

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 Convolutional Neural Networks	10
2.3 Neural Compute Stick 2	13
3 Realisierung Objekt Erkennung	17
3.1 Datensatz	17
3.2 Object Detection Modelle	19
3.3 Training	20
3.4 Inferenz	21
4 Evaluierung	25
4.1 Evaluierungs Metriken	25
4.2 Vergleich der Modelle	27
4.3 Optimierungen: Faster R-CNN	30
4.4 Inferenz zeit	34
5 Entwicklung der Anwendung	37
5.1 Hardware	37
5.2 Software	38
6 Test und Validierung	43
7 Zusammenfassung und Ausblick	45
A Beispiel für ein Kapitel im Anhang	49
A.1 Bsp für ein Abschnitt im Anhang	49

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.

aus 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 die Inferenzzeit einhergeht war dies ein 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.

Kapitel 2

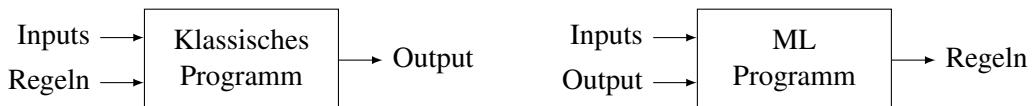
Grundlagen

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.



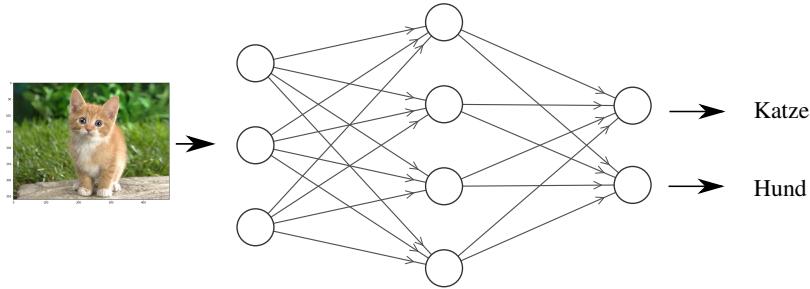
Das ableiten der Regeln erfolgt beim Machine Learning in einem iterativen Prozess, welcher als Training bezeichnet wird. Dabei werden die Zusammenhänge zwischen In- und Output Daten als mathematische Funktion betrachtet, welche numerisch angenähert wird.

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.

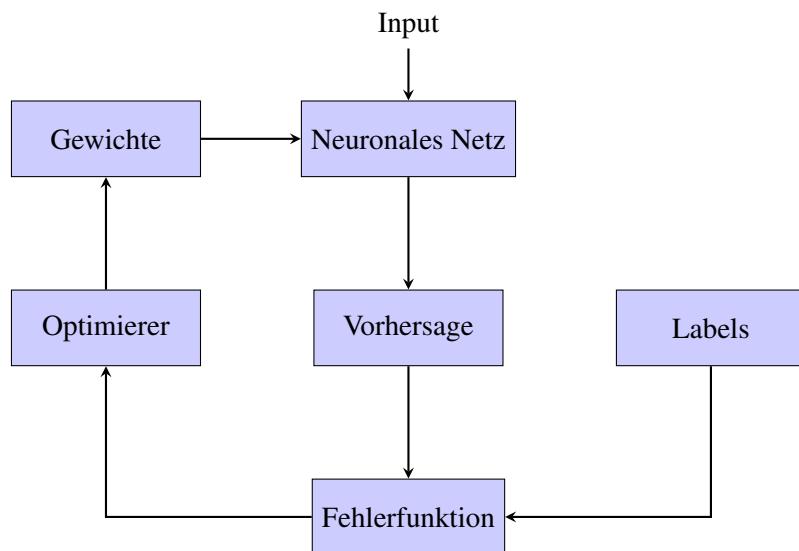


Abbildung 2.1: Trainingsablauf NN

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 ??

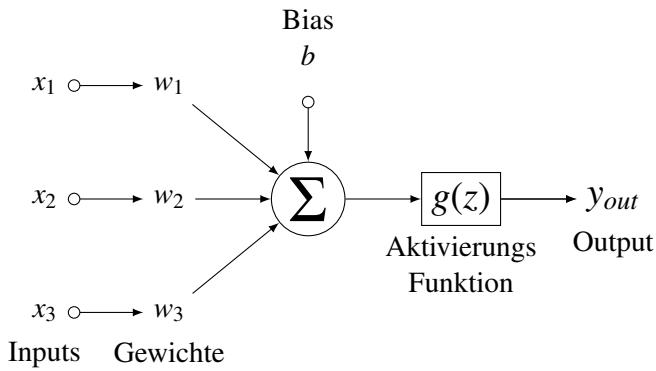


Abbildung 2.2: Einzelnes Perzeptron

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.

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

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

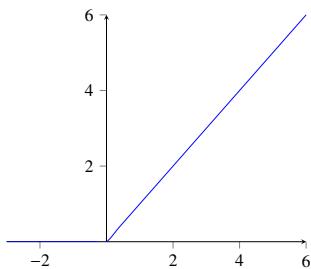


Abbildung 2.3: ReLU Funktion

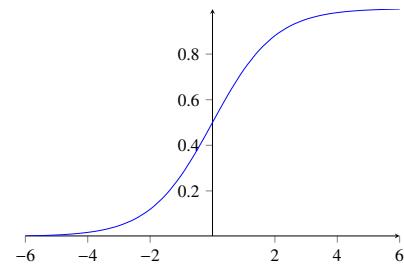


Abbildung 2.4: Sigmoid Funktion

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.

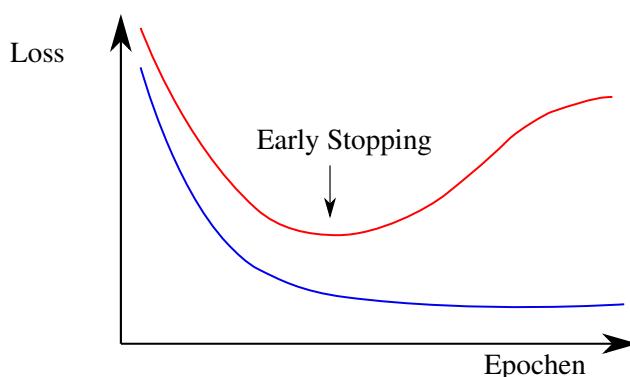


Abbildung 2.5: Overfitting, anhand der Losskurven

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.

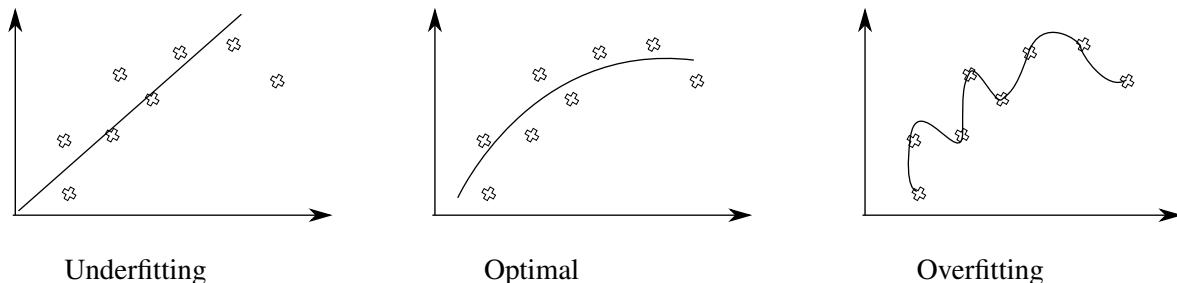


Abbildung 2.6

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.

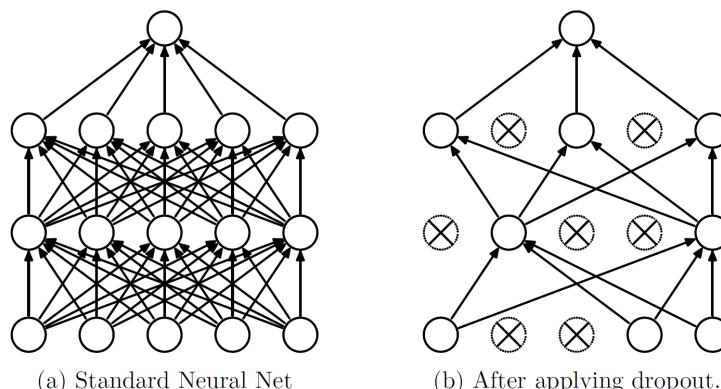


Abbildung 2.7: Dropout, Quelle: [1]

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.

2.1.3 Machine Learning Frameworks

Machine Learning Algorithmen beinhalten eine Vielzahl an komplexen Berechnungsschritten und Parametern. Um diese nicht jedesmal von Grund auf neu implementieren zu müssen, bieten Frameworks eine vereinfachte Möglichkeit die Modelle zu konstruieren.

Einige der bekannten Open Source Frameworks 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.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) erweitern die in Abschnitt 2.1.1 beschriebenen *Feed Forward Netze* um zusätzliche Schichten, die vor der eigentlichen Klassifizierung ausgeführt werden und Merkmale aus den Input Daten herausextrahieren.

Diese Schichten enthalten den Input als zwei dimensionale Matrix und führen darauf eine mathematische Faltungsoperation aus.

CNNs kommen größtenteils in der Bilderkennung zum Einsatz, weitere Anwendungsgebiete sind die Spracherkennung.

Anstatt dass alle Neuronen zweier benachbarter Schichten durch gewichtete Parameter miteinander verbunden sind, stellen kleinere, sogenannte Filter Matrizen, die Parameter dar.

Diese werden zeilenweise über das Input Bild geschoben, wobei an jeder Stelle eine mathematische Faltung mit dem überlappten Bereich des Input Bildes durchgeführt wird. Abbildung 2.2 und 2.2 veranschaulichen diesen Vorgang.

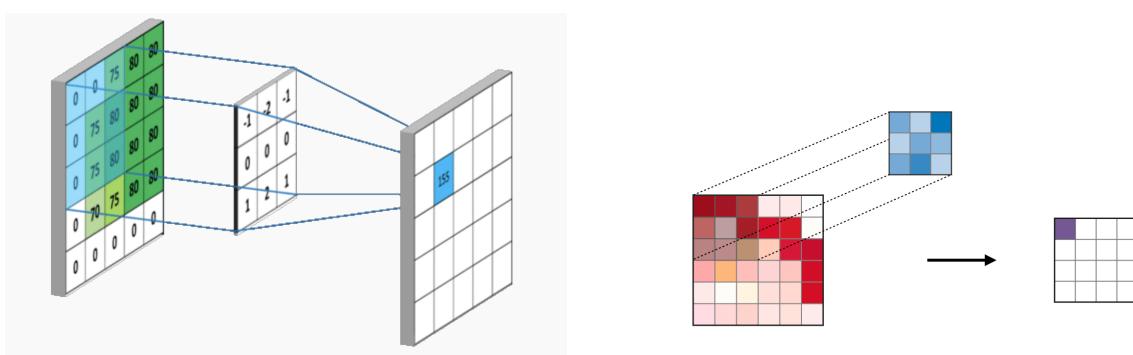


Abbildung 2.8: Faltung, [2]

Jedes Ergebnis einer Faltung ergibt einen Wert für die nächste Schicht, die dadurch ein Korrelationsverhältnis zwischen Filter Matrix und Input Bild erhält.

So werden in den Filtern definierte Muster, wie beispielsweise vertikale Linien, aus dem Inputbild herausgefiltert und in den Folgeschichten, als Feature Maps abgebildet.

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

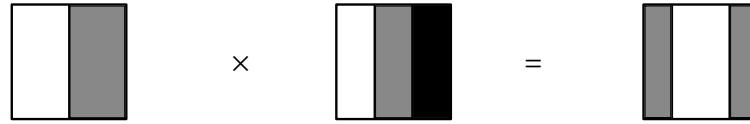


Abbildung 2.9

In Gleichung 2.7 ist die Faltung eines Input Bildes mit einem Filter zur Extrahierung vertikaler Linien dargestellt. Da pro Zeile vier Faltungen angewendet werden, entsteht eine 4×4 Matrix. Soll die Ausgangsgröße beibehalten werden, kann *zero padding* verwendet werden.

Durch die Faltung entsteht so eine räumliche Invarianz für das zu erkennende Objekt im Input Bild.

Den Convolutional Layern folgen meist Pooling Layer, zur Reduktion der Dimension und eine ReLU Aktivierungsfunktion.

Beim Pooling wird eine bestimmte Anzahl an Werten zusammengefasst, indem entweder das Maximum oder der Mittelwert verwendet werden.

Durch hintereinanderschaltung mehrerer Convolutional Blöcke können immer komplexere Muster aus dem Input Bild herausgezogen werden.

