Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Studiengang Künstliche Intelligenz (Master)

Studienarbeit Deep Vision

von

Patrick Sabau

Hand Pose Estimation

Bearbeitungszeitraum: von 25. Mai 2022
bis 01. Juli 2022

Prüfer: Prof. Dr. phil. Tatyana Ivanovska

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bildverarbeitung ist heutzutage für den durchschnittlichen Bürger immer noch ein nicht vertrauter Begriff. Jedoch sind viele Menschen durch Themen wie autonomes Fahren oder auch Instagram-Filter mit diesem Thema in Berührung getreten. Allerdings bietet dieser Themenbereich ein viel größeres Potenzial, als weitläufig bekannt ist. Dieses Projekt zeigt einen diese Potenziale: Pose Estimation von (interagierenden) Händen.
Pose Estimation beschreibt dabei, das extrahieren der Lage sogenannter Schlüsselpunkte (Keypoints). Diese 2D oder auch teilweise 3D Koordinaten können aus unterschiedlichen Quellen extrahiert werden, von simplen 2D RGB Bildern bis hin zu komplexer Sensorik wie Lidar.
OpenPose [Cao 18] ist beispielsweise ein Projekt, welches 2D Koordinaten eines gesamten menschlichen Körpers erkennt, wohingegen es auch Versuche gibt, nur einzelne Körperteile, wie z.B. Hände, genauer zu betrachten. Für Hände gibt es eine Vielzahl an Datensätzen und Modellen. Diese können in der Größe (Anzahl der Datenpunke), der Erstellung (Kameraaufnahmen oder synthetische 3D Modelle) oder auch in der Art der Labels (2D und/oder 3D) variieren. Hier ein paar Beispiele:

- FreiHand: RGB Bilder aus verschiedenen Kamerarblickwinkeln [Zimm 19]
- Ego3D: synthetische Bilder aus der Ego-Perspektive [Lin 20]
- BigHand2.2M: Tiefenbilder mit 3D Koordinaten [Yuan 17]

Im Folgenden wird ein im Jahr 2020 erschienener Datensatz InterHand2.6M [Moon 20b] genutzt um 2D Pose Estimation an einzelnen und auch interagierenden Händen zu entwickeln. Dieser Datensatz wurde von Meta (damals noch Facebook) veröffentlicht. Ein möglicher Use Case, der für das Unternehmen relevant ist, ist die 3D Erkennung der Hände um diese dann mittels Virtual Reality im sogenannten 'Metaverse' korrekt zu visualisieren.

Kapitel 2

Datensatz

Wie bereits in Chapter 1 erwähnt, wird der InterHand2.6M Datensatz [Moon 20b] genutzt. Dabei handelt es sich um Bilder von Händen, die zeitgleich von 6 verschieden Kameras aus unterschiedlichen Perspektiven aufgenommen wurden.

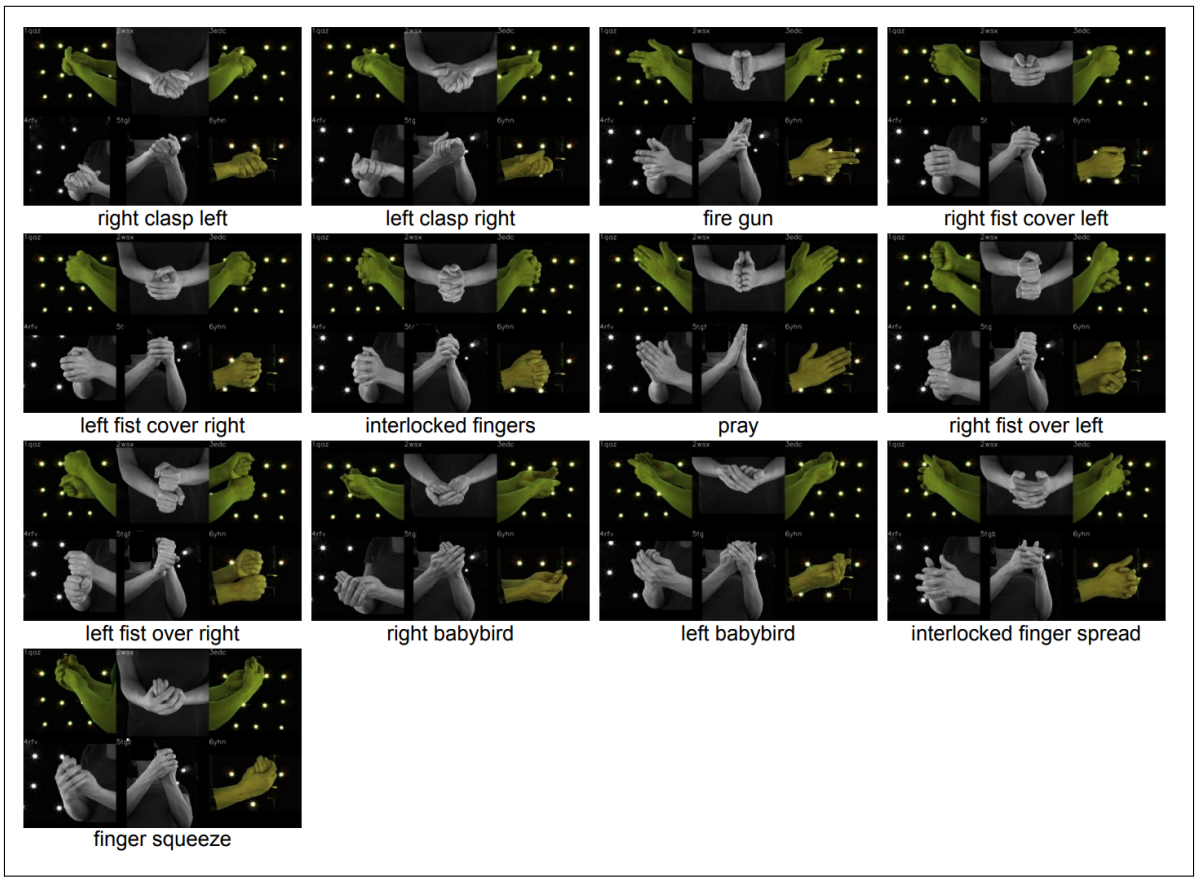

Abbildung 2.1: Beispiele InterHand2.6M Bilder[Moon 20b]

