an jeden datenpunkt anpassen kann, damit kein generalisierbare Aussage für neue Daten mehr getroffen werden kann.

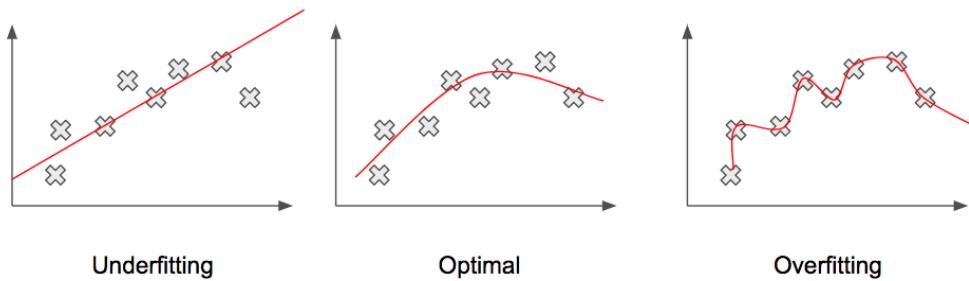


Abbildung 2.3: Quelle: [1]

Um Overfitting, auch wenn nicht mehr Trainingsdaten zur Verfügung stehen zu vermeiden, gibt es verschiedene Techniken:

### *Augmentierung*

Bei Augmentierung werden aus den vorhandenen Daten künstlich mehr Daten generiert, in dem an den Bildern geometrische Transformationen oder Manipulationen der Pixelwerte vorgenommen werden.

### *Regularisierung der Parameter (L1/L2)*

Bei Regularisierung wird an die Lossfunktion als weiterer Term eine Aufsummierung der Gewichte gehängt, wodurch diese bei der Minimierung klein gehalten werden, wodurch weniger Potential zur Überanpassung da ist.

$$J(w) = E + \lambda \sum_i w_i^2 \quad (2.8)$$

### *Dropout*

Beim Dropout werden zufällig Gewichte zu 0 gesetzt.

### *Early Stopping*

Stoppen des Trainings, wenn sich Overfitting einstellt.

## 2.2 Computer Vision

Deep Learning + Techniken der Bildverarbeitung

### 2.2.1 Convolutional Neural Networks

Für die Bilderkennung werden üblicherweise Convolutional Neural Networks (CNNs) verwendet. Hierbei handelt es sich um eine Erweiterung der in ?? beschriebenen Neuronalen Netze. Beim CNN sollen vor der Klassifikation, Merkmale des Input Bildes, die spezifisch für eine Klasse sind herausgezogen werden.

Dafür werden über das Bild zeilenweise Filtermatrizen mit kleinerer Dimension (3x3, 5x5) geschoben und eine math Faltung angewendet. Die Ergebnisse der Faltungen ergeben eine sog Feature Map, in welcher Muster die sowol in Filter Matrix als auch in input Bild auftreten, verstärkt dargestellt werden.  
Die Werte der Filter Matrizen entsprechen den zu lernenden Gewichten und werden mithilfe der Backpropagation angepasst.

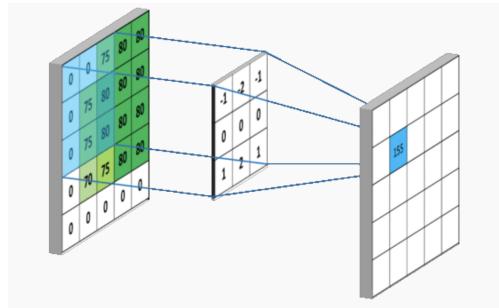


Abbildung 2.4: Faltung, [2]

Durch die hintereinanderschaltung mehrerer Convolutional Layern lassen sich so immer komplexere Merkmale des Input Bildes in den Feature Maps heraus extrahieren.

Durch Subsampling Methoden wie Max Pool Layer zwischen den Convolutional Layern verkleinert sich die Dimension der Feature Maps in jeder Schicht.

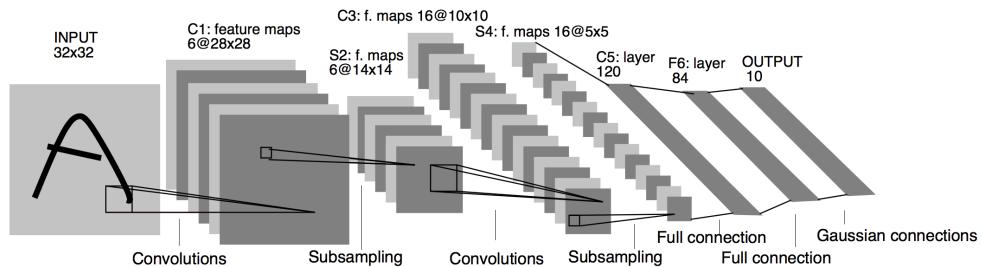


Abbildung 2.5: Faltung, [3]

Vorteile der CNNs sind der geringere Rechenaufwand durch die gemeinsame Nutzung der Parameter der Filter Matrizen und die durch die Faltung zustande kommende räumliche Invarianz für das zu erkennende Objekt auf dem Bild.

Um die Features, welche insbesondere in den vordersten ConvLayern für alle Klassen sehr ähnlich sind, nicht bei jedem Modell neu lernen zu müssen, wird häufig *Transfer Learning* angewendet, d.h. es werden die auf ein allg. Datenset wie z.B. ImageNet vortrainierten Gewichte verwendet und müssen so nur noch etwas für den eigenen Datensatz fine-tuned werden.

## Architekturen

Nach der in Abbildung ?? dargestellten, erfolgreichen ersten Veröffentlichung eines CNN von Yann Lecun 1998 [3] wurden viele weitere Architekturen entwickelt.