Die Features des letzten Convolutional Layers werden dann einem *Fully Connected Layer* zur Klassifikation übergeben.

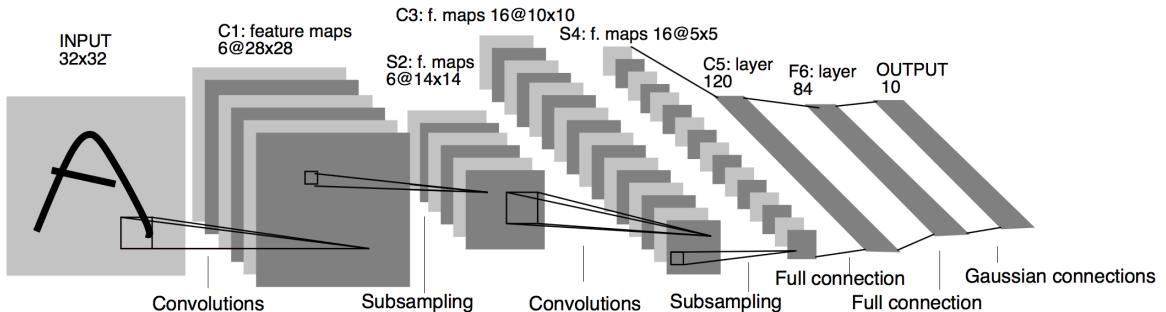


Abbildung 2.10: Faltung, [3]

Ein wesentlicher Vorteil gegenüber einem reinen *Feed Forward Network* ist der, dass durch die gemeinsame Nutzung der Parameter in den Filtern ein geringerer Rechenaufwand entsteht.

Die Werte der Filter Matrizen, welche die zu findenden Muster bilden, werden über die Backpropagation eingelernt.

Da diese, insbesondere in den vorderen Convolutional Layern für die meisten Klassen sehr ähnlich sind, können vortrainierte Filter verwendet werden.

Durch das sogenannte *Transfer Learning* müssen die Gewichte dann nur noch leicht für den eigenen Datensatz angepasst werden.

2.2.1 Architkturen

Nachdem 1998 das erste CNN (Abbildung ??) von Yann LeCun in [3] vorgestellt wurde, gab es eine Vielzahl an Weiterentwicklungen, welche genauere und effizientere Netzwerke hervorbrachten. Gemessen und verglichen werden die Ergebnisse häufig an der *Large Scale Visual Recognition Challenge (ILSVRC)* [4]. Bekannte Modelle, welche die Challenge in den letzten Jahren gewinnen konnten sind wie in [5] dargestellt.

- **AlexNet**, von Alex Krizhevsky (2012) mit einer ähnlichen Struktur wie das LeNet, jedoch Tiefer und mit mehreren Convolutional Layer übereinander.
- **ZF Net**, von Matthew Zeiler and Rob Fergus (2013), welches das AlexNet durch Vergrößerung der mittleren Convolutional Layer und Verkleinerung von Filter und Stride in den ersten Layern optimierte.
- **VGGNet** von Karen Simonyan and Andrew Zisserman (2014), zeigte das ein Tieferes Netz (16 bis 19 Conv Layer) reduzierter Filter Größe (3×3) bessere Ergebnisse erzielt.
- **GoogleLeNet**, auch bekannt als Inception, von Szegedy et al (2014), konnte mit den Inception Modulen, welche im folgenden genauer erläutert werden, die Zahl der Parameter deutlich verringern.
- **ResNet** von Kaiming He et al (2015), enthalten *Residual Blocks*, in denen auf das Ergebnis zusätzlich der Input Wert addiert wird.

GoogleLeNet (Inception)

Eine Möglichkeit die Genauigkeit eines Deep Learning Modell zu verbessern, ist es mehr Schichten und/oder eine größere Anzahl an Neuronen pro Schicht zu verwenden.

Der Nachteil dabei ist der größere Rechenaufwand, sowie die höhere Chance des Overfittings.

Das in [6] vorgestellte GoogleLeNet hat mit den Inception Modulen einen neuen, effizienteren Ansatz gefunden die Komplexität und damit die Genauigkeit zu erhöhen.

Die Module bestehen aus parallel ausgeführten Convolutional Layern unterschiedlicher Kernel Größe ($1 \times 1, 3 \times 3, 5 \times 5$)

Diesen werden zu Dimensionsreduktion häufig 1 Kernel vorgeschaltet.

Um die Effizienz weiter zu Steigern wurden in der zweiten Variante des GoogleLeNet [7] neben anderen Verbesserungen, die 5×5 Filter durch $2 \times 3 \times 3$ Filter ersetzt.

Am Ende eines Inception Moduls werden diese in einem *DepthConcat* Layer wieder zusammengeführt.

Dadurch können Features auf unterschiedlicher Skalen besser zusammengesetzt und erkannt werden.

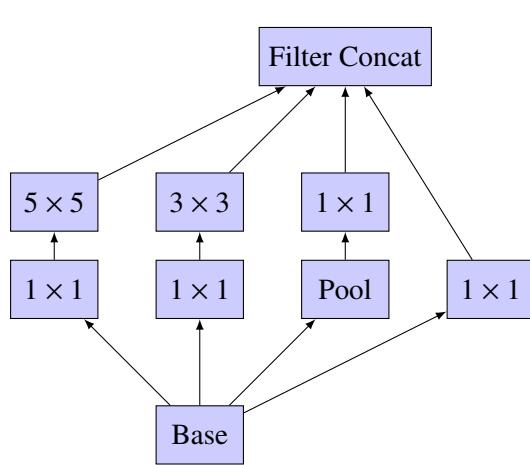


Abbildung 2.11: Inception Module V1

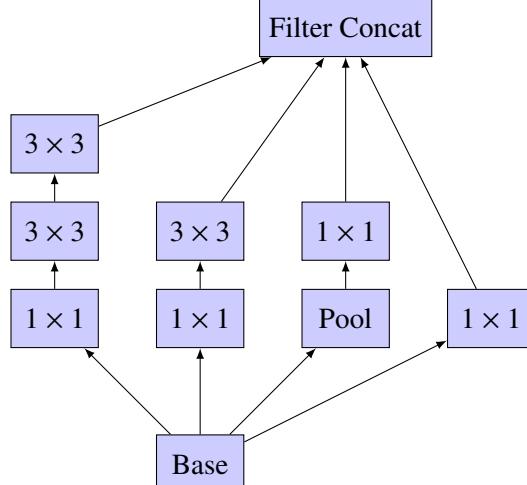


Abbildung 2.12: Inception Module V2

Mobilenet

[8]

Geringere Komplexität für Mobile oder embedded geräte, indem die normale Faltungsoperation mit der *Depthwise Separable Convolutions* ersetzt wird. Diese besteht aus einer *Depthwise Convolutions*, welche die Faltung auf die drei Farbchannel separat anwendet und einer 1times1 *pointwise convolution*, welche die input channel kombiniert.

In der zweiten version [9] enthalten die Hauptbestandteile eine 1times1 conv mit relu, die 3time3 depthwise gefolgt von einer weitere 1time1 ohne aktivierungsfunktion.
eine *residual connection* unterstützt den gradientenfluss.

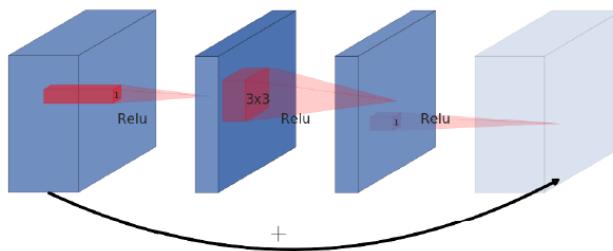


Abbildung 2.13: MobilenetV2

2.2.2 Objekterkennung

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, dass dem Modell neben den Klassenlabels auch die Koordinaten der Bounding Boxen für das training mit übergeben werden. Bounding Boxen umrahmen wie in Abbildung ?? das Objekt auf dem Bild und werden meist über Regressionsverfahren angenähert.

Das gesammt Framework zur Objekterkennung verwendet dabei ein Basis CNN z.B. eines der in 2.2.1 dargestellen.

Das finden von Regionen, welche Objekte enthalten, kann dabei in einem seperaten, Netzwerk, welches Selective Search oder Vorschlagsgenerierer verwendet, stattfinden.



Abbildung 2.14: Unterschied: Classification - Detection

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. [10]



Abbildung 2.15: nesc2

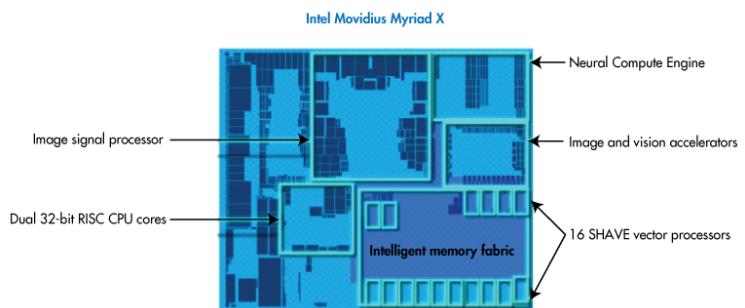


Abbildung 2.16: myriad

2.3.1 OpenVino Toolkit

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

Dieses ist eine Plattform zur Otimierung 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.

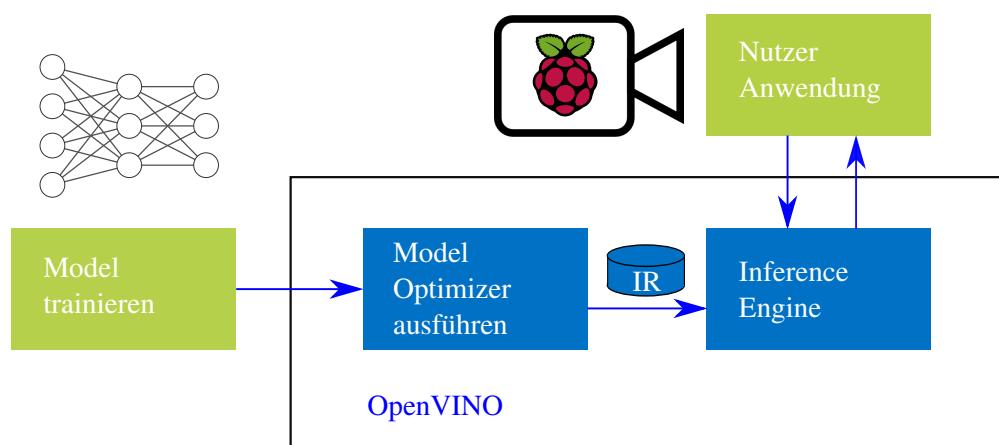


Abbildung 2.17: OpenVINO Workflow

Kapitel 3

Realisierung Objekt Erkennung

3.1 Datensatz

Um ein Deep Learning Modell richtig trainieren zu können, wird eine große Menge an gelabelten Trainingsdaten benötigt. Handelt es sich dabei um ein, wie in Abschnitt ?? beschriebenen Modell zur Objekterkennung müssen die Labels neben der Klasse, auch die Koordinaten, die angeben wo sich das Objekt im Bild befindet. Indem die, sogenannten Bounding Boxes, mit trainiert werden, kann das Modell auch die Lokalisierung der Objekte vorhersagen. Die Trainingsdaten können entweder selber erstellt, oder aus frei zugänglichen Datensätzen wie z.B. *ImageNet*, *COCO*, oder *OpenImages* aus dem Internet heruntergeladen werden.

Für die Arbeit wurden aus dem Open Source Dataset *OpenImages* von Google [11], welches 600 gelabelte Klassen enthält, die 9 Klassen *Brown bear*, *Deer*, *Fox*, *Goat*, *Hedgehog*, *Owl*, *Rabbit*, *Raccoon* und *Squirrel* heruntergeladen und für das Training verwendet. Für die Evaluierung des Trainings wurde der Datensatz in ein Trainings, einen Validierung und einen Testset aufgeteilt, mit dem Verhältnis 80%, 10%, 10%. Je Klasse variierte die Anzahl an Bildern zwischen 200 und 2000, wodurch die in Abbildung 3.1 dargestellte unausgeglichenheit der Klassen zustande kam. Um diese auszugleichen, wurden die Datei, wie im nächsten Abschnitt genauer erklärt wird, so augmentiert, dass je Klasse 3000 Bilder vorhanden waren, was zu einer in Abbildung 3.2 dargestellten Verteilung der Klassen führte.

