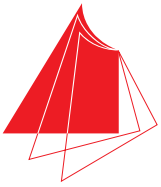

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

2D-Keypoint-Detektion mit Mask R-CNN

Lukas Stürmlinger

Projektarbeit im Wintersemester 2020/21
Studiengang Informationstechnik (Master)

Fakultät Elektro- und Informationstechnik
Hochschule Karlsruhe

8. Februar 2021

Betreuer
Prof. Dr. Manfred Strohrmann
Dr. Kawther Aboalam

Inhaltsverzeichnis

1 Projektbeschreibung

Die vorliegende Projektarbeit ist Teil eines Gesamtprojekts in Kooperation der Hochschule Karlsruhe mit einer externen Firma. Dieser Abschnitt führt in das Gesamtprojekt ein und nennt die Ziele der Projektarbeit.

1.1 Gesamtprojekt

Für sportmedizinische Bewegungsanalysen werden zur Zeit physische Marker verwendet, die dem Sportler am Körper angebracht werden müssen. Ziel dieser Analysen ist die Vorbeugung von Haltungsschäden und die Verbesserung der sportlichen Leistung durch Korrekturen von Bewegungsabläufen.

In Zusammenarbeit mit der Hochschule Karlsruhe soll eine Software entwickelt werden, die Analysen ohne angebrachte Marker durchführen kann. Dazu sollen *Deep-Learning*[1]-Algorithmen bestimmte Gelenkpositionen, sogenannte *keypoints* auf digitalen Bildern markieren und so die Bewegungsanalyse erleichtern.

1.2 Ziele und Aufgaben der Projektarbeit

Die Projektarbeit startet zeitgleich mit dem Gesamtprojekt. Ziel ist es, ein lauffähiges Programm zu erstellen, das die Inferenz und das Training korrekt auf dem COCO-Datensatz ausführt. Als Grundlage dient ein Softwareprojekt zur *keypoint*-Detektion, das in Python mit den Modulen Tensorflow und Keras programmiert wurde. Inzwischen sind diese beiden Module zusammengeführt und verbessert worden, sodass Anpassungen des veralteten Softwareprojekts vorzunehmen sind.

Ein weiteres Ziel ist es, den Umgang mit dem Software-Projekt und den verwendeten Programmen und Werkzeugen in Form von Anleitungen zu erklären, um so die Einarbeitungszeit für zukünftige Arbeiten zu verkürzen. Gerade beim Programmieren mit Python können sich wegen kontinuierlicher Änderungen von Modulen Probleme beim Ausführen ergeben. Deshalb ist eine genaue Beschreibung der Entwicklungsumgebung unverzichtbar.

Aus den gesetzten Zielen ergeben sich folgende Aufgaben:

[1] Fachbegriffe, die in der einschlägigen englischen Literatur verwendet werden, werden der besseren Verständlichkeit nicht übersetzt, sondern kursiv gesetzt.

Das Mask R-CNN ist ein komplexes Netz, welches sich über mehrere Jahre Forschungsarbeit entwickelt hat. Ein gutes Grundverständnis über Prinzipien und Methoden der Arbeit mit neuronalen Netzen in der Bildverarbeitung ist somit essenziell für der Implementierung.

Für die Arbeit an der Software ist eine Einarbeitung in die Arbeitsweise von Tensorflow und Keras erforderlich. Außerdem müssen vertiefte Kenntnisse in der Programmiersprache Python erworben und das Arbeiten mit sog. virtuellen Programmierumgebungen erlernt werden. Für eine komfortable Versionsverwaltung der Software ist außerdem eine Einarbeitung in GitHub Desktop nötig.

Zudem müssen vorhandene Programme des Software-Projekts mit Kommentierungen ergänzt und verschiedene Konfigurationen getestet und dokumentiert werden. Auch die Einrichtung von Tensorflow erfordert bei Verwendung einer GPU einige zu beachtende Schritte, die ermittelt und getestet werden müssen.

2 Zweidimensionale Keypoint-Detektion von Menschen

Das Erkennen von *keypoints* an Menschen findet in vielen Branchen Verwendung u. a. in der Film- und Gamingindustrie, sowie bei der Sportanalyse im Profibereich. Wegen der schnellen wissenschaftlichen Fortschritte in den letzten Jahren ist eine Erkennung mit Hilfe von *machine learning* in Echtzeit technisch möglich geworden.

Prinzipiell unterscheidet man zwei Strategien zur Erkennung von *keypoints* aus Bildern. Bei der *bottom-up*-Methode werden zuerst sämtliche *keypoints* gesucht und danach Personen zugeordnet. Diese Methode ist sehr schnell, weil die Detektionsgeschwindigkeit praktisch nicht von der Personenanzahl in einem Bild abhängt. Im Gegensatz dazu werden bei der *top-down*-Methode erst Personen in einem Bild gesucht und daraufhin die zugehörigen *keypoints* detektiert[1]. Mask R-CNN verfolgt letztere Strategie.

3 Mask R-CNN

3.1 Die Entwicklungsstufen von Mask R-CNN

Ein Blick auf die zeitlichen Entwicklungsstufen soll dem besseren Verständnis der grundsätzlichen Arbeitsweise dieses neuronalen Netzes dienen.