Hierbei kann es sich um eine rechte, eine linke oder aber auch um ein paar interagierender Hände handeln. Abbildung 2.1 zeigt einige Posen von interagierenden

Händen aus den 6 verschiedenen Blickwinkeln. Um als interagierend gelabelt zu werden, müssen lediglich 2 Hände auf dem Bild sein, egal ob diese sich berühren oder nicht. Eine mögliche Erklärung dafür ist, dass je nach Blickwinkel die Hände auf dem Bild überlappt dargestellt sind, es jedoch zu aufwändig ist, jeden Blickwinkel manuell zu prüfen. Der ganze Prozess wird dadurch also vereinfacht.
Der Datensatz wurde als Video aufgenommen und in einzelne Bilder zerlegt. Für das Projekt wurde die 5fps (1 Sekunde Video wird zu 5 Bildern umgewandelt) Variante ausgewählt. Diese enthält nur $1/6$ der Bilder der 30fps Variante, jedoch sind es immer noch 2,6 Millionen Bilder, die sich mehr unterscheiden als die der 30fps Variante.
Von den knapp 2,6 Millionen Bildern wurden 650.000 von Menschen annotiert, wohingegen der Rest durch Maschinen errechnet wurde (z.B. durch die unterschiedlichen Kamerablickwinkel). Der Fokus der menschlichen Annotierung lag dabei auf den interagierenden Händen, vermutlich da die Komplexität (durch z.B. Überlappung) höher ist. Von den 1,4 Millionen Bildern einzelner Hände wurden ca. 170.000 Bilder menschlich annotiert ($\sim$ 12%), wohingegen von den 1,2 Millionen interagierenden Händen ca. 475.000 menschlichen Ursprungs ($\sim$ 40%) sind. Erwähnenswert ist auch, dass bei den interagierenden Händen, welche von Maschinen annotiert wurde, ist teilweise nur eine Hand gelabelt. Die Keypoints der zweiten Hand sind außerhalb des Bildes.
Eine Aufteilung in Train (1.360.000 Bilder, $\sim$52%), Val (380.000 Bilder, $\sim$15%) und Test (850.000 Bilder, $\sim$33%) liegt bereits vor und wird auch im weiteren Verlauf so genutzt.

Kapitel 3

Methoden

Zur Pose Estimation werden zwei separate Funktionen benötigt: Object Detection der Hand und anschließend die Lokalisierung der Keypoints. Für die erste Aufgabe wird ein YOLOv5 Modell [Joch 20] und ein FasterRCNN Modell [Ren 15] verglichen, welche potenziell Austauschbar sind. Die Lokalisierung der Keypoints übernehmen dabei selbst kreierte Modelle, die für die spezifischen Handtypen ausgelegt sind. Abbildung 3.1 zeigt eine Übersicht der beschriebenen Architektur.

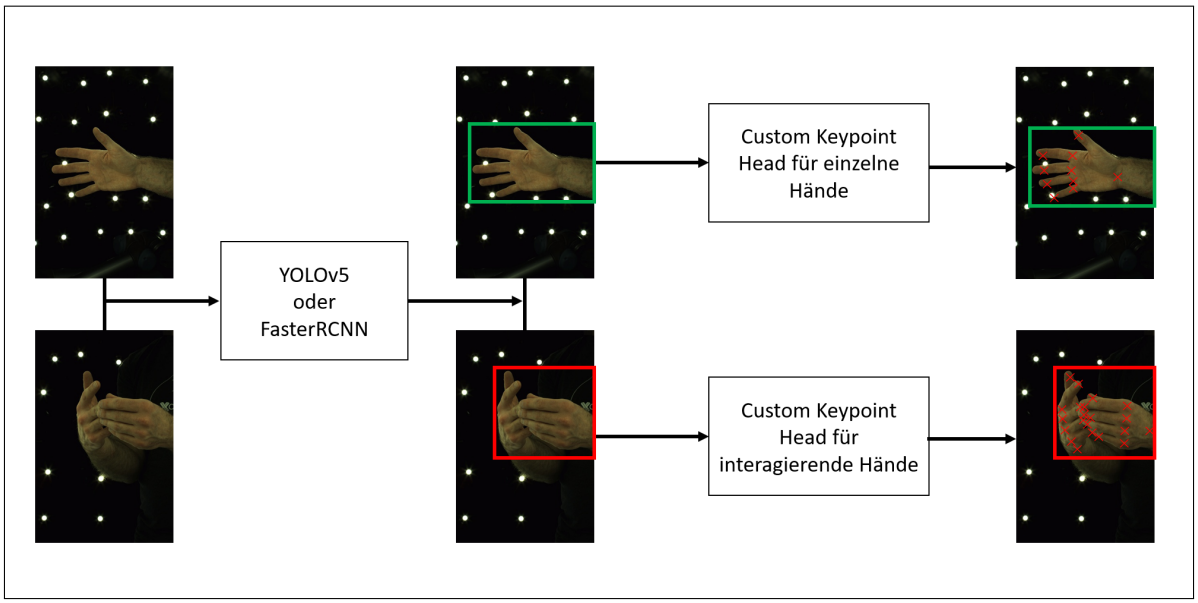

Abbildung 3.1: Übersicht Architektur

Als Alternative zu dieser Architektur, wird noch ein KeypointRCNN [He 17] trainiert. Dabei handelt es sich um Abwandlung von MaskRCNN, eine Weiterentwicklung der FasterRCNN Architektur, welche ursprünglich für Image Segmentation entwickelt wurde. Diese Architektur wird detailliert in Kapitel 3.5 beschrieben.

