

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	9
2.3 Neural Compute Stick 2	11
3 Anforderungen und Analyse	13
3.1 Ziel der Arbeit	13
3.2 Related Work	13
4 Realisierung Objekt Erkennung	15
4.1 Datensatz	15
4.2 Training	16
4.3 Inferenz	17
5 Evaluierung	19
5.1 Evaluierungs Metriken	19
5.2 Vergleich der Modelle	20
5.3 Optimierungen: Faster r-CNN	23
5.4 Inferenz zeit	25
6 Entwicklung der Anwendung	27
6.1 Aufbau/Hardware	27
6.2 Implementierung/Software	28
7 Test und Validierung	31
8 Zusammenfassung und Ausblick	33
A Beispiel für ein Kapitel im Anhang	37
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

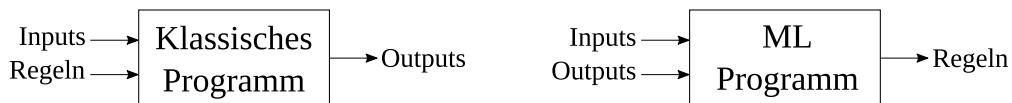
Grundlagen

Dieses Kapitel beschreibt in den ersten Abschnitten die Grundlagen des Machine Learnings, und die Funktionsweise Künstlicher Neuronaler Netze, insbesondere für Computer Vision Anwendungen. Der letzte Abschnitt behandelt die für die Arbeit verwendete KI taugliche Hardware, den Neural Compute Stick 2.

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 um daraus Regeln für Zusammenhänge zwischen Ein- und Ausgabe Daten abzuleiten.

Dadurch unterscheidet sich das Vorgehen wesentlich zur klassischen Programmierung, bei der die Regeln vorab definiert werden müssen.



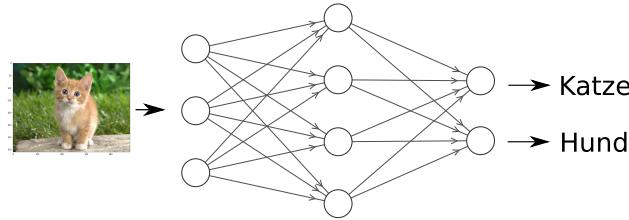
Das ableiten der Regeln erfolgt beim Machine Learning in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei werden die Zusammenhänge zwischen In- und Output Daten als mathematische Funktion betrachtet, welche numerisch angenähert wird.

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 der das Programm keine Labels erhält, sondern diese selber finden soll, oder das *Reinforcement Learning*, bei dem das Programm durch Interaktion mit der Umwelt bestimmte Aufgaben lernt. 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 Gewichtung der Parameter erfolgt dabei in einem iterativen Trainingsprozess, welcher aus den drei Schritten:

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

bestehet und in Abbildung ?? schematisch dargestellt ist.

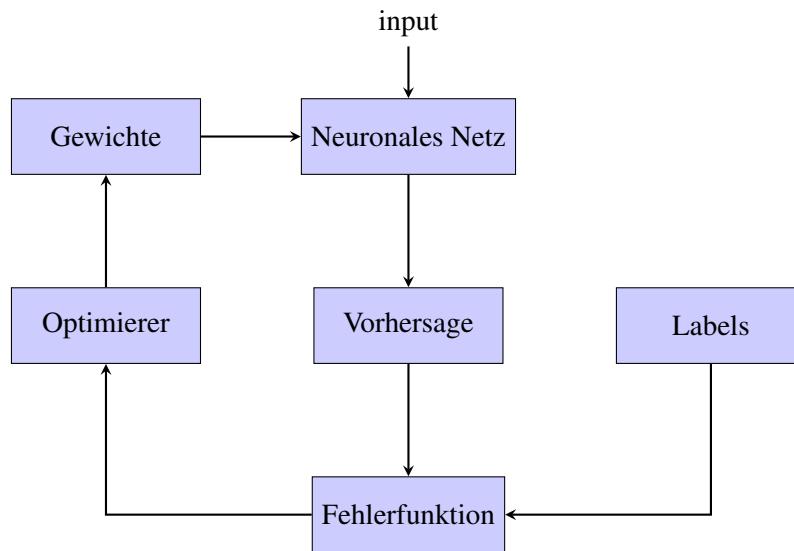


Abbildung 2.1: Trainingsablauf NN

Durch mehrfaches Durchlaufen der Schritte wird die Fehlerfunktion soweit minimiert, sodass das Modell auch für neue Input-Daten die richtigen Aussagen treffen kann. Die mathematischen Berechnungen der einzelnen Schritte werden im folgenden erläutert.

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 ??

Um den Forward Pass für eine gesamte 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 mit der Gewichtsmatrix W ergibt die Werte des Vektors z der aktuellen Schicht,

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

Dieser wird dann elementweise einer nichtlinearen Aktivierungsfunktion $g(z)$ übergeben werden.

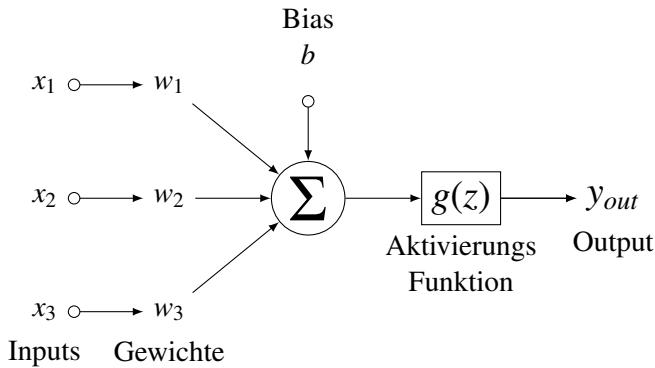


Abbildung 2.2: Einzelnes Perzepton

Für mittlere Schichten (Hidden Layer) wird dabei oft die in 2.1.1 dargestellte *ReLU Funktion* verwendet welche positive Werte beibehält und negative Werte zu 0 setzt.

Das hat den Vorteil, ...

Um für die Outputs einen Wahrscheinlichkeits Wert zwischen 0 und 1 zu erhalten, wird in der letzten Schicht für eine binäre Klassifikation die Sigmoid Funktion (??) verwendet, welche S-Förmig zwischen 0 und 1 verläuft.