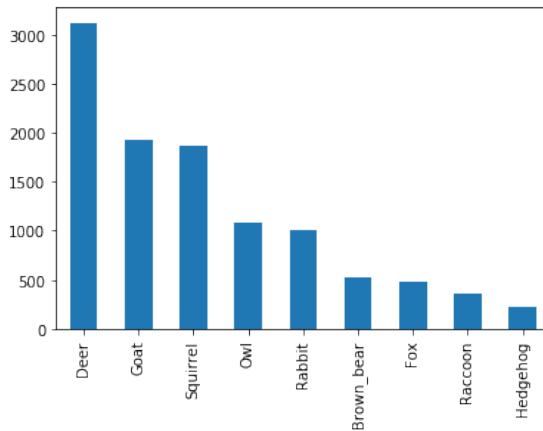


Abbildung 3.1: ohne aug

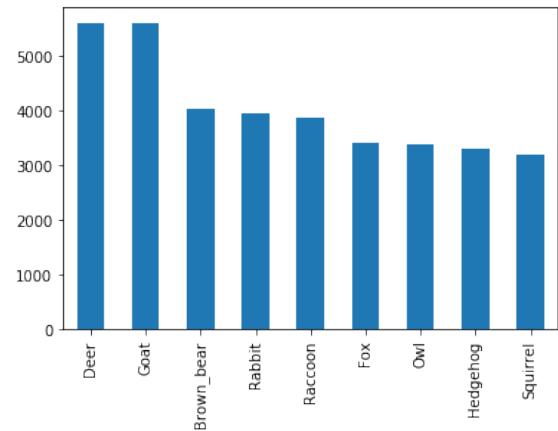


Abbildung 3.2: mit aug

Der Grund dafür, dass in den Histogrammen mehr Klassen, als es tatsächlich Bilder waren angezeigt werden, kommt daher, dass sich häufig mehrere Tiere der selben Klasse in einem Bild befinden. Ein weiterer Grund für die Augmentierung der Daten war die vorsorgliche Maßnahme gegen Overfitting.

3.1.1 Augmentierung

Das augmentieren von Bilddaten für Deep Learning, ist eine effektive Technik, durch künstliches generieren neuer Daten aus den vorhandenen, um Overfitting zu verhindern und Inbalancen der Klassen untereinander auszugleichen. Die vermehrung der Daten erfolgt dabei in dem die Bilder geometrischen Transformationen oder manipulationen der Pixelwerte unterzogen werden.

Die Augmentierung des OpenImages Datensatzes wurde mithilfe eines Python Scripts, welches die Library *imgaug* [12] verwendet, durchgeführt. Dabei wurde je zu augmentierendem Bild eine geometrische und eine Pixel Transformation angewendet. Diese wurden zufällig aus einer Auswahl an Augmentern, angewendet.

Folgend ist ein Codeausschnitt des Python Scripts dargestellt, welcher die verwendeten Augmentierungs-techniken zeigt.

```
import imgaug.augmenters as iaa

color_augmenters = [
    iaa.Dropout(p=(0, 0.1)),
    iaa.CoarseDropout((0.01, 0.05), size_percent=0.1),
    iaa.Multiply((0.5, 1.3), per_channel=(0.2)),
    iaa.GaussianBlur(sigma=(0, 5)),
    iaa.AdditiveGaussianNoise(scale=((0, 0.2*255))),
    iaa.ContrastNormalization((0.5, 1.5)),
    iaa.Grayscale(alpha=((0.1, 1))),
    iaa.ElasticTransformation(alpha=(0, 5.0), sigma=0.25),
    iaa.PerspectiveTransform(scale=(0.15)),
    iaa.MultiplyHueAndSaturation((0.7))
]

geometric_augmenters = [
    iaa.Affine(scale=((0.6, 1.2))),
    iaa.Affine(translate_percent=(-0.3, 0.3)),
    iaa.Affine(shear=(-25, 25)),
    iaa.Affine(translate_percent={"x": (-0.3, 0.3), "y": (-0.2, 0.2)}),
    iaa.Fliplr(1),
    iaa.Affine(scale={"x": (0.6, 1.4), "y": (0.6, 1.4)})
]
```

In Abbildung ?? sind beispielhaft 9 dieser zufällig augmentierten Bilder der Klasse Fuchs dargestellt.

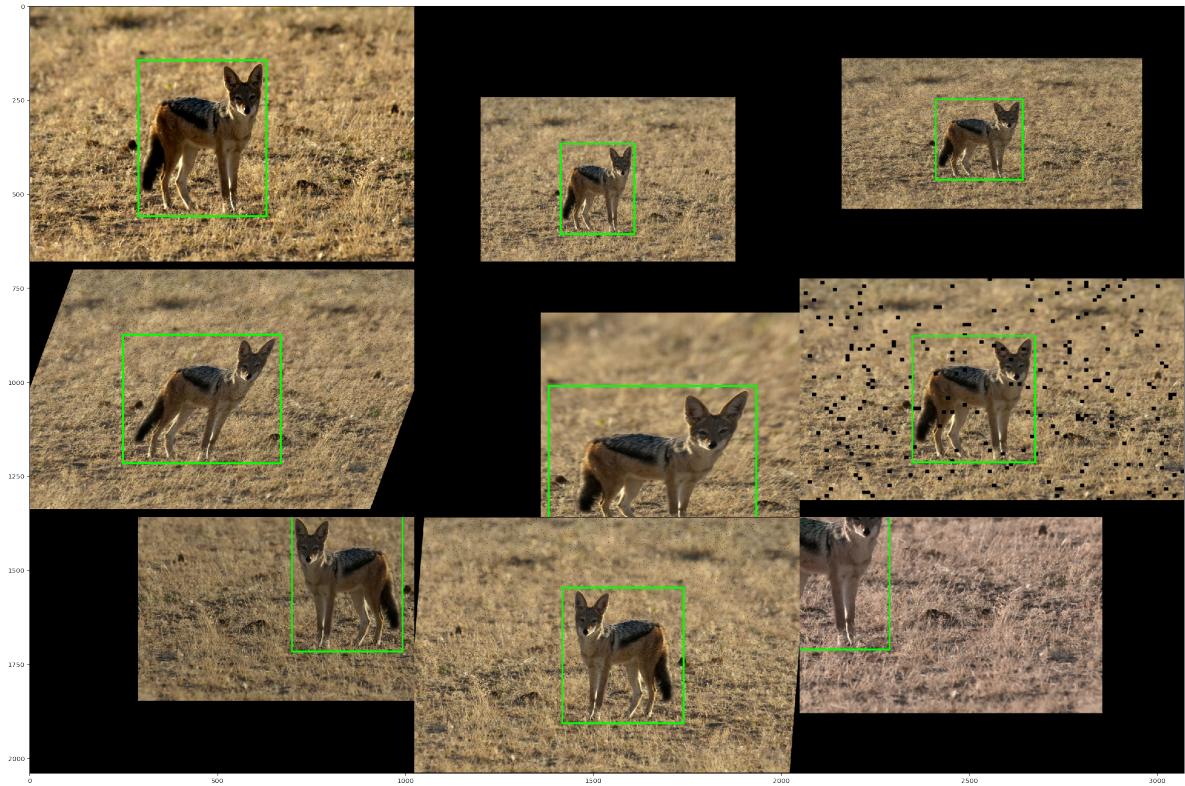


Abbildung 3.3: Anwendung von Augmentierungstechniken

3.2 Object Detection Modelle

Object Detection Modelle lassen sich, wie in [13] beschrieben wird, in Einstufige- und Zweistufige Detektoren einteilen.

Zweistufige Detektoren generieren in der ersten Stufe, eine Auswahl an räumlichen Vorschlägen, in welchen Objekte enthalten sein können. In der zweiten Stufe, werden die Vorschläge zur Feature Extraction einem CNN übergeben, welchem ein Netz zur Klassifizierung (zb SVM) und eines zur annäherung der Bounding Box Koordinaten (Regressor) folgen.

Einstufige Verfahren verwenden kein separates Netz zur Vorschlagsgenerierung. Stattdessen wird das gesamte Bild, gitterartig unterteilt, als potentielle Regionen für Objekte betrachtet und hinsichtlich vorhanden sein einer Klasse je Region klassifiziert. Dadurch sind diese zwar wesentlich schneller, aber auch ungenauer.

Für die Thesis wurden beide Ansätze verwendet und für die Umsetzbarkeit als Wildtierdetektor mit dem NCS2 hinsichtlich Genauigkeit und Inferenzzeit verglichen.

Die dafür verwendeten Modelle werden im folgenden näher erläutert.

Faster R-CNN

Das Faster R-CNN [14], dargestellt in Abbildung ??, verwendet ein zweistufiges Verfahren zu Objekterkennung.

Die Vorschlagsgenerierung erfolgt mithilfe eines Region Proposial Networks (RPN), welches auf einem *fully convolutional network* basiert. Über die daraus generierten Feature Maps werden im Sliding-Window Verfahren vordefinierte Anker Boxen konvolviert. Der daraus resultierende Feature Vektor wird einem binären klassifikator (*cls layer*), welcher angibt ob sich ein Objekt in dem Vorschlag befindet, sowie einem Bounding Regresor (*reg layer*) übergeben.

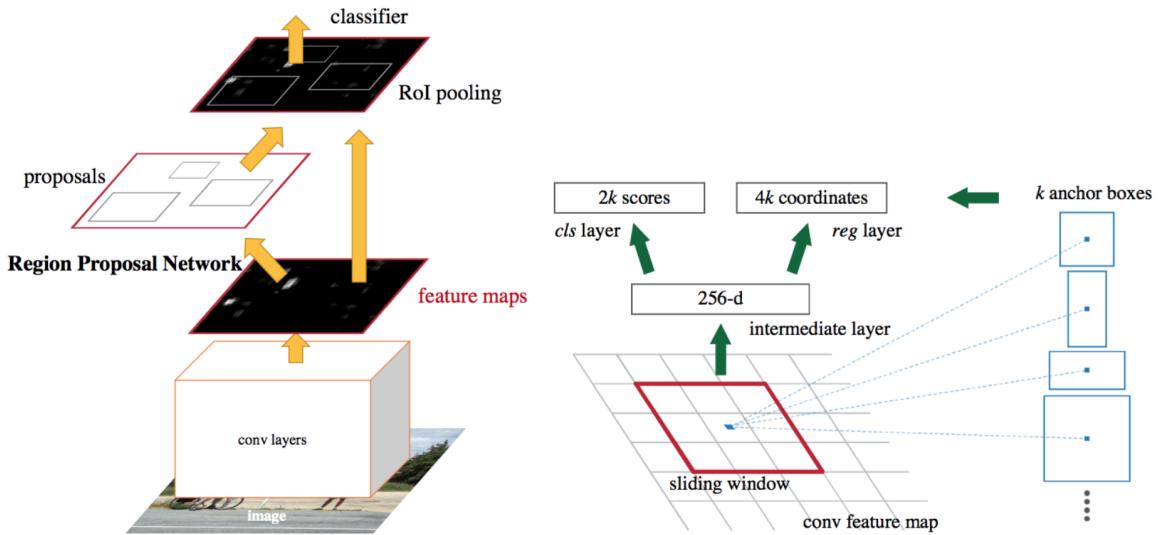


Abbildung 3.4: Faster R-CNN Architektur, Quelle: [15]

SSD: Single Shot MultiBox Detector

Der Single Shot MultiBox Detector (SSD) [16] ist ein einstufiges Verfahren zur Objekterkennung, welches das Input Bild gitterartig unterteilt. In jeder Zelle des Gitters werden default Anker Boxen unterschiedlicher Skalierungen definiert.

Indem an das Backbone CNN weitere *Extra Feature Layer* verschiedener Größe angehängt werden, kann dieses für jede default Box eine Klassifizierung, in Form eines *confidence scores* sowie eine Lokalisierung, in Form eines Offsets zur default Box vornehmen.

Diese werden zur finalen Detektion einem *non-maximum suppression* Layer [17] übergeben, welcher alle zu einer Klasse gehörenden Boxen zusammenführt.

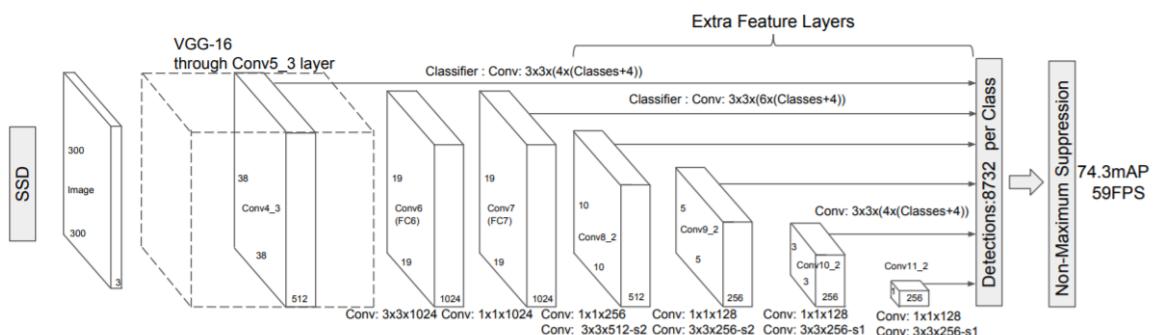


Abbildung 3.5: SSD, quelle: [18]

3.3 Training

Das Training der Deep Learning Modelle erfolgte mithilfe des Frameworks Tensorflow, welches auch von Open Vino für den Neural Compute Stick unterstützt wird. Dabei wurde eine speziell für Object Detection entwickelte API von Tensorflow verwendet.