3.1 Dataset

Die Bilder des InterHand2.6M Datensatz können über ein Python Skript, zu finden unter scripts/download_data.py, heruntergeladen werden. Die Annotation müssen jedoch separat über die InterHand2.6M Website [Moon 20a] heruntergeladen werden.

Die Daten können als ein PyTorch Dataset geladen werden. Dabei werden alle Annotationen geladen und vorverarbeitet. Während der Laufzeit werden die Bilder erst aus dem Filesystem geladen, was Ressourcen wie den Arbeitsspeicher schont.

```
1 from dataset import dataset
2 data = dataset.Dataset(mode='train',
3                        transform = None,
4                        only_keypoints = False,
5                        limit_handedness = 'all')
```

Listing 3.1: Dataset

Die Keypoints der jeweiligen 2D Bilder sind nicht direkt in den Annotationen gespeichert. Stattdessen sind Informationen über die Kameraposition (Kameramatrix) und die globalen 3D Koordinaten enthalten. Mithilfe dieser, können die Keypoints auf das 2D Bild projiziert werden.
Über den Parameter 'mode' kann gesteuert werden, welcher der Datensätze (Train, Test oder Val) geladen werden soll. Zusätzlich kann 'limit_handedness' dazu genutzt werden, um nur Hände eines bestimmten Typs (interagierend, einzeln, links oder rechts) in dem Datensatz zu berücksichtigen. Als Standard 'transform' wird eine Konvertierung zu einem PyTorch Tensor genutzt, jedoch kann bei Bedarf auch bspw. ein Resize übergeben werden.
Je nach Modell muss der Parameter 'only_keypoints' gesetzt werden. Ist dieser nicht gesetzt, so wird neben dem Bild als Input, ein Dict als Target übergeben. Dieses ist im COCO Format [Lin 14] und enthält unter anderem Felder über die Bounding-Box der Hände, das Label der Hände und die Keypoints der Hände.
Ist der Parameter auf True gesetzt, so wird das Bild um die Bounding-Box herum ausgeschnitten. Somit fällt der Großteil des Hintergrunds weg. Zusätzlich werden die Keypoints der Hände normalisiert, d.h. diese haben einen Wert zwischen 0 und 1, je nachdem wie weit diese von den Rändern entfernt sind. Ist das ausgeschnittene Bild zum Beispiel 500x500 Pixel groß und ein Keypoint befindet sich an Pixel [100 , 200] in diesem Ausschnitt, so wird dieser als [0.2 , 0.4] kodiert.

Das YOLOv5 Modell benötigt nicht als PyTorch Dataset, sondern in einer einzigen Ordnerstruktur. Über das Skript scripts/create_yolo_annotations.py kann diese Ordnerstruktur erzeugt werden. Jedoch benötigt dieser Vorgang ziemlich viel Zeit, da das Filesystem (Windows) nicht mit so viele Daten in einem einzigen Order zurechtkommt. Die Erstellung des Val-Datensatzes mit ca. 380k Bildern benötigte nur eine halbe Stunde, wohingegen der Train-Datensatz knapp 45 Stunden brauchte, 90 mal solange

trotz des eigentlich nur 3 mal so großen Datenmenge.

3.2 YOLOv5

YOLO steht für You Only Look Once, was auch direkt eine Besonderheit dieses Object Detection Networks ausmacht: Es reicht ein einziger Durchlauf des Bildes durch das Modell [Redm 16]. In diesem Durchlauf, wird das Bild in ein Grid der Größe S x S eingeteilt. Ist der Mittelpunkt einer Label-Bounding-Box in dieser Zelle, so ist diese Verantwortlich das Objekt zu erkennen. Dabei werden B Bounding Boxen in verschiedenen Größen und Längen-Breiten-Verhältnissen gebildet. Diese enthalten neben der Mittelpunkt, sowie Höhe und Breite der Box, noch den Confidence Score für entsprechendes Objekt. Abbildung 3.2 zeigt, wie diese Bounding Boxen auf einem Beispielbild aussehen.

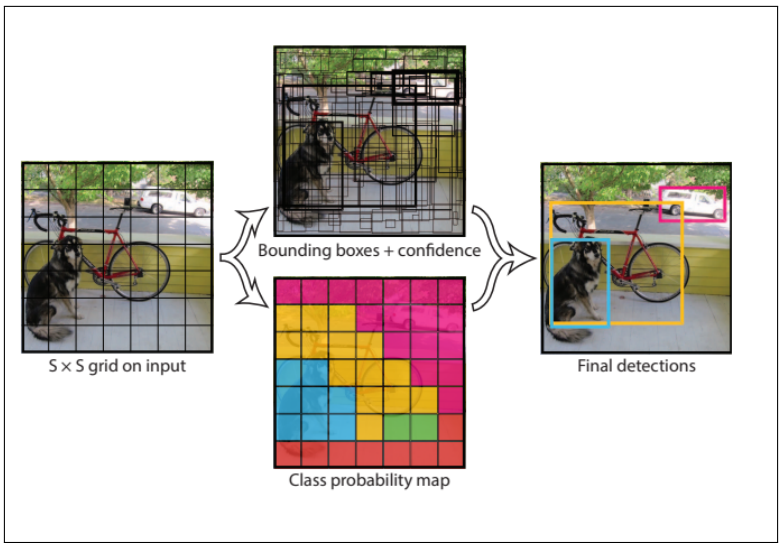

Abbildung 3.2: Bounding Box Erkennung YOLO [Redm 16]

YOLO sagt dies Koordinaten dieser Bounding Boxen direkt im Forward-Pass des Modells vorher, was in der Architektur in Abbildung 3.3 ersichtlich ist. Als Feature Network werden einige Convolution Layers genutzt, die anschließend als Input für Fully Connected Layers dienen. Diese FC Layers predicted dabei die Koordinaten der Bounding Boxes.