Diese werden anhand der ImageNet Challenge ILSVRC [?] bewertet

Die bekanntesten Gewinner Modelle sind wie in [4] aufgeführt:

- Alexnet (2012), mehrere conv layer hintereinander
- GoogleLeNet (2014), Inception Module
- VGGNet 2014
- ResNet (2015),

## 2.2.2 Objekt erkennung

Neben der Information, was sich auf einem Bild befindet möchte man bei der Object Detection auch herausfinden wo sich das Objekt befindet. Dafür wird ein in 2.2.1 beschriebenes CNN als Basis zusammen mit weiteren Mechanismen, auf die in 3.2 genauer eingegangen wird, verwendet.



Abbildung 2.6: Unterschied: Classification - Detection

## 2.2.3 Deep Learning Lerining Frameworks

- Frameworks allgemein
- TensorFlow

## 2.3 Hardware/NCS2

Da das Training und die Inferenz von Deep Learning Algorithmen sehr rechenintensiv ist, werden entsprechend leistungsfähige Prozessoren benötigt. Dabei ist die Ausführung auf einer GPU (Graphical Processor Unit) meist effizienter als auf einer CPU (Central Processor Unit). Anwendungen auf eingebetteten Systemen wie z.B. einplatinen Computern wie dem in der Arbeit verwendeten Raspberry Pi kommen dabei schnell an ihre Grenzen. Möchte man dennoch die Daten auf dem Gerät verrechnen und nicht an eine Cloud senden, bieten verschiedene KI Beschleuniger die Möglichkeit die Inferenz des Deep Learning Modells auf externer Hardware auszuführen. Einer davon ist der in der Arbeit verwendete Neural Compute Stick 2 von Intel.

Dieser basiert auf der Movidius Myriad X Vision Processing Unit (VPU) [5]



Abbildung 2.7: NCS2

## 2.3.1 OpenVino Toolkit

Die Implementierung der Inferenz des trainierten Models wurde mithilfe des OpenVino Toolkits vorgenommen, eine Anwendung zur Optimierung und Ausführung von CNNs auf Intel Hardware.

Es vereinfacht und optimiert damit die Verbindung zwischen Training des Models und bereitstellen in einer Anwender Applikation, wie in Abbildung 2.8 schematisch dargestellt.

Das Toolkit besteht im Wesentlichen aus den zwei Komponenten *Model Optimizer* und *Inference Engine*

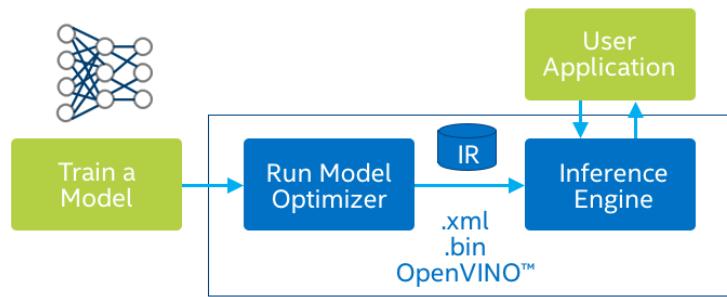


Abbildung 2.8: Workflow: OpenVINO Toolkit

Mit dem Model Optimizer können Netze die in den Frameworks TensorFlow, Caffe, MXNet, Kaldi oder ONNX trainiert wurden in die von OpenVINO verwendete Intermediate Representation des Modells gebracht werden.

Diese ist ein Framework unspezifisch Dateiformat, welches aus einer .xml Datei für die Struktur/Architektur des Modells und einer .bin Datei für die trainierten Gewichten besteht.

Die InferenceEngine ist eine Runtime welche eine API für die Sprachen C++ und Python zur Integration und Nutzung der Inferenz in der Anwendung bereitstellt.

Dafür werden die IR Dateien des Models in ein Hardware spezifisches Plugin geladen. Dieses kann die User Applikation für die Inferenz von Image Classification, ObjectDetection sowie Instance Segmentation Modellen nutzen.

- weitere hardware
  - Raspberry
  - RPiCam mit IR Cut Switch
  - WiFi Stick



# Kapitel 3

## Anforderungen und Analyse

### 3.1 Ziel der Arbeit

Wie in der Einleitung 1 beschrieben, soll ein CNN Basiertes System zur Wildtiererkennung entwickelt werden, das für die Inferenz den Neural Compute Stick 2 verwendet. Dabei sollte neben der reinen Erkennung auch eine Lokalisierung der erkannten Tiere im Bild stattfinden. Gängige Techniken dafür werden im nächsten Abschnitt erläutert.

Dabei soll das Deep Learning Modell im Rahmen der gegebenen Möglichkeiten und Limitierungen der Hardware möglichst genau und Robust sein, sodass es auch für die graustufen Bilder der Infrarot Kamera zuverlässig funktioniert. Da eine erhöhte Genauigkeit auch immer mit einer größeren Latenz für der Inferenzzeit einhergeht war dies ein mit zu berücksichtigender Punkt.

Neben Training und Evaluierung eines geeigneten Deep Learning Modells, war die Implementierung der Anwendung, welche die Inferenz des Modells ausführt ein weiterer Bestandteil der Arbeit.

Diese soll voll autonom auf dem Raspberry Pi laufen, über eine mobile Netzwerk Verbindung verfügen und mittels eines geeigneten Kommunikations Protokolls die die erkannten und abgespeicherten Bilder an einen Heim PC senden. Des Weiteren sollte eine geeignete Kamera verwendet werden, die sowohl normale, als auch Infrarot Aufnahmen machen kann.

### 3.2 Related Work

Für die Objekterkennung werden häufig End-to-End Lösungen verwendet, Modelle die sowohl Klassifikation als auch Lokalisierung durchführen. Diese verwenden meist eines der im Abschnitt 2.2.1 erläuterten Basis CNNs als *Feature Extractor* und darauf aufbauend ein Framework für die Lokalisierung.

Diese lassen sich, wie in [6] beschrieben, in einstufige und zweistufige Verfahren gliedern.