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

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

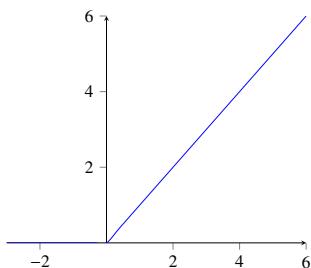


Abbildung 2.3: ReLU Funktion

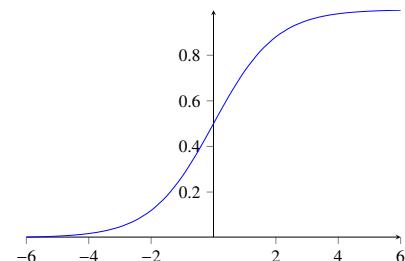


Abbildung 2.4: Sigmoid Funktion

Für eine Kategorische Ausgabe mit mehr als zwei Werten wird die Softmax (2.2) Funktion verwendet, welche eine Wahrscheinlichkeits Verteilung über alle Ausgabe Neuronen generiert.

$$g(z) = \frac{e^z}{\sum e^x} \quad (2.2)$$

Fehlerbestimmung

Die Abweichung des geschätzten Werts, welche an den Neuronen der letzten Schicht vorliegen, zum tatsächlichen Werten, dem Label, wird mithilfe einer geeigneten Fehlerfunktion bestimmt. Für eine Lineare Regression wird dabei z.B. der absolute oder quadratischen Abstand verwendet.

Für Klassifikationsmodelle wird meistens eine logarithmische Fehlerberechnung, wie die *Cross Entropy Funktion* 2.3 verwendet.

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

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 Fehlerfunktion L für jede Schicht partiell nach den Gewichten w abgeleitet, was wie in gl. 2.4 dargestellt mithilfe der Kettenregel über die Aktivierungsfunktion z geschieht.

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

Mit dem ermittelten Gradienten werden dann die Gewichte nach Gleichung 2.5 angepasst.

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

wobei die *Lernrate* η 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 Fehlerfunktionen wie in 2.5 dargestellt, findet eine Überanpassung (*Overfitting*) des Modells an die Trainingsdaten statt.

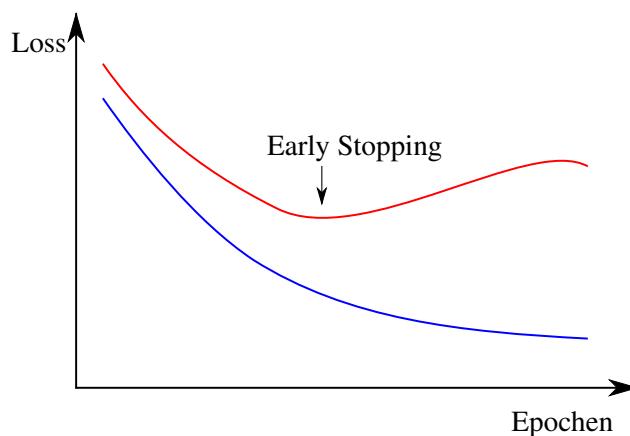


Abbildung 2.5: Overfitting, Lossfunktion, Quelle: [1]

Gründe dafür sind zu wenig Trainingsdaten oder ein zu komplexes Modell, welches sich aufgrund der vielen Parameter wie bei einer Funktionen hohen Grades, die Möglichkeit hat sich an jeden Datenpunkt anzupassen und damit nicht mehr generalisierbare Aussagen für neue Datenpunkte treffen kann.

Stehen nicht mehr Trainingsdaten zur Verfügung, kann Overfitting mit einer der folgenden Techniken verhindert werden.

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.

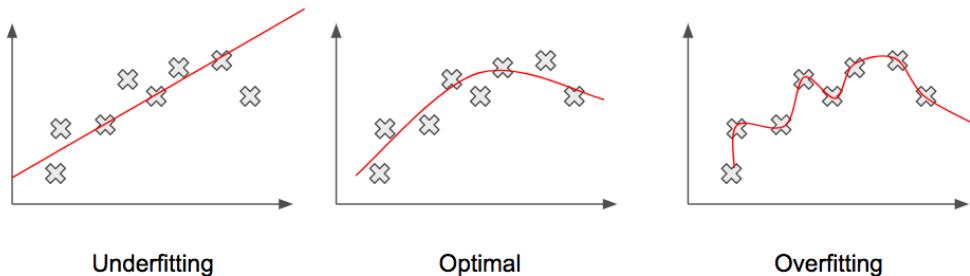


Abbildung 2.6: Quelle: [2]

Regularisierung der Parameter (L1/L2)

Bei Regularisierung wird der Lossfuction als weiterer Term eine aufsummierung aller Gewichte hinzugefügt, wodurch diese bei der Minimierung des Lossfunktion klein gehalten werden und damit einhergehend weniger potential zum Overfitting besteht.

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

Dropout

Beim Dropout werden in mit einer bestimmten Wahrscheinlichkeit einige Neuronen Neuronen (Bsp 50%) zu 0 gesetzt, um alternative Gewichtsanpassungen zu erzwingen.

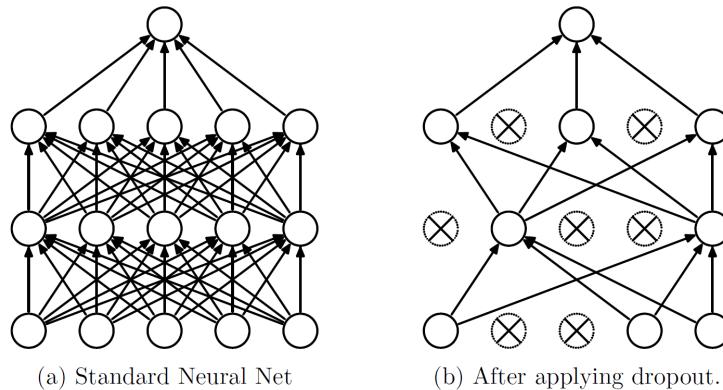


Abbildung 2.7: Dropout, Quelle: [3]

Early Stopping

Beim early Stopping wird das Traing an der Stelle unterbroche, an der die Lossfunktion ihr Minimum erreicht hat, markierte stelle in 2.5.

2.2 Computer Vision

Computer Vision bezeichnet die Erkennung und Verarbeitung des Inhalts einer Bild- Oder Video Datei. Es kommen dabei Techniken der Bilverarbeitung zusammen mit Deep Learnig Algorithmen zum Einsatz.