YOLOv5 ist eine Weiterentwicklung des ursprünglichen YOLO Modells. Diese enthält diversen Verbesserungen, die teils auch über die vorherigen YOLO Varianten hinzugefügt wurden So werden bspw. seit YOLO9000 (YOLO2) eine Optimierung der Bounding Box Vorschläge, sogenannte Anchor Boxen, genutzt [Redm 17]. Die wichtige Neuerung von YOLO in Version Nummer 5 ist die Verwendung eines CSP Backbone. Dabei handelt es sich um Convolutional Layers, bei denen einzelne Layers mehrfach

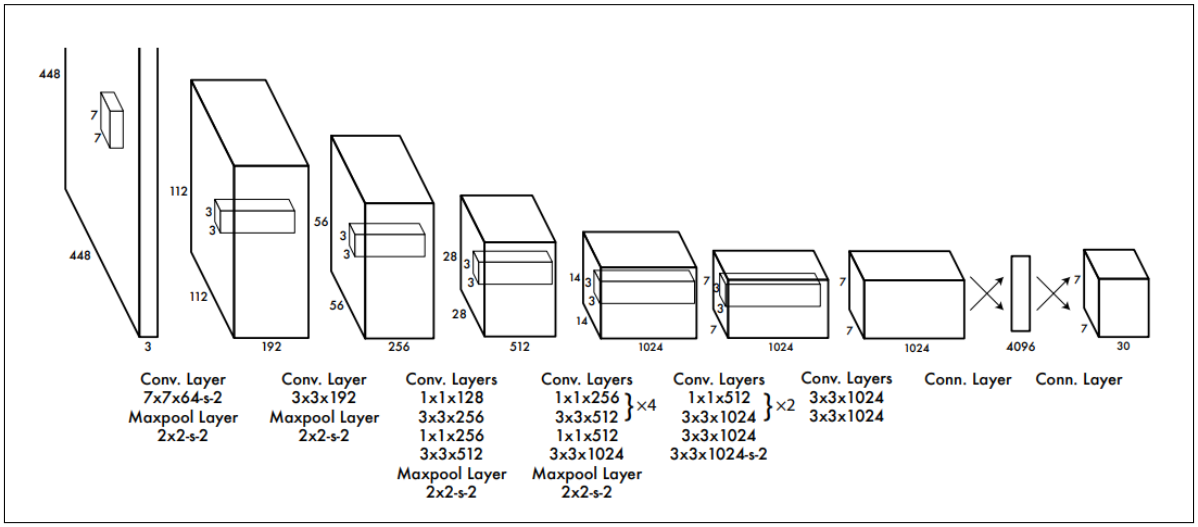

Abbildung 3.3: Architektur YOLO [Redm 16]

verwendet werden. Ziel dabei ist es, das Netzwerk dazu zu bringen Features wiederzuverwenden, sowie Probleme wie das Vanishing Gradient Problem zu verringern.

Um YOLOv5 zu trainieren, muss zuerst der Preprocessing Step aus Kapitel 3.1 ausgeführt werden. Anschließend muss eine Konfiguration als .yaml Datei erzeugt werden, welche sich bereits unter yolo_data/yolo_dataset.yaml befindet. Diese enthält neben den Pfad zu den Datensätzen auch die Labels der Klassen. Nach dem Klonen des git-Repos, kann das Training über die Konsole gestartet werden. Alternativ kann das Training auch über das Notebook Training.ipynb ausgeführt werden (dieses führt bei Bedarf auch nur den Konsolenbefehl aus).

```
1 python .\yolov5\train.py --batch 2 --epochs 2 \\
2     --data yolo_data/yolo_dataset.yaml \\
3     --weights yolov5s.pt --freeze 8
```

Listing 3.2: YOLO Training

3.3 FasterRCNN

Eine Alternative zu YOLO bietet die Klasse der RCNN Modelle. FasterRCNN ist dabei der Nachfolger von FastRCNN und dieses wiederum von RCNN [Ren 15]. Im Gegensatz zu YOLO reicht es jedoch nicht, dass das Bild ein einziges Modell einmal durchläuft. Stattdessen besteht FasterRCNN aus 3 Teilen: Feature Network, Region Proposal Network und einem Region of Interest Pooling. Abbildung 3.4 zeigt das Zusammenspiel dieser Komponenten.

Als Feature Network dienen einige Convolutional Layers. Hier können bereits vortrainierte Modelle (bzw. Convolution Layer dieser Modelle), bspw. ResNet50 genutzt werden. Das Region Proposal Network (RPN) lässt sogenannte Anchor Boxen in verschiedenen Größe, Längen-Breiten-Verhältnissen und Ausrichtungen über das Bild

