

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

1 Einleitung	3
2 Grundlagen	5
2.1 Machine Learning	5
2.2 Computer Vision	8
2.3 Hardware/NCS2	10
3 Anforderungen und Analyse	11
3.1 Ziel der Arbeit	11
3.2 Related Work	11
4 Realisierung Objekt Erkennung	13
4.1 Datensatz	13
4.2 Training	14
5 Evaluierung	17
5.1 Evaluierungs Metriken	17
5.2 Vergleich der Modelle	18
5.3 Optimierungen: Faster r-CNN	19
6 Entwicklung der Anwendung	23
6.1 Aufbau	23
6.2 Raspberry Pi Kamera	25
6.3 Server-Client-Connection	25
6.4 Anwendung gesamt	25
7 Test und Validierung	27
8 Zusammenfassung und Ausblick	29
A Beispiel für ein Kapitel im Anhang	33
A.1 Bsp für ein Abschnitt im Anhang	33

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

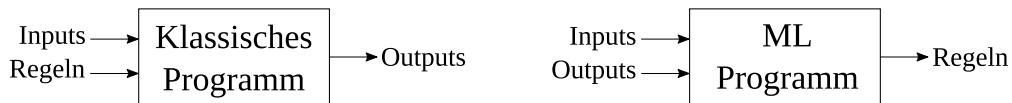
Grundlagen

Im folgende Kapitel wird zunächst auf die Grundlegenden Techniken des Machine Learings, insbesondere auf die für die Bilderkennung verwendeten Convolutional Neural Networks eingegangen. Anschließend wird es um die verwendete Hardware, den Neural Compute Stick 2 un seine Anwendungen gehen.

2.1 Machine Learning

Beim Machine Lerining, welches ein Teilgebiet der Computerwissenschaften ist, geht es um Algorithmen, die Zusammenhänge in großen Datenmengen erkennen sollen, ohne explizit darauf programmiert worden zu sein.

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



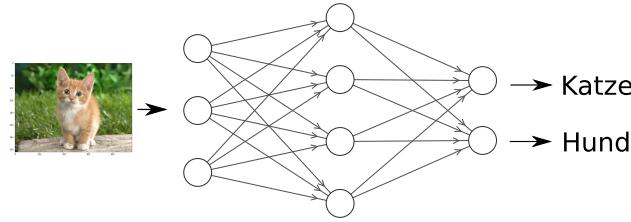
Das herleiten der Regeln erfolgt beim Machine Learning dabei in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei soll eine math. Funktion, welche die Zusammenhänge beschreibt numerisch angenähert werden. Ist der Zusammenhang linear, spricht man von einer Regression, handelt es sich um Kategorische, liegt ein Klassifizierung problem vor.

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

Da hier jedoch ausschließlich mit dem Supervised Learing gearbeitet wurde, werden diese Techniken 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 Learings 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 und sich in die drei Schritte:

- Feed Forward anhand aktueller Gewichte vorhersage aus den Inputs treffen
- Lossfuction Abweichung zu tatsächlichen Werten bestimmen
- Backpropagation Minimierung der Fehlerfunktion durch Anpassung der Gewichte

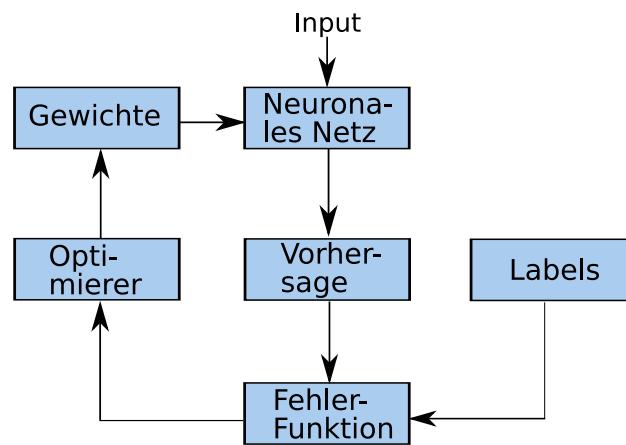


Abbildung 2.1: Trainingsablauf NN

Durch häufiges Wiederholen dieser Schritte kann die Fehlerfunktion soweit minimiert werden, dass das Modell auch für neue Input-Daten die richtigen Aussagen treffen kann.