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks sind eine Erweiterung der in ?? beschriebenen Neuronalen Netze und besonders geeignet für die Bilderkennung. Befor die Klassifizierung mittels Fully Connected Network stattfin-

det, sollen für die Klasse spezifische Merkmale aus dem Input Bild heraus extrahiert werden. Dafür werden über das Bild zeilenweise Filtermatrizen mit kleinerer Dimension (3×3 , 5×5) geschoben und eine math Faltung angewendet. Die daraus entstehenden Feature Maps sind Matrizen, in denen korellationen zwischen fileter ung input in form von Mustern abgebildet werden. Die Werte der Filter Matrizen entsprechen den zu lernenden Gewichten und werden mithilfe der Backpropagation angepasst.

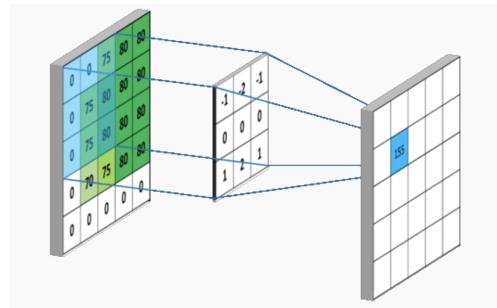


Abbildung 2.8: Faltung, [4]

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 Ferture Maps in jeder Schicht.

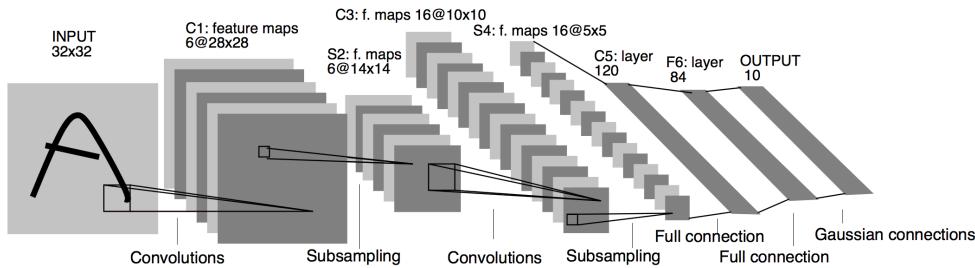


Abbildung 2.9: Faltung, [5]

Vorteile der CNNs, verglichen mit den in Abschnitt 2.1.1 beschriebenen *Feed Forward Networks* sind der geringere Rechenaufwand durch die gemeinsame Nutzung der Parameter durch die 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 Convolutional Layern für alle Klassen sehr ähnlich sind, nicht bei jedem Modell neu lernen zu müssen, wird häufig *Transfer Learning* angewendet, eine Technik, bei der vortrainierte Filter verwendet und durch Fine Tuning an den eigenen Datensatz angepasst werden.

Architekturen

Seit der Einführung des ersten CNN 1998 von Yann LeCun [5], welches in Abbildung ?? dargestellt ist, wurde eine Vielzahl an neuen Architekturen für genauere und effizientere Ergebnisse entwickelt.

Um diese einheitlich miteinander zu vergleichen und zu bewerten wurde die *Large Scale Visual Recognition Challenge (ILSVRC)* [6] von ImageNet gegründet.

Bekannte Modelle, welche die Challenge gewonnen haben sind, wie in [7] nachzulesen ist:

- Alexnet (2012), mehrere conv layer hintereinander
- ZF Net (2013)
- GoogleLeNet (2014), Inception Module
- VGGNet 2014
- ResNet (2015)

2.2.2 Objekt erkennung

Neben der Information, was sich auf einem Bild geht bei der Objekt Erkennung auch darum herausfinden wo sich das Objekt auf dem Bild befindet.

Dafür wird ein in 2.2.1 beschriebenes CNN als Basis in einem Framework welches die Lokalisierung vornimmt verwendet.



Abbildung 2.10: Unterschied: Classification - Detection

2.2.3 Machine Learning Frameworks

Deep Larning Algorithmen beeinhalten eine vielzahl an komplexen Berechnungsschritten und Parametern. Um diese nicht jedesmal von Grund auf neu implementieren zu müssen bieten Deep Learning Frameworks eine vereinfachte Möglichkeit die Modelle zu konstruieren.

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

TensorFlow, welches in der Thesis verwendet wurde, stammt von Google und ist ein aufgrund seiner hohen Flexibilität besonders in der Forschung oft verwendetes Framework.

2.3 Neural Compute Stick 2

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

Anwendungen die auf eingebetteten Systemen laufe, wie etwa auf einem Raspberry Pi, kommen dabei schnell an ihre Grenzen.

Häufig werden daher die Bilddaten über eine Cloud zur verrechnung/inferenz an einen Leistungsstärkeren Rechner gesendet.

Sollen die Daten, wie dies beim Edge Computing der Fall ist, auf dem Gerät direkt verarbeitet werden, gibt es speziell für die Inferenz von Deep Learing Algorithmen Ausgelegt Hardware, welche NN spezifische Berechnung besonders effizient durchführen kann.

Diese können entweder als eigenständiges *System on Chip* (SoC) System wie zB der *Nvidia Jetson TX2* agieren oder zusammen mit einem Host, wie das bei dem in der Arbeit verwendeten Neural Compute Stick 2 der Fall ist.

Dieser verwendet den Movidius Myriad X Vision Processing Unit (VPU) welcher neben der Neural Compute Engine zur beschleunigten berechnung Neuronalen Netzen, einen Bildbeschleuniger, 16 SHAVE Prozessoren, Bildsignalprozessoren udn RISC CPU Core besitzt. [8]



Abbildung 2.11: ncsc2

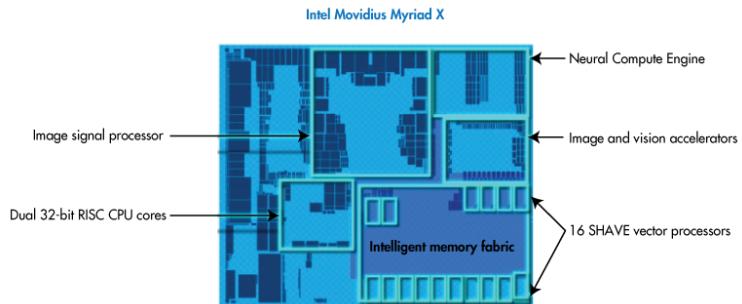


Abbildung 2.12: myriad

2.3.1 OpenVino Toolkit

Um die Inferenz eines trainierten Deep Learning Modells auf dem Neural Compute Stick ausführen zu können, wird das Toolkit *OpenVino* verwendet.