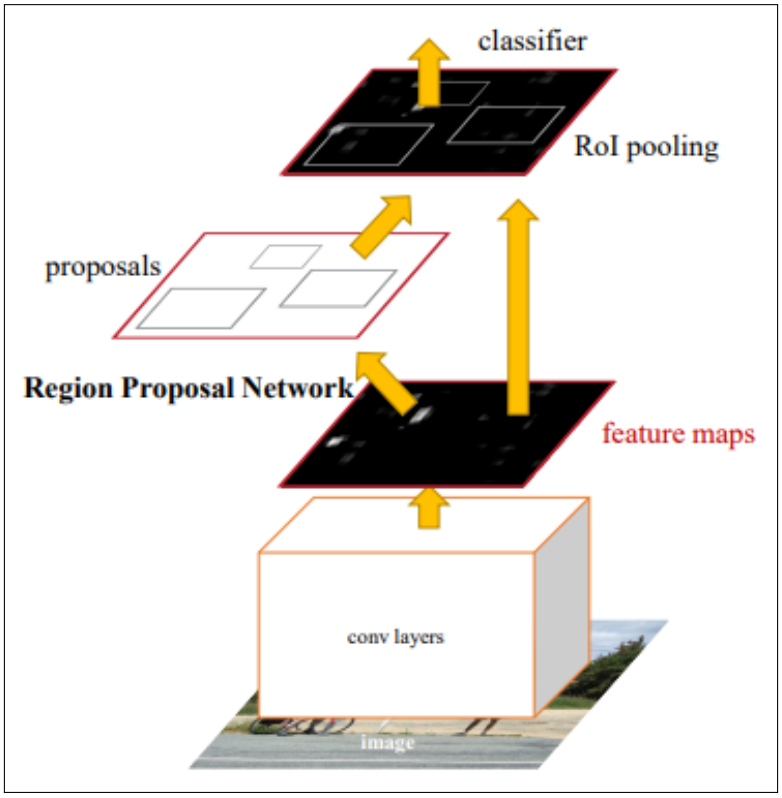

Abbildung 3.4: Architektur FasterRCNN [Ren 15]

laufen (wie ein Sliding Window). Für jede dieser Anchor Boxen wird dabei predicted, ob ein Objekt darin enthalten ist (Vordergrund) oder nicht (Hintergrund). Das Region of Interest Pooling (ROI Pooling) ist der letzte Schritt vor der Klassifikation. Diese Schicht ist dabei ähnlich wie ein einfaches Max-Pooling. Der Output des ROI Pooling kann nun als Input für einen Klassifikator genutzt werden, der die 'Vordergrund'-Objekte nun den richtigen Klassen zuordnet.

PyTorch bietet bereits eine Klasse für FasterRCNN Modelle an, welche auch in diesem Projekt genutzt wird. Für dieses Modell wird ein Dataset im COCO-Format benötigt, welches bereits in 3.1 beschrieben wurde. Das Notebook Training.ipynb enthält alle Schritte zur Erstellung, Anpassung und zum Trainieren des FasterRCNN Modells für Hände.

3.4 Keypoint Head

Sowohl YOLOv5 als auch FasterRCNN haben als Output eine Bounding Box um die Hände, sowie ein Label um welche Art von Hand (links, rechts oder interagierend) es sich handelt. Um nun Keypoints zu bestimmen, wird ein neues Modell benötigt. Dabei handelt es sich um ein Standard Convolutional Network, welches einige Convolutional Layers gefolgt von einigen Dense Layern beinhaltet. Die Convolutional Layers werden dabei aber von einem VGG16-Net übernommen. Die Fully Conected Layers werden

von Grund auf an trainiert, wobei die Anzahl der Outputs der doppelten Anzahl der Keypoints entspricht. Dies liegt daran, dass für jeden Keypoint eine x und eine y-Koordinate bestimmt wird. Tendenziell wäre es möglich für jeden Hand Typen ein eigenes Modell zu trainieren, jedoch wurden lediglich 2 Modelle genutzt: Eines für interagierende Hände (42 Keypoints) und eines für einzelne Hände, egal ob linke oder rechte Hand (21 Keypoints). Über entsprechende Parameter des PyTorch Dataset (siehe Kapitel 3.1), wird das gecroppte Bild sowie die normalisierten Keypoints ausgegeben. Diese werden bei diesen Modellen zum Training genutzt. Als Loss Funktion wird hierbei ein simpler MeanSquarredError genutzt, als Optimizer dient Adam.

