

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	10
2.3 Neural Compute Stick 2	11
3 Anforderungen und Analyse	15
3.1 Ziel der Arbeit	15
3.2 Related Work	15
4 Realisierung Objekt Erkennung	17
4.1 Datensatz	17
4.2 Training	18
4.3 Inferenz	19
5 Evaluierung	21
5.1 Evaluierungs Metriken	21
5.2 Vergleich der Modelle	22
5.3 Optimierungen: Faster R-CNN	25
5.4 Inferenz zeit	28
6 Entwicklung der Anwendung	31
6.1 Aufbau/Hardware	31
6.2 Implementierung/Software	32
7 Test und Validierung	39
8 Zusammenfassung und Ausblick	41
A Beispiel für ein Kapitel im Anhang	45
A.1 Bsp für ein Abschnitt im Anhang	45

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.

Kapitel 2

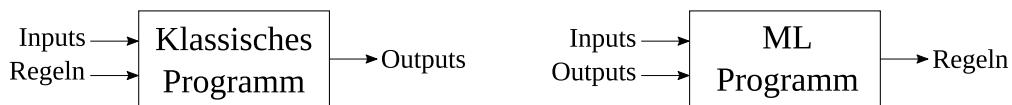
Grundlagen

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.



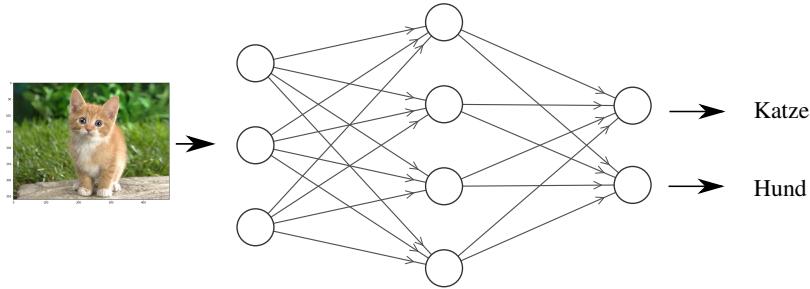
Das ableiten der Regeln erfolgt beim Machine Learning in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei werden die Zusammenhänge zwischen In- und Output Daten als mathematische Funktion betrachtet, welche numerisch angenähert wird.

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.

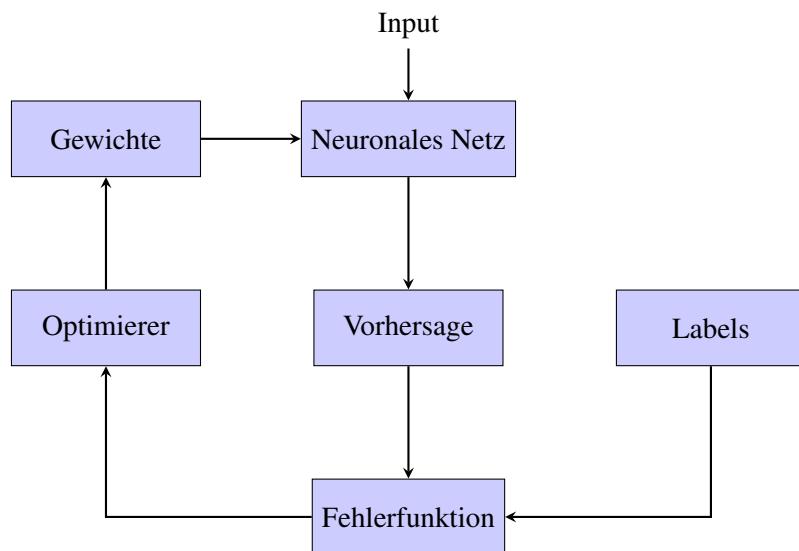


Abbildung 2.1: Trainingsablauf NN

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 ??

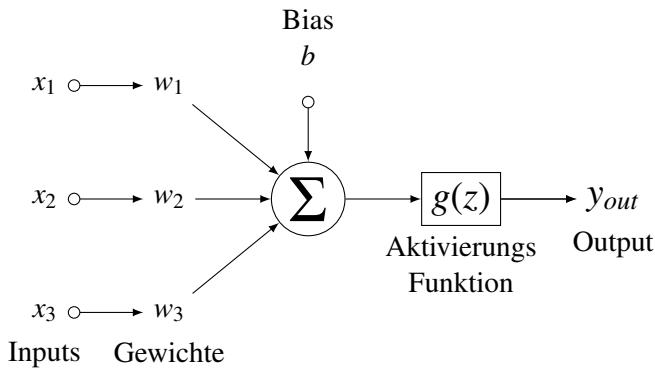


Abbildung 2.2: Einzelnes Perzeptron

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 mit der Gecwichtsmatrix 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 wird.

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.

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

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

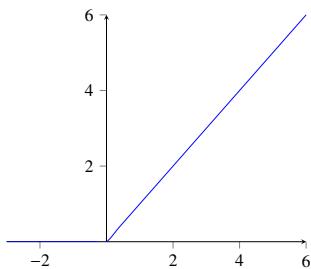


Abbildung 2.3: ReLU Funktion

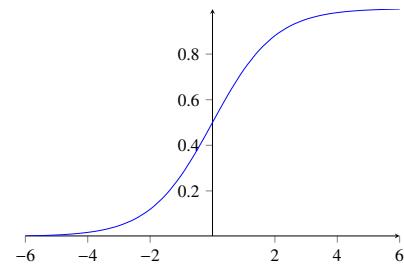


Abbildung 2.4: Sigmoid Funktion

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 Fehlerfunktion wie in 2.5 dargestellt, findet eine Überanpassung (*Overfitting*) des Modells an die Trainingsdaten statt.

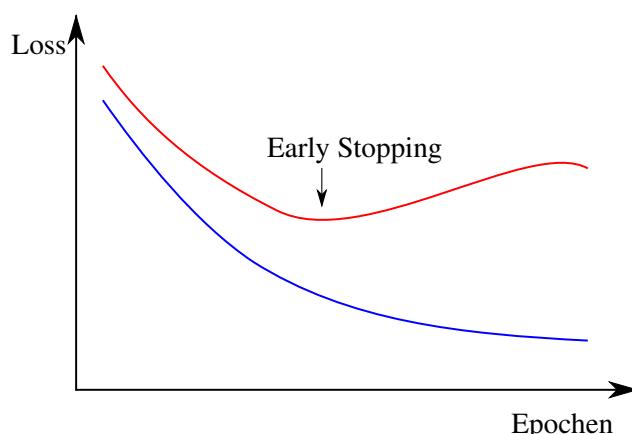


Abbildung 2.5: Overfitting, Lossfunktoin, Quelle: [1]

Gründe dafür sind zu wenig Trainingsdaten oder ein zu komplexes Model, 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.

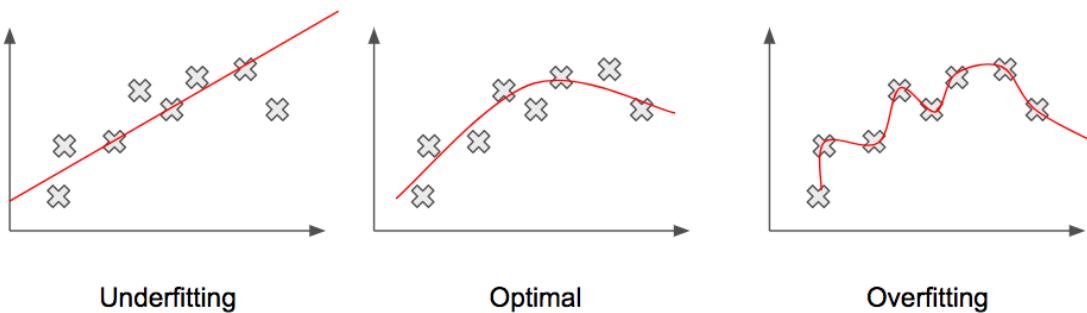


Abbildung 2.6: Quelle: [2]

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.