Dieses ist eine Plattform zur Optimierung und Inferenz von CNN basierten Modellen auf unterschiedlicher Intel Hardware.

Dabei wird ein eigenes Dateiformat verwendet, die *Intermediate Representation* (IR), welche die Struktur/Architektur des Modells in einer .xml Datei und die trainierten Parameter/Gewichte in einer Binary (.bin) Datei abbildet.

Mit dem *Model Optimizer* können Modelle der Frameworks TensorFlow, Caffe, ONNX, Kaldi, oder MX-NET in das IR Format konvertiert werden.

Um diese dann auf die entsprechende Hardware zu laden und anwendbar zu machen, wird die auch in OpenVino enthaltene *InferenceEngine* verwendet.

Diese bietet eine API mit der aus der Anwendung heraus in den Programmiersprachen C++ oder Python auf die Funktionen der InferenceEngine zugegriffen werden können.

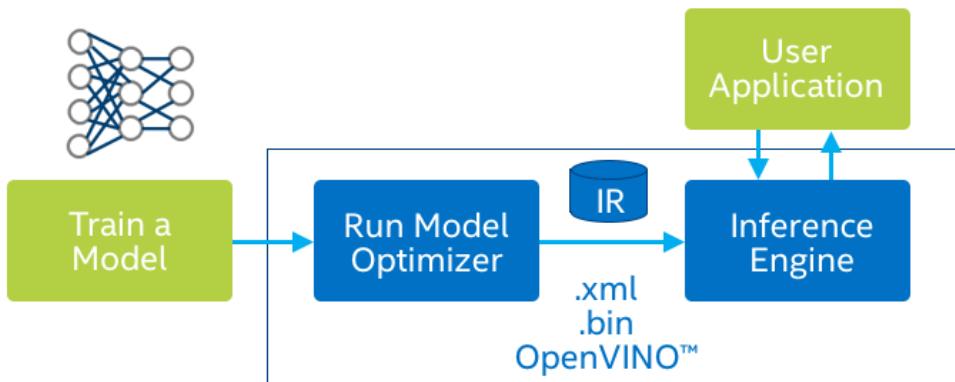


Abbildung 2.13: Workflow: OpenVino Toolkit

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

Wie in 2.2.2 erwähnt, werden für die Object Detection neben einem CNN zur Feature Extraction weitere Strukturen/Framework benötigt zur Lokalisierung des Objekts im Bild

Die Entwicklung immer genauerer und effizienterer Frameworks ist Gegenstand aktueller Forschung und lassen sich wie in dem Übersichtspaper [9] beschrieben in die zwei Grundarten *Region proposal based* und den *Regression/Classification Based* einteilen.

3.2.1 Faster R-CNN

Region Proposial Based Ansatz, ... [10] bei dem das RPN ein Bild als Input erhält und daraus mögliche Regionen für Objekte im Bild findet.

hier in einem Fully Convolutional realisiert, über dessen Feature Maps im Sliding Window Verfahren vordefinierte Anker Boxen konvolviert werden. Der resultierende Vektor wird dann in einen binären FC Layer (cls) zur Klassifizierung und einen Box-Regressor Layer (reg) für die Koordinaten gegeben.

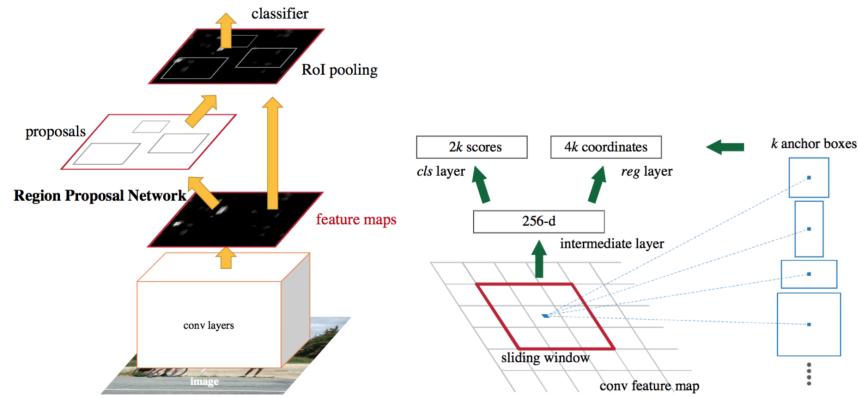


Abbildung 3.1: Faster R-CNN, quelle: [11]

3.2.2 SSD: Single Shot MultiBox Detector

Ist ein Regression/Classification Based Ansatz ... [12]

Dabei werden dem Backbone CNN weitere feature Layer verschiedener Größe angehängt, welche zusammen mit default anker boxen und einem score für die Anwesenheit des Objekts in der Box in einen non max suppression layer gegeben, welcher die finalen predictions bestimmt.

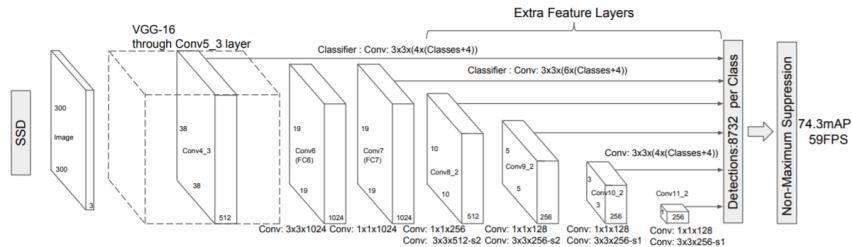


Abbildung 3.2: SSD, quelle: [13]

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* [14] 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 Sklierung, 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* [15] konnten so verschiedene Augmentierungstechniken auf das Datenset angewendet werden.