2014 stellten Gershwick et al [2] ein Regionen-basiertes Faltungsnetz (*region-based convolutional network, R-CNN*) zur Objekt-Detektion vor. Objekt-Detektion bedeutet, dass es Objekte auf Fotos klassifiziert und diese mit einem Rechteck (*bounding-box*) markiert. Das Netz arbeitet Regionen-basiert. Anhand grober Regionen-Vorschläge (*region proposals*), die mit einer klassischen Bildverarbeitungsmethode, z. B. *selectiv search* generiert werden, berechnet das Netz für jede vorgeschlagene Region (*region of interest*, *RoI*) dessen Objektklasse und *bounding-box* (Abbildung 1 links). Die finale *bounding-box* ist eine Verfeinerung des Regionen-Vorschlags (*bounding-box regression*).

Nachteilig an dieser Architektur ist die Verarbeitungsgeschwindigkeit. Weil bei einem einzigen Bild unter Umständen tausende Regionen, die potenziell Objekte enthalten, vorgeschlagen werden und jede Region ein tiefes Faltungsnetz durchlaufen muss, entsteht ein hoher Rechenaufwand.

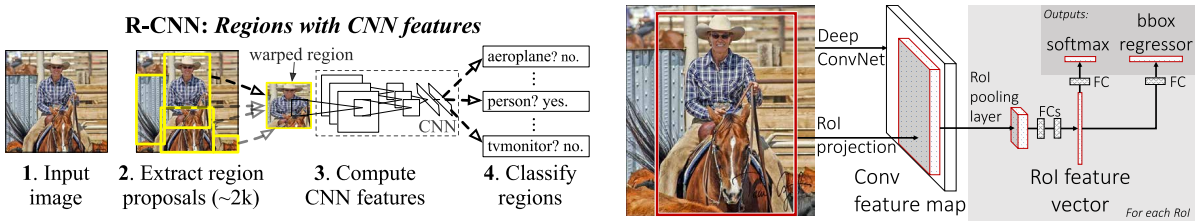

Abbildung 1: Methode von R-CNN [2] und und Fast R-CNN [3]. Während beim R-CNN jedes einzelne RoI ein tiefes Faltungsnetz durchläuft, muss es das gesamte Bild beim Fast R-CNN nur einmal.

Dieses Problem wird vom Nachfolger, dem Fast R-CNN [3] gelöst. Anstatt tausender Regionen-Vorschläge durchläuft nun das gesamte Bild ein Faltungsnetz nur einmal. Die RoIs werden auf die resultierende *feature map* projiziert und wieder einzeln klassifiziert bzw. regressiert (Abbildung 1 rechts). Genau wie das R-CNN ist das Fast R-CNN von guten Regionen-Vorschlägen abhängig.

Der Nachfolger, das Faster R-CNN [4] generiert RoIs mit Hilfe eines *sliding window*, das Regionen einer *feature map* als „Objekt“ oder „Hintergrund“ klassifiziert und *bounding-boxen* generiert. Das *sliding window* ist ein flaches Faltungsnetz, das trainiert werden kann. Es wird auch *region proposal network* (*RPN*) genannt. Somit arbeitet das gesamte Faster R-CNN mit *machine learning*-Algorithmen.

Das Elegante an dem Faster R-CNN ist die effiziente Nutzung der *backbone feature maps* für das RPN und das Detektions-Netzwerk. Ein Bild durchläuft somit nur einmal ein tiefes Faltungsnetz, wonach aus dessen *feature maps* sowohl Regionen-Vorschläge als auch Objektpositionen berechnet werden (siehe Abbildung 2).

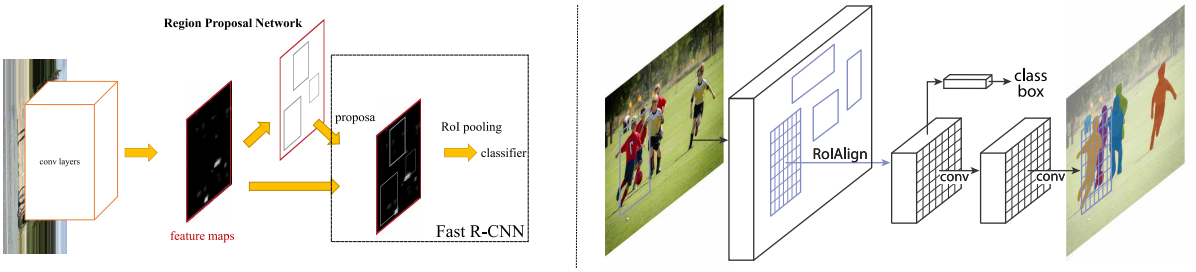

Abbildung 2: Methode von Faster R-CNN und Mask R-CNN: links: Faster R-CNN [4, S. 3] generiert Regionen-Vorschläge aus *feature maps*. rechts: Mask R-CNN [5, S. 1] mit Blick auf den Masken-Zweig.

Das Mask R-CNN [5] ist im wesentlichen ein Faster R-CNN, das um einen Masken-Zweig erweitert wurde. Eine Maske ist eine binäre Karte, die auf Pixelebene Objekte in einem Bild mit Einsen markiert. Die Null stellt den Hintergrund dar. Jede Objektinstanz erhält eine eigene Maske.