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 (Bsp 50%) zu 0 gesetzt, um alternative Gewichtsanpassungen zu erzwingen.

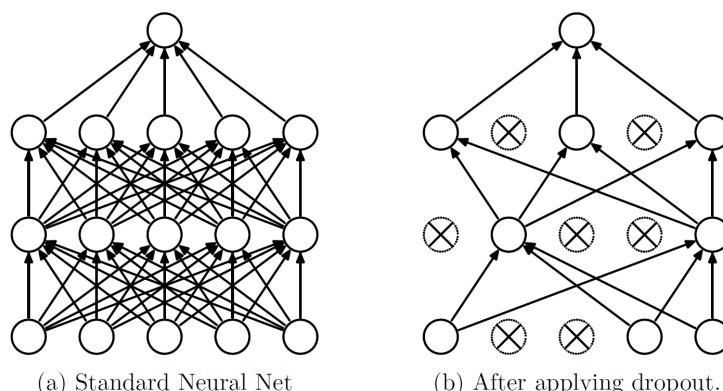


Abbildung 2.7: Dropout, Quelle: [3]

Early Stopping

Beim early Stopping wird das Training an der Stelle unterbrochen, 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 Bildverarbeitung zusammen mit Deep Learning 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. Bevor die Klassifizierung mittels Fully Connected Network stattfindet, 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 Faltung angewendet. Die daraus entstehenden Feature Maps sind Matrizen, in denen Korrelationen zwischen Filterung input in Form von Mustern abgebildet werden. Die Werte der Filter Matrizen entsprechen den zu lernenden Gewichten und werden mithilfe der Backpropagation angepasst.

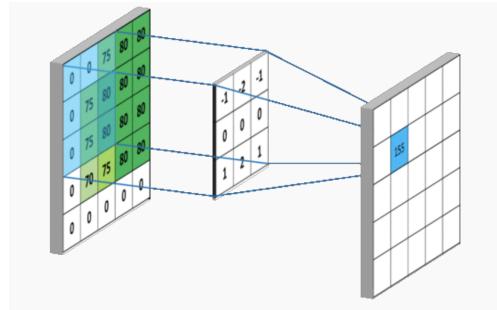


Abbildung 2.8: Faltung, [4]

Durch die hintereinanderschaltung mehrerer Convolutional Layer 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.

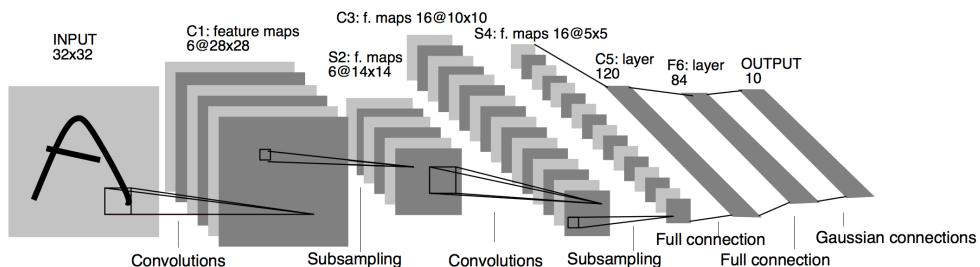


Abbildung 2.9: Faltung, [5]

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.

Architkturen

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.

Zur bewertung dieser anhand einheitlicher Kriterien, wurde die *Large Scale Visual Recognition Challenge (ILSVRC)* [6] von ImageNet gegründet.

Bekannte Modelle, welche die Challenge gewonnen haben sind, wie in [7] aufgelistet 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 befindet geht es bei der Objekt Erkennung auch darum herausfinden wo sich das Objekt auf dem Bild befindet.

Dafür wird die CNN Architektur so erweitert, das das Modell Neben den Input Bildern auch die Koordinaten für das Training erhält, und dadurch diese mitlernt.

Dafür gibt es verschiedene Ansätze, welche im Abschnitt 3.2 genauer beschrieben werden.



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 Learning Algorithmen ausgelegte Hardware, welche NN spezifische Berechnung besonders effizient durchführen kann.

Diese können entweder als eigenständiges *System on Chip* (SoC) System wie z.B. 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 und RISC CPU Core besitzt. [8]



Abbildung 2.11: ncsc2

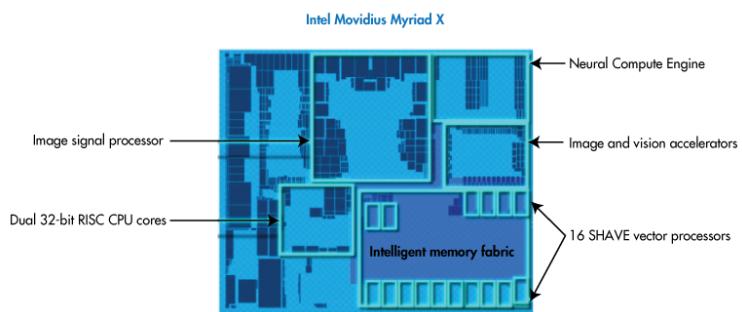


Abbildung 2.12: myriad

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.

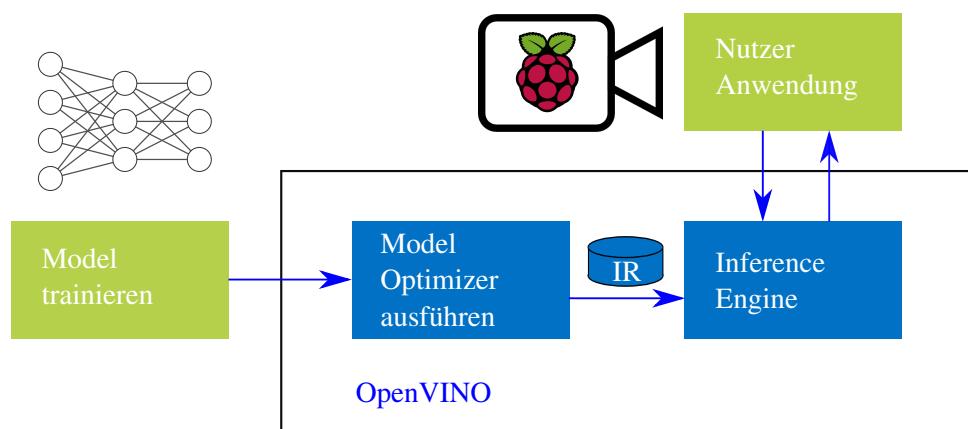


Abbildung 2.13: OpenVINO Workflow

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.

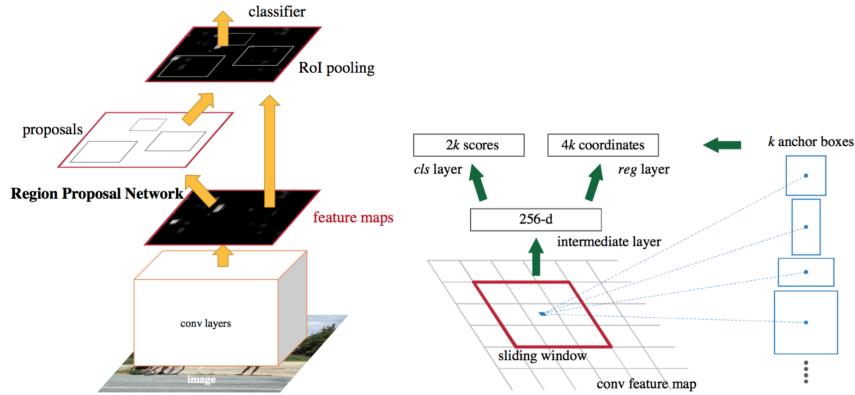


Abbildung 3.1: Faster R-CNN, quelle: [11]

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.