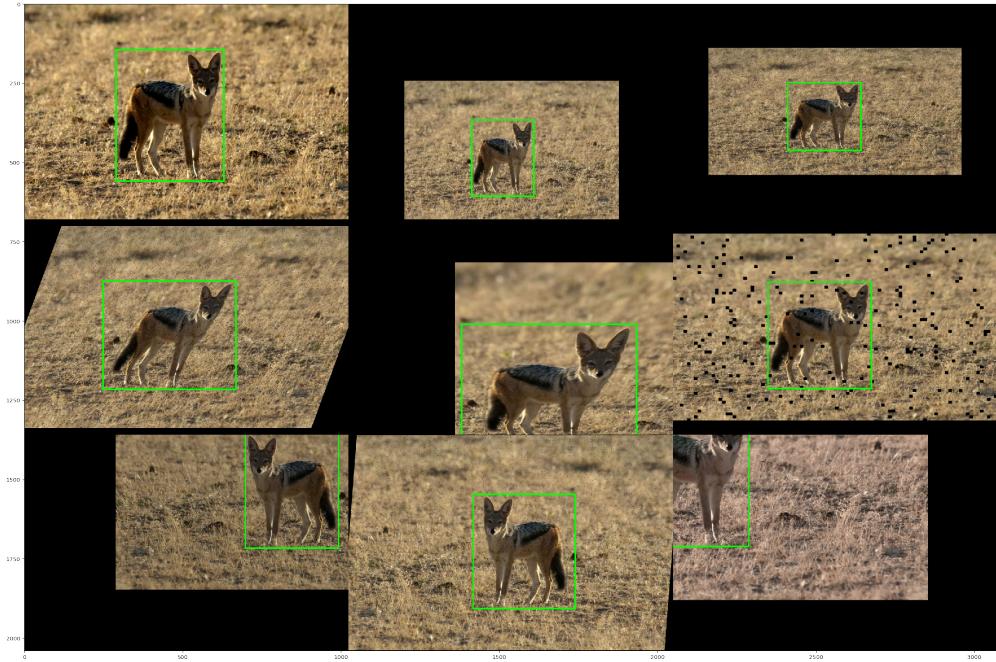


Abbildung 4.1: Anwendung von Augmentierungstechniken

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 [16] 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 geeignete Gpu zur Verfügung stellt.

4.2.2 Trainingsworkflow

Das Ergebnis des Trainings kann neben der Auswahl eines geeigneten Modells, 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 Konfiguration des Trainings erkannt und durch Anpassen von entweder Datensatz, Modell oder Parameter korrigiert/verbessert werden. Es ergibt sich folgender Workflow.

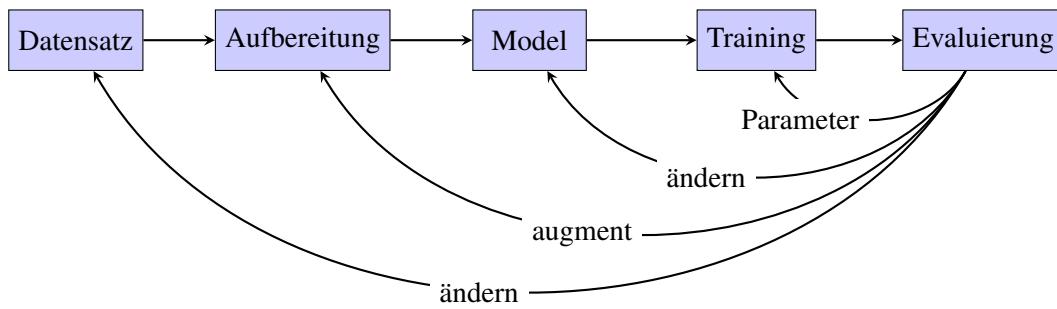


Abbildung 4.2: Trainingsworkflow

Die Ergebnisse werden im nächsten Kapitel (5) diskutiert.

4.3 Inferenz

- Tensorflow graph exportieren/einfrieren
- mit model opt in ir Format
- mit inf engine inferieren

4.3.1 OpenVinos InferecneEngine

Folgend der prinzipielle Ablauf der Inference Engine für die Inferenz eines Modells auf dem Neural Compute Stick

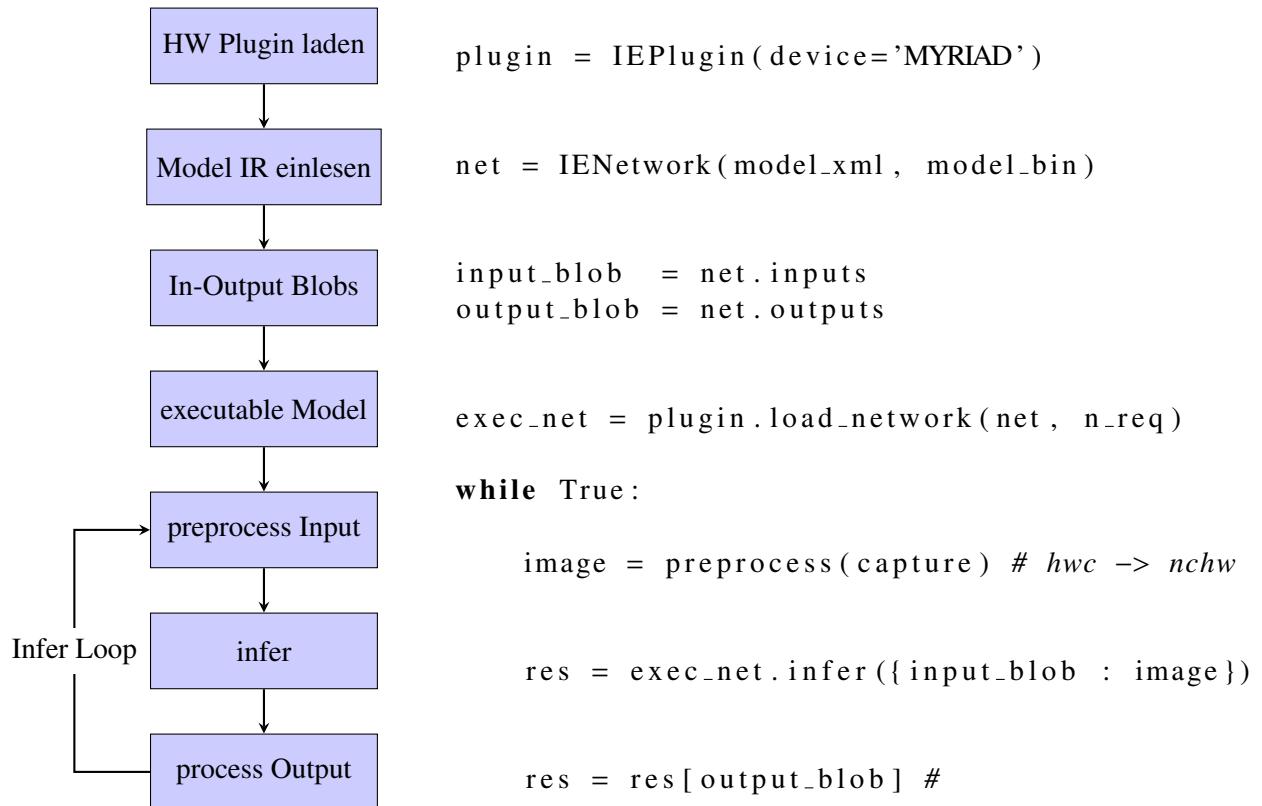


Abbildung 4.3

Output Format variiert je nach Modell, Image Classification, ObjectDetection sowie Instance Segmentation. 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

Asynchrone Inferenz

Um die Ausführungszeit der Inferenz zu verringern, können, je nach Hardware Optimierungen wie Asynchrone Inferenz, Inferenz requests auf mehreren Threads, oder Batching des Inputs vorgenommen werden.