Vorwärts

Im Vorwärtsdurchgang wird der Input durch alle Schichten hindurch gereicht, um in der letzten Schicht den gewünschten Output zu liefern. Dabei erhält jedes Neuron wie in ?? dargestellt, die Ausgabewerte aller Neuronen der vorherigen Schicht, summiert diese auf und über gibt den Wert an eine Aktivierungsfunktion, die den Wert auf einen bestimmten Bereich skaliert.

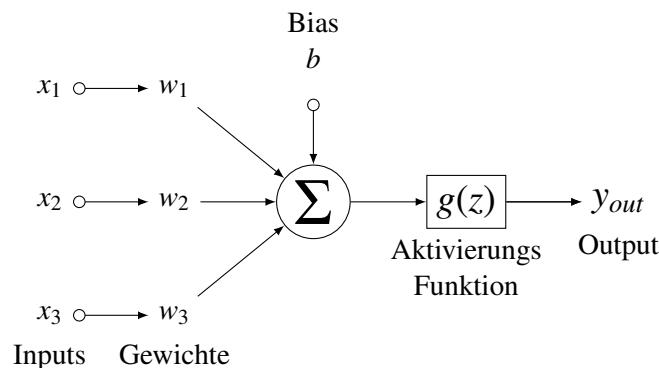


Abbildung 2.2: Einzelnes Perzeptron

Die Berechnung des Vorwärtsdurchgangs einer ges Schicht aus Neuronen als Vektor z dargestellt, erfolgt wie in [1] beschrieben, über die Matrixmultiplikation des Vektors x der vorherigen Schicht mit der Gewichtsmatrix W .

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

und wird anschließend elementweise einer nichtlinearen Aktivierungsfunktion $g(z)$ übergeben.

Für Hidden Schichten wird dabei üblicherweise 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 haben soll wird für die letzte Schicht bei einer binären Klassifikation die *Sigmoid* Funktion 2.3

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

verwendet und für kategorische Ausgaben die *Softmax* 2.4 Funktion.

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

Fehlerfunktion

Die Abweichung der Schätzung, welche an den Neuronen der letzten Schicht vorliegen, zu den tatsächlichen Werten, den Labels, wird mithilfe geeigneter Fehlerfunktion bestimmt. Für Ragression z.B. abs oder rms und für Kategorisch häufig logarithmisch.

hier am beispiel einer binären klassifikation (erg 0 oder 1) mit log loss (crossentropy) dargestellt.

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

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 sie sich im nächsten Durchgang minimiert. Dafür wird die die Fehlerfunktion für jede Schicht partiell nach den Gewichten abgeleitet, was wie in gl. 2.6 dargestellt mithilfe der Kettenregel für die Aktivierungsfunktion geschieht.

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

Damit werden die Gewichte dann nach Gleichung 2.7 angepasst.

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

wobei die *Leariningrate* η die Schrittweite mit der die Anpassung vorgenommen werden soll angibt.

2.1.2 Validierung und Overfitting

um überprüfen zu können ob ein Modell die Trainingsdaten tatsächlich generalisiert hat, dh auch für neue daten anwendbar ist, oder diese nur auswendig gelernt hat, wird häufig der Datensatz in einen Trainingsanteil und einen Testanteil aufgeteilt.

Mit dem Testdatensatz wird dann schon während des Trainings regelmäßig zwischen geprüft, verringert sich irgendwann nur noch der fehler der trainingsdaten, findet overfitting statt.

Häufig sind zu wenige Trainingsdaten oder zu komplexe/überparametrisierte Modelle und damit zuviele Freiheitsgrade, Grund für Overfitting.