#### Region Based CNNs

Bei den zweistufigen handelt es sich um Regionbased CNNs, die Regionen für mögliche Box Locations mithilfe RPN (Region Proposia Network) oder selective search Verfahren finden soll, um diese dann zu klassifizieren bzw. box regression. Die aktuellste Version davon ist das in Abbildung 5.6 schematisch dargestellte *Faster R-CNN* [7]

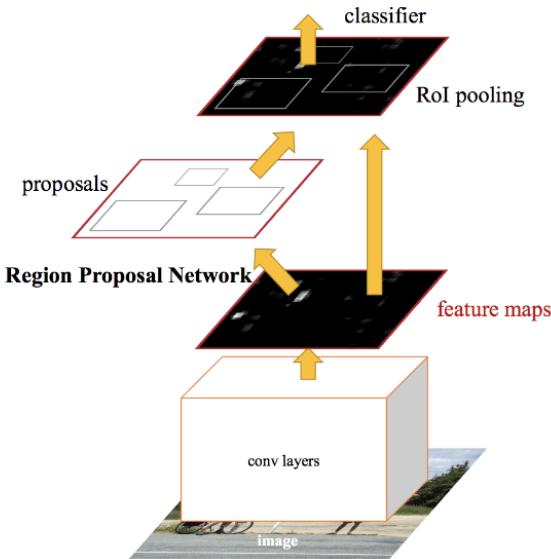


Abbildung 3.1: Faster R-CNN, [7]

### Single Shot Detectoren

Einstufige Verfahren wie Single Shot Detectoren (SSD) führen die lokalisierung zusammen mit dem Feature Extractor aus, indem verschiedene scalierungen/ausmaße der Convolutional Layer in den Classifier/Regressor gegeben werden. Dadurch sind diese wie in Abbildung 3.2 zu erkennen ist, zwar schneller, jedoch auch ungenauer.

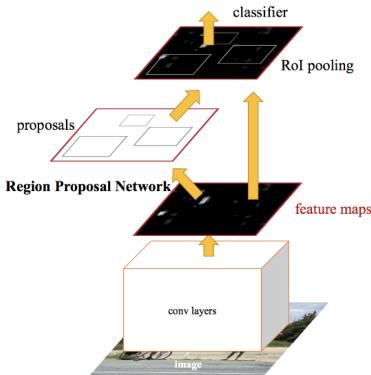


Abbildung 3.2: Faster R-CNN, [7]

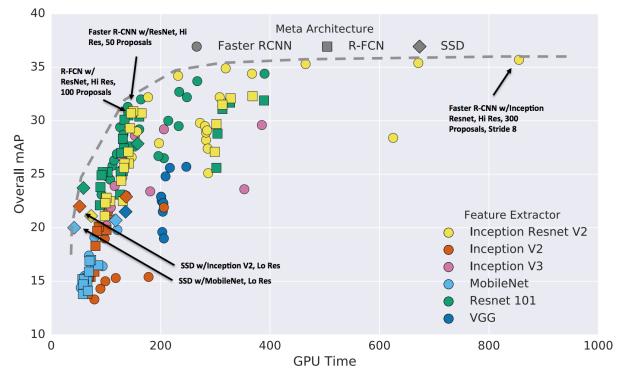


Abbildung 3.3: Geschw vs Genauigkeit, [8]

# Kapitel 4

## Realisierung Objekt Erkennung

### 4.1 Datensatz

Um ein Deep Learning Modell richtig trainieren zu können, wird eine große Menge an gelabelten Trainingsdaten benötigt. Im Falle der Objekterkennung enthalten die labels neben der Klasse auch die Koordinaten der Bounding Boxen, welche das zu erkennende Objekt auf dem Bild umrahmen.

Für die vorliegende Arbeit wurden dafür aus dem Open Source Dataset *OpenImages* [9] von Google 9 Klassen, welche Wildtiere enthalten heruntergeladen.

Dieses besteht aus einem Trainingsset mit 200 bis 2000 Bildern pro Klasse sowie kleineren Test- und Validierungssets. Um für alle Klassen die gleiche Anzahl an Trainingsdaten zu erhalten und um Overfitting zu verhindern wurden wie im folgenden beschrieben die Trainingsdaten augmentiert.

#### 4.1.1 Augmentierung

Augmentierung ist eine Technik, mit der aus den vorhandenen Daten künstlich mehr Daten generiert werden können. Dafür werden z.B. geometrische Transformationen, wie etwa Skalierung, Verschiebung, Rotieren und Spiegelungen oder Manipulationen der Pixelwerte zur Veränderung der Farbwerte, Helligkeit, Kontrast oder Rauschen vorgenommen.

Mithilfe eines Python Scripts und der Library *imgaug* [10] konnten so verschiedene Augmentierungstechniken auf das Datenset angewendet werden.