Weitere Änderungen gegenüber Faster R-CNN ist die Verwendung von *RoIAlign* anstatt *RoIPooling*. So werden Rundungen bei der Extraktion von RoIs aus der *backbone feature map* vermieden, die gegenüber der neuen Methode zu schlechteren Genauigkeiten bei den Masken führt. Eine effektivere Nutzung der *backbone feature maps* wird durch die Verwendung eines *feature pyramide networks (FPN)* erzielt.

Im Folgenden werden das *feature pyramid network*, das *region propsal network*, *RoIAlign* und die Umsetzung des Masken-Zweigs detaillierter erklärt. Es wird gezeigt, dass der Masken-Zweig die Markierung von *keypoints* ermöglicht.

3.2 Feature Pyramid Network (FPN)

Backbone feature maps werden durch die sog. *backbone* erzeugt. Diese ist ein tiefes Faltungsnetz und hat die Aufgabe, *features* aus einem Bild zu extrahieren. Features sind charakteristische Informationen über ein Bild [6, S. 97]. Es ist daher wichtig, möglichst viele davon zu finden, um Objekte sicher zu detektieren. Nachfolgend werden *feature mapsm maps* und *features* gleichbedeutend verwendet.

In Mask R-CNN arbeitet ein ResNet als *feature* Extraktor. Das Netz besteht aus fünf Stufen (*stages*). Nach jeder Stufe verdoppelt sich die Anzahl der *features*, während sich Breite und Höhe halbieren [7, S. 473]. Jede Stufe beinhaltet mehrere Faltungsschichten. Abbildung 3 zeigt die Abmessung der *feature maps* eines ResNet. Dieses wird auch in dem später beschriebenen Software-Projekt verwendet.

Wegen der abnehmenden Größe und Tiefe der *feature maps* wird diese Architektur als pyramidal bzw. als *feature map*-Pyramide bezeichnet. Am Fuße haben die *features* eine hohe Auflösung, beinhalten jedoch nur semantisch schwache Merkmale (z. B. einfache Formen). Zur Spitze hin nimmt die Auflösung ab und die Aussagekraft der Merkmale nimmt zu (z. B. komplexe Muster) [8, S. 2]. Faster R-CNN nutzt nur die *features* aus Stufe 4 (C_4). Diese hat eine relativ geringe Auflösung und macht eine genaue Detektion von kleinen Objekten schwierig.

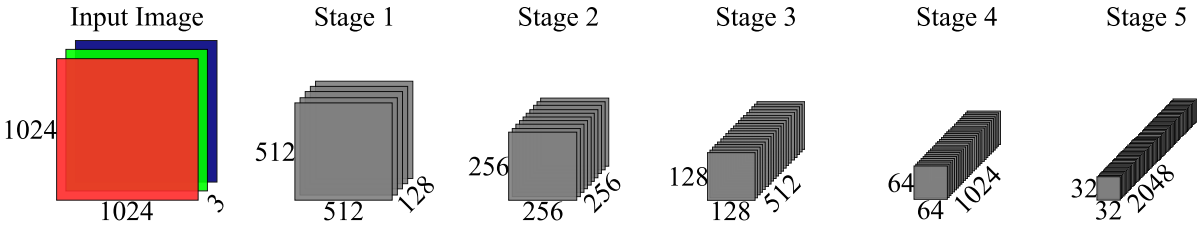

Abbildung 3: Die Stufen eines ResNet.

Um dieses Problem zu mildern, haben Lin et al [8] ein *feature pyramid network* entwickelt, das die *feature maps* der einzelnen Stufen kombiniert. Sie verwenden dazu eine *top-down* Architektur mit lateralen Verbindungen.

Abbildung 4 beschreibt die Erzeugung der *pyramid feature maps* (P_2 bis P_5) aus den ResNet-*features* (C_2 bis C_5). Links ist der *bottom-up*-Zweig zu sehen. Dieser entspricht der eigentlichen Erzeugung der ResNet-*features*. Dann werden rechts die *pyramid feature maps* von Oben nach Unten (*top-down*) gebildet. Zunächst wird die Tiefe von C_5 mittels 1×1-Faltung von 2048 auf 256 reduziert. Dies entspricht nun P_5. P_4 wird erzeugt, indem P_5 (zweifach vergrößert) auf C_4 addiert wird. Auch bei C_4 wurde die Tiefe vor dem Addieren auf 256 reduziert.

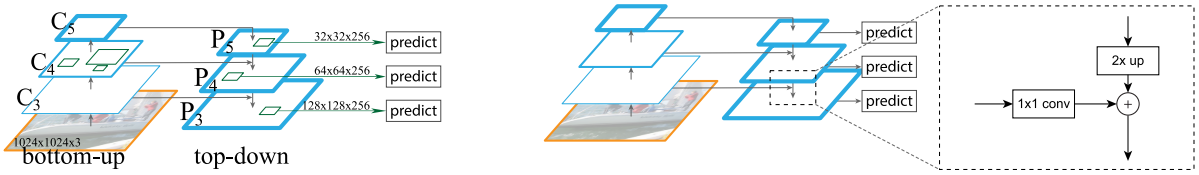

Abbildung 4: Feature Pyramid Network: links: bottom-up und top-down Pfad, rechts: Details der lateralen Verbindungen. Zur besseren Ansicht werden nicht alle vier *feature maps* gezeigt.[8]

P_3 und P_2 werden auf die gleiche Weise erzeugt. Sie sind die Summe der darüber liegenden *features*. Sie besitzen unterschiedliche Auflösungen, jedoch durchgehend hohe semantische Aussagekraft.