Techniken um Overfitting zu vermeiden sind z.B.

- Augmentierung der Daten
- Regularisierung der Parameter (L1/L2)
- Dropout
- early stopping

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

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.

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

Beim Dropout werden zufällig Gewichte zu 0 gesetzt.

early stopping: stoppen des Trainings, wenn sich Overfitting einstellt.

2.2 Computer Vision

Was ist Computer Vision, Einordnung in ML

2.2.1 Convolutional Neural Networks

Für die Bilderkennung werden typischerweise 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 sowohl 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.

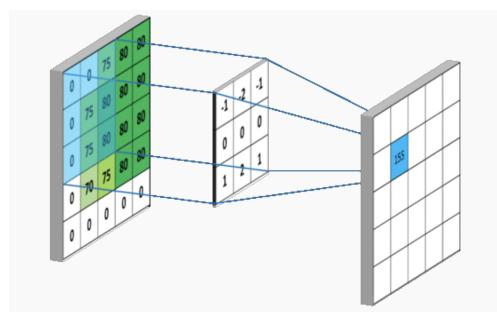


Abbildung 2.3: Faltung, [2]

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

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

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.

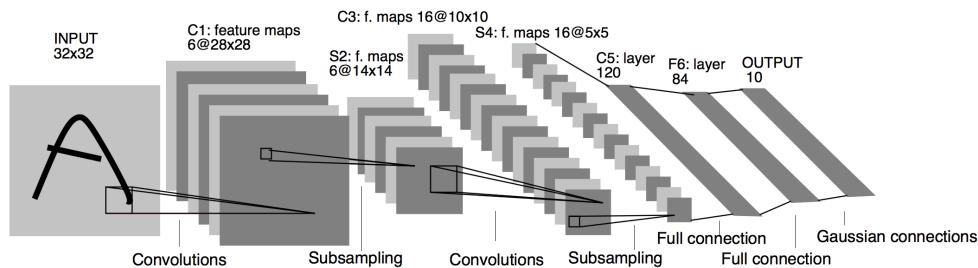


Abbildung 2.4: Faltung, [3]

auf dem Bild.

Um die Features, welche insbesondere in den vordersten ConvLayern für alle Klassen sehr ähnlich sind, nicht bei jedem Modell von Grund auf neu lernen zu müssen, wird häufig *Transfer Learning* angewendet, d.h. es werden die auf einem allg. Datensatz wie z.B. ImageNet vortrainierten Gewichte verwendet und müssen so nur noch leicht 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 2.5: Unterschied: Classification - Detection

2.2.3 Deep Learning Frameworks

kurz Frameworks all.

dann speziell tensorflow (kein obj det api)

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 2.6: 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 2.7 schematisch dargestellt.

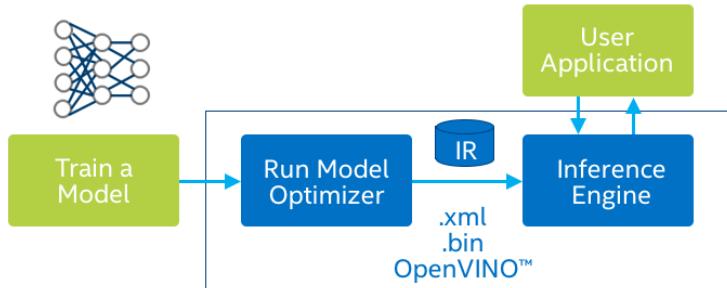


Abbildung 2.7: Workflow: OpenVino Toolkit

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

•

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.

Anforderungen an die Erkennung waren ein möglichst hohe Genauigkeit und Robustheit des Models zu schaffen, sodass diese auch für die weniger detailreichen graustufen Bilder der Infrarot kamera noch funktioniert. Echtzeit Inferenz war nicht notwendig, dennoch sollten alle relevanten Informationen verarbeitet werden können.