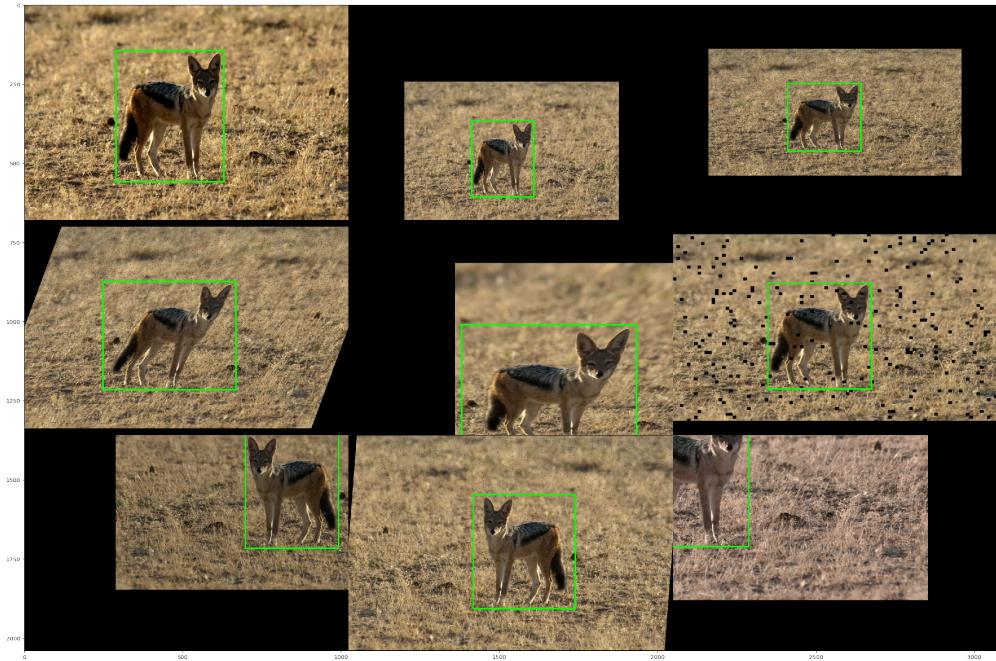


Abbildung 4.1: Anwendung von Augmentierungstechniken

## 4.2 Training

Da der Neural Compute Stick mit OpenVino ein eigenes Datei Format für die trainierten Modelle verwendet, musste bei der Auswahl eines Frameworks sowie Models auf die Kompatibilität zu OpenVino geachtet werden.

### 4.2.1 Tensorflow Object Detection Api

Die Tensorflow Object Detection Api ist unter den Research Modellen [11] des offiziellen Tensorflow Repository zu finden und enthält implementierungen einiger gängiger Object Detectin Modelle, wie Single Shot Detectors (SSD) und Faster R-CNNs mit verschiedenen Basis CNNs mit vortrainierten Gewichten.

Um die Modelle trainieren zu können, mussten zunächst die Trainingsdaten in das binary Dateiformate TFRecords umgewandelt werden, welches die Api verwendet. Dieses ist eine Serialisierte darstellung der Bilder und Labels als Protocol Buffer für effizienten Zugriff auf diese.

Trainiert wurde mit hilfe *Google Colab*, eine cloudbasierte VM, welche eine geeignete Gpu zur Verfügung stellt.

### 4.2.2 Trainingsworkflow

Das Ergebnis des Trainings kann neben der Auswahl eines geeigneten Modell, sowie auswahl und aufbereitung des Datensatzes auch durch anpassungen sogenannter Hyperparameter beeinflusst werden.

Mit diesen können in 2.1.2 beschriebene Verfahren realisiert werden.

Durch eine Evaluierung zur Laufzeit des Trainings können so Fehler in der Konfigureation des Trainings erkannt und durch anpassen von entweder Datensatz, Modell oder Parameter korrigiert/verbessert werden. Es ergibt sich folgender Workflow.

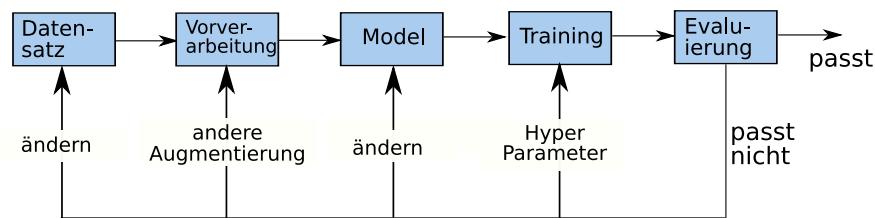


Abbildung 4.2: Trainingsworkflow

Die Ergebnisse werden im nächsten Kapitel (5) diskutiert.



# Kapitel 5

## Evaluierung

Bevor die Ergebnisse der trainierten Modell diskutiert werden, behandelt dieses Kapitel zunächst einige der für die Objekterkennung üblicherweise verwendeten Metriken zur Messung der Genauigkeit.

Anschließend werden die Modell in 5.2 miteinander verglichen und in 5.3 optimierungsverfahren für das Faster R-CNN Model ausgewertet.

### 5.1 Evaluierungs Metriken

#### Mean Average Precision (mAP)

Als Metrik für die Genauigkeit eines Object Detection Models dient die *Mean Average Precision (mAP)*, welche sowohl Klassifizierung als auch Lokalisierung mit einbezieht und sich aus folgenden Werten berechnen lässt.

- *True Positive (TP)*:
- *True Negative (TN)*:
- *False Positive (FP)*:
- *False Negative (FN)*:

Zur bestimmung dieser Werte wird die *Intersection over union* verwendet, welche Überlappungsgrad der gelabelten (Ground Truth) und der geschätzte Boundig Box zu dem Gesamtbereich beider Boxen darstellt. Beträgt dieser mehr als ein Bestimmter Threshold, häufig 50% gilt die Schätzung als *True Positive*, andernfalls als *False Positive*.

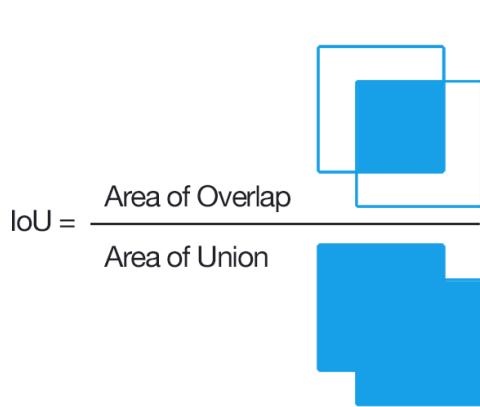


Abbildung 5.1: Intersection over Union

		geschätzter Wert
tatsächlicher Wert	p'	n
	True Positive	False Negative
n'	False Positive	True Negative

Abbildung 5.2: Confusion Matrix

Daraus lassen sich dann Precision und Recall berechnen.