Die Backbone liefert, wie oben beschreiben, *features* für das RPN und den Detektor. Im Gegensatz zu Faster R-CNN, das nur C_4 verwendet, stehen mit dem FPN vier *feature maps*

zur Verfügung. Die Auswahl der jeweiligen *map* ist abhängig von der Objektgröße im Bild. In Unterabschnitt 3.4 wird näher darauf eingegangen.

3.3 Region Proposal Network (RPN)

Ein RPN generiert aus einem Bild einen Satz rechteckiger Regionen-Vorschläge. Jeder Vorschlag enthält u. a. einen *objectness score*, der die Wahrscheinlichkeit für das Vorhandensein eines Objekts angibt.

Die Erzeugung der Regionen-Vorschläge wird wegen der besseren Darstellbarkeit anhand einer *backbone* ohne FPN erklärt (d. h. wie im Faster R-CNN).

Zunächst wird eine 3×3-Faltung durchgeführt (Abbildung 5). Dies reduziert die C_4 *feature map* von 1024 auf 256. In der Literatur [4, S. 3] wird diese Faltung *sliding-window* genannt, weil der 3×3-Filter wie ein Fenster aussieht, der zeilenweise über die *feature map* gleitet. Für jede Fensterposition werden $k = 9$ Vorschläge gemacht. Basis bilden sog. Ankerboxen (*anchors*), die es in drei unterschiedlichen Größen mit jeweils drei verschiedenen Seitenverhältnissen gibt (9 Ankerboxen). Diese heißen so, weil sie wie ein Anker auf dem *sliding-window* sitzen.

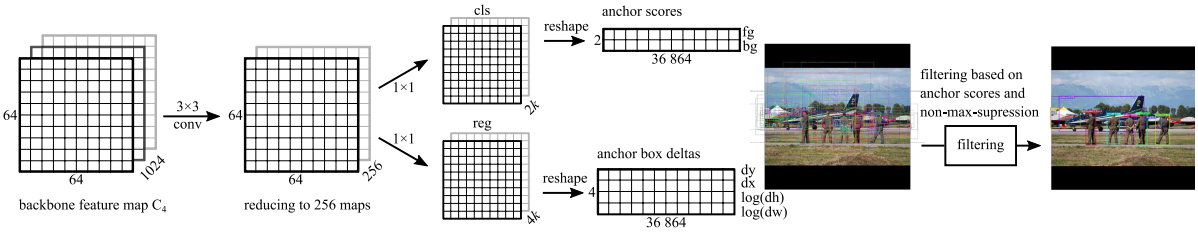

Abbildung 5: Region Proposal Network: Mit Hilfe von Ankerboxen werden eine *features* systematisch nach Objekten abgesucht. Fotos der Ankerboxen aus [9].

Bildlich stelle man sich 9 verschieden große Rechtecke vor, die nacheinander auf einem festen Raster über ein Bild geschoben werden, beginnend mit Rechteck 1. Für jede Position bewertet man, ob und wie gut das Rechteck ein Objekt im Bild bedeckt. Dies drückt das RPN mit dem *objectness score* aus, der die Wahrscheinlichkeit für ein Objekt und nicht-Objekt innerhalb einer Ankerbox angibt. Nachdem alle Ankerboxen über die *feature map* geglitten sind, gibt es für jede Position 9 *scores*.

Außerdem berechnet das RPN für jede Position und jede Ankerbox deren Abweichung (*anchor-box deltas*) zu einem Objekt. Weil die Ankerboxen auf einem festen Raster über die *feature map* gleiten, bedecken sie ein Objekt im Allgemeinen nie ganz genau. Die Abweichungen werden relativ zu den Ankerboxen gemacht und bestehen aus vier Zahlen: Abweichung des

Mittelpunkts ($\mathrm{d}y$ und $\mathrm{d}x$) und der Höhe bzw. Breite ($\ln \mathrm{d}h$ und $\ln \mathrm{d}w$). Diese werden später mit den zugehörigen Ankerboxen verrechnet.

Zur einfacheren Darstellung des *objektness scores* und der *anchor-box deltas* wurde eine Umformung vorgenommen (Abbildung 5 Mitte). Dadurch wird deutlich, dass es insgesamt $64 \times 64 \times 9 = 36\,864$ Bewertungen gibt. Diese müssen noch gefiltert werden, sodass nur diejenigen Ankerboxen übrig bleiben, die tatsächlich Objekte einschließen. Dies wird anhand hoher *objectness scores* und *non-maximum-suppression (NMS)* durchgeführt. Abbildung 5 zeigt Ankerboxen vor und nach der Filterung. Diese Boxen werden als RoIs an Detektor und Masken-Zweig weitergegeben.

Anzumerken ist, dass die Ankerboxen nur während des Trainings eine Rolle spielen. Sie werden mit den annotierten Boxen verglichen, mit denen die Filter des *sliding-windows* gelernt werden. Bei der Inferenz entsteht die Bewertung nur aufgrund der gelernten Gewichte.