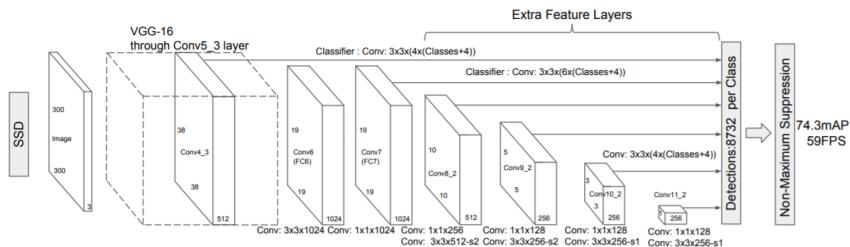


Abbildung 3.2: SSD, quelle: [13]

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.

Dabei hat die Anzahl an Bildern pro Klasse zwischen 200 und 2000 für die Trainingssätze variiert. Die Test und Validierungssätze waren entsprechend kleiner.

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 in der Bilder/Objekterkennung 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 an den Bildern vorgenommen.

Mithilfe eines Python Scripts in welchem die Library *imgaug* [15] verwendet wurde, konnten so verschiedene Augmentierungstechniken auf das Datenset angewendet werden.

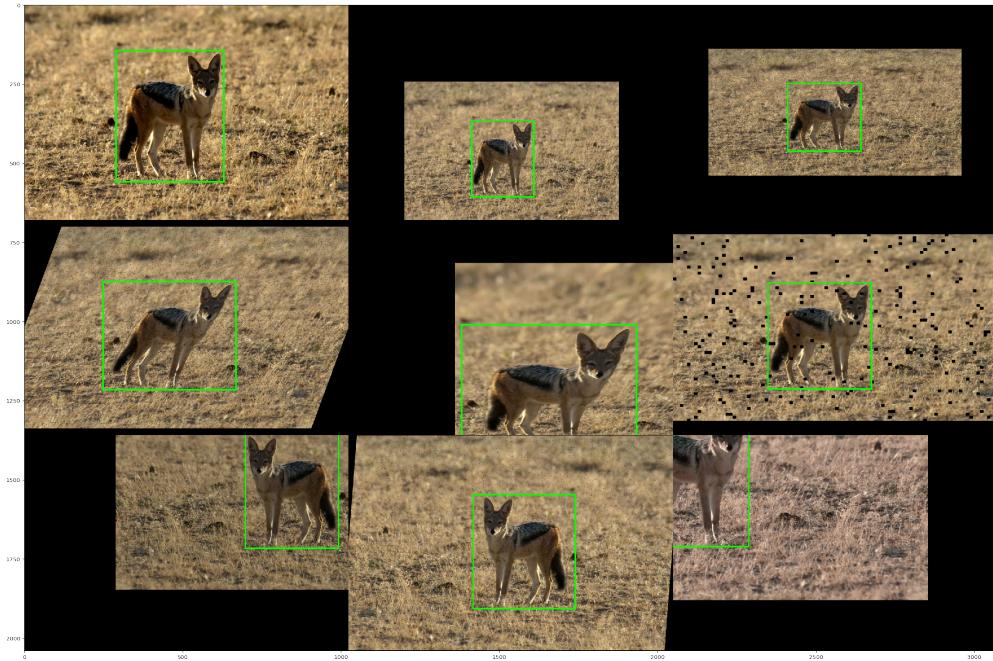


Abbildung 4.1: Anwendung von Augmentierungstechniken

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.

Um unabhängig von der leistungsfähigkeit der GPU des eigenen Rechners trainieren zu können wurde *Google Colabe* verwendet. Dabei handelt es sich um ein kostenlosen, Cloudbasierten Service welcher eine VM (Virtual Machine) mit GPU in Form eines Jupyter Notebooks zur Verfügung stellt.

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 mit vortrainierten gewichten zur Feature Extraction.

Für die Arbeit wurde die Architekturen *Single Shot Detector* (SSD) und *Faster R-CNN* verwendet.

Beim SSD wurde das *MobilenetV2* sowie das *InceptionV2* als Basis CNN verwendet.

Bei Faster R-CNN nur *InceptionV2*.

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 welche einen schnelleren Zugriff auf die Daten ermöglicht.

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.

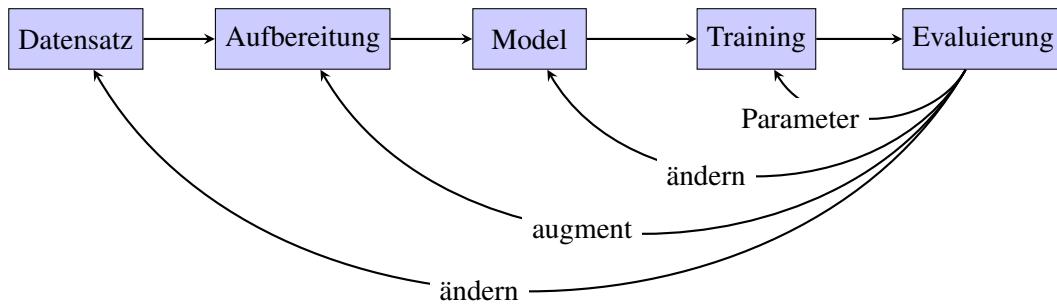


Abbildung 4.2: Trainingsworkflow

Die Ergebnisse werden im nächsten Kapitel (5) diskutiert.

4.3 Inferenz

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

4.3.1 OpenVino InferecneEngine

Die einzelnen Schritte, welche für die Inferenz eines Modells auf dem Neural Compute Stick mithilfe der InferecneEngine durchgeführt werden sind im folgenden dargestellt.

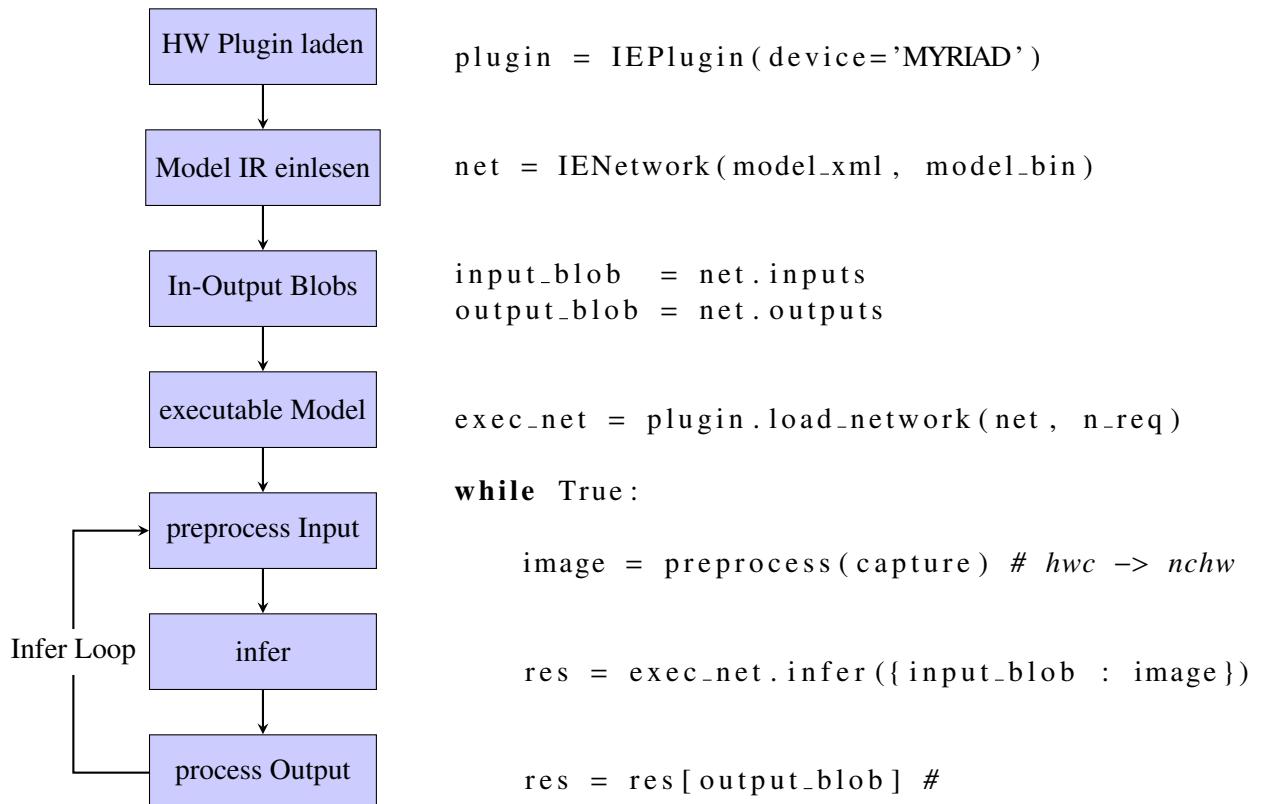


Abbildung 4.3

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

Die koordinaten können dann zur veranschaulichung wieder in das Ausgangsbild gezeichnet werden.