*Recall* welcher angibt wie viele Objekte das Modell gefunden hat

$$Recall = \frac{TP}{TP + FN} \quad (5.1)$$

TP + FN allen Objekten im Bild entspricht, somit also das Verhältnis der Gefunden zu allen Objekten im Bild

*Precision* die angibt mit welcher Genauigkeit die Objekte gefunden wurden

$$Precision = \frac{TP}{TP + FP} \quad (5.2)$$

also die richtig geschätzten durch alle gemachten schätzungen und gibt somit die genauigkeit mit der das Model die Objekte findet an.

*Average Precision*

$$AveragePrecision = \frac{1}{N} \sum (Precision(Recall)) \quad (5.3)$$

Daraus kann nun die durchschnittliche Precision für alle Recall Werte bestimmt werden, welche als *Average Precision* bezeichnet wird und die Genauigkeit des Models bezogen auf eine bestimmte Klasse angibt  
*mean Average Precision* Für alle Klassen gemittel erhält man die mittlere durchschnittliche Precision (mAP)

## Fehlerfunktion (Loss)

Die Fehlerfunktion setzt sich aus einem Lokalisierungs und einem Klassifizierungsfehler zusammen. Die Lokalisierung erfolgt über eine Lineare Regression zur Annäherung der Bounding Boxes and die richtigen Koordinaten.

Bei Faster R-CNN werden diese beiden Loss Werte dann sowohl für RPN(1st stage) als auch für classifier(2nd stage) also insgesamt 4 loss werte verwendet.

## 5.2 Vergleich der Modelle

In diesem Abschnitt werden die beiden für das Training verwendeten Objec Detection Architekturen SSD und Faster R-CNN miteinander verglichen.

Für SSD wurden einmal Mobilenet und einmal InceptionV2 als Basis CNN verwendet und für Faster R-CNN nur InceptionV2.

### 5.2.1 Evaluierung/Validierung

Die Evaluierungsergebnisse beziehen sich auf das Testsets aus dem Open Images Datensatz.

SSD wird mit Batchsize=12 75k durchläufe trainiert.

Faster R-CNN mit Batchsize=1 für 200k durchläufe.(1 weil dynamische input size)  
eine epoch sind (teps mal batchsize)/samples

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

Man kann erkennen das der Fehler (Loss) durch Augmentierung etwas verringert wurde. Insbesondere beim Faster R-CNN, welches aufgrund der höheren Komplexität, schneller zur Überanpassung neigt. Der Verlauf des Trainings ließ sich mithilfe des Evaluierungstools Tensorboard Visualisieren, Plots für die drei Konfigurationen des Faster R-CNN sind in Abbildung ?? und ?? dargestellt.

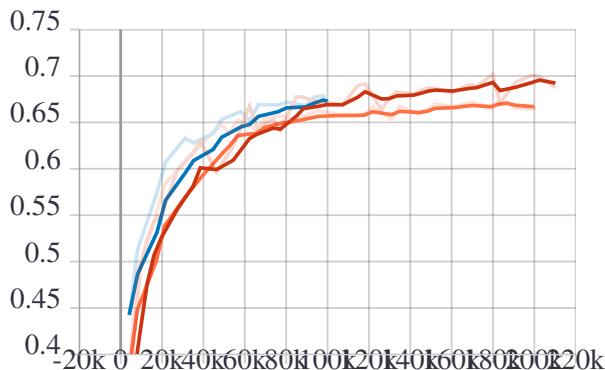


Abbildung 5.3: mAP

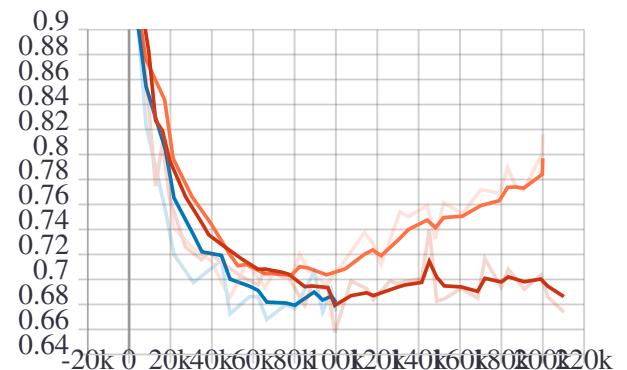


Abbildung 5.4: Loss

● Ohne   ● Early Stopping   ● Augmentierung

hier die zwei Techniken zur Vermeidung von Overfitting: Augmentierung und Loss gegenübergestellt. Augmentierung erzielt bessere Ergebnisse, da durch vorzeitiges Stoppen des Trainings auch keine weitere Verbesserung bezüglich mAP mehr stattfinden konnte.

## 5.2.2 Test Inferenz

Um eine bessere Vorstellung davon zu bekommen wie sich die unterschiedlichen Ergebnisse in mAP und Loss in der Anwendung tatsächlich bemerkbar machen, wurde die Inferenz der drei Modelle für verschiedene Test Bilder durchgeführt.

Dafür wurden die Modelle in die für OpenVINO benötigte Intermediate Representation umgewandelt um dann mithilfe eines Python Scripts in die InferenceEngine geladen zu werden.

Mithilfe dieses Scripts konnten nun die Inferenz für folgende Testszenarien:

- inferenz auf Test Set *OpenImages*
- inferenz auf *The iWildCam 2019 Dataset*[12]
- inferenz auf eigene Bilder

durchgeführt und die Ergebnisse miteinander verglichen werden.