Besitzt die Backbone ein FPN, stehen fünf *feature maps* (P_2 bis P_6) zur Verfügung. Tabelle 1 vergleicht beide Varianten. Für jede Ebene der Pyramide werden Ankerboxen mit fester Größe, jedoch variierendem Seitenverhältnis generiert. Die weitergegebenen RoIs enthalten zusätzlich zu ihrer Abmessung und Bildposition die Information, aus welcher Ebene eine Bewertung stammt.

Tabelle 1: Vergleich zwischen den RPN-Varianten

	single scale	FPN
Backbone feature maps	C_4 $64 \times 64 \times 1024$	P_2 bis P_6 $254 \times 256 \times 256$ $128 \times 128 \times 256$ $64 \times 64 \times 256$ $32 \times 32 \times 256$ $16 \times 16 \times 256$
Anchor aspect ratios	3	3
Anchor scales	3	1
Anchors per location (k)	9	3
Number of anchors	$64 \cdot 64 \cdot 3 = 36\,864$	$(256^2 + 128^2 + 64^2 + 32^2 + 16^2) \cdot 3$ $= 261\,888$

3.4 RoIPooling und RoIAlign

RoIPooling und *RoIAlign* extrahieren aus den *features* Regionen-Vorschläge und skalieren sie auf eine feste Größe. Dies ist notwendig, weil Detektor und Masken-Zweig eine feste Eingangsgröße von 7×7 bzw. 14×14 erwarten.

RoIAlign ist eine Weiterentwicklung von *RoIPooling*. Es verzichtet auf Rundungen, was vor allem für eine genaue Maskierung notwendig ist [5, S. 1]. Beide Verfahren werden in Abbildung 6 gegenübergestellt.

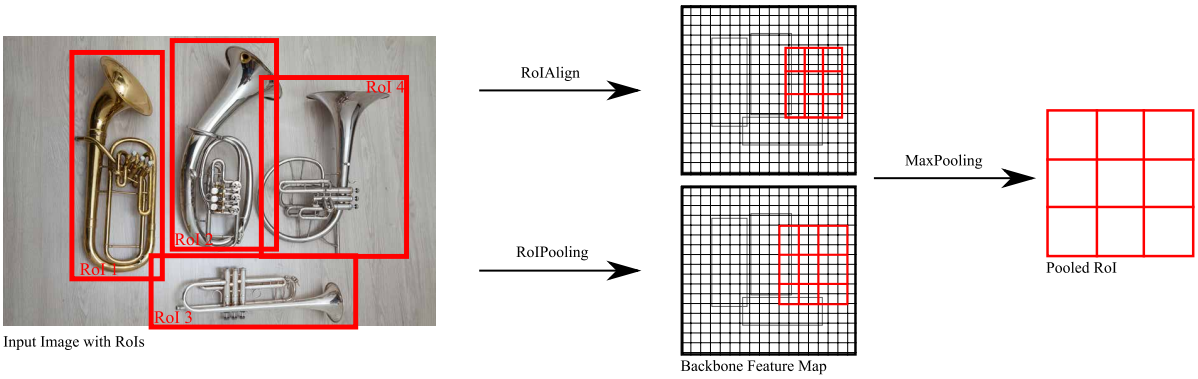

Abbildung 6: RoIAlign vs. RoIPooling: RoIPooling rundet bei der RoI-Box und beim Gitter. RoIAlign interpoliert die Werte im Gitter.

Die in Bildkoordinaten vorliegenden RoI-Boxen werden auf die *backbone feature map* projiziert. Im Beispiel ist die Abmessung von C_4 16-mal kleiner als die des Eingangsbildes, deshalb sind die Koordinaten durch 16 zu teilen ($1024 \div 64 = 16$). Im allgemeinen Fall sind die sich ergebenden Koordinaten gebrochene Zahlen. *RoIPooling* wandelt hier, im Gegensatz zu *RoIAlign*, in ganze Zahlen um, sodass die Box genau in das Raster der *feature map* fällt (vgl. Abbildung 6 oben). Damit eine feste Ausgangsgröße entsteht, wird jede Box gerastert (im Beispiel 3×3). Auch hier rundet *RoIPooling* auf ganze Pixel. Zum Schluss wird das Maximum aus jedem Feld gespeichert. Hier muss *RoIAlign* interpolieren, weil das Gitter der RoI-Box Pixelteile bedeckt. Dieser Vorgang wird für C_4, 1024-mal durchgeführt und schließlich zur Weiterverarbeitung an Detektor und Masken-Zweig weitergegeben.

Arbeitet nach der *backbone* ein FPN, muss entschieden werden, aus welcher *pyramid feature map* (P_2 bis P_5) ein RoI extrahiert wird. Lin et al [8, S. 4] verwenden dazu eine Formel, die in Abhängigkeit der RoI-Größe eine Ebene in der Pyramide auswählt. Je kleiner ein Objekt im Bild, desto höher sollte die Auflösung der *feature map* sein. D. h. es wird eine weiter unten in der Pyramide liegende *map* ausgewählt. Da ein Bild viele verschieden große RoIs enthalten kann, wird aus verschiedenen Stufen extrahiert. Dies ist in Abbildung 4 links angedeutet.