Die Entwicklung der autonome laufenden Anwendung für den Raspberry, sowie die Integration des trainierten Modells in diese war ebenso wichtiger Bestandteil der Arbeit.

Hierbei waren die Anforderungen eine geeignete Kamera (infrarot) auszusuchen, sowie eine Kommunikations/Benachrichtigungs möglichkeit über Netzwerk Verbindung zu schaffen.

3.2 Related Work

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

Diese lassen sich in einstufige und zweistufige Verfahren gliedern [6]. 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 3.2 schematisch dargestellte *Faster R-CNN* [7]

Einstufige Verfahren wie Single Shot Detectoren (SSD) führen die Lokalisierung zusammen mit dem Feature Extractor aus, indem verschiedene Skalierungen/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.

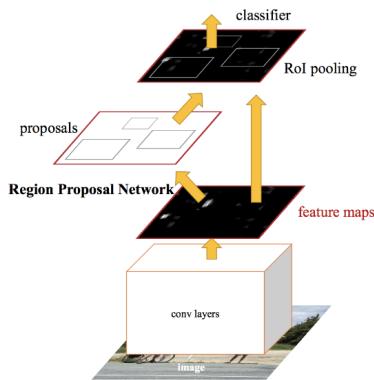


Abbildung 3.1: Faster R-CNN, [7]

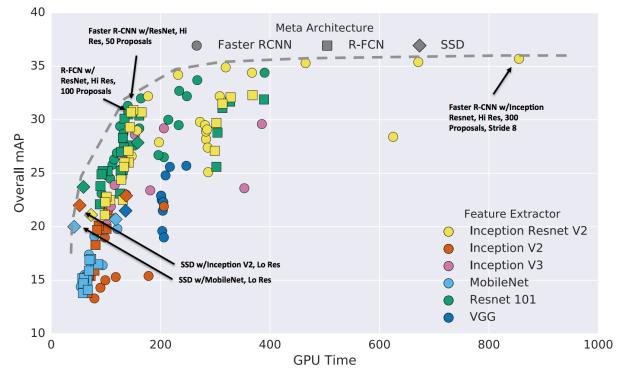


Abbildung 3.2: Geschw vs Genauigkeit, [8]

Kapitel 4

Realisierung Objekt Erkennung

4.1 Datensatz

Um ein Deep Learning Modell vernünftig 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.

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

Dieses besteht aus einem Trainingsset mit 200 bis 2000 Bildern pro Klasse sowie einem Test- und einem Validierungsset. 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 aus den vorhandenen Daten künstlich mehr Daten zu generieren, indem diese leicht abgeändert werden. Im Falle des für diese Arbeit verwendeten OpenImages Datensatzes wurden Geometrische Transformationen wie Sklierung, Verschiebung, Rotieren oder Spiegeln und Manipulationen der Pixelwerte wie ändern der Farbwerte, Helligkeit, Kontrast oder Rauschen vorgenommen.