## test set

Fast alles wird erkannt.

grund: große ähnlich der art der bilder in trainings und testsatz  
tiere sind überwiegend mittig zentriert im bild. Abb. 5.5

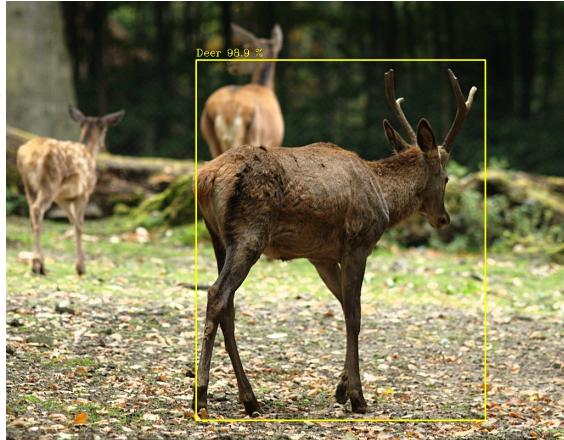


Abbildung 5.5: SSD

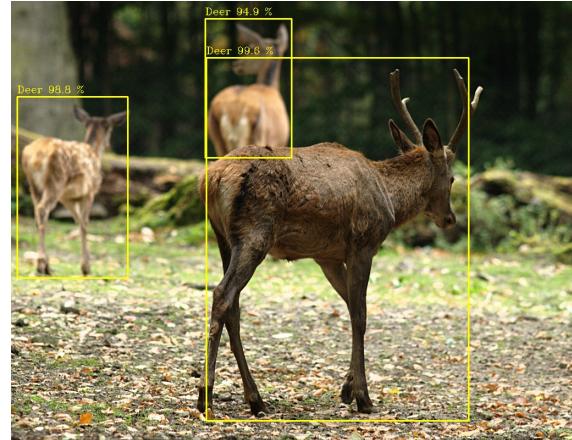


Abbildung 5.6: Faster R-CNN

Wenn mehrere Tiere im Bild sind, diese weiter weg oder schlechtere Bildqualität, erkennt Faster besser. Abb 5.6

## eigene

bilder guter qualität, jedoch unterschied zu openimages dataset bezogen auf background umgebung.

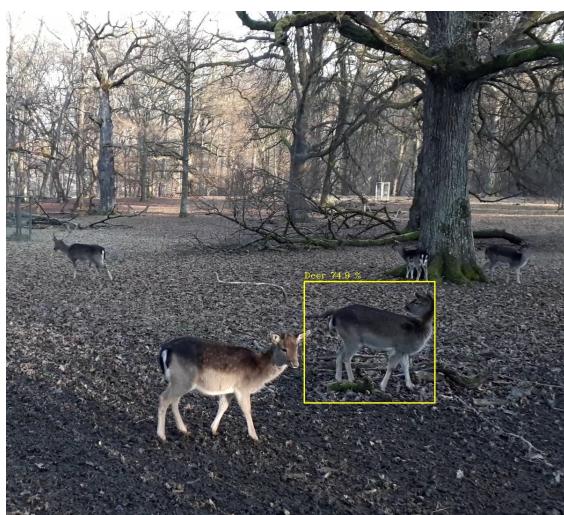


Abbildung 5.7: SSD Mobilnet

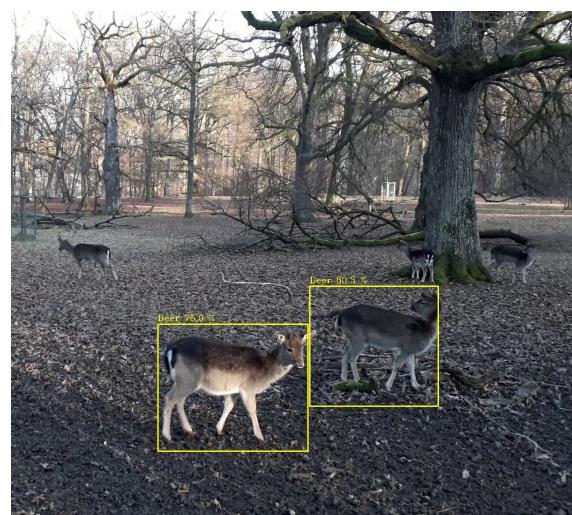


Abbildung 5.8: SSD Inception

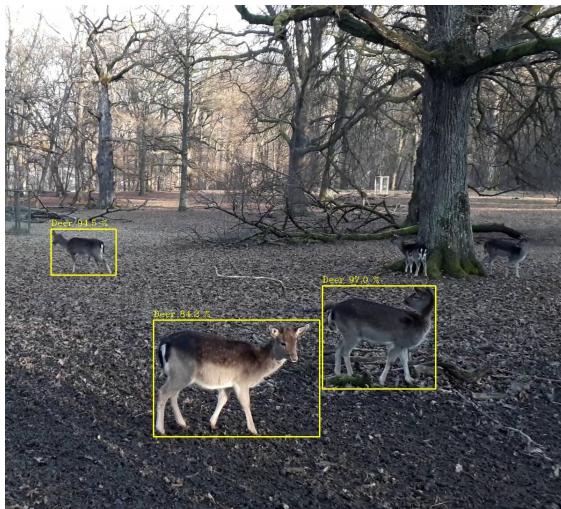


Abbildung 5.9: Faster R-CNN + Early Stopping

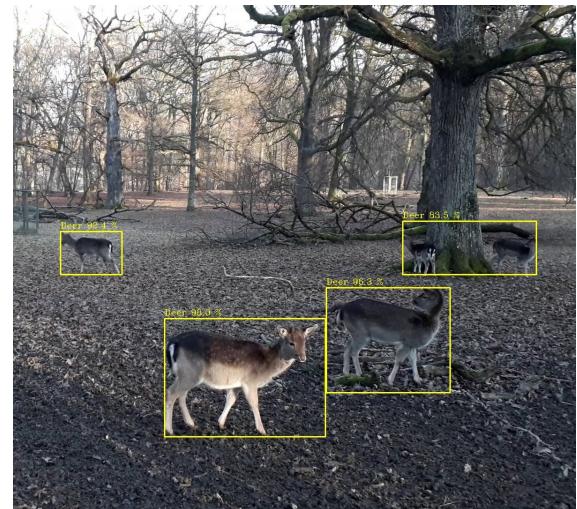


Abbildung 5.10: Faster R-CNN + Aug

### kaggle set

Aufnahmen von ... enthält viele dunkle nacht aufnahmen geringer qualität. entspricht daher am ehesten den real world bedingungen. wenig wird erkannt. am besten erkennt faster + aug. daher wurde versuch faster rcnn zu optimieren und robuster zu machen, was im nächsten Abschnitt beschrieben ist. evtl noch bilder.