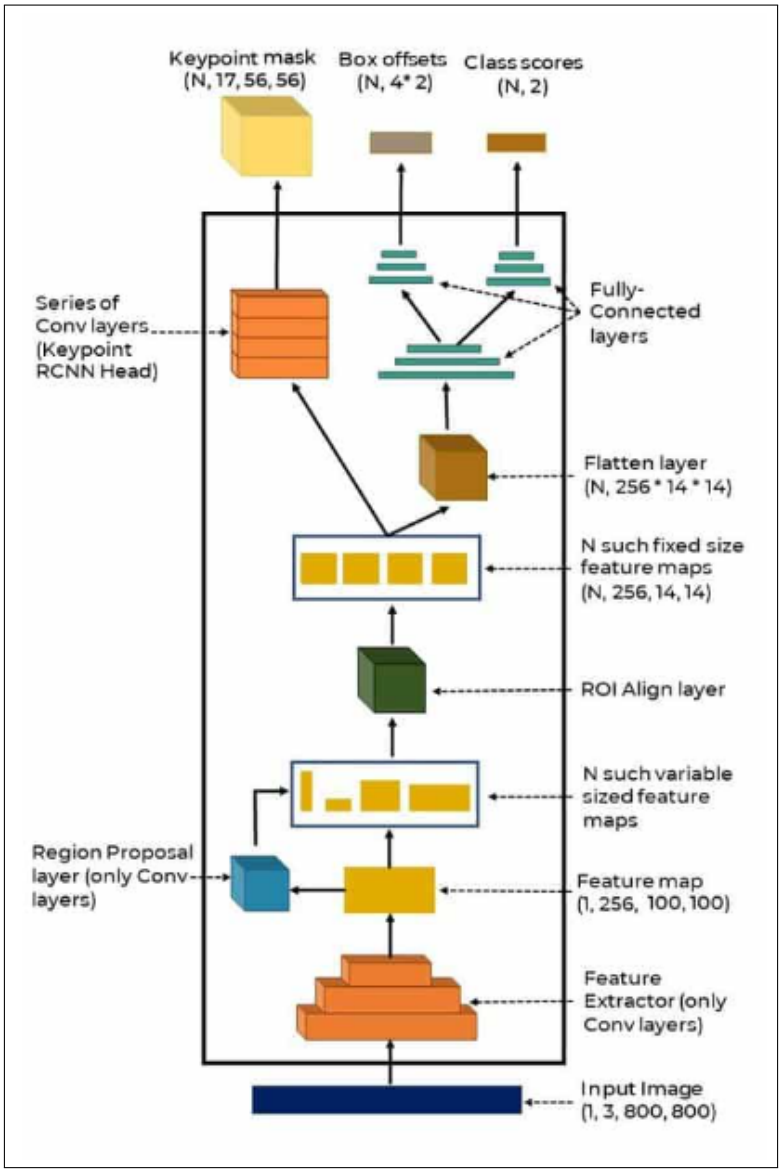

Abbildung 3.5: Architektur KeypointRCNN[Pati 21]

3.5 KeypointRCNN

Eine Erweiterung der bereits beschriebenen FasterRCNN Architektur ist MaskedRCNN [He 17]. Es handelt sich dabei nicht zwingend um eine direkte Weiterentwicklung, da MaskedRCNN nicht für Object Detection sondern für Image Segmentation genutzt wird. Dabei wird jeder Pixel einer entsprechenden Klasse zugeordnet. MaskedRCNN baut jedoch auf der FasterRCNN Architektur auf mit einigen Anpassungen. Anstelle der ROI Pooling Layer besitzt MaskedRCNN eine ROI Align Layer. Diese unterscheiden sich durch die Feinheit der weitergegeben Informationen: ROI Pooling nutzt den maximalen Wert, wohingegen ROI Align einen Mittelwert jedes Feldes bildet. Dadurch gehen weniger Informationen verloren. Nach der ROI Align Layer, gibt es noch zusätzlich einen sogenannten MaskedRCNN Head. Dabei handelt es sich um ein Fully Convolutional Network, bei dem jeder Channel des Outputs einer Klasse entspricht.
Eine Variation dieses Netzwerks ist das KeypointRCNN. Es besitzt die gleiche Architektur, jedoch ist der Output des MaskedRCNN Heads unterschiedlich. Anstatt das jeder Channel eine Klasse repräsentiert, kodiert jeder Channel einen Keypoint. Abbildung 3.5 zeigt die Architektur eines KeypointRCNN bei dem 17 Keypoints eines Menschen aus dem COCO Datensatz predicted werden.
PyTorch bietet auch hier bereits eine Klasse für KeypointRCNNs an. Alle benötigten Schritte zur Modellerstellung und zum Training finden sich ebenfalls in dem Notebook Training.ipynb.

3.6 Training

Das Training fand parallel auf 2 Systemen statt: einem privaten Rechner mit einer RTX 2060 Super und einem Hochschulrechner mit einer RTX 2080 Super. Da der Hochschulrechner eine geteilte Ressource ist, wurde das Training teilweise abgebrochen (z.B. wenn sich ein Student im Rahmen einer Vorlesung angemeldet hatte), wodurch das ganze Training neu gestartet werden musste. Um solche Dinge zu vermeiden, wurde die Evaluierung mit dem Validation Datensatz bei FasterRCNN und KeypointRCNN weggelassen. Beide Modelle trainierten für 4 Epochen, was beim FasterRCNN Modell knapp 100 Stunden und beim KeypointRCNN Modell knapp 200 Stunden benötigte. Auch die Custom Keypoint Modelle wurden nur über 5 Epochen trainiert.

Kapitel 4

Ergebnisse

Die Evaluation wurde primär durch ausprobieren getestet. Das Notebook Visualize.ipynb enthält Funktionen um jede der Varianten zu visualisieren.

4.1 YOLOv5

YOLOv5 ist das Modell, welches in diesem Projekt am wenigsten optimiert wurde. Dies lag an der hohen Trainingszeit, durch die hohe Auslastung des Filtersystems. Es wurde ein YOLOv5s Modell (relativ kleines Modell) über 2 Epochen mit einigen Frozen Layern des Backbones trainiert. Leider sind die Resultate von diesem nicht zu gebrauchen. Es werden anstatt die selbst definierten Labels immer noch Objekte der Kategorie 'Person' erkannt, was vermutlich durch entsteht, dass es sich um ein vortrainierte Modell handelt. Im weiteren Verlauf wurde deshalb FasterRCNN anstatt YOLO genutzt.

4.2 FasterRCNN + Keypoint Head

Durch eine Aneinanderreihung des FasterRCNN Modells und der beiden Custom Keypoint Detection Heads, erhält man die beschriebene Architektur aus Kapitel 3 (Abbildung 3.1). Dabei detektiert FasterRCNN die Bounding Box der Hände sowie ob es sich dabei um einzelne oder interagierende Hände handelt. Nachdem der Bildausschnitt innerhalb der Box ausgeschnitten wurde, wird dieses an entsprechendes Modell weitergegeben. Die Keypoint-Predictions müssen jedoch vor der Visualisierung transformiert werden, da diese in einem Wertebereich zwischen 0 und 1 liegen.
Abbildung 4.1 zeigt einige gute Beispiele für die Erkennung einzelner Hände. Die Keypoints sind relativ nah an den tatsächlichen Keypoints (Gelenken), jedoch oftmals einige Pixel versetzt. Leider ist nicht jedes Ergebnis so gut, wie bei den dargestellten Beispielen. Bei einigen Bildern sind die Keypoints geclustert in einer Ecke. Jedoch ist der Ansatz vielversprechend und könnte durch zusätzliche Optimierungen, wie bspw Training mit mehreren Epochen, verbessert werden.