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*.

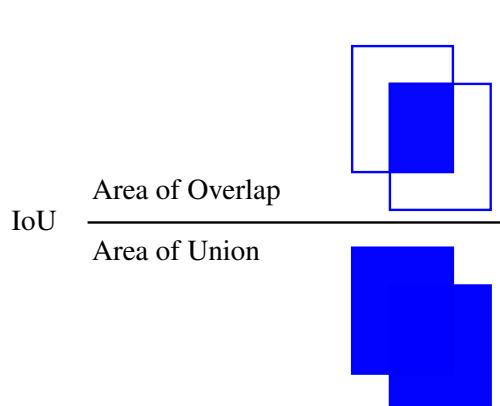


Abbildung 5.1: Intersection over Union

		Geschätzter Wert
Tatsächlicher Wert	p'	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 insgesammt 4 loss werte verwendet.

5.2 Vergleich der Modelle

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

5.2.1 Evaluierung/Validierung

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

SSD wurd 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-CN +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.

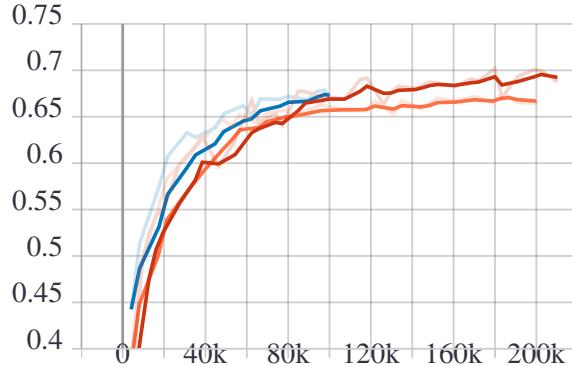


Abbildung 5.3: mAP

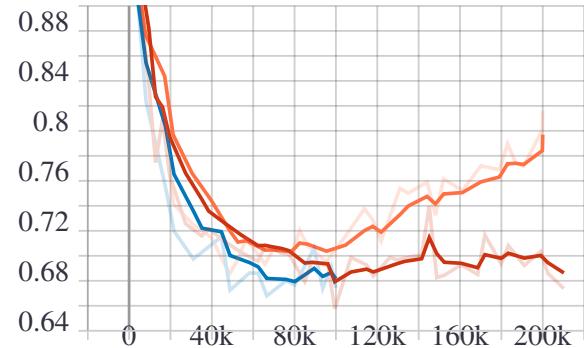


Abbildung 5.4: Loss

● 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*[17]
- inferenz auf eigene Bilder

Test Set (*OpenImages*)

Sowohl SSD als auch Faster R-CNN erkennen das Meiste, da Gr. Ähnlichkeit zu Trainingssatz.