3.5 Keypoint-Detektion mit dem Masken-Zweig

Eine Maske weist jedem Pixel eines Bildes eine Objektklasse zu. Hierbei unterscheidet man zwischen semantischer Segmentierung (*semantic segmentation*) und Instanz-Segmentierung (*instance segmentation*). Ein Beispiel für semantische Segmentierung zeigt Abbildung 7.

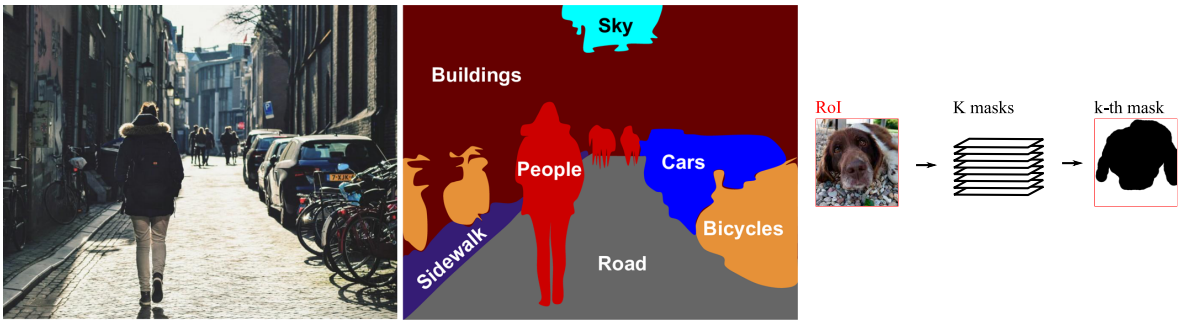

Abbildung 7: Segmentierung und Maskierung: links und Mitte: semantische Segmentierung [7, S. 493], rechts: Maskenprädiktion mit Mask R-CNN.

Alle Objektklassen sind zu einer Maske zusammengefasst. Einzelne Instanzen sind nicht unterscheidbar. Bei der Instanz Segmentierung besäße z. B. jedes Fahrrad eine eigene Maske (Abbildung 2 rechts und Mitte).

Mask R-CNN führt eine Instanz-Segmentierung durch. Jede vorgeschlagene Region enthält nur eine Objektinstanz, für die dann die Maske berechnet wird. He et al heben heraus, dass der Erfolg des Masken-Zweigs darauf fußt, dass er unabhängig vom Detektor arbeitet. So werden parallel zur Detektion K Masken für jedes RoI berechnet. Dabei steht K für die Anzahl der Objektklassen (z. B. Hund, Katze, Maus, . . .). Ausgewählt wird dann nur die k-te Maske, wobei k die von Detektor ermittelte Klasse ist (Abbildung 7 rechts).

Für die *keypoint*-Detektion wird ebenfalls der Masken-Zweig benutzt. RoIs enthalten jedoch nur die Objektklasse „Person“. Hierfür werden wieder K Masken berechnet. K steht dieses Mal für die Anzahl der *keypoints* (z. B. linke Schulter, rechte Schulter,. . .). Dabei enthält jede Maske nur ein einziges Pixel, das als Vordergrund markiert ist und damit die Position des jeweiligen *keypoint* repräsentiert.

4 COCO-Datensatz

COCO ist ein Bilddatensatz mit Annotationen. COCO steht für *common objects in context*. Es sind komplexe Alltagssituationen mit gewöhnlichen Objekten in ihrer natürlichen Umgebung

zu sehen. Dies steht im Kontrast zu Fotos, die nur einzelne Objekte zeigen, wie z. B. Produktfotos oder Portraits [10, S. 1]. Bei den 80 unterschiedlichen Objektklassen haben Personen für die *keypoint*-Detektion eine besondere Bedeutung. Neben *bounding-box*, Klassenlabel und Masken für jede Instanz sind auch 17 *keypoints* pro Person annotiert. Abbildung 8 zeigt links ein Bild mit Personen. Zur besseren Erkennbarkeit sind die *keypoint* mit Linien verbunden.

Abbildung 8 zeigt rechts einen Auszug aus der Annotierungsdatei für *keypoints*. Unter „Annotations“ werden *keypoint*-Koordinaten und -Anzahl einer Instanz angegeben. In „Categories“ befindend sich die Bezeichnungen z. B. „linke Schulter“, usw.

```
annotation{
  "keypoints"          : [x1,y1,v1,...],
  "num_keypoints"      : int,
  "[cloned]"           : ...,
}

categories[{
  "keypoints"          : [str],
  "skeleton"           : [edge],
  "[cloned]"           : ...,
}]

"[cloned]": denotes fields copied from object detection annotations defined above.
```

Abbildung 8: COCO Datensatz: links: Visualisierung der *keypoints* [11], rechts: Aufbau der Annotationen [12].

COCO enthält ca. 330 000 Bilder mit ca. 250 000 Personen, die für das Training eines Mask R-CNN Netzes verwendet werden können. Für das Einlesen und Verarbeiten der Annotationen stellen die Betreiber die „PyCOCOTools“ API zur Verfügung. Installationshinweise befinden sich in Abschnitt 6.

Um *deep computer vision* voranzutreiben, werden unter den Betreibern jährlich Wettbewerbe veranstaltet. Deren Ziel ist es ein Netz zu trainieren, das z. B. *keypoints* detektiert. Zu diesem Zweck lassen sich mit den PyCOCOTools Modelle bewerten. Die genauen Bewertungskriterien können auf der COCO-Webseite nachgelesen werden.