## 5.3 Optimierungen: Faster r-CNN

Um bessere Ergebnisse für das Faster R-CNN zu erzielen wurde zunächst das im vorherigen beschriebene Faster + Aug 400k statt 200k durläufe trainiert.

### 5.3.1 Validierung

Trotz Augmentierung hat das Modell ab 250k schritten overfittet. Betrachtet man die einzelnen Loss Kurven, aus denen sich der gesammt Loss zusammensezt (insgsamt 4 bei faster R-CNN, 2mal rpn, 2mal fully connected am ende) stellt man fest, dass nur das Rpn netz sich überanpasst.

Da die L2 Regulierung durch anpassung des Config Files für die Einstellungen der Model Parameter, auf die Stufen separat angewendet werden kann, wurde diese auf die erste stufe (RPN) mit einem Faktor von 0,001 angewendet, wodurch sich das Overfitting verhindern ließ.

Auf der anderen Seite hat sich jedoch eine leichte verschlechterung des mAP ergeben.

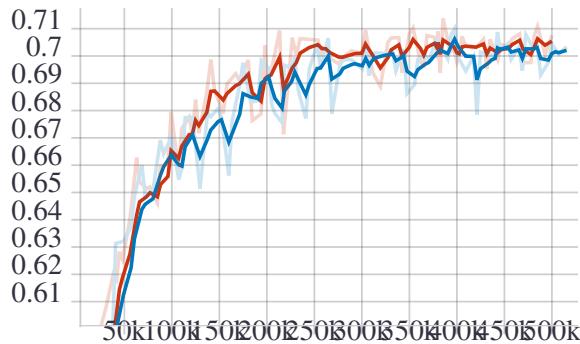


Abbildung 5.11: mAP



Abbildung 5.12: Total Loss



Abbildung 5.13: Classifier Loss

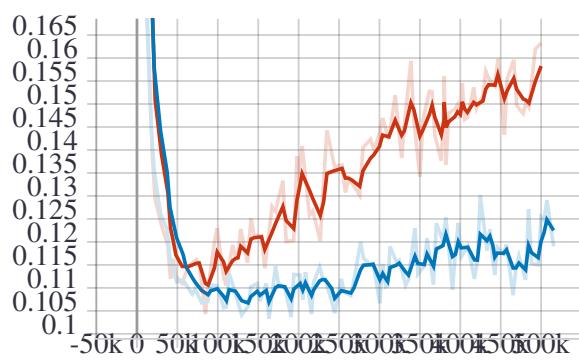


Abbildung 5.14: RPN Loss

weitere Trainings Konfigurationen waren

- dropout
- l2 (rpn loss anschauen)
- mehr daten (4000 statt 3000)

und sind in tabelle so und so dargestellt.

>400k	mAP	Loss (Gesammt)	Loss (RPN)
Augmentierung	0,7	0,74	0,12
+Dropout	0,7	0,73	
+L2 Reg (0.01)	0,7	0,69	
+L2 Reg (0.02)	0,69	0,7	

Tabelle 5.1: Regularisierungen

### 5.3.2 Test Inferenz

da nicht sehr eindeutig, welche optimierung die beste ist, wird auch hier test inferenz für kaggle und eigen durchgeführt.

### 5.3.3 Graustufen

da kamere a graustufen bilder liefert, wurde getestet, ob ein training in graustufen bilder zu besseren erg führt, ... nicht so.

### 5.3.4 weitere tabellen

Regularisierung	$mAP_{orig}$	$mAP_{handy}$	$Loss_{orig}$	$Loss_{handy}$
Early Stopping (100k steps)	0.6715	0.4265	0.6742	0.267
Augmentierung (200k steps)	0.6914	0.4537	0.6738	0.2503

Tabelle 5.2: Regularization

### 5.3.5 Graustufen/Infrarot Bilder

Modell	Dataset	mAP	Loss
rgb	original	0.6556	0.1451
	handy	0.4155	0.2389
gray 1 channel	original	0.5625	0.1716
	handy	0.3226	0.2747
gray 3 channel	original	0.664	0.1653
	handy	0.438	0.2492

Tabelle 5.3: Grayscale

## 5.4 Inferenz zeit

inferenz zeiten wurden in FPS für 100 bilder gemessen. Bei mehr als ein Inferenz Request wurde die Asynchrone Api der InferenceEngine verwendet. Folgende Tabelle Zeigt für verschiedene Modelle die Inferenz zeit auf ncs2 über raspberry pi für 100 frames.

Model	Asynchronge 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



# Kapitel 6

## Entwicklung der Anwendung

In diesem Kapitel wird die Entwicklung der Anwendung als Autonomes Edge System auf dem Raspberry Pi zusammen mit dem Neural Compute Stick 2 und einer geeigneten Kamera beschrieben. Ebenso wird die Integration des trainierten Tensorflow Models in die Applikation sowie die Implementierung der Netzwerk Verbindung zu dem System beschrieben.

### 6.1 Aufbau

Die Anwendung soll auf dem ein Platinen Computer Raspberry Pi 4 Abbildung ?? laufen, an den die nötigen Komponenten angeschlossen werden. Dazu gehören der Neural Compute Stick 2, zur Ausführung der Inferenz, ein Kamera Modul, mit welchem die Bilder aufgenommen werden, sowie ein WiFi Stick und Powrebank.

Der Neural Compute Stick wird über USB angeschlossen und kann nach installation des OpenVino Toolkits ?? verwendet werden.