Um unabhängig von der Leistungsfähigkeit der GPU des eigenen Rechners zu sein, wurde das Training in der Cloudbasierten Virtual Machine Google Colab [19] durchgeführt, welche kostenlos eine für Deep Learning geeignete GPU zur Verfügung stellt.

3.3.1 Tensorflow Object Detection Api

Die Tensorflow Object Detection Api ist unter den Research Modellen [20] des offiziellen Tensorflow Repository auf GitHub zu finden und enthält implementierungen einiger gängiger Object Detectin Modelle mit vortrainierten gewichten zur Feature Extraction. Für die Arbeit wurde die im Abschnitt ?? erläuterten 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*, da hierfür das MobilenetV2 nicht verfügbar war. Weitere Basis CNN sind unter Anderem das Resudial Network, welche jedoch sehr Rechenaufwendig und nicht für den Neural Compute Stick kompatibel ist.

Die Festlegung der Parameter für die Modelle kann dabei in einer Konfigurationsdatei vorgenommen werden. 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.

Nach herunterladen des Modell und festlegen einiger Parameter was in einem Jupyter Notebook der Colab VM erfolgte, konnte das Training gestartet werden.

Das erfolgte über einen Command dem die TfRecord Files als Argument übergeben wurden. Die Trainierten Gewichte wurden während des Trainings regelmäßig abgespeichert.

Mithilfe des Evaluierungstools Tensorboard konnte der Trainingsfortschrit mitverfolgt und ausgewertet werden.

3.3.2 Trainingsworkflow

Durch die in Tensorboard visualisierte beobachtung der Trainingsverläufe, konnten so schon während des traings fehlerhafte Einstellung der Trainingskonfiguraron festgestellt und korrigiert werden.

So ergibt sich der in ... dargestellte Workflow, in dem durch unterschiedliche Konfigurationen der für das Training benötigten Schritte, das Optimalste Endergebniss erzielt werden kann.

Anpasungen die für die trainierten Modelle SSD und Faster R-CNN vorgenommen wurden bezogen sich auf unterschiedliche Augmentierung der Trainingsdaten, sowie verschiedene einstellungen der Hyperparameter.

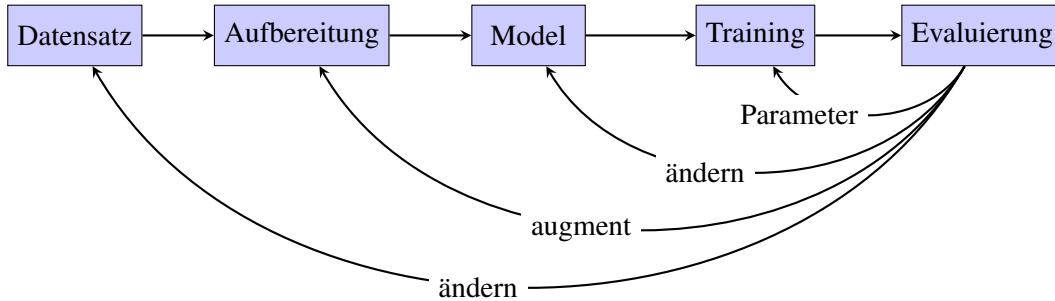


Abbildung 3.6: Trainingsworkflow

Die Ergebnisse der jeweiligen Ansätze werden im nächsten Kapitel diskutiert.

3.4 Inferenz

Die Anwendung eines fertig trainierten Modells für neue Inputdaten, wird als Inferenz bezeichnet. Zur Ausführung dieser auf dem NCS2, wird wie in den Grundlagen beschriebene das Toolkit *OpenVino* von Intel verwendet.

Dafür musste zunächst der trainierte Tensorflow Graph exportiert, d.h. die aktuellen Werte der Gewichte konstant gesetzt werden.

Mit dem ModelOptimizer, der ebenfalls im OpenVino Toolkit enthalten war konnte das Tensorflow Modell in die Intermediate Representation (IR), bestehend aus einer xml und einer bin Datei konvertiert werden. Diese können nun von der InfereneEngine zur Inferenz weiter verwendet werden.

InferecneEngine

Die für die Inferenz eines Modells im IR Formal auf dem NCS2 notwendigen Schritte, welche die InferecneEngine ausführt, sind in Abbildung 3.7 schematisch dargestellt.

Daneben ist jeweils (teils vereinfacht) die entsprechende Codezeile in Python abgebildet.

Zunächst muss das Zielgerät spezifiziert (*HW Plugin laden*) und das Modell anhand der IR Dateien definiert werden (*Model IR einlesen*), um daraus das Ausführbare Modell erzeugen zu können (*executable Model*).

In-und Outputblob sind die Dimensionen welche das Modell in den In- und Output Schichten hat und an die das Array des zu inferierenden Bilds angepasst werden müssen, was im Schritt *preprocess Input* geschieht. Nachdem die Inferenz abgeschlossen ist, können die Ergebnisse weiter verarbeitet werden, handelt es sich die Inferenz eines Videos oder Kamera Streams, werden die Schritte preprocess, Inferenz und process Output in einer Schleife wiederholt.

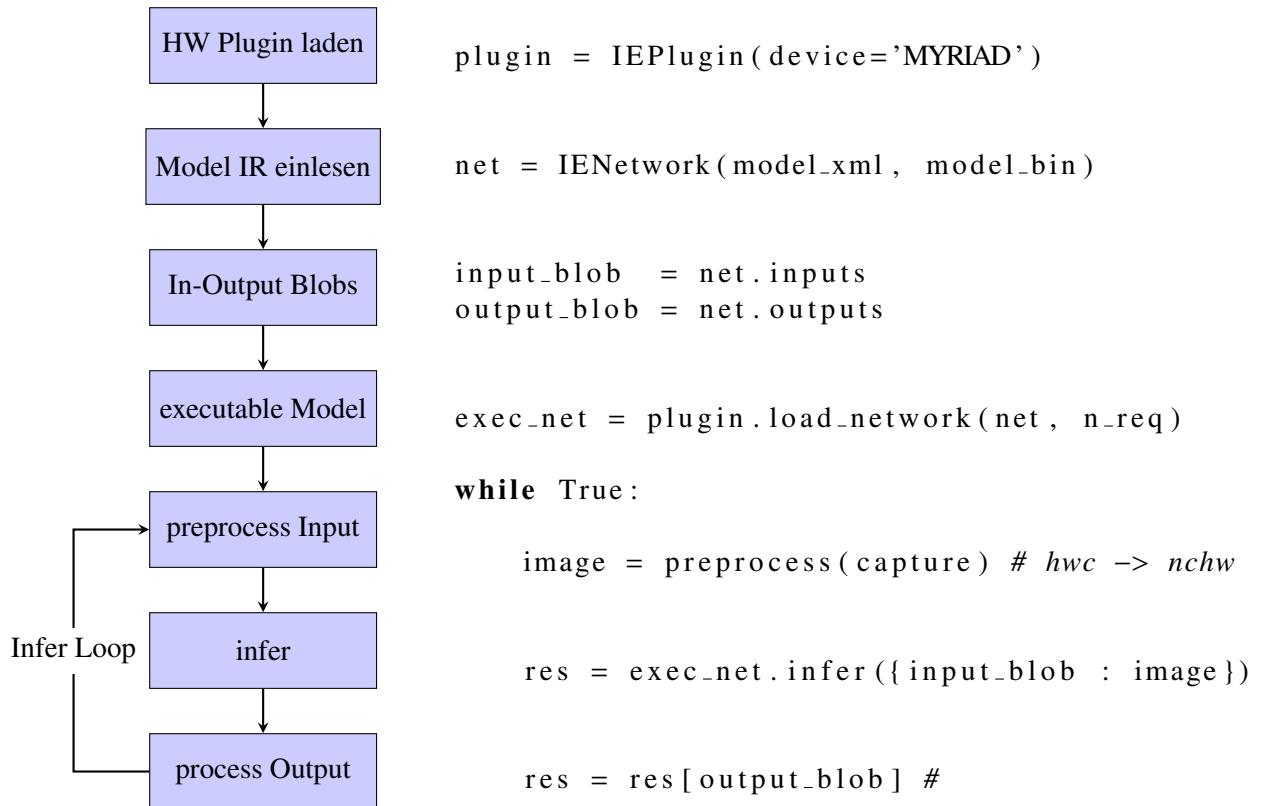


Abbildung 3.7

Die Form des Inferenz Ergebnisses hängt von der Art des verwendeten Deep Learning Modells ab, welche z.B. Image Classification, ObjectDetection, oder Instance Segmentation sein können.

Für Object Detection Modelle enthält das Ergebnis Datenstrukturen, welche den Index der Klassen, zugehörige Wahrscheinlichkeit, sowie Boxkoordinate der geschätzten Objekte im Bild enthalten.

Indem man für alle Schätzungen die in einem Bild gemacht werden einen Threshold für die Wahrscheinlichkeit festlegt (zb. 0,7), können die sinnvollen Ergebnisse herausgefiltert werden.

Zur Veranschaulichung können die Koordinaten dann als Bounding Box in das Inferierte Bild gezeichnet werden.

Kapitel 4

Evaluierung

Dieses Kapitel beschreibt die Auswertung der Ergebnisse der trainierten Modelle.

Dafür werden im ersten Abschnitt zunächst die Metriken erklärt, anhand denen die Evaluierung erfolgte.

Im zweiten Abschnitt werden die beiden verwendeten Model Architekturen SSD und Faster R-CNN hinsichtlich dieser Metriken, sowie anhand inferenzergebnisse verglichen.

Der dritte Abschnitt beschreibt Methoden mit denen die Ergebnisse das Faster R-CNN optimiert werden konnten und im vierten Abschnitt werden die Modelle noch einmal miteinander verglichen, dieses mal hinsichtlich der Inferenzzeit.

4.1 Evaluierungs Metriken

Mean Average Precision (mAP)

Zur Messung der Genauigkeit der Object Detection Modelle wurde die *Mean Average Precision (mAP)* verwendet.

Diese bezieht sowohl Klassifizierungs- als auch Lokalisierungsgenauigkeit mit ein und lässt sich den Folgenden Werten errechnen.

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

Die festlegung, für True Positive Werte wird dabei über die Intersection over Union ermittelt.

Diese ist durch den Überlappungsgrad der gelabelten (Ground Truth) und der geschätzte Boundig Box zu dem Gesamtbereich beider Boxen definiert.

Beträgt dieser mehr als ein bestimmter Threshhold, häufig 50%, gilt die Schätzung als *True Positive*, andernfalls als *False Positive*.

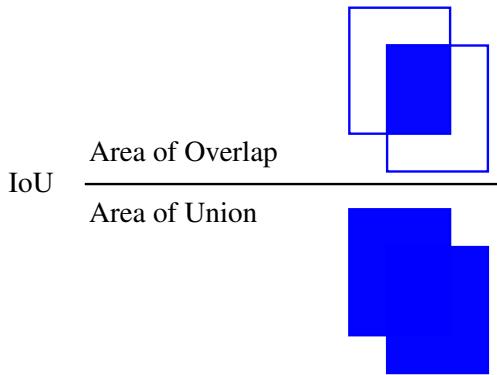


Abbildung 4.1: Intersection over Union

		Geschätzter Wert	
		p	n
Tatsächlicher Wert	p'	True Positive	False Negative
	n'	False Positive	True Negative

Abbildung 4.2: Confusion Matrix

Anhand dieser, in der Confusion Matrix dargestellten, Werte lassen sich *Precision* und *Recall* berechnen. Dabei ist der Recall definiert durch das Verhältniss der richtig gefundenen zu allen im Bild befindlichen Objekten, oder anders ausgedrückt die True Positives zu True Positive + False Negatives wie in Gleichung 4.1 dargestellt.

$$\text{Recall} = \frac{TP}{TP + FN} \quad (4.1)$$

Im Gegensatz zu Recall welcher die Trefferquote angibt, gibt die Precision die Genauigkeit an mit der die Objekte gefunden werden.