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

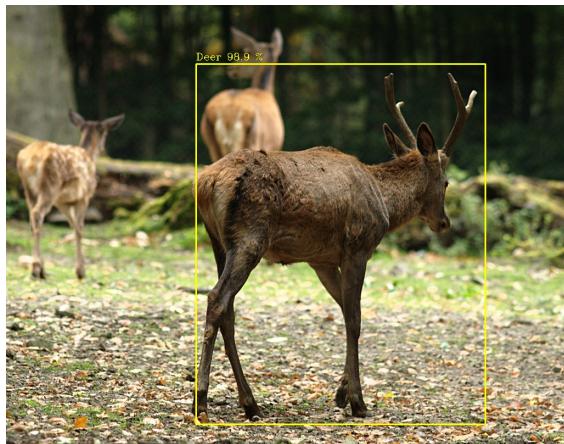


Abbildung 5.5: SSD

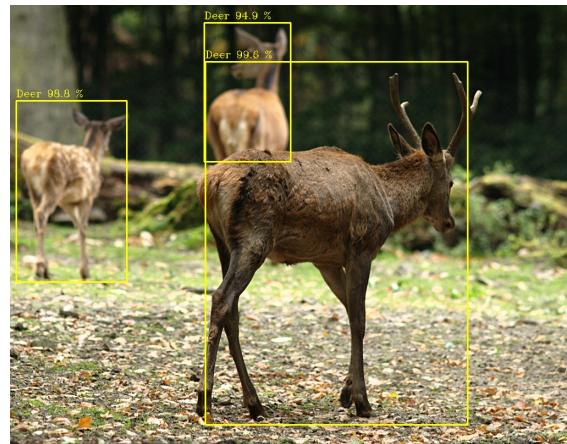


Abbildung 5.6: Faster R-CNN

Eigene Aufnahmen

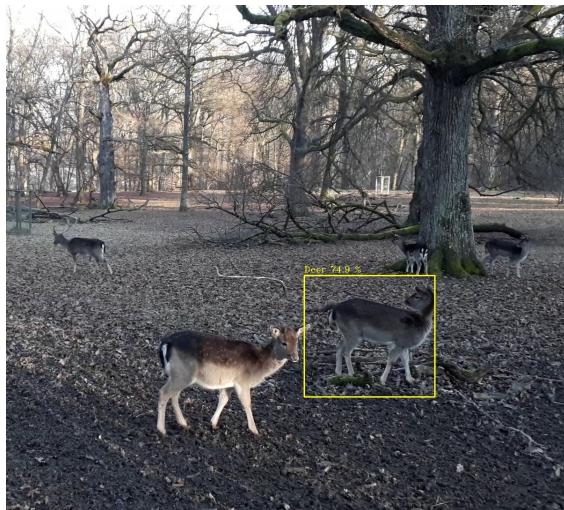


Abbildung 5.7: SSD Mobilnet

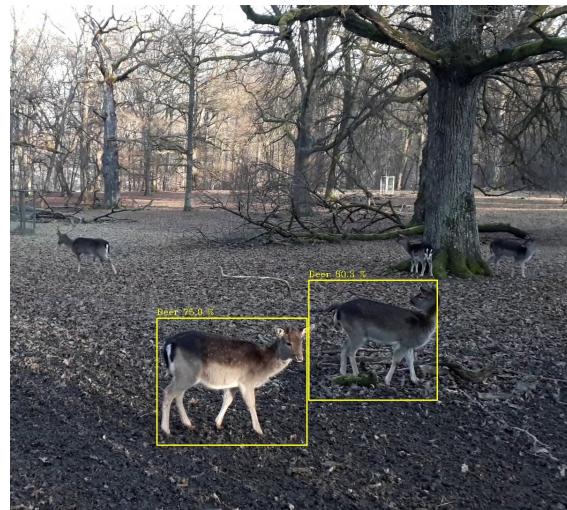


Abbildung 5.8: SSD Inception

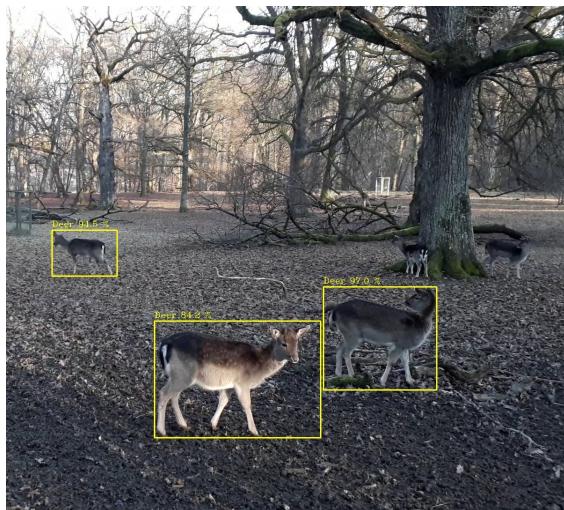


Abbildung 5.9: Faster R-CNN + Early Stopping

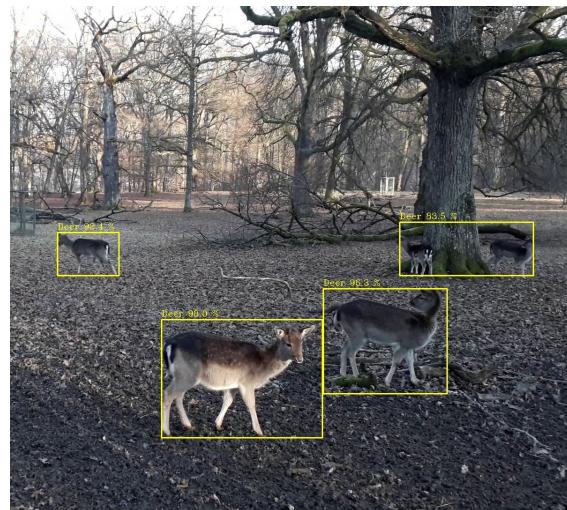


Abbildung 5.10: Faster R-CNN + Aug

iWildCam

Aufnahmen von ... enthält viele schlecht beleuchtete nacht Aufnahmen mit Infrarot Kamera, also ähnlich dem gewünschten Anwendungsfall. Auch hier erkennt das Faster R-CNN am beste, jedoch schneiden hier alle schlecht ab.

Daher wurde, wie im nächsten Abschnitt beschrieben, versuch das Faster R-CNN noch weiter zu optimieren.

5.3 Optimierungen: Faster R-CNN

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

Auch hier werden zunächst die Ergebnisse zunächst wieder anhand der Eval Metriken für den Validierungsdatensatz mithilfe Tensorboard dargestellt und anschließend mit Testdatensatz sowie weitere testinferez durchgeführt.

5.3.1 Verschiedene Augmentierungen

Der erste Anstz war es dieses mit variierendem Augmentierungsgrad für insgesamt mehr Trainingsdurchläufe (500 statt 200) zu trainieren.

Im vorherigen Abschnitt wurde für die Augmentierung der Daten je Bild zufällig aus einer Auswahl eine geometrische und eine Pixelbezogene augmentierungstechnik angewendet, sodass für jede Klass 3000 Samples generiert wurden.

Als variation wurde hier nun einmal die gleiche Augmentierungsstrategie für 4000 samples und einmal nur einer Augmentierung pro Bild für 3000 Samples angewendet.

Die Trainingskurven aus Tensorboard sind für mAP und Loss sind in den Plots 5.3.1 und 5.3.1 dargestellt.

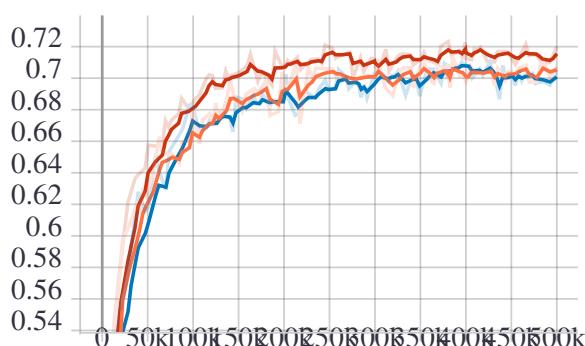


Abbildung 5.11: mAP

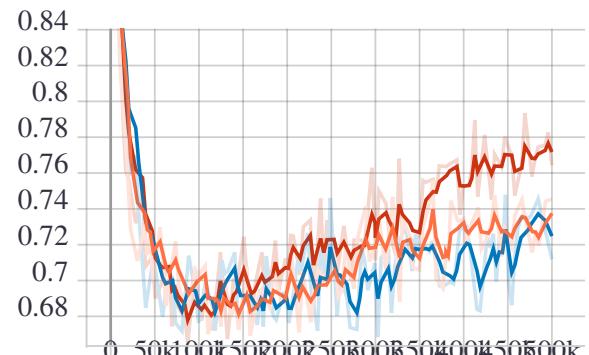


Abbildung 5.12: Loss

● 1 je bild ● 3000 samples ● 4000 samples

Es ist zu erkennen, dass sich durch stärkere augmentierung der Trainingsdaten der Loss reduzieren lässt, dadurch jedoch auch der mAP abnimmt.

5.3.2 weitere Regularisierungen

Um das trotz Augmentierung zu stande kommende Overfitting zu vermeiden, wurden nun zusätzlich die L2 Regularisierung angewendet. Diese soll wie in den Grundlagen (2.1.2) beschrieben, durch anfügen der aufsummierten Gewichte an die Loss Funktion, die Überanpassung eindämmen.

In der Konfigurationsdatei des Fater R-CNN kann dies durch setzen des bestimmten Parameters für sowohl die erste Stufe, das RPN (Region Proposal Network) als auch für die 2. Stufe, das Klassifizierungsnetz separat gesetzt werden.

Ebenso lassen sich die beiden Losskurven, aus denen sich der gesammt Loss zusammensetzt, seperat anzeigen, wie in Plot 5.15 und 5.16 zu erkennen ist.

Dadurch ließ sich feststellen, dass Overfitting nur beim Loss des RPNs stattfindet, weshalb der L2 Parameter auch nur für die erste Stufe das RPN eingestellt wurde. Dabei wurde der Fa $\lambda = 0.001$ verwendet.

Schaut man nun wieder die beiden Losskurven an, zeigt sich deutlich der Effekt, den die L2 Regularisierung auf den RPN Loss hat, was sich dann auch im Gesamtloss durch eine leichte Verbesserung bemerkbar machte.