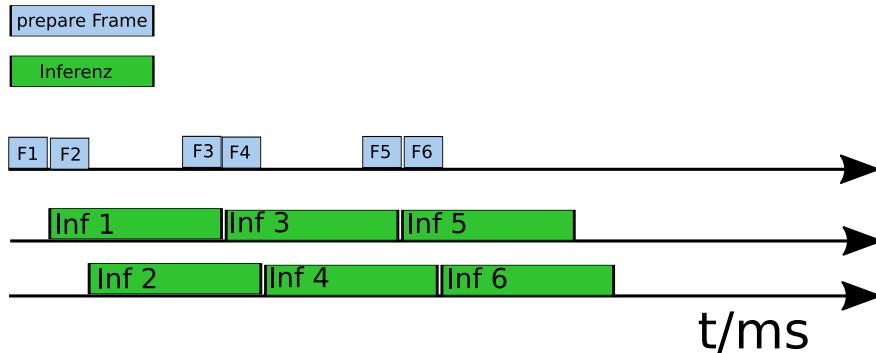


Abbildung 4.4: Asynchron und mehrere Inferenz Requests

In Abbildung 4.4 dargestellt die Asynchrone ausführung zwischen dem aufbereiten der Frames und der Inferenz, sowie mehrere parallel ausgeführte Inferenz Requests.

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 Threshhold, häufig 50% gilt die Schätzung als *True Positive*, andernfalls als *False Positive*.

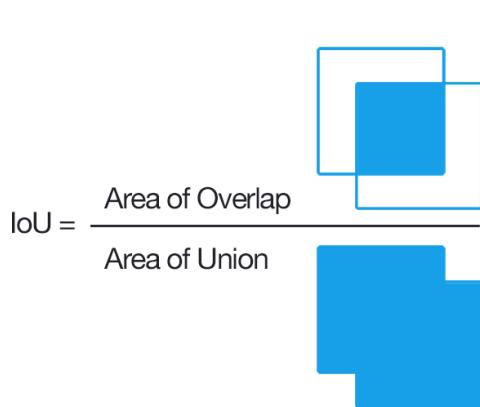


Abbildung 5.1: Intersection over Union

		geschätzter Wert
tatsächlicher Wert	p'	p
	n'	n
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

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

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 insgesammt 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 epoche 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.

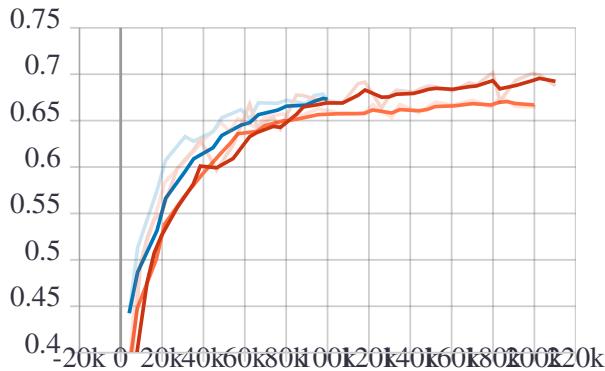


Abbildung 5.3: mAP

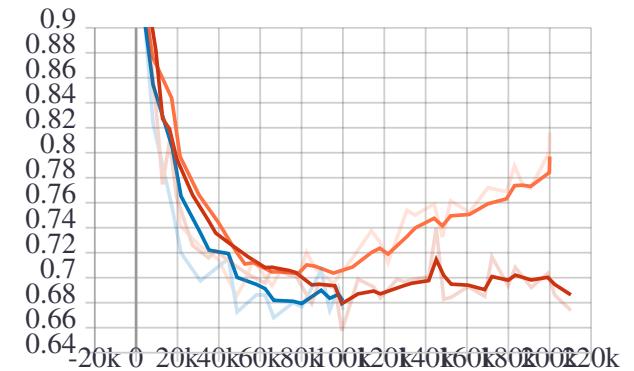


Abbildung 5.4: Loss

● 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-Skripts in die InferenceEngine geladen zu werden.

Mithilfe dieses Skripts konnten nun die Inferenz für folgende Testszenarien:

- Inferenz auf Test Set *OpenImages*
- Inferenz auf *The iWildCam 2019 Dataset*[17]
- 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

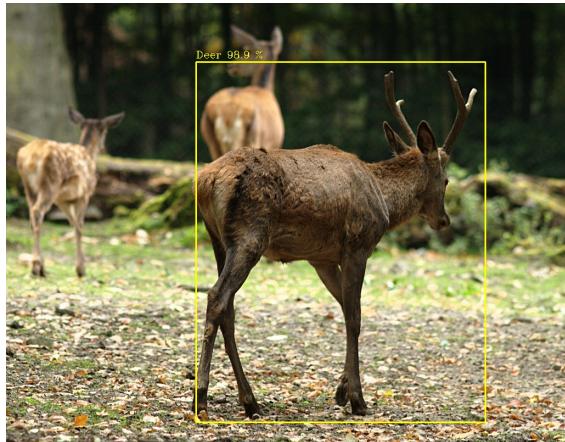


Abbildung 5.5: SSD

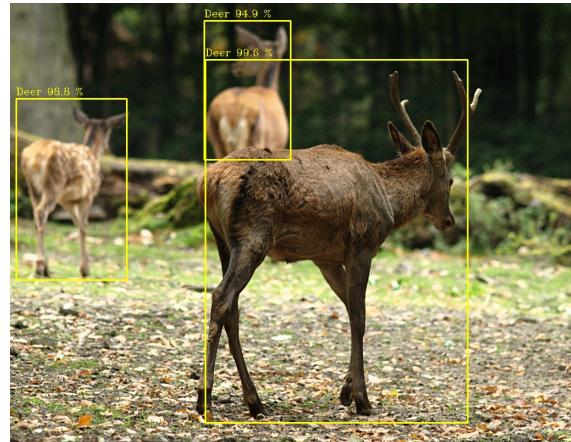


Abbildung 5.6: Faster R-CNN

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.