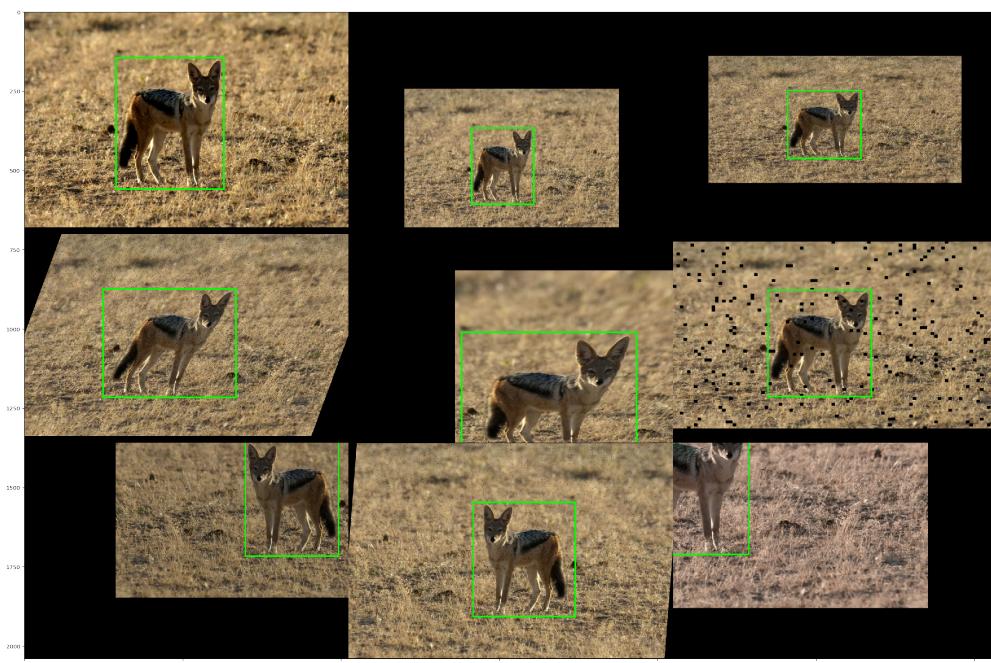


Abbildung 4.1: Anwendung von Augmentierungstechniken

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.

Verwendete wurde daher das unterstützte Framework Tensorflow, um zwei Ansätze zu verfolgen. Der eine verwendet die Api Keras, und versucht neben der Klassifikation die Lokalisierung mit dieser vorzunehmen, der andere verwendet eine speziell für die Objekterkennung konzipierte Api für wie in Abschnitt 3.2 beschriebene End-to-End lösungen.

4.2.1 Tensorflow Object Detection Api

Die Tensorflow Object Detection Api ist unter den Research Modellen [10] 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 Dateiformat 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 ... Gpu zur Verfügung stellt.

4.2.2 Trainingsoptimierung

Um zu einem guten Ergebnis zu kommen werden, wie in ... dargestellt, häufig Anpassung bezogen auf Datensatz Augmentierungstechniken, verschiedene Modelle und Basis CNNs, sowie dessen Hyperparameter vorgenommen.

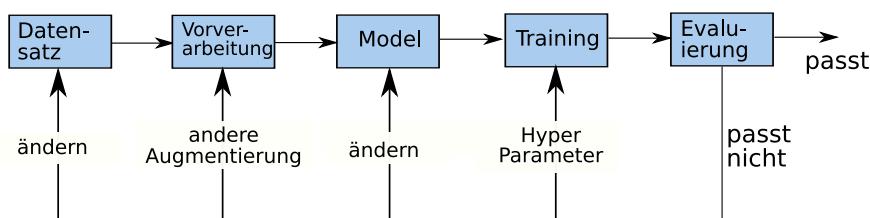


Abbildung 4.2: Trainingsworkflow

- Datensatz
 - Augmentierungstechniken
 - * Geometrische
 - * Farbwerte
 - Graustufen
 - * 1 Channel
 - * 3 Channel
- Modelle
 - Faster R-CNNs
 - * InceptionV2
 - * Resnet
 - SSD
 - * Mobilenet
 - * InceptionV2

- Hyperparameter
 - Dropout
 - L2 Realisierung
 - Early Stopping

Kapitel 5

Evaluierung

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 berechnet.

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

Zur bestimmung von TPs 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 Threshhold, häufig 50% gilt die Schätzung als *True Positive*, andernfalls als *False Positive*.