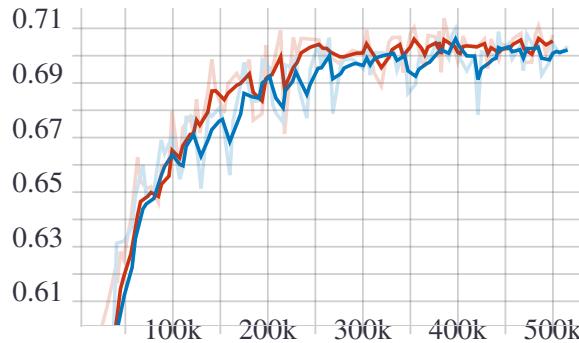


Abbildung 5.13: mAP

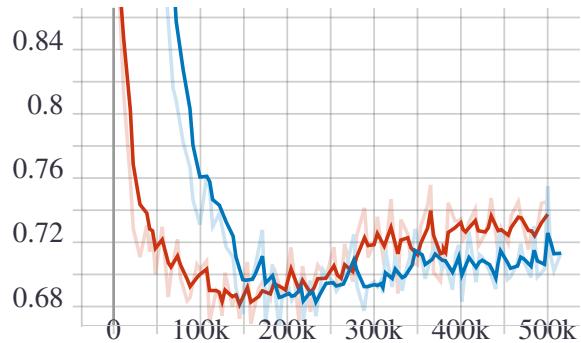


Abbildung 5.14: Total Loss

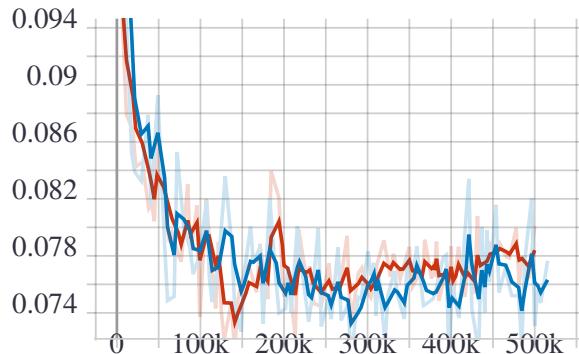


Abbildung 5.15: Classifier Loss

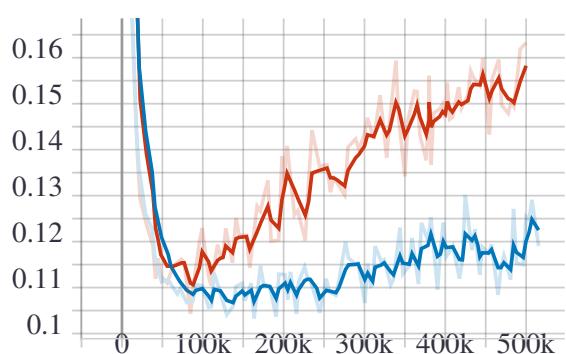


Abbildung 5.16: RPN Loss

● nur Augmentierung

● Augmentierung+L2 Regulierung

Dies wurde ebenso auf das Training mit weniger stark augmentiertem Datensatz angewendet. Weitere Einstellungen waren $\lambda = 0.002$ und das ebenfalls in 2.1.2 beschriebene Dropout. Die Ergebnisse sind noch einmal zusammenfassend in Tabelle ... dargestellt.

5.3.3 Test Inferenz

Auch hier wurde nun wieder zur besseren Auswertung der Ergebnisse die Inferenz Testweise auf die drei Datensätze Test Set des trainierten Open Images Datensatzes, eigen Aufnahmen, sowie des *iWildCam* Datensatzes 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



Abbildung 5.17: 3000 Samples Abbildung 5.18: 4000 Samples Abbildung 5.19: 50% Augment
Bei der L2 Regulariesierung konnte für das iWildCam keine Verbesserung festgestellt werden, teilweise wurden die Bilder besser erkannt, teilweise schlechter, so dass im Mittel keine Verbesserung entstand.
Für die eigenen Bilder waren die Ergebnisse sogar schlechter.

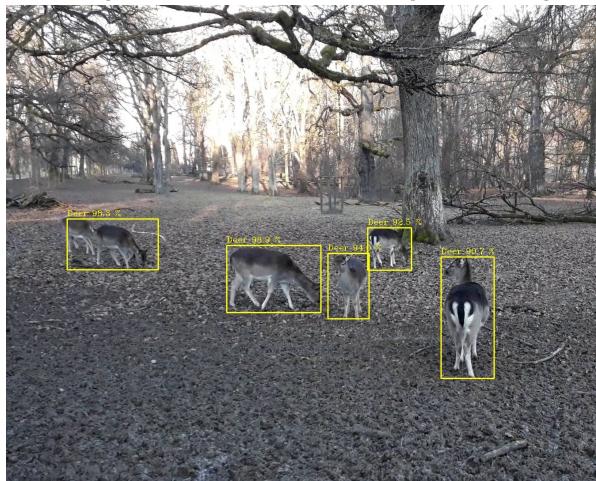


Abbildung 5.20

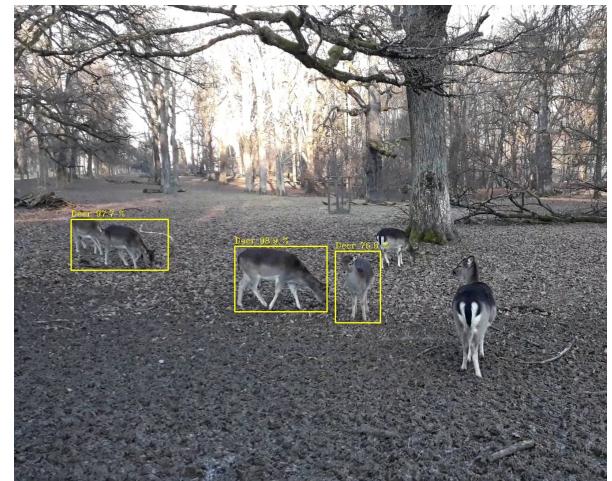


Abbildung 5.21

Daraus lässt sich schließen, dass die Art/Qualität/Varianz der Daten den größeren Einfluss auf das Ergebnis haben und mit den Hyperparametern wenn überhaupt

5.3.4 Graustufen

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

5.4 Inferenz zeit

Neben der Genauigkeit ist die Ausführungszeit welche ein Model für die Inferenz benötigt, ein weiteres Kriterium für die Auswahl des Modells gewesen. Je nach Anwendungsart muss diese in Echtzeit erfolgen oder nicht.

Ein Faktor von dem die Inferenzzeit abhängt ist die verwendeten Hardware sowie Library. Für den Neural Compute Stick können dafür OpenCV oder OpenVino verwendet werden, wobei mit OpenVino die Möglichkeit zur asynchronen Inferenzausführung sowie mehreren Inferenz Requests wodurch sich die Inferenzzeit optimieren lässt.

Der zweite Faktor wird durch die Komplexität des CNNs und der zur Objekterkennung verwendeten Architektur bestimmt.

Üblicherweise sind Komplexere Modelle wie Faster R-CNN genauer jedoch auch langsamer.

Im folgenden soll daher zunächst die Asynchrone Inferenz näher erläutert werden und anschließen ein Verfahren zur Untersuchung des Einflusses der Art der trainierten Modelle auf die Inferenz Zeit.

5.4.1 Asynchrone Inferenz

Bei einem synchronen Inferenzablauf kann immer nur entweder inferiert oder die Daten vor- und nachverarbeitet werden. Da die Inferenz jedoch auf dem Chip des NCS2 und nicht auf dem PC oder Raspberry läuft, kann diese auch ungehindert parallel dazu ablaufen, was in OpenVino mit der Asynchronen API erreicht werden kann.

Algorithm 1 Synchrone Inferenz

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

Algorithm 2 Asynchrone Inferenz

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

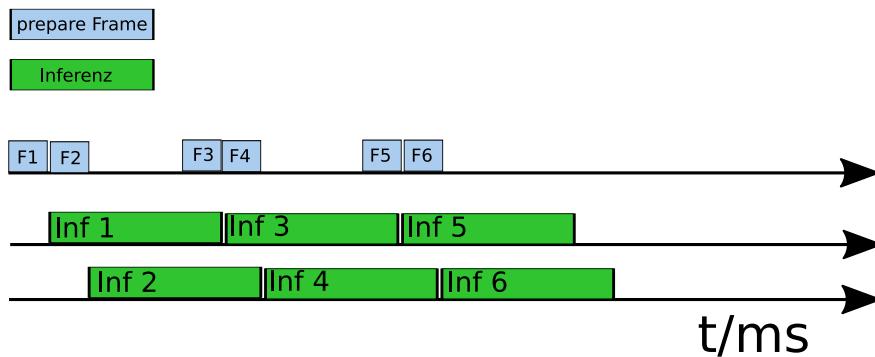


Abbildung 5.22: Asynchron und mehrere Inferenz Requests

5.4.2 Vergleich der Modelle

Mithilfe eines Python Scripts wurde nun die Inferenz für 100 Bilder ausgeführt und daraus die durchschnittliche Anzahl an Frames pro Sekunde berechnet (FPS).

Tabelle ?? zeigt die Ergebnisse für die Ausführung auf einem Raspberry Pi für die trainierten Modelle mit variierenden Inferenz Requests von einem bis vier.

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

Tabelle 5.2: Vergleich Inferenz Zeiten Modelle

Insbesondere zwischen den Objekt Detection Frameworks Faster R-CNN und SSD besteht ein großer Unterschied. Möchte man die Inferenz für eine Echtzeitanwendung kommen nur SSD in Frage. Des Weiteren ist festzustellen, dass eine Erhöhung der Inferenz Requests nur bis 3 sinnvoll ist.