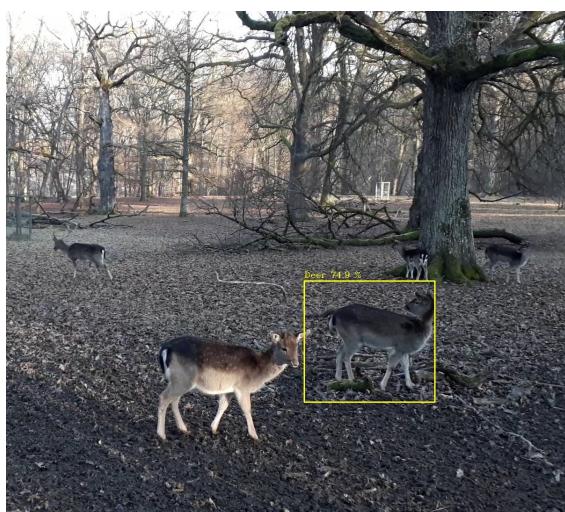


Abbildung 5.7: SSD Mobilnet

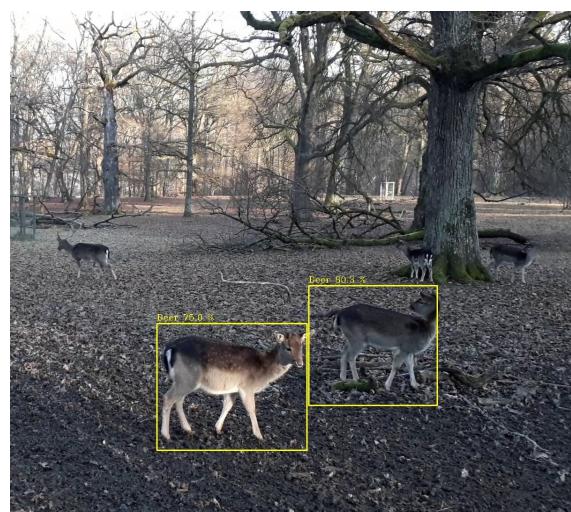


Abbildung 5.8: SSD Inception

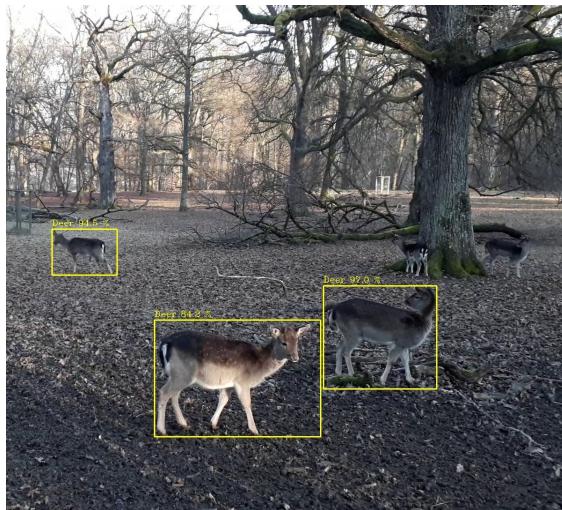


Abbildung 5.9: Faster R-CNN + Early Stopping

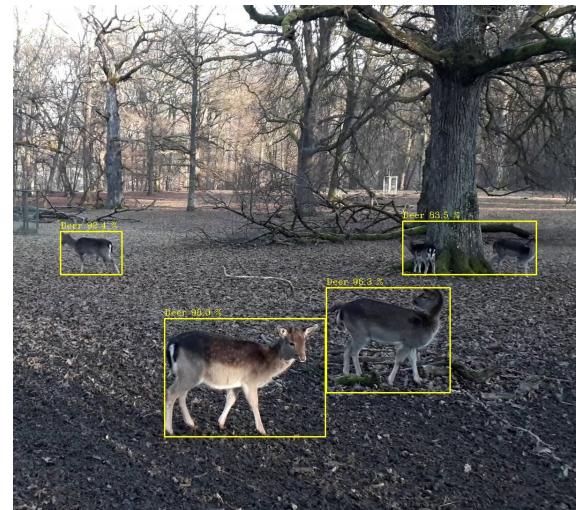


Abbildung 5.10: Faster R-CNN + Aug

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 Verschiedene Augmentierungen

Folgende Plots zeigen Faster R-CNN mit einer Augment pro Bildv, zwei Augment pro Bild (3000Samples) und zwei Augment pro Bild (4000Samples)



Abbildung 5.11: mAP

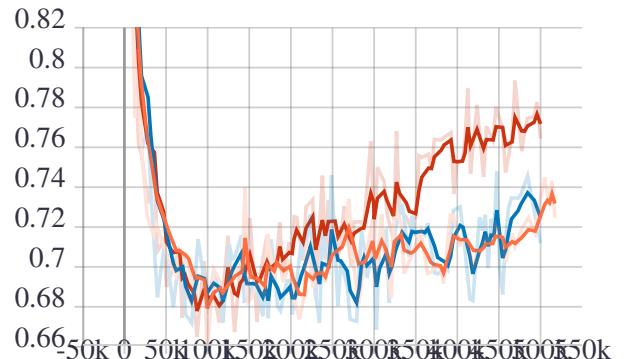


Abbildung 5.12: Loss

● Ohne ● Early Stopping ● Augmentierung

Mehr Augmentierung führt zu weniger loss, jedoch auch schlechterem mAP.

5.3.2 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 seperat 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.