Die Precision ist durch das Verhältnis der Richtigen Schätzungen bezogen auf alle gemachten Schätzungen definiert, was durch die True Positives durch die True Positives + False Positives wie in Gleichung 4.2 dargestellt ausgedrückt wird.

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4.2)$$

Stellt man den Recall, welcher die Trefferquote angibt, und der Precision welche die Genauigkeit der Treffer angibt gemeinsam dar ergibt sich die *Precision-Recall-Kurve*, dessen Fläche die *Average Precision* für eine Klasse darstellt. Für alle Klassen im Mittel ergibt sich so die *mean Average Precision*.

$$\text{AveragePrecision} = \sum (\text{Precision}(Recall)) \quad (4.3)$$

$$mAP = \frac{1}{N} \sum AP \quad (4.4)$$

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 an die richtigen Koordinaten.

4.2 Vergleich der Modelle

Dieser Abschnitt beschreibt, wie die beiden für das Training verwendeten Object Detection Architekturen Single Shot Detector (SSD) und Faster R-CNN mit verschiedenen basis CNNs und Datensatzaufbereitungen miteinander verglichen und ausgewertet wurden.

4.2.1 Evaluierung/Validierung

Die im Folgenden dargestellte Ergebnisse beziehen sich auf das Validierungsanteil des für das Training verwendeten OpenImages Datensatzes.

Die berechnung anhand der in 4.1 erläuterten Metriken konnte dabei mithilfe Tensorflow durchgeführt und während des Trainings mit in dem Evaluierungs-Tool Tensorboard visualisiert werden werden.

Der Single Shot Detector wurde mit den Basis CNNs MobilenetV2 und InceptionV2 trainiert, das Faster R-CNN nur mit dem InceptionV2. Dabei wurden jeweils einmal der augmentierte une einmal der originale Datensatz verwendet.

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

Anhand der in Tabelle ?? dargestellten Ergebnisse ist zu erkennen, das sich mit der zweistufigen Model Architekture des Faster R-CNN bessere Ergebnisse als mit dem einstufigen SSD erzielen ließen. Dieser war besonders deutlich anhand der Loss Werte festzustellen.

Desweiteren werden unter den SSD Konfigurationen mit dem InceptionV2 als Basis CNN bessere Ergebnisse als mit dem weniger komplexen MobilenetV2 erreicht.

Bei allen Modellen war durch die Augmentierung eine Verbesserung des Loss Wertes festzustellen.

Bei den Varianten mit der SSD Architektur führt die Augmentierung jedoch auch zu einer Verringerung der mean Average Precision (mAP), was auf die weniger komplexe Model Struktur zurückzuführen sein. Kann. Je mehr Parameter einem Model zu Verfügung stehen, desto besser kann es sich an die trainingsdaten anpassen, desto eher findet jedoch auch eine Überanpassung (Overfitting) statt, was hier bei dem komplexeren Faster R-CNN Modell zu beobachten ist.

Der Plot in Abbildung ?? zeigt den Trainingsverlauf der Faster R-CNN Trainingskonfigurationen anhand der Losskurve.

Für das Training mit dem Original Datensatz nimmt diese ab ca. 100k Iterationen wieder zu, wohingegen der Loss beim Training mit augmentierten Datensatz den Wert relativ beibehalten kann.

Early Stopping war, beim Faster R-CNN Modell, ein weiter Ansatz Overfitting zu vermeiden. Dabei wird, bevor der Loss Wert anfängt sich wieder zu verschlechtern, das Training abgebrochen.

Anhand der Losskurve ?? ist zu erkennen, dass sich dadurch die gleichen Werte wie durch Augmentierung erreichen ließ. Auf der anderen Seite kann der mAP Wert, wie in Abbildung ?? zu erkennen ist durch das frühzeitige Stoppen seinen Endwert nicht erreichen.

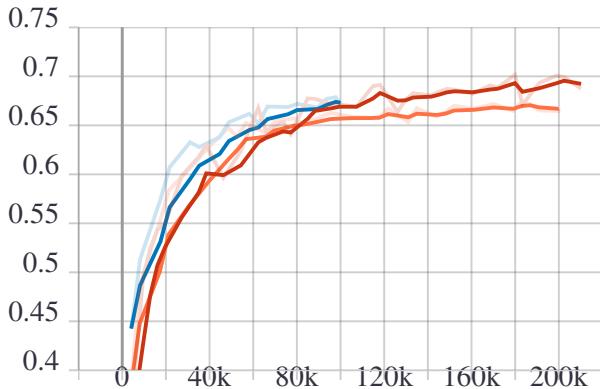


Abbildung 4.3: mAP

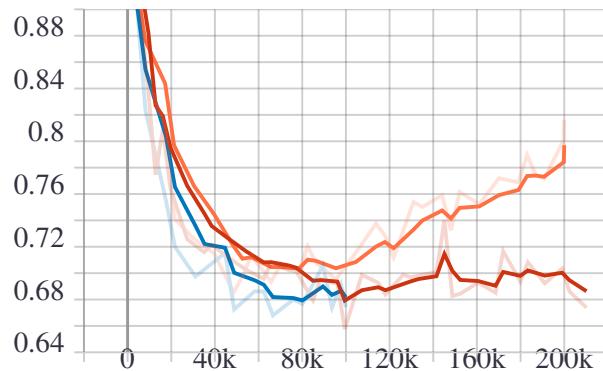


Abbildung 4.4: Loss

● Ohne ● Early Stopping ● Augmentierung

Daraus lässt sich schließen, dass Augmentierung das bessere Verfahren gegen Overfitting ist. Um herauszufinden ob sich die Vermutung bestätigen lässt und wie sehr sich die Unterschiedlichen Ergebnisse zwischen SSD und Faster R-CNN in der Praktischen Anwendung bemerkbar machen wurde die Inferenz der Trainierten Modelle auf verschiedene Bilder testweise ausgeführt.

4.2.2 Test Inferenz

Um die Inferenz testweise für verschiedene Bilder ausführen zu können wurden die trainierten Modelle wie in 3.4 beschrieben zunächst mithilfe des Model Optimizers in die *Intermediat Representation (IR)* und anschließend mit der Inferenz Engine in einem Python Script zur inferenz auf den NCS2 geladen.

Mithilfe des Scripts wurde dann die Inferenz für die verschiedenen Model- und Datensatzkonfigurationen zunächst auf den Testanteil des für das Training verwendeten OpenImages Datensatzes ausgeführt. Da dieser sehr recht ähnlich zum Trainingsdatensatz ist wurden, um die Robustheit der Modelle gegenüber anderen Daten testen zu können, die Inferenz zum einen auf selbst aufgenommene Bilder und zum anderen auf einen neuen Datensatz, dem *iWildCam 2019 Dataset* [21] ausgeführt. Dabei wurden nur die auf augmentierte Daten trainierten Modelle berücksichtigt.

OpenImages Test Set

Die Inferenzergebnisse des OpenImages Test Sets ergaben, das in den meisten Fällen sowohl das SSD als auch das Faster R-CNN die Tiere in den Bildern richtig erkannten.

Waren die Tiere jedoch sehr weit weg, mehrere gleichzeitig im Bild oder mit schlechter Qualität abgebildet, vielen die Ergebnisse beim Faster R-CNN besser aus, wie in den Abbildungen 4.5 und 4.6 zu erkennen ist. Der Unterschied zwischen MobilenetV2 und InceptionV2 beim SSD sowie zwischen Early Stopping und Augmentierung beim Faster R-CNN war hier eher gering, weshalb diese hier nicht extra abgebildet wurden.

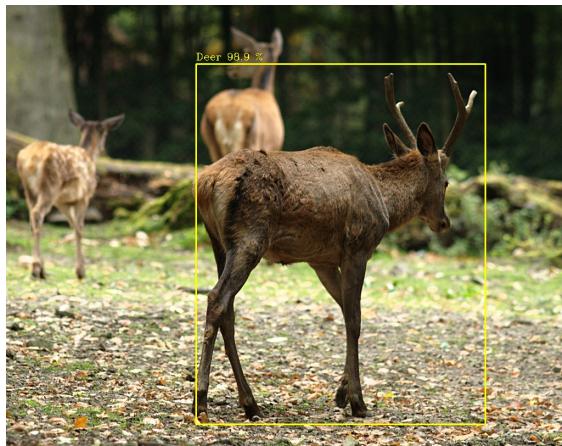


Abbildung 4.5: SSD

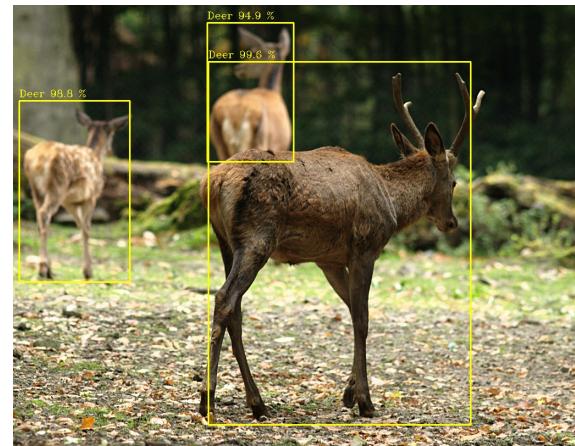


Abbildung 4.6: Faster R-CNN

Eigene Aufnahmen

Die eigenen aufgenommenen Bilder enthielten im Gegensatz zum OpenImages nur Reh Bilder, weshalb hier nur die eine Klasse getestet werden konnte.

Hier war ein deutlicher Unterschied der Modell festzustellen, in aufsteigender Reihenfolge lieferten SSD + MobilenetV2, SSD + InceptionV2, Faster R-CNN mit Early Stopping und Faster R-CNN mit Augmentierten Daten wie in den Abbildungen 4.7 bis 4.10 festzustellen ist, bessere Ergebnisse.

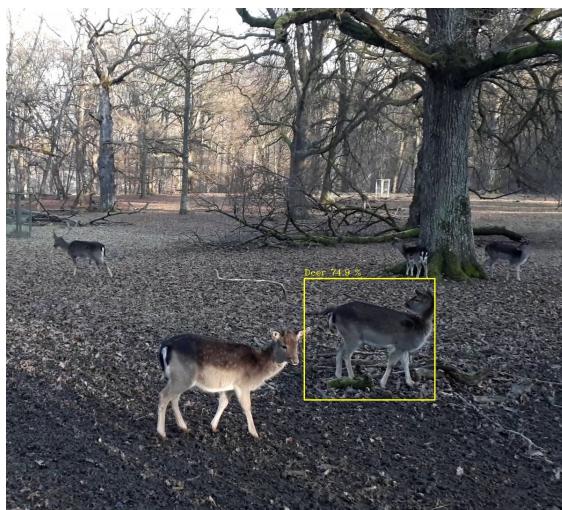


Abbildung 4.7: SSD Mobilnet

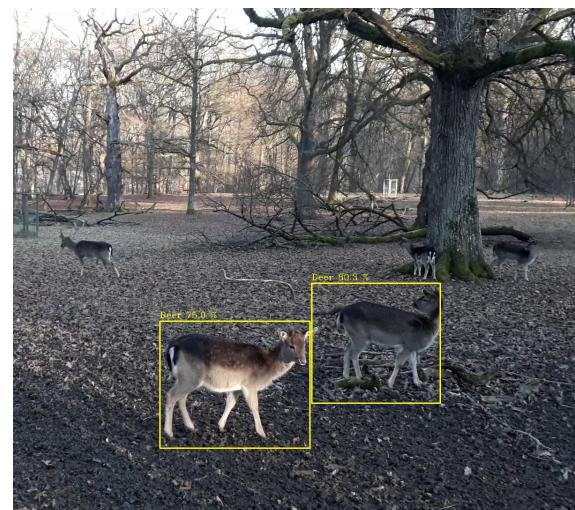


Abbildung 4.8: SSD Inception

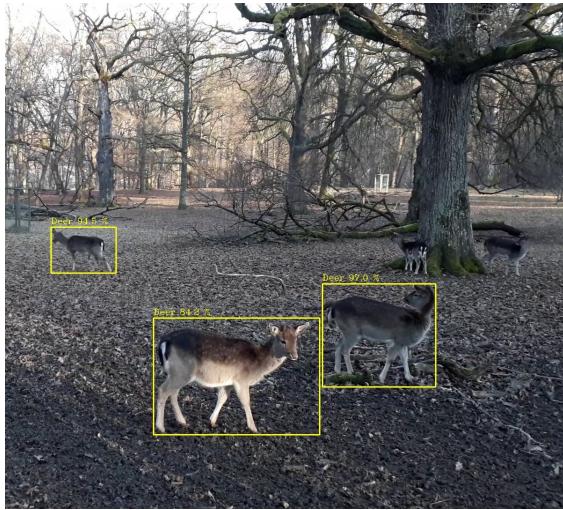


Abbildung 4.9: Faster R-CNN + Early Stopping

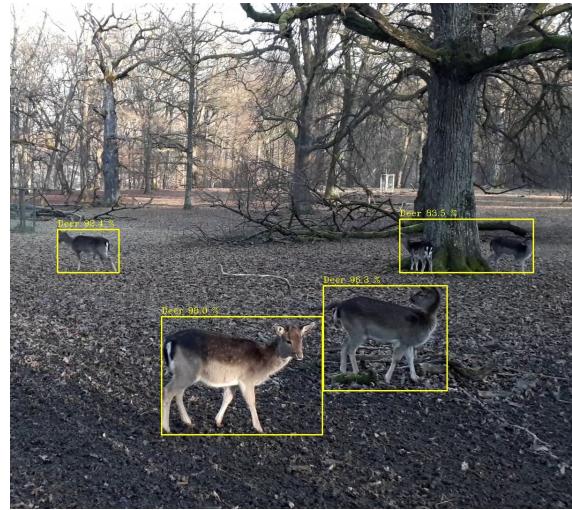


Abbildung 4.10: Faster R-CNN + Aug

Auch hier fällt auf das besonders bei Tieren auf kleineren Skalen die Faster R-CNN Architekturen bessere Ergebnisse liefern konnten.

iWildCam

Aus der von Kaggle veröffentlichten *iWildCam 2019 Competition* wurden die Klassen, welche sich mit für das Training verwendeten überschnitten, heruntergeladen und testweise inferiert.