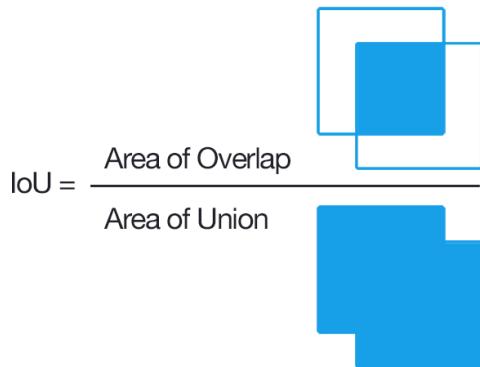


Abbildung 5.1: Intersection over Union

		geschätzter Wert	
		p	n
tatsächlicher Wert	p'	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

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.1}$$

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

- Tab vgl SSD, Faster, mit/ohne Aug, für mAP, Loss
- Overfitting bei Faster: Aug vs Early Stopping

5.2.1 Evaluierung/Validierung

hier mit validierungs set

	mAP	Loss	Augmentation
SSD Mobilenet	0,62	3,56	neim
	0,61	3,42	ja
SSD Inception	0,65	4,41	nein
	0,62	3,71	ja
Faster R-CNN	0,67	0,82	neim
	0,69	0,67	ja

Tabelle 5.1: Vergleich Modelle

Augmentierung hat nur effekt bei Faster R-CNN, da komplexeres Modell

- Ohne
- Early Stopping
- Augmentierung

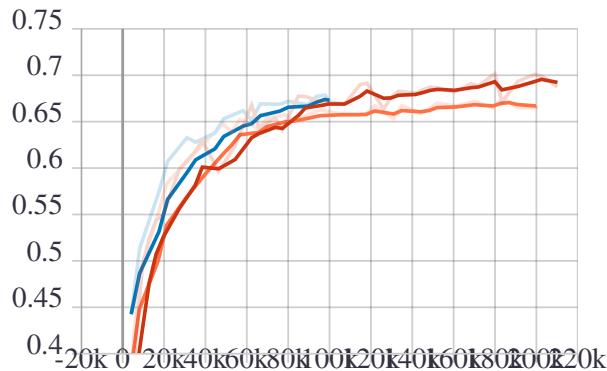


Abbildung 5.3: mAP

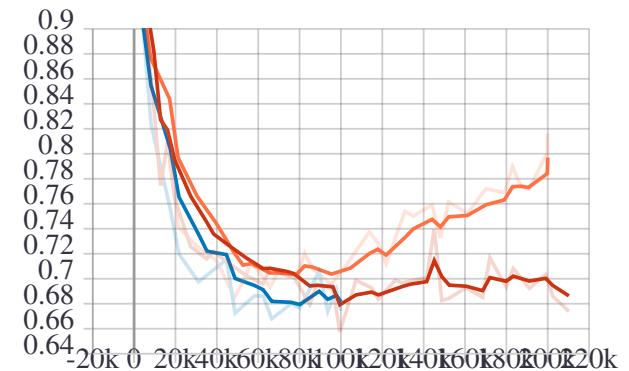


Abbildung 5.4: Loss

- Early Stopping
 - mAP: 0,67
 - Loss: 0,69

Loss Werte sind für Aug und Early Stopping gleich. jedoch kann mAP bei Training mit Augmentierung noch weiter ansteigen.

weitere Beobachtung: wenn eval für anderes Datenset angewendet wird ist auch der Loss des Augmentierten Datensets besser.

5.2.2 Test Inferenz

hier mit test set und weitere test script mit open vino zum vgl der modelle bzg infer ergebnisse (bilder/zeit)

test set

kaggle set

eigene

5.2.3 Inferenz zeit

inferenz zeiten wurden in FPS für 100 bilder gemessen. Bei mehr als ein Inferenz Request wurde die Asynchrone Api der InferenceEngine verwendet.

hier noch für Raspberry

Modus	Synchron	Asynchron		
		1	2	3
Inferenz Requests	1	2	3	4
SSD MobilenetV2	12,63	25,14	33,46	33,63
SSD InceptionV2	10,71	21,22	28,10	28,29
Faster R-CNN Incept.	0,55	0,61	0,68	0,72

Tabelle 5.2: Vergleich von Inferenz Zeiten

5.3 Optimierungen: Faster r-CNN

mit faster r-cnn wurden nun noch optimierungen durchgeführt, da trotz augmentierung bei mehr epochen overfitting auftrat.