Bei der Kamere handelt es sich um ein Infrarot Fähiges *RaspberryPi Camera Module* welches zusammen mit zwei Infrarot LEDs montiert wird. 6.2



Abbildung 6.1: Raspberry Pi 4

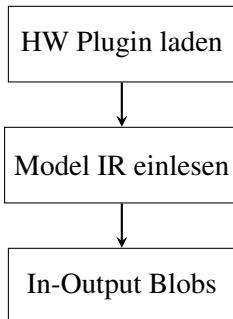
- Netzwerkverbindzuung:
  - GSM Module
  - WiFi-Stick
- Powrebank/Akku aus Sp. Verbrauch von:
  - NCS2
  - Kamera
  - LEDs
  - WiFi Stick

## Implementierung

Die Implementierung der Inferenz wurde in Python vorgenommen.

Dafür waren folgende Schritte nötig:

1. HW Plugin laden
2. Model IR einlesen
3. In-Output Blobs allokieren
4. ausführbares Model laden
5. inferenz request abgeben
6. Bild als Array in Input Blob laden
7. Inferenz
8. Output verarbeiten, wieder zu Schritt 6



Blobs sind In-Output Tensoren

```

plugin = IEPlugin(device='MYRIAD')
net = IENetwork(model=model_xml, weights=model_bin)
input_blob = next(iter(net.inputs))
exec_net = plugin.load_network(network=net)
infer_request = exec_net.requests[request_id]
# bild mit opencv als numpy arra laden und von hwc nach nchw umstellen
res = exec_net.infer(inputs={input_blob: image})
# res enthaelt liste mit allen erkannten klassen auf dem Bild
# fuer Objekt Detection zusaetglich noch Bounding Box koordinaten
  
```

Die Inferenz kann entweder Synchron oder Asynchron ausgeführt werden. Der programmatische Ablauf der hier verwendeten asynchronen Inferenz ist im Folgenden als Pseudocode dargestellt.

```

while true do
    capture frame;
    populate Next InferRequest;
    start Next InferRequest; // asynchroner aufruf
    if wait for Current done then
        // wird in eigenem verarbeitet
        display Current;
    end
    swap Current and Next InferRequests;
end
  
```

### Algorithm 1: Asynchrone Inferenz

Das Ergebnis eines InferRequest für Object Detection Modelle enthält eine Liste mit allen möglichen erkannten Objekten, jedes davon bestehend aus einem Array mit den Indices:

0. batch index
1. class label
2. Wahrscheinlichkeit
3.  $x_{min}$  Box Koordinate
4.  $y_{min}$  Box Koordinate
5.  $x_{max}$  Box Koordinate
6.  $y_{max}$  Box Koordinate

Mit über die Wahrscheinlichkeit ließen sich die Ergebnisse nach einem bestimmte Threshhold ausfiltern. Die Box Koordinaten wurden in Prozent der Bild- Breit/Höhe angegeben wodurch sie wieder in die Original bildgröße für die Bounding Boxes übertragen werden konnen.

## 6.2 Raspberry Pi Kamera

Bei der Kamera handelt es sich um das OV5647 5MP Modul mit regelbarem Infrarotfilter. Zusammen mit zwei Infrarot LEDs von der Firma Quimat Abbildung ??

Wird der Infrarotfilter ausgeschaltet ist es durch die Infrarot LEDs mit 850nm welligen Licht möglich auch bei Dunkelheit Aufnahmen zu machen, die in Graustufen Werten dargestellt werden.

## 6.3 Server-Client-Connection

## 6.4 Anwendung gesamt

Da die Inferenz sehr rechenaufwendig ist, sollen die Frames der Kamera nur dann inferiert werden, wenn eine Bewegung stattfindet. Dafür wurde der Inferenz ein Bewegungsmelder vorgeschaltet. Dieser wurde mithilfe der Library OpenCV implementiert, indem zu Beginn des Kamereastrams ein Referenzbild gespeichert wurde, mit dem die aktuellen Frames verglichen werden. Ist der absolute Abstand der einzelnen Array Elemente/Werte der Bilder größer als ein bestimmter Threshhold, wird dies als Bewegung gewertet.



## **Kapitel 7**

# **Test und Validierung**



## **Kapitel 8**

# **Zusammenfassung und Ausblick**

Die Zusammenfassung bildet mit der Einleitung den Rahmen der Arbeit. Sie greift zu Beginn die Aufgabenstellung auf und beschreibt dann die wesentlichen Punkte des Lösungsweges und die erzielten Ergebnisse kurz und knapp, so dass diese in kürzester Zeit erfasst werden können.

Anschließend werden noch kurz offene Punkte, Verbesserungen oder Weiterentwicklungen diskutiert.

Insgesamt sollten Zusammenfassung und Ausblick anderthalb Seiten nicht überschreiten. In der Regel ist eine Seite ausreichend.



# Literaturverzeichnis

- [1] M. Deshp and e, “A Guide to Improving Deep Learning’s Performance.”
- [2] M. S. Researcher, PhD, “Simple Introduction to Convolutional Neural Networks.”
- [3] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” p. 46.
- [4] “Stanford - CS231n Convolutional Neural Networks for Visual Recognition.”
- [5] M. Häußermann, “Funktion und Effizienz von Hardware für Deep Neural Networks.”
- [6] L. Weng, “Object Detection Part 4: Fast Detection Models.”
- [7] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,”
- [8] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, “Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3296–3297, IEEE.
- [9] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Mallochi, T. Duerig, and V. Ferrari, “The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale,”
- [10] 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.
- [11] “TensorFlow Object Detection Api.” [https://github.com/tensorflow/models/tree/master/research/object\\_detection](https://github.com/tensorflow/models/tree/master/research/object_detection).
- [12] S. Beery, D. Morris, and P. Perona, “The iwildcam 2019 challenge dataset,” *arXiv preprint arXiv:1907.07617*, 2019.



## **Anhang A**

### **Beispiel für ein Kapitel im Anhang**

#### **A.1 Bsp für ein Abschnitt im Anhang**