Die Bilder stellen dabei aufnahmen von Wildtier Kameras aus dem Süd- und Nordwesten Amerikas dar, welche aus der iNaturalist Datenbank und von Microsoft DatAirSim stammen. [21].

Darunter enthalten waren viele Nachtaufnahmen, welche wenig beleuchtet und mit Infraro Kameras aufgenommen und in Graustufen dargestellt waren.

Da die Ergebnisse bei allen vier Modellen deutlich schlechter als bei dem OpenImages Datensatz und den eigenen Bildern waren, wurde, versucht diese durch weitere Optimierungen zu verbessern.

4.3 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.

Zunächst werden auch in diesem Abschnitt die Ergebnisse der anhand der Evaluierungsmaßen und Trainingsverläufen der Tensorboard Visualisierungen aufgeführt und anschließend anhand testweise ausgeführter Inferenzen weiter verglichen.

4.3.1 Verschiedene Augmentierungen

Der erste Ansatz war, das Faster R-CNN mit variierendem Augmentierungsgrad der Daten für insgesamt mehr Iterationen (500k statt 200k) zu trainieren

Dabei wurde wieder das im Abschnitt 3.1.1 erläuterte Augmentierungsverfahren angewendetet, mit den variationen

1. nur eine (zufällige) Augmentierung pro Bild, anstatt zwei
2. 4000 Bilder pro Klasse anstatt 3000 generieren

Die Trainingsergebnisse für 500k Iterationen sind anhand der Trainingsverläufe von Loss und mAP in den Plots 4.3.1 und 4.3.1 dargestellt.

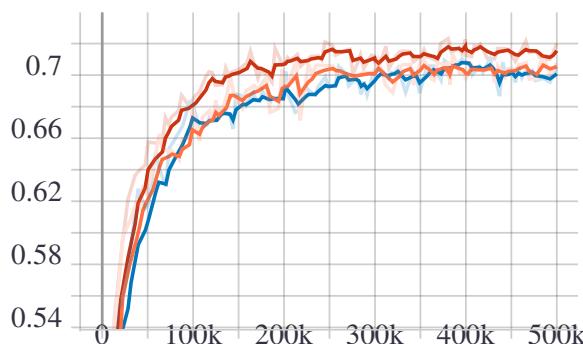


Abbildung 4.11: mAP

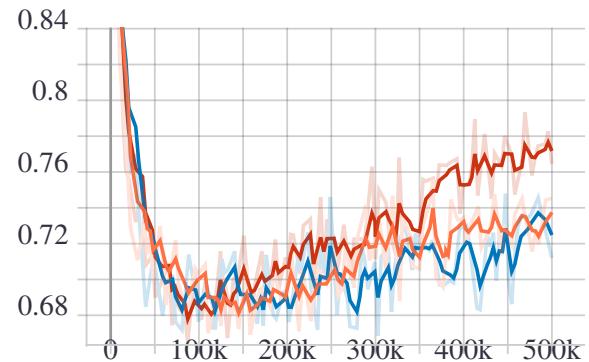


Abbildung 4.12: Loss

● 1 je bild ● 3000 samples ● 4000 samples

Aufgrund des länger durchgeföhrten Trainings ist bei allen Konfigurationen im, gegensatz zu den Ergebnissen des vorherigen Abschnitts, Overfitting zu erkennen.

Dieses fällt, wie man es auch erwarten würde, bei den weniger augmentierten Datensätzen stärker aus. Auf der anderen Seite kann dabei das einen besseren mAP erreichen.

Was sich positiver auf des Inferenzergebnis auswirkt soll auch hier wieder die Testinferenz, ausgeführt auf den iWildCam Datensatz veranschaulicht zeigen.



Abbildung 4.13: 3000 Samples



Abbildung 4.14: 4000 Samples



Abbildung 4.15: 50% Augment

Abbildung 4.3.1 bis 4.3.1 zeigen beispielhaft die Inferenz ergebnisse. Weniger stark augmentierte Daten haben bei dem zum original Datenset recht verschiedenen iWildCam Datensatz den effekt, das die Tiere schlechter bis gar nicht mehr erkannt wurden gehabt.

4.3.2 Verschiedene 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 reduzieren.

In der Konfigurationsdatei des Faster R-CNN kann dies durch setzen eines bestimmten Parameters für sowohl die erste Stufe des Netzes, dem RPN (Region Proposal Network), als auch für die zweite Stufe, dem

Klassifizierungsnetz, separat eingestellt werden.

Ebenso lassen sich die beiden Losskurven, aus denen sich beim Faster R-CNN der gesammt Loss zusammensetzt, separat anzeigen was in den Plots 4.18 und 4.19 dargestellt ist.

Durch die getrennte beobachtung der Loss Kurven ließ sich feststellen, dass Overfittin nur beim Loss des RPNs stattfindet, weshalb der Parameter für die L2 Regulierung dann nur für die erste Stufe mit einem Faktor von $\lambda = 0.001$ gesetzt.

Vergleicht man wieder die wieder die Losskurven, ist deutlich zu erkennen das sich das Overfitting im Regoin Proposal Network reduzieren ließ 4.19, wodurch sich auch der gesammt Loss verbesserte 4.17.

Auch hier ging die Verbesserung des Loss Wertes mit einer leichten Verschlechterung des mAP einher, wie in 4.16 zu erkennen ist.

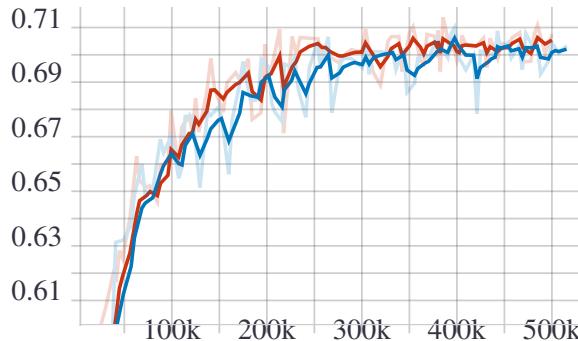


Abbildung 4.16: mAP

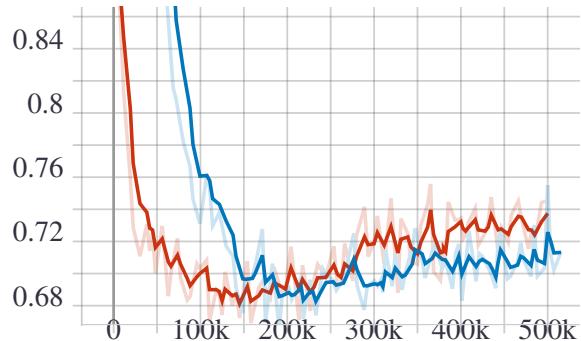


Abbildung 4.17: Total Loss

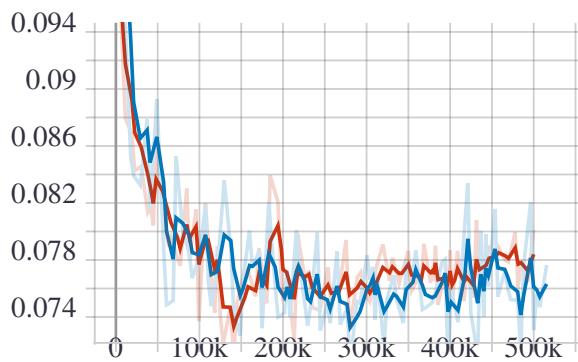


Abbildung 4.18: Classifier Loss

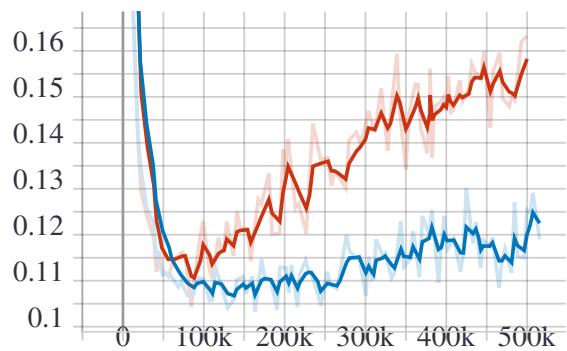


Abbildung 4.19: RPN Loss

● nur Augmentierung

● Augmentierung+L2 Regulierung

Um die Auswirkung der in den Trainingsverläufen zu erkennenden Effekten zu überprüfen, wird hier nun auch wieder die Inferenz auf die drei Testdatensetze ausgeführt. Beispielhaft, sind Ergebnisse davon für eigene Bilder in den Abbildungen 4.3.2 und 4.3.2 dargestellt.

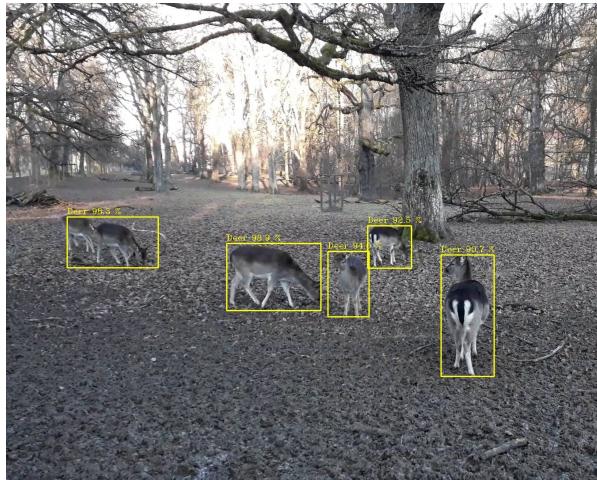


Abbildung 4.20

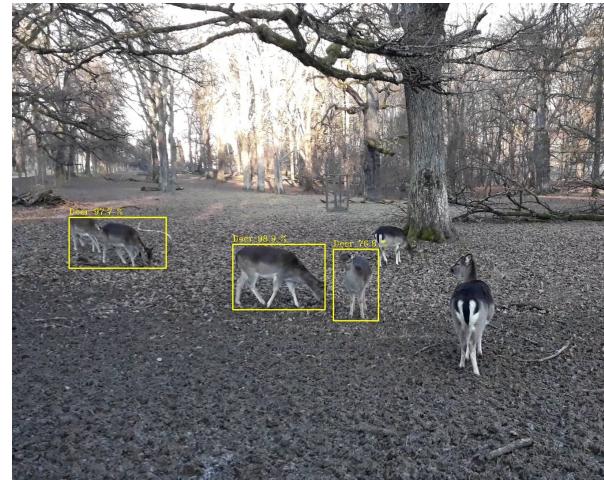


Abbildung 4.21

Daraus ergab sich, dass eine L2 Regulierung der Modelle in den meisten Fällen keinen Einfluss auf die Ergebnisse hatte, falls doch führte sie tendenziell eher zu einer Verschlechterung.