5.3.1 Validierung

bei faster ζ 400k steps:

- dropout
- l2 (rpn loss anschauen)
- mehr daten (4000 statt 3000)
- early stopping (bei zb 300000)

hier loss und mAP für 500k steps



Abbildung 5.5: mAP: nur Augmentierung

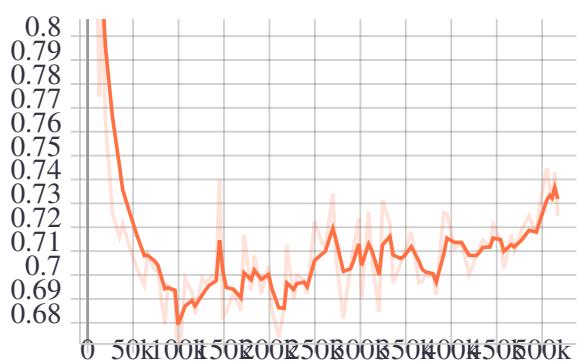


Abbildung 5.6: Loss: nur Augmentierung

und hier die stufen im detaeil

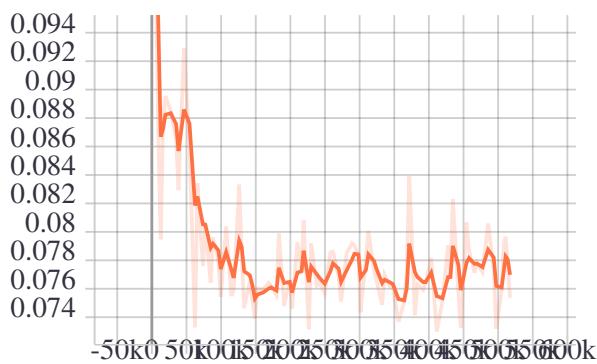


Abbildung 5.7: Classifier Loss: nur Augmentierung

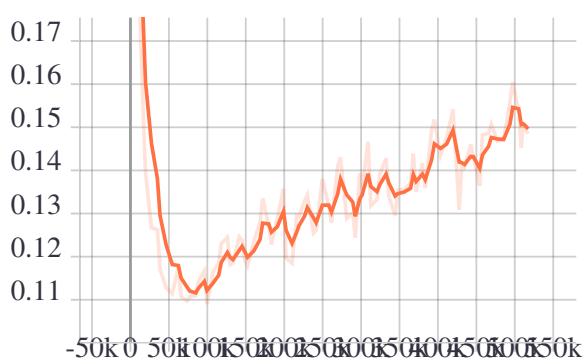


Abbildung 5.8: RPN Loss: nur Augmentierung

durch l2 regulierung in der ersten stufe (rpn), konnte dass overfitting verringert werden.

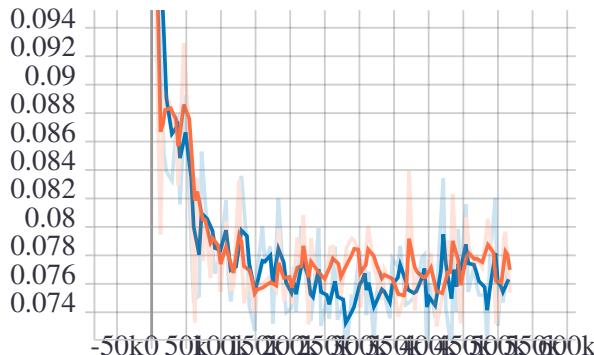


Abbildung 5.9: Classifier Loss: nur Augmentierung

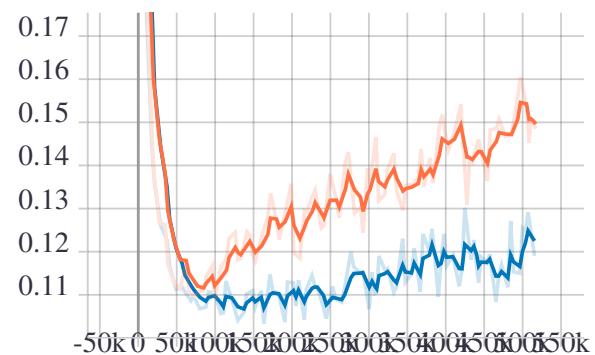


Abbildung 5.10: RPN Loss: nur Augmentierung

Auch auf den gesamten Loss hatte es einen geringen Einfluss.