Kapitel 6

Entwicklung der Anwendung

Dieses Kapitel beschreibt die Entwicklung der Anwendung, welche als autonomes System dass auf einem Raspberry Pi 4 läuft.

Dazu gehören die Einrichtung einer infrarotfähigen Kamera, die Implementierung der Inferenz mit OpeVino und die Verbindung zu einem Pc zur übertragung der Daten mit geeignetem Protokolls.

6.1 Aufbau/Hardware

Der Aufbau besteht aus einem Raspberry Pi, auf welche der Anwendungscode läuft, dem NCS2 für die Inferenz, welcher per USB mit dem Raspberry verbunden wird.

Desweiteren wurde ein Raspberry Pi Kamera Modul mit 5MP OV5647 Sensor der Marke Longrunner verwendet.

Dieses ermöglicht durch zu und abschalten eines Infrarot Filters vor die Linse zwischen Tag und Nachtsicht zu wechseln.

Das Schalten wird dabei über einen Helligkeitssensor automatisch geregelt. Im Infrarotmodus befindet sich der Infrarot Filter nicht vor der linse, wodurch auch die etwas längeren (850nm) Elektromagnetischen Wellen als die des Sichtbaren Lichts aufgenommen werden können.

Durch zwei Infrarot LEDs des gleichen Spektrums, können so auch in der Dunkelheit Bilder aufgenommen werden, ohne Sichtbare Beläuchtung, was Tiere verscheuchen würde.

Das Kamera Modul wird über die CSI (Camera Serial Interface) Schnittstelle mit dem Raspberry Pi verbunden.

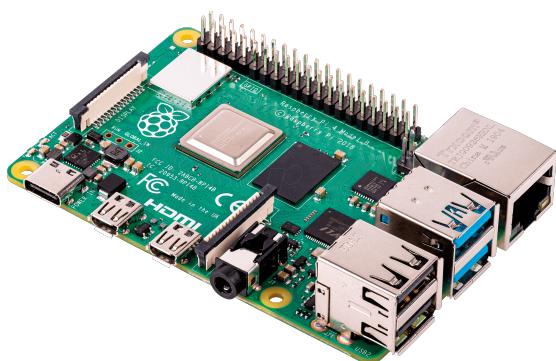


Abbildung 6.1: Raspberry Pi 4

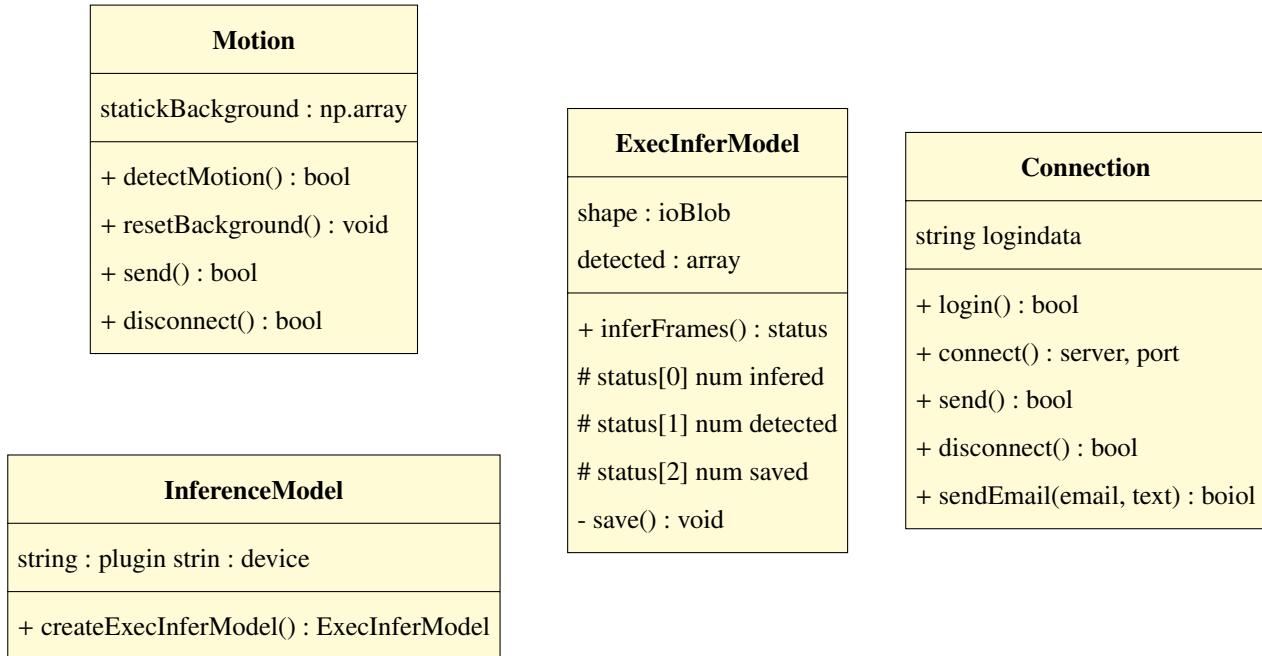


Abbildung 6.2: Longrunner Kamera Modul

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

6.2 Implementierung/Software

Die Implementierung der Anwendung wurde in Python vorgenommen und besteht aus den drei Scripten `main.py`, `detection.py` und `connection.py`. Welche Folgend dargestellte Klassen implementieren.



Dabei führt die Main Anwendung eine Dauerschleife aus, in der die Frames der Kamera erhalten werden.

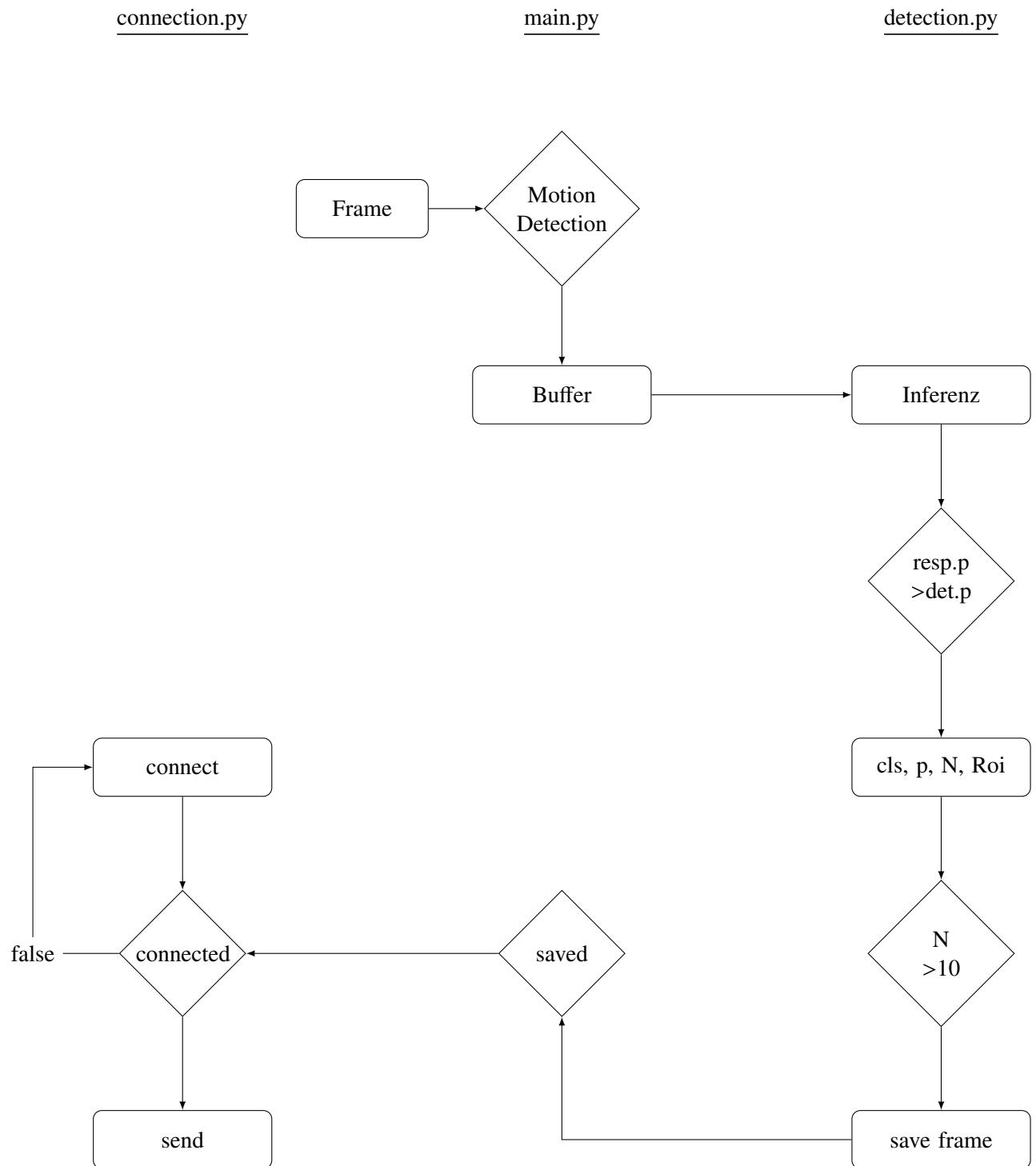
Die inferenz wird über das detection Script, welches die InferenceEngine implementier ausgeführt. Das senden erkannter Ergebnisse erfolgt dann mithilfe des Detection Scripts.