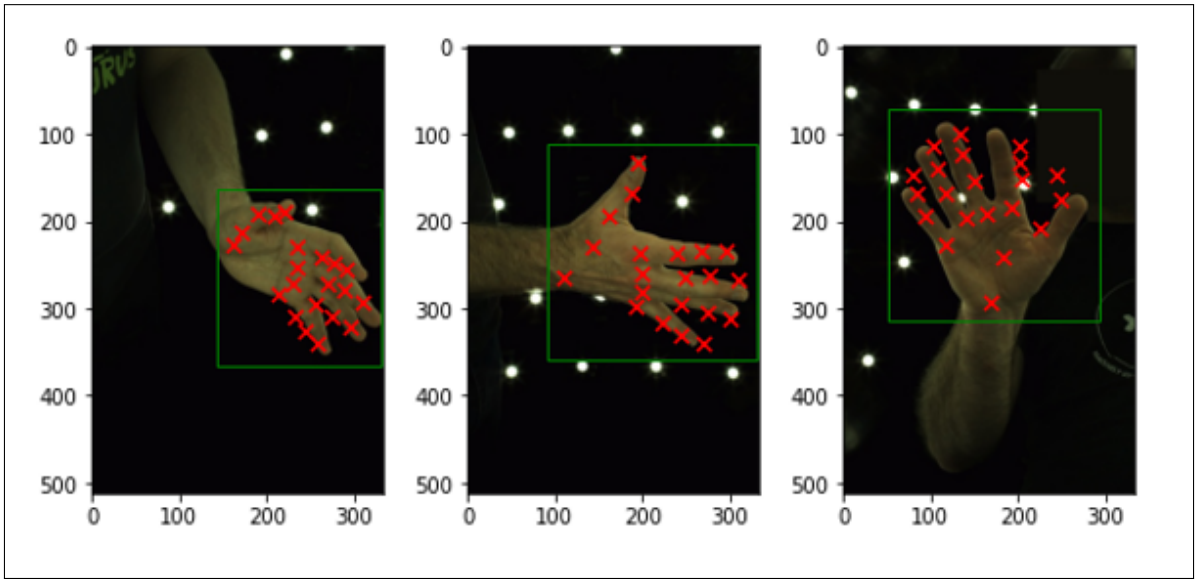

Abbildung 4.1: FasterRCNN+Keypoint Head, einzelne Hand

Anders sieht es jedoch bei interagierenden Händen aus,dort sind die Ergebnisse deutlich schlechter. Wie in Abbildung 4.2 sichtbar, sind die Keypoints oftmals geclustert. Manchmal sind einige Keypoints auch in der oberen linken Ecke. Dies liegt daran, dass bei einigen Labels eine der beiden Hände fehlt, und alle Keypoints deshalb die Koordinaten (0,0) besitzen. Durch mehrere und größere Schichten könnten sich die Ergebnisse verbessern. Eine andere Alternative wäre es, zusätzlich noch Convolution Layers zu nutzen, welche die Keypoints als Heatmap lokalisieren, ähnlich wie bei KeypointRCNN.

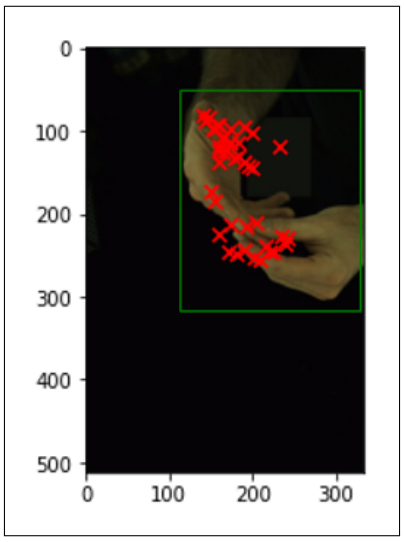

Abbildung 4.2: FasterRCNN+Keypoint Head, interagierende Hände

4.3 KeypointRCNN

Die besten Ergebnisse liefert KeypointRCNN. Da dieses Modell immer die gleiche Anzahl an Keypoints vorhersagt, wurde die Anzahl auf 42, 21 Keypoints pro Hand, gesetzt. Abbildung 4.3 zeigt einige Beispiele von interagierenden Händen. Die Keypoints liegen sehr genau auf den Gelenken, welche durch diese kodiert werden. Sogar wenn eine Hand die andere verdeckt (sichtbar in der Visualisierung rechts), werden die Keypoints der zweiten Hand korrekt dargestellt.
Sofern nur eine Hand im Bild ist, werden die Keypoints in der Regel aufeinander