Weitere angewendete Regularisierungen, die jedoch auch zu keiner nennenswerten Veränderung des Trainingsergebnisses führten waren Dropout und L2 mit $\lambda = 0,02$ und sind in Tabelle ?? zusammen mit den anderen Ergebnissen dargestellt.

	mAP	Loss
Augmentierung (50%)	0,72	0,76
+ L2 Reg ($\lambda = 0,001$)	0,71	0,75
Augmentierung (normal)	0,7	0,74
+ Dropout	0,7	0,73
+ L2 Reg. ($\lambda = 0,001$)	0,7	0,69
+ L2 Reg ($\lambda = 0,002$)	0,69	0,7
Augmentierung (4000 Samples)	0,7	0,71

Tabelle 4.1: Regularisierungen

Aus den Ergebnissen lässt sich schließen, dass die Art der Aufbereitung der Daten den größeren Einfluss auf das Ergebniss haben und durch Einstellung der Hyperparameter wenn überhaupt noch Finetuning betrieben werden kann.

4.3.3 Graustufen

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

4.4 Inferenz zeit

Neben der Genauigkeit war die Ausführungszeit welche ein Model für die Inferenz benötigt, ein weiteres Kriterium für die Auswahl des zu verwendenden Modells. Einer der Faktoren, der die Inferenzzeit beeinflusst, ist die verwendete Hardware sowie Library. Die Hardware war mit dem Neural Compute Stick 2 festgelegt, als Library kamen dabei OpenCV oder OpenVino in Frage, wobei mit OpenVino die Möglichkeit zur asynchronen Inferenzausführung sowie mehreren Inferenz Requests besteht, wodurch sich die Inferenzzeit optimieren ließ.

Ein weiter Faktor ist die Komplexität der CNNs und die zur Objekterkennung verwendeten Architektur.

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

Um den Effekt, den die drei unterschiedlichen für das Training verwendeten Varianten SSD + MobilenetV2, SSD + InceptionV2 und Faster R-CNN + InceptionV2 auf die Inferenzzeit haben zu untersuchen, wurden diese durch Messungen der FPS verglichen.

Dabei wurde die Asynchrone Inferenz mit unterschiedlicher Anzahl an Inferenz Requests verwendet, weshalb in diesem Abschnitt zunächst auf die Funktionsweise der Asynchronen Inferenzausführung mit OpenVino eingegangen wird.

4.4.1 Asychrone Inferenz

Wird die Inferenz im Synchronen Modus ausgeführt, kann immer nur entweder inferiert oder die Bilddaten vor- bzw. nachverarbeitet werden.

Vorverarbeitung der Bilder beeinhaltet dabei z.B. die Umwandlung des von der Kamera gelieferten Bildformats in das für das jeweilige Modell richtige Input Format. Die Nachbereitung bezieht sich auf das verarbeiteten der Inferenz Ergebnisse in der Anwendung.

Die Implementierung in OpenVino erfolgt dementsprechend sequentiell, wie im Algorithmus 1 dargestellt ist.

Anhand des zeitlichen Ablaufs, welcher in Abbildung 4.22 zur Veranschaulichung abgebildet ist, sind die Abschnitte, in denen keine Inferenz stattfindet gut zu erkennen.

Algorithm 1 Synchrone Inferenz

```

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

```

Preprocessing

Process Output

Infer Frame

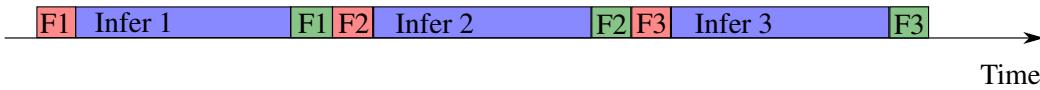


Abbildung 4.22

Da die Inferenz auf dem Myriad Chip des Neural Compute Sticks und nicht auf dem ausführenden PC oder Raspberry läuft, kann diese auch ungehindert parallel dazu ablaufen.

In OpenVino wird das mithilfe der Asynchronen API erreicht, welche über einen bestimmten Funktionsaufruf in einem separaten Thread gestartet wird.

Indem man vor Erhalt und Verarbeitung eines aktuellen Inferenz Ergebnisses den Request für den nächsten Durchlauf aufgibt, wie in Allgorithmus 2 als Pseudocode dargestellt, kann der in Abbildung ?? dargestellte Zeitliche Ablauf erreicht werden.

Algorithm 2 Asynchrone Inferenz

```

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

```

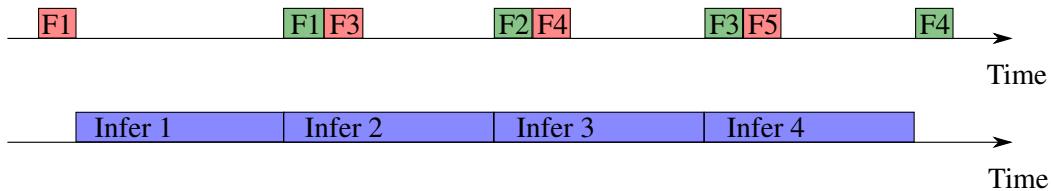


Abbildung 4.23

Die hier mit *Current* und *Next* bezeichneten Inferenz Requests entsprechen den indices der jeweiligen Requests und können (theoretisch belieb) erweitert werden. Dadurch wird erreicht, dass die Inferenz zusätzlich zum sowohl asynchron zum restlichen Programmcode abläuft, als auch auf mehreren Threads parallel.

4.4.2 Vergleich der Modelle

Mithilfe eines Python Scripts in welchem die Asynchrone Inferenz für eine variabel einstellbare Anzahl an inferenz Requests implementiert wurde, konnte für die drei Modelle die durchschnittliche Anzahl an Frames die pro Sekunde inferiert werden können ermittelt werden.

Diese wurde auf dem Raspberry Pi und dem Neural Compute Stick ausgeführt und liefert die in Tabelle ?? dargestellten Ergebnisse.

Model	Asynchrone Inferenz Requests			
	1	2	3	4
SSD MobilenetV2	19,5	35,2	40,6	40,3
SSD InceptionV2	15,6	27,7	31,1	31,7
Faster R-CNN Incept.	0,63	0,67	0,75	0,74

Tabelle 4.2: Vergleich Inferenz Zeiten Modelle

Die asynchrone inferenz ausführung führte bei alle Modellen für bis zu 3 inferenz Requests zu besseren Ergebnissen. Ab dann war eine Erhöhung nicht merh sinnvoll.

Ein Unterschied in der benötigten Zeit macht sich besonders stark zwischen SSD Faster R-CNN Architekturen bemerkbar.

Für Real-Time Anwendungen kommen daher nur die mit SSD Trainierten Modelle in Frage.

Ob und wie das Fater R-CNN trotz der recht langsamen Inferenzzeit für die Anwendung verwendet werden können, wird im nächsttem Kapitel erläutert werden.

Kapitel 5

Entwicklung der Anwendung

Dieses Kapitel beschreibt die Realisierung der Anwendung als autonomes Kamera System zur Wildtiererkennung, welches auf einem Raspberry Pi 4 läuft.

Dabei wird es neben der Implementierung der Inferenz für eines der traininierten Modelle, auch um die Auswahl einer nachtsichtgeeigneten Kamera und der implementierung einer Kommunikationsmöglichkeit zur übermittlung der Daten gehen.

5.1 Hardware

Der Aufbau der Anwendung besteht aus einem Raspberry Pi 4, auf dem der Programmcode läuft, sowie dem Neural Compute Stick 2 für die Inferenz, welcher über eine USB Schnittstelle mit dem Raspberry Pi verbunden wird.

Zur aufnahme der Bilder wurde ein Raspberry Pi Kamera Modul mit 5MP OV5647 Sensor der Marke Longrunner verwendet. Dieses ermöglicht durch mechanisches zu und abschalten eines Infrarot Filters vor die Linse zwischen Tag und Nachtsicht zu wechseln. Der dafür verwendete Magnetschalter wird dabei über einen Helligkeitsensor getriggert. Im Infrarotmodus befindet sich der Filter nicht vor der Linse, wodurch neben den elektromagnetischen Wellen des Sichtbaren Lichts auch die etwas langwelligeren (850nm) des Infrarot Spektrums auf die Linse treffen und verarbeitet werden können.

Zudem verfügt die Kamera über zwei Infrarot LEDs, sodass auch Aufnahmen bis zu 3m Entfernung in völliger Dunkelheit gemacht werden können. Diese haben den vorteil gegenüber normalen Scheinwerfern, das die Tiere von keiner Sichtbaren Lichtquelle gestört oder verscheucht werden. verbunden wird das Kamera Modul über die CSI (Camera Serial Interface) Schnittstelle des Raspberry Pi's.

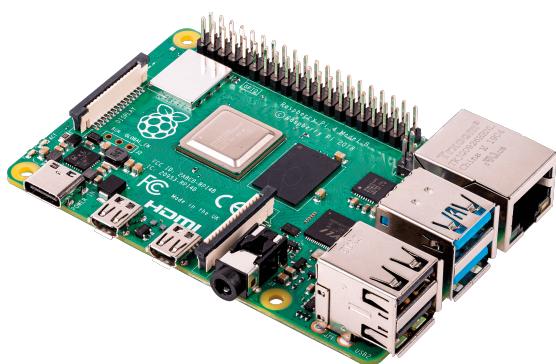


Abbildung 5.1: Raspberry Pi 4



Abbildung 5.2: Longrunner Kamera Modul

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

5.2 Software

Die Implementierung der Anwendung für den Raspberry Pi wurde in Python vorgenommen. Dabei sind die Funktionalitäten zur Objekerkennung in einem Script *detection.py* und die für die Verbindung und Senden der Daten in einem *connection.py* Script aufgeteilt.

Der Kamera Inputstream ist in einem *main.py* Script implementiert, von dem aus auch die in Abbildung 5.3 dargestellten Klassen, welche in *detection.py* und *connection.py* definiert sind verwendet werden.

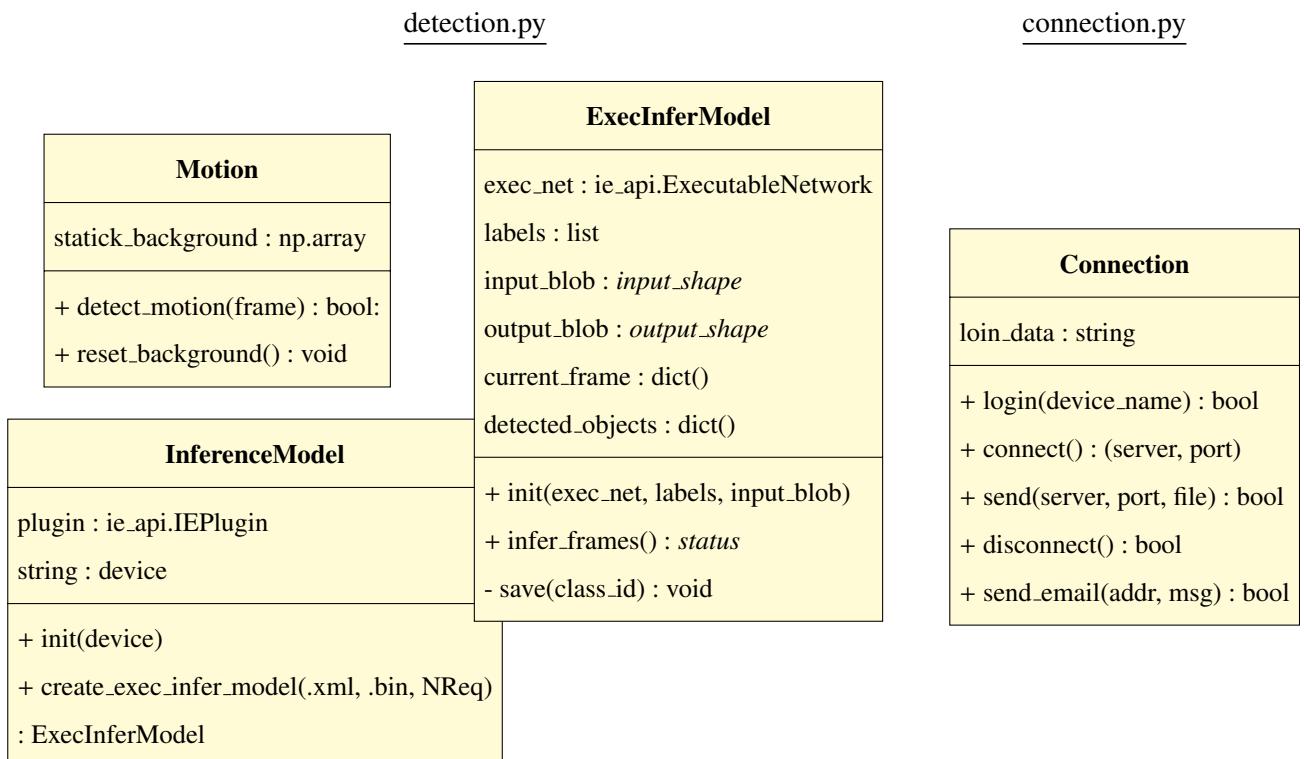


Abbildung 5.3: Klassendiagramm

Motion ist eine Klasse zur Erkennung von Bewegung im Kamera Input Stream, *InferenceModel* und *ExecInferModel* dienen der Inferenz und *Connection* dem Aufbau einer Verbindung zu einem Pc sowie dem Senden der Daten über diese Verbindung.