5 Das Softwareprojekt Mask R-CNN

5.1 Einleitung

He et al [5] haben ihr Netz mit der API „Caffe“ implementiert. Wegen der immer größeren Beliebtheit von Tensorflow und Keras veröffentlichte Waleed Abdulla [9] 2017 sein Projekt mit den genannten APIs. Es ist, abgesehen von kleinen Änderungen, stark am Original orientiert. Das Projekt ist sehr umfangreich und bietet komfortable Möglichkeiten, die *backbone*

oder Hyperparameter zu variieren, Modelle zu analysieren und Ergebnisse zu visualisieren. Außerdem ist das Training mit COCO-Bildern und -Annotationen vorbereitet.

Chiao Li alias Superlee506[13] modifizierte Abdullas Projekt für die *keypoint*-Detektion. Er fügte u. a. den *keypoint*-Zweig hinzu und ermöglichte es, *keypoint*-Annotationen einzulesen.

Adam Kelly alias akTwelve[14] stellte Abullas Mask R-CNN für die Verwendung mit Tensorflow 2.0 und tf.keras um.

5.2 Tensorflow und Keras

Keras ist eine high-level Software API, die es ermöglicht, neuronale Netze zu bauen, zu trainieren und zu evaluieren. Ursprünglich war Keras ein eigenes Modul und nutzte für seine Tensoroperationen u. a das Backend Tensorflow. Seit Ende 2016 ist Keras ein Teil des Tensorflow Moduls (tf.keras), was u. a. Versionskompatibilitäten erheblich erleichtert.

Beim gesamten Programmierungsprozess werden Tensoren verwendet. Das sind spezielle Tensorflow Objekte, die den gleichen Verarbeitungsregeln, wie Numpy-Arrays folgen. Bilder und *feature maps* werden durch sie dargestellt. Mit einer GPU lassen sie sich zudem effizient verarbeiten.

5.3 Ausgewählte Funktionen von tf.keras

Grundsätzlich ist die Vorgehensweise für das Erstellen eines Modells in 4 Schritte unterteilt:

1. *layer:* Festlegen der Schichten eines Modells
2. *compile:* Festlegen der Loss-Funktion, des Optimierers und der Metrik
3. *fit:* Trainieren mit dem Trainingsdatensatz
4. *predict:* Testen des Modells durch Inferenz

Je nach Komplexität der Modellstruktur bietet tf.keras mehrere Möglichkeiten den Aufbau der Schichten zu programmieren:

1. Sequential API
2. Functional API
3. Subclassing API

Die Sequential API ist für einfache Netze geeignet, deren Schichten ausnahmslos sequenziell, also hintereinander geschaltet sind. Man gibt der Reihe nach eine Auswahl vorgefertigter Schichten an. Vorgänger- und Nachfolgeschicht sind durch die Reihenfolge der Angabe festgelegt. Wegen der verzweigten Struktur des Mask R-CNN wird diese Programmierungsart im Softwareprojekt nicht verwendet. Sie wird hier nur der Vollständigkeit halber erwähnt.

tf.keras bietet verschiedene Arten von Schichten in Form von Objektklassen an. Weil die Objekte wie Funktionen mit Ein- und Ausgängen aufgerufen werden, nennt man diese Art der Programmierung Functional API. Anhand Listing 1 wird die Verwendung erklärt.

Listing 1: Functional API [7, S. 309]

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Listing 1 zeigt die Umsetzung einer Netzstruktur, die nicht ausschließlich sequenziell ist. Zu Beginn wird eine Eingangsschicht instanziiert, mit der die Form des Trainingsdatensatzes angegeben wird. Es folgen zwei dichte Schichten. Hierbei werden deren Eigenschaften, wie z. B. die Anzahl der Neuronen und die Aktivierungsfunktion in der ersten Klammer angegeben, der Eingang bzw. die Vorgängerschicht in der zweiten Klammer.

Der *concatenate layer* fügt hier im Beipiel zwei Tensoren zusammen. `input_` und `hidden2` werden daher beide als Eingang der Schicht angegeben. Zuletzt wird die gesamte Struktur in einem Modell-Objekt gespeichert.

Soll ein Modell spezielle Schichten enthalten, die nicht standardmäßig in tf.keras implementiert sind, bietet die Subclassing API Möglichkeiten, diese zu erstellen. Dafür erbt der *Custom Layer* vom dafür vorgesehenen Keras Layer Objekt. Ist dieser fertig programmiert, kann er auf die oben erklärte Weise verwendet werden.

In einem Custom Layer sind u. a. die Methoden `__init__()` und `call()` für die innere Struktur der Schicht verantwortlich. `__init__()` nimmt Einstellungsparameter und `call()` den Eingangstensor auf. Listing 2 zeigt ein einfaches Beispiel.