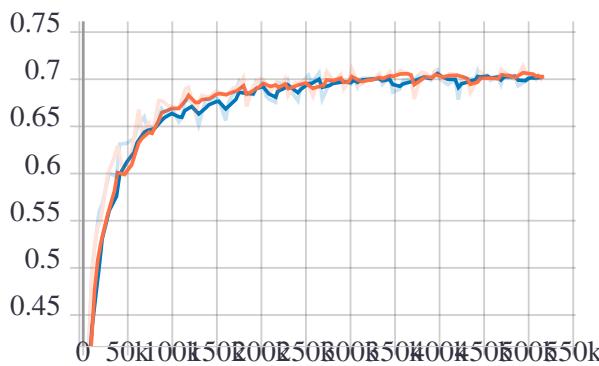


Abbildung 5.11: mAP: nur Augmentierung

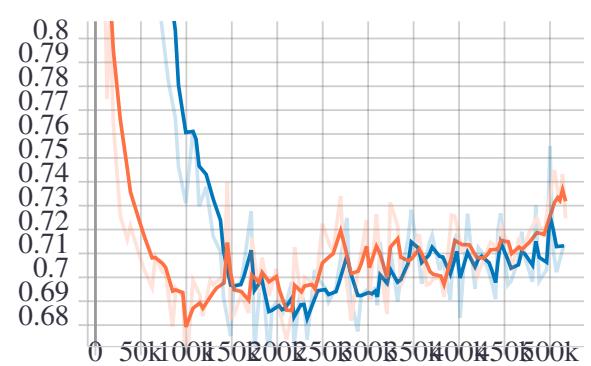


Abbildung 5.12: Loss: nur Augmentierung

weitere Regularisierungstechniken, die angewendet wurden sind Dropout und L2 mit $\lambda = 0.02$ und sind in ?? 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 5.3: Regularisierungen

5.3.2 Test Inferenz

da nicht sehr eindeutig, welche Optimierung die beste ist, wurde auch hier Test Inferenz für Kaggle und Eigen durchgeführt.

5.3.3 Graustufen

da kamereea 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 5.4: 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 5.5: Grayscale

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

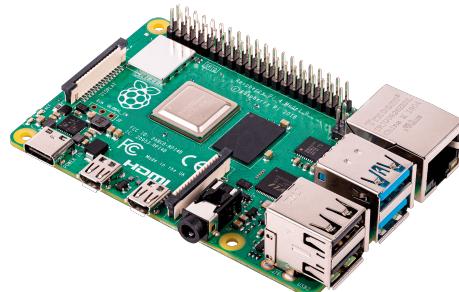


Abbildung 6.1: Raspberry Pi 4

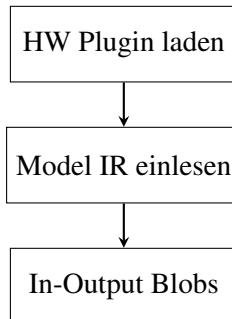
- Netzwerkverbindzuung:
 - GSM Module
 - WiFi-Stick
- Powrebank/Akku aus Sp. Verbrauch von:
 - NCS2
 - Kamera
 - LEDs
 - WiFi Stick

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] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Adaptive Computation and Machine Learning, The MIT Press.
- [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] “https://github.com/tensorflow/models/tree/master/research/object_detection.”

Anhang A

Beispiel für ein Kapitel im Anhang

A.1 Bsp für ein Abschnitt im Anhang