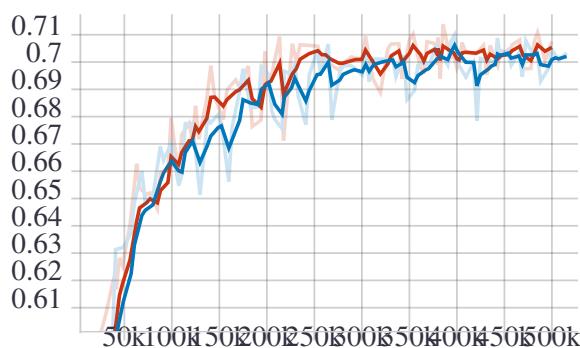


Abbildung 5.13: mAP

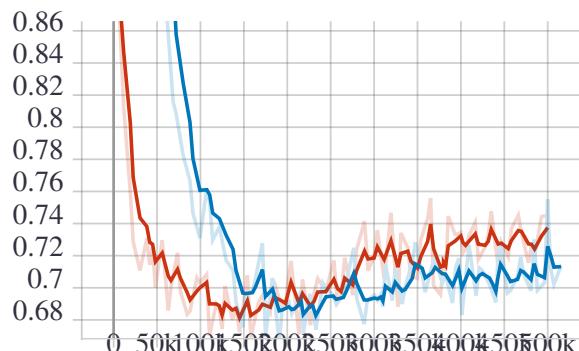


Abbildung 5.14: Total Loss



Abbildung 5.15: Classifier Loss

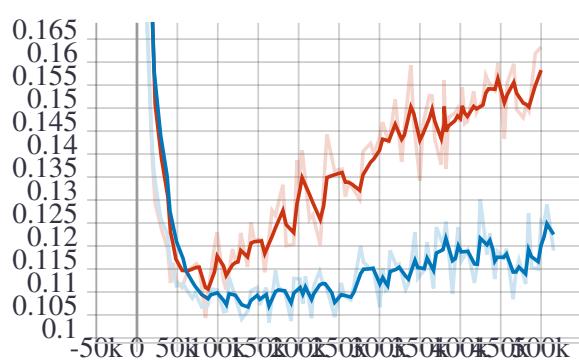


Abbildung 5.16: RPN Loss

weitere Trainings Konfigurationen waren

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

und sind in tabelle so und so dargestellt.

5.3.3 Test Inferenz

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

>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

5.3.4 Graustufen

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

5.3.5 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

5.3.6 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	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

Kapitel 6

Entwicklung der Anwendung

Dieses Kapitel beschreibt die Entwicklung der Anwendung, als autonomes System welches auf dem Raspberry Pi läuft, Bilder über eine infrarotfähige Kamera aufnimmt und diese auf dem Neural Compute Stick 2 mit dem trainierten Modell inferiert. Desweiteren wird die Implementierung der verbindugn zu einem Host Pc beschrieben.

6.1 Aufbau/Hardware

Im folgenden werden die zur Realisierung der Anwendung verwendeten Komponenten erläutert.

Raspberry Pi

Der Anwendungs Code läuft auf dem Raspberry Pi. Der Neural Compute Stick 2 wird über eine USB Schnittstelle verbunden.



Abbildung 6.1: Raspberry Pi 4

Kamera

Verwendet wurde ein Raspberry Pi Kamera Modul mit 5MP OV5647 Sensor und automatschem umschalten der Infrarot Funktion der Marke Longrunner verwendet. Dieses verfügt zusätzlich über zwei Infrarot LEDs, mit 850 nm wellenlänge.

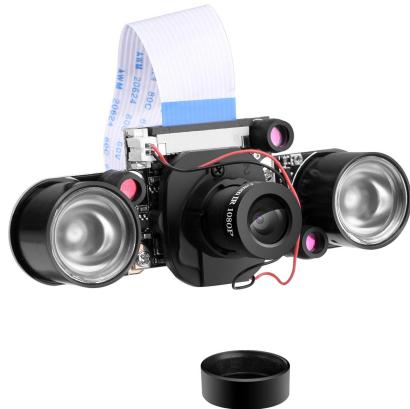


Abbildung 6.2: Longrunner for Raspberry Pi 4 Camera Kamera Module

Für das Umschalten in den Infrarot Modus, wird der Infrarot Filter, welcher sich für normale Aufnahmen vor der Linse befindet entfernen, was über einen magnethaltschalter der mit einem Helligkeitssensor getriggert wird erfolgt.

sonstiges

- Internet Stick
- Hülle
- Powerbank

6.2 Implementierung/Software

Bestht aus 3 Scripten:

- main
 -
- detection
 - init
 - motion
 - create exec net
 - infer frames
 - process result
- connection
 - init
 - login
 - connect
 - send
 - disconnect

Inferenz

Um nicht durchgehend die Input Frames welche die Kamera liefert inferieren zu müssen, wurde mithilfe der Library OpenCV ein bewegungserkennung implementiert. Diese speichert bei start der Anwendung ein Frame als Referenz ab, und kann damit alle weiteren Input frames vergleiche, indem der Abstand der einzelnen Pixel werte berechnet und gemittelt wird.

Der Ablauf der reinen Asynchronen Inferenz ist grob in folgendem Pseudocode dargestellt.

Algorithm 1 Asynchrone Inferenz

```

while true do
    capture Frame
    if Frame has Motion then
        Buffer ← Frame
    end if
    for reqId = 0 to reqMax do
        if Model.requests[reqId].wait(0) then
            result = Model.requests[reqId].output
            inferedFrames ← (result, currentFrames[reqId])
            if Buffer not empty then
                currentFrames[reqId] ← Buffer
                inFrame = preprocess: currentFrames[reqId]
                Model.inferAsync(reqId, inFrame)
            end if
        end if
    end for
    return inferedFrames
end while

```

wobei die wait Funktion mit Timeout = 0 nicht blockierend ist.

Dadurch war es möglich trotz langsamerer inferenz zeit als capture zeit, durch zwischenspeichern alle frames zu inferieren, unter der Annahme, das nur zeitweise bewegung erkannt und damit inferiert werden muss.

Connection

hier prinzip remote Proxy verbindung über SSH und SCP Protokol erklären sowie implementierung mit remote it api und scp command.

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] Gringer, “Overfitting svg.” https://de.m.wikipedia.org/wiki/Datei:Overfitting_svg.svg, 2007.
- [2] M. Deshp and e, “A Guide to Improving Deep Learning’s Performance.”
- [3] R. Maksutov, “Deep study of a not very deep neural network. Part 5: Dropout and Noise.”
- [4] M. S. Researcher, PhD, “Simple Introduction to Convolutional Neural Networks.”
- [5] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” p. 46.
- [6] “ImageNet Large Scale Visual Recognition Competition (ILSVRC).”
- [7] “Stanford - CS231n Convolutional Neural Networks for Visual Recognition.”
- [8] M. Häußermann, “Funktion und Effizienz von Hardware für Deep Neural Networks.”
- [9] A. Ouaknine, “Review of Deep Learning Algorithms for Object Detection.”
- [10] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,”
- [11] “Object Detection for Dummies Part 3: R-CNN Family.”
- [12] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” vol. 9905, pp. 21–37.
- [13] “SSD : Single Shot Detector for object detection using MultiBox.”
- [14] 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,”
- [15] 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.
- [16] “TensorFlow Object Detection Api.” https://github.com/tensorflow/models/tree/master/research/object_detection.
- [17] 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