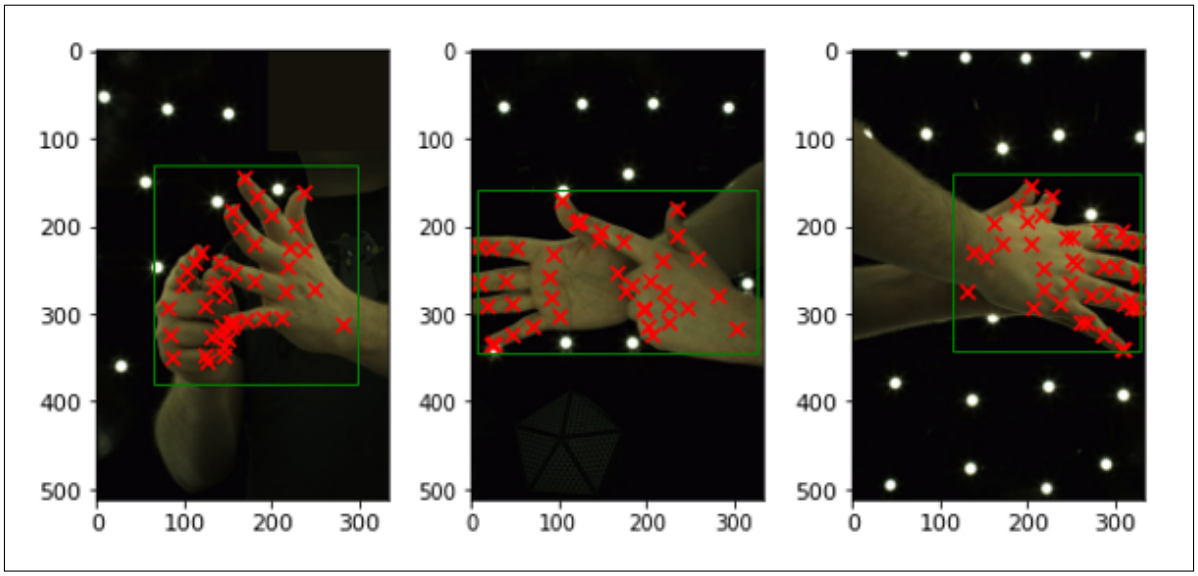

Abbildung 4.3: KeypointRCNN, interagierende Hände

predicted, wie es Abbildung 4.4. Dadurch sind es quasi 2 Keypoints pro Gelenk. In einem Post-Processing Schritt könnte man dies bei Bedarf raus filtern.

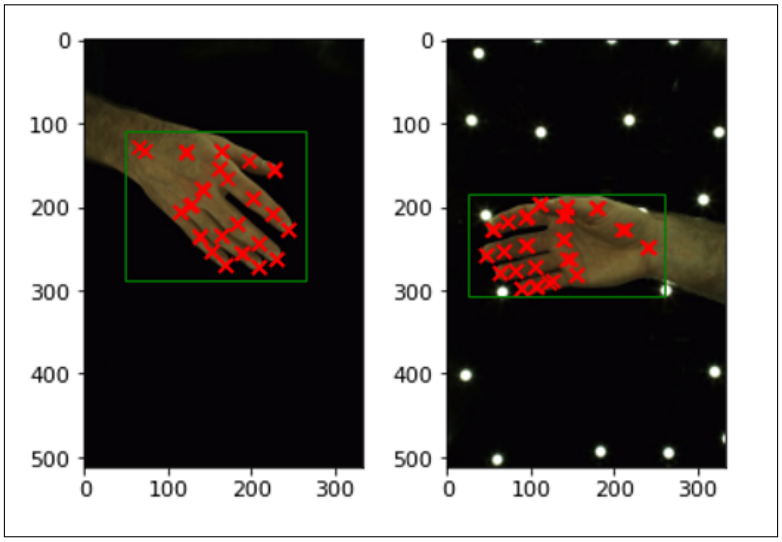

Abbildung 4.4: KeypointRCNN, einzelne Hand

Kapitel 5

Zusammenfassung

KeypointRCNN Modelle sind eine großartige Möglichkeit sehr gute Ergebnisse im Bereich Keypoint Lokalisierung / Pose Estimation zu erhalten. Durch die direkte Integration in PyTorch sind diese Modelle dazu auch sehr einfach zu erstellen und zu trainieren.
Eigene Modelle benötigen mehr Entwicklungsaufwand um vergleichbare Ergebnisse zu liefern, tendenziell sind diese sogar merklich schlechter. Hier bietet es sich an, dass direkt in PyTorch verfügbare FasterRCNN gegenüber YOLO zu bevorzugen. Eine Ausnahme bei der YOLO benutzt werden sollte, ist wenn die Echtzeit-Fähigkeit benötigt wird. YOLO Modelle schaffen teilweise 100+ Frames pro Sekunde, wohingegen je nach Größe des FasterRCNN Modell, die Anzahl der Frames nur Zehntel beträgt.

Literaturverzeichnis

[Cao 18] Z. Cao, G. Hidalgo, T. Simon, S. Wei, and Y. Sheikh. "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields". *CoRR*, Vol. abs/1812.08008, 2018.

[He 17] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. "Mask R-CNN". *CoRR*, Vol. abs/1703.06870, 2017.

[Joch 20] G. Jocher. "ultralytics/yolov5". https://github.com/ultralytics/yolov5, 2020.

[Lin 14] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. "Microsoft COCO: Common Objects in Context". *CoRR*, Vol. abs/1405.0312, 2014.

[Lin 20] F. Lin, C. Wilhelm, and T. R. Martinez. "Two-hand Global 3D Pose Estimation Using Monocular RGB". *CoRR*, Vol. abs/2006.01320, 2020.

[Moon 20a] G. Moon. "InterHand2.6M". https://mks0601.github.io/InterHand2.6M/, 2020.

[Moon 20b] G. Moon, S.-I. Yu, H. Wen, T. Shiratori, and K. M. Lee. "InterHand2.6M: A Dataset and Baseline for 3D Interacting Hand Pose Estimation from a Single RGB Image". In: *European Conference on Computer Vision (ECCV)*, 2020.

[Pati 21] C. Patil and V. Gupta. "Human Pose Estimation using Keypoint RCNN in PyTorch". https://learnopencv.com/human-pose-estimation-using-keypoint-rcnn-in-pytorch, 2021.

[Redm 16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[Redm 17] J. Redmon and A. Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

[Ren 15] S. Ren, K. He, R. B. Girshick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". *CoRR*, Vol. abs/1506.01497, 2015.

[Yuan 17] S. Yuan, Q. Ye, B. Stenger, S. Jain, and T. Kim. "BigHand2.2M Benchmark: Hand Pose Dataset and State of the Art Analysis". *CoRR*, Vol. abs/1704.02612, 2017.

[Zimm 19] C. Zimmermann, D. Ceylan, J. Yang, B. C. Russell, M. Argus, and T. Brox. "FreiHAND: A Dataset for Markerless Capture of Hand Pose and Shape from Single RGB Images". *CoRR*, Vol. abs/1909.04349, 2019.

Abbildungsverzeichnis

Listingverzeichnis