Durch geeigneten Implementierung des Applikationsablaufes, sollte eine Möglichkeit gefunden werden, trotz langsamer Inferenzzeit, mit dem Faster R-CNN alle relevanten Frames, also die in denen Tiere zu vermuten sind, inferieren zu können. Dafür wurde die Annahme gemacht, dass zur Laufzeit der Anwendung nicht durchgehend inferiert werden muss, sich also Zeitweise keine Tiere und damit auch keine Bewegung vor der Kamera befinden.

Um Bewegungen feststellen zu können, wurde, mithilfe der Library OpenCV ein Bewegungsmelder implementiert. Dieser speichert zu begin des Kamera Streams ein Referenz Frame ab, mit dem alle weiteren Frames verglichen werden. Beträgt der Abstand einzelnen Pixelwerte im Graustufenbereich mehr als ein bestimmter Threshhold, wurde eine Bewegung erkannt. Indem nun die Frames, welche der Kamera Stream permanent liefert, zunächst auf Bewegung überprüft wurden, ließ sich unnötiges inferieren, was Zeit und Leistung kostet, vermeiden. Frames die bewegung enthalten und aufgrund der langsameren inferenzzeit

des Faster R-CNN nicht sofort inferiert werden können, werden in einem Buffer zwischen gespeichert und in Phasen zu denen keine Bewegung stattfindet inferiert.

Dafür musste der im Abschnitt 4.4 beschriebene Asynchrone Inferenz Ablauf dahingehend angepasst werden, dass keine Blockieren mehr stattfindet, diese also komplett zeitasynchron zu den Input Frames ablaufen kann. Der Gesamtablauf der Applikation ist in Abbildung 5.4 schematisch als Flussdiagramm dargestellt.

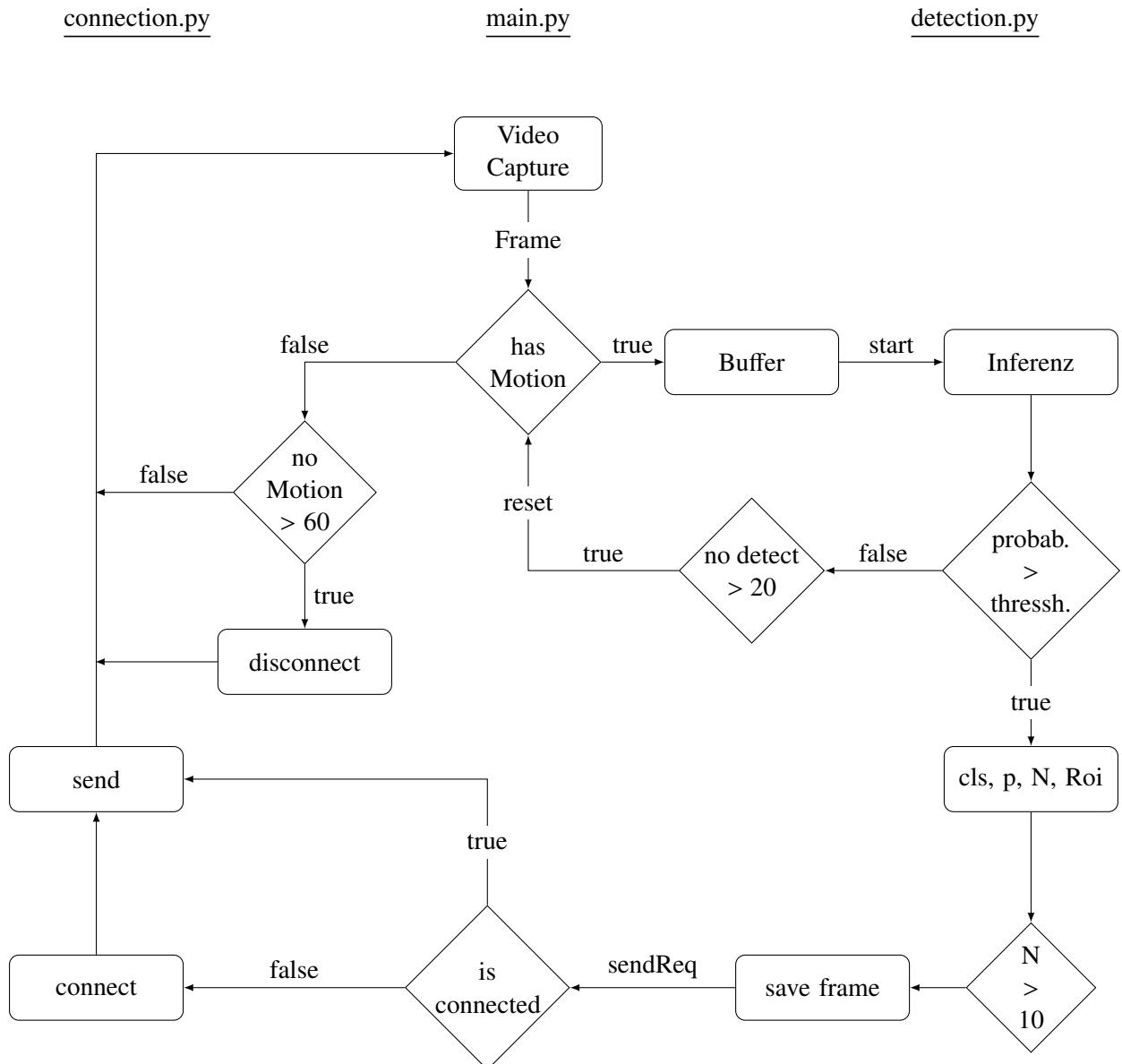


Abbildung 5.4: Main Applikation

Wird in inferierten Frames mehrfach nichts erkannt, wird das referenz Frame des Bewegungsmelders neu gesetzt, da vermutlich sich das Hintergrundbild geändert hat.

Erkannte Objekt werden in einem Dictionary zusammen mit Klassenname (cls), Wahrscheinlichkeit(p) Anzahl an Erkennungen (N) sowie die Bounding Box Koordinaten (Roi, Region of Interest) abgespeichert.

Nach 10 Erkennungen des selben Objekts, wird dieses als lokale Bilddatei abgespeichert und ein Send-Request an die Main zurückgegeben.

Diese prüft ob eine Verbindung zu einem anderen Gerät besteht, stellt diese gegebenenfalls her und sendet die Bilder.

Um nicht permanent die Verbindung zu einem Pc aufrecht erhalten zu müssen, was das Datenvolumen des mobilen Internets, des Raspberry schneller aufbrauchen würde, wird diese nach einer bestimmte zeit ohne Bewegung getrennt.

Inferenz

Der in Abschnitt 4.4 beschriebene asynchronen Inferenzablauf wurde dahingehend angepasst, dass eine beliebige Anzahl an Inferenz Requests verwendet werden können und dass das Warten auf ein Inferenz Ergebnis nicht mehr blockierend ist. Dafür wurde der Timeot in der Wait Funktion auf *0ms* gesetzt. Durch vorheriges abspeichern der jeweiligen zu inferierenden Frames, kann eine richtige Zuordnung der Inferenz Ergebnisse zu den Frames erfolgen. Der Pseudocode in Algorithmus 3 stellt grob den Inferenzablauf dar.

Algorithm 3 Asynchrone Inferenz, ohne Blockierung

```

while true do
    capture FRAMES
    for all InferRequests do
        if wait for InferRequest == 0 then
            Result ← InferRequest.output
        end if
        if Buffer not empty then
            preprocess InferRequest
            start InferRequest
        end if
        if Result not NULL then
            process Result
        end if
    end for
end while
```

Connection

Um die Bilder mit erkannten Tieren an ein anderes Gerät z.B. einen Pc senden zu können, musste eine Verbindung hergestellt werden, die auch über verschiedene Netzwerke hinweg funktioniert.

Um unabhängig von Router Konfigurationen des Heimnetzes zu sein, wurde mithilfe des Dienstes *remot3.it* [22] eine Cloudbasierte Remote Verbindung hergestellt. Mit dieser war es möglich eine Proxy SSH Verbindung, über das Internet herzustellen, ohne von einer Firewall blockiert zu werden.

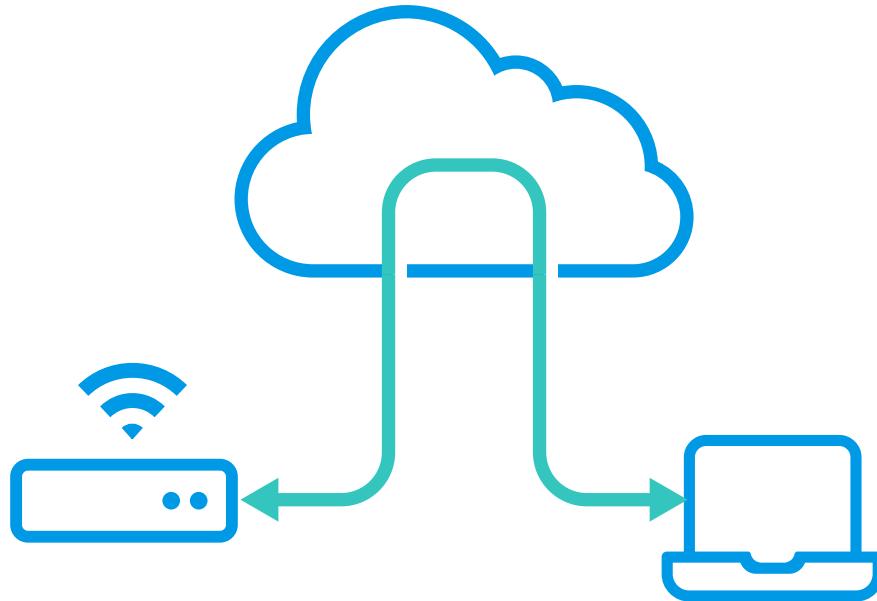


Abbildung 5.5: Prinzip Proxy Verbindung

Da die Daten vom Raspberry Pi aus automatisiert gesendet werden sollten, wurde der Pc als Remote Gerät implementiert. Gesendet wurden die Daten über das Secure Copy Protocol welches das hergestellte Secure Shell Protocol (SSH) verwendet. Dieses lässt sich über folgendes Kommando, welches im *connectin.py* Script ausgeführt wird verwenden:

```
$ scp -P port file.jpg user@proxyserver /zielpfad/file.jpg
```

Server und Port werden dabei von remoteit generiert, *file.jpg* ist das zu sendende Bild und *user* der Nutzernname des Geräts, an welches gesendet wird. Um das Einloggen sowie den Verbindungsau- und Abbau über remote.it zu einem Gerät zu automatisieren bietet remote.it eine API mit der über Post- und Get-Requests die Befehle dafür programmatisch aufgerufen werden können.

Kapitel 6

Test und Validierung

Kapitel 7

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] R. Maksutov, “Deep study of a not very deep neural network. Part 5: Dropout and Noise.”
- [2] M. S. Researcher, PhD, “Simple Introduction to Convolutional Neural Networks.”
- [3] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” p. 46.
- [4] “ImageNet Large Scale Visual Recognition Competition (ILSVRC).”
- [5] “CS231n Convolutional Neural Networks for Visual Recognition.” <http://cs231n.github.io/convolutional-networks/#layers>.
- [6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,”
- [7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,”
- [8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,”
- [9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,”
- [10] M. Häußermann, “Funktion und Effizienz von Hardware für Deep Neural Networks.”
- [11] 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,”
- [12] 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.
- [13] X. Wu, D. Sahoo, and S. C. H. Hoi, “Recent Advances in Deep Learning for Object Detection,”
- [14] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,”
- [15] “Object Detection for Dummies Part 3: R-CNN Family.”
- [16] 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.
- [17] J. Hosang, R. Benenson, and B. Schiele, “Learning non-maximum suppression,” in *CVPR*, 2017.
- [18] “SSD : Single Shot Detector for object detection using MultiBox.”

- [19] “Google Colaboratory.” <https://colab.research.google.com/notebooks/intro.ipynb#recent=true>.
- [20] “TensorFlow Object Detection Api.” https://github.com/tensorflow/models/tree/master/research/object_detection.
- [21] S. Beery, D. Morris, and P. Perona, “The iwildcam 2019 challenge dataset,” *arXiv preprint arXiv:1907.07617*, 2019.
- [22] “remot3.it” <https://remote.it/>.

Anhang A

Beispiel für ein Kapitel im Anhang

A.1 Bsp für ein Abschnitt im Anhang