Listing 2: Custom Layer via subclassing

```
class CustomLayer(keras.layers.Layer):
    def __init__(self, factor):
        super(CustomLayer, self).__init__()
        self.factor = factor
```

```
    def call(self, inputs):
        a = inputs[0]
        b =inputs[1]
        c = (a+b)*self.factor
        return[a, c]

output = CustomLayer(7)(input_)
```

Zunächst wird eine Unterklasse des Keras Objekts `keras.layers.Layer` erstellt. In der Initialisierung wird die Initialisierungsmethode der Superklasse aufgerufen und der Parameter „factor“ in eine Variable gespeichert. Die eigentlichen Berechnungen werden in der `call()` Methode ausgeführt und anschließend zurückgegeben. Der Aufruf der Schicht ist in Listing 2 unten zu sehen.

5.4 Struktur des Software-Projekts

In den nachfolgenden Abschnitten wird die Hauptarbeit der Projektarbeit dargelegt. Während zuvor deskriptiv an das Mask R-CNN herangeführt und Software-Grundlagen erklärt wurden, folgt nun eine detaillierte Beschreibung der Software, deren Aufbau und Umgang. Dazu werden notwendige Python-Skripte erklärt und Verbesserungsvorschläge gegeben. Im Anhang befinden sich weitere stichpunktartige Anleitungen, die für das Arbeiten mit der Software nützlich sind.

Für die Implementierung des Mask R-CNN Netzes gibt es fünf Python-Skripte:

1. `model.py`
2. `config.py`
3. `utils.py`
4. `coco.py`
5. `visualize.py`

`model.py` enthält die Schichten des Mask R-CNN Netzes. Die enthaltene Klasse `MaskRCNN` wird für das Training und die Inferenz instanziiert und repräsentiert die Keras Model Klasse.

`config.py` enthält alle Einstellungen und Hyperparameter des Netzes. Hier können komfortable Parameter des umfangreichen Mask R-CNN Netzes vorgenommen werden.

`utils.py` enthält allgemeine Funktionen, die für das Training des Netzes notwendig sind. Dazu gehören Skalierungen von Bildern und Masken, sowie Berechnungen für das RPN.

Außerdem ist die Basisklasse `Dataset` für den Umgang mit verschiedenen Datensätzen inkludiert.

`coco.py` ist eine Unterklasse von `config.py` und enthält wiederum eine Unterklasse von `Dataset`. Hier sind zusätzliche Hyperparameter speziell für den COCO-Datensatz enthalten, z. B. Anzahl der Objektklassen und *keypoints*. Zudem werden mit den PyCOCOTools Annotationen im COCO-Format dem Netz zugänglich gemacht.

`visualize.py` stellt Funktionen zur Visualisierung der *bounding-boxen*, Masken und *keypoints* zur Verfügung.

Im nächsten Abschnitt wird genauer auf die Implementierung der Klasse `MaskRCNN` eingegangen.

5.5 Klasse MaskRCNN

Die Klasse `MaskRCNN` ist das Herzstück der Implementierung. In dessen `build()` -Funktion werden die Einzelteile als Funktionen und Klassen des Netzes aufgebaut (siehe Abschnitt 3). Wegen des großen Code-Umfangs wird der Aufbau tabellarisch dargestellt. Tabelle 2 stellt die einzelnen Teile des Netzes dar und zeigt die verwendeten Klassen und Funktionen für deren Implementierungen. Die Spiegelstriche kennzeichnen den Aufruf durch die übergeordnete Funktion bzw. Klasse. Tabelle 2 soll den Umgang mit dem Code, besonders in der Einarbeitungszeit erleichtern.

5.6 Inferenz

Für das Ausführen der Inferenz stehen zwei Notebooks zu Verfügung:

1. `inference_coco_human_pose.ipynb`
2. `inference_human_pose.ipynb`

Im Ersten können Masken und *keypoints* aus Bildern des COCO-Validation-Datensatzes prädiziert und mit dessen Annotationen verglichen werden. Im Zweiten lässt sich die Inferenz an beliebigen Bildern testen.

Die Notebooks enthalten ausführliche Kommentare, die deren Verwendung erklären. Deshalb wird an dieser Stelle nur auf die wichtigsten Punkte eingegangen.

Tabelle 2: Implementierung von Mask R-CNN

Paper	Software
ResNet	resnet_graph – conv_block – identity_block
Feature Pyramid Network	build() in MaskRCNN
Region Proposal Network	build_rpn_model – rpn_graph
Detector	fpn_classifier_graph – PyramidROIAlign DetectionLayer – refine_detections_graph DetectionKeypointTargetLayer – detection_keypoint_targets_graph
Mask branch	build_fpn_mask_graph – PyramidROIAlign
Keypoint branch	build_fpn_keypoint_graph – PyramidROIAlign

Voraussetzungen

Voraussetzung für das erfolgreiche Ausführen ist die Verwendung der richtigen Python Module und deren Versionen. In Abschnitt 6 wird die Einrichtung einer virtuellen Umgebung beschrieben. Des Weiteren müssen zwei Pfade angegeben werden, die unter dem Markdown *Set path to model and images* beispielhaft angegeben sind:

1. Pfad zum Modell (h5-Datei)
2. Pfad zum COCO- bzw. Bilder-Ordner

Hierbei ist bei der COCO-Inferenz die Struktur innerhalb des COCO-Ordners wie angegeben einzuhalten. Es gibt jeweils einen Ordner mit den Trainings-, Test- und Validierungsbildern sowie einen mit den Annotationen.

Modell im Inferenz-Modus erstellen

Der Code `model = modellib.MaskRCNN(mode='inference',...)` erstellt eine