Um trotz der langsamen Inferenz Zeit das Faster R-CNN verwenden zu können, galt es die Anwendung und die Zeitliche ausführungen der Inferenz so zu gestalten, das möglichst alle Frames, in welchen Tiere zu beobachten sein können, auch inferiert werden.

Dafür wurde die Annahme gemacht, dass zur Laufzeit der Anwendung häufig Zeiten sind in denen nichts zu inferieren ist, was durch eine Bewegungserkennung herausgefunden werden konnte.

Desweiteren wurde die Inferenz so implementiert, dass sie im gegensatz zur üblichen verwendung an keiner stelle Blockiert, wodurch durch zwischen Speichern von Bildern in denen Bewegung erkannt wurde trotz langsamerer Inferenzzeit alle Frames inferiert werden können.

Folgendes Diagramm zeigt schmematsch den groben Ablauf davon.



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.

```
def infer_frames (Buffer , threshold):
    for idx , inferRequest in all inferRequests:
        status = inferRequest.wait(0) # nicht blockierend
        if status not ready:
            continue

        if idx in currentFrames:
            results = inferRequest.output
            frame = currentFrames [idx]

        if Buffer not empty:
            currentFrames [idx] = Buffer.pop()
            infer_frame = preprocess (currentFrames [idx])
            inferRequest.async_infer (infer_frame)

        if results or frame is None:
            continue

        for obj in all results:
            Class , Roi , Proba <- obj
            if Proba < threshhold:
                continue

            coords <- Roi , frame . shape

            inferred_frame = draw_rect (frame , coords)

            if proba > detectedObjects . proba
                replace detectedObjects

        if number of detections > x:
            send (frame)
```

Algorithm 3 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

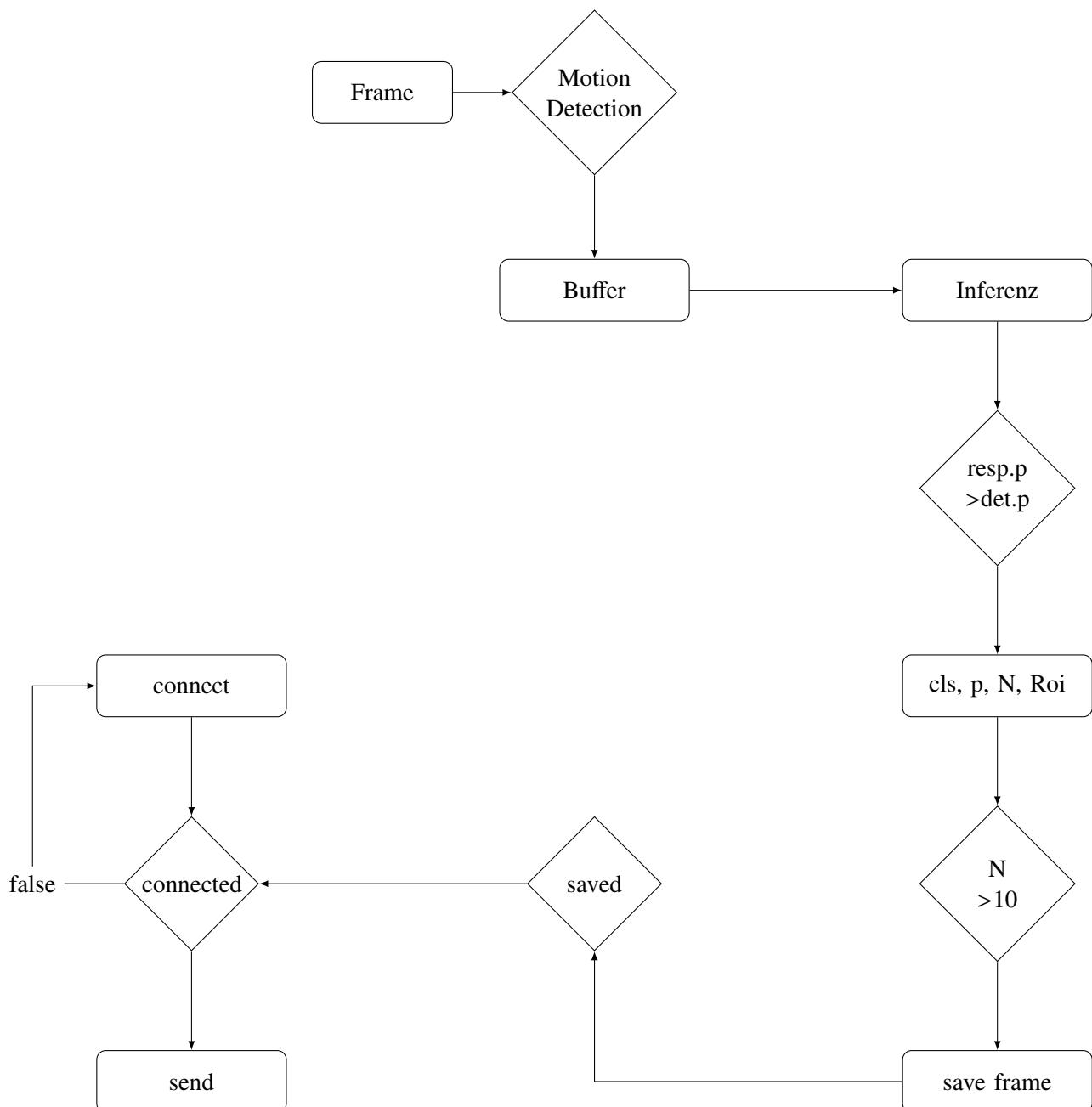
- remote Proxy verbinung über SSH
- mit SCP Protokol send

6.2.1 all

connection.py

main.py

detection.py



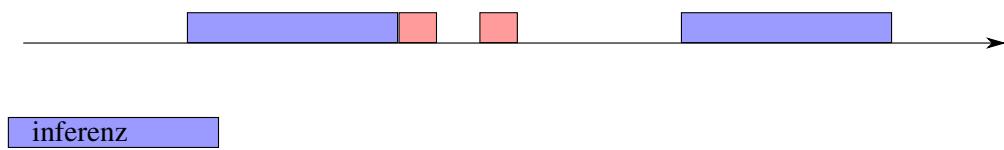


Abbildung 6.3

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. Mal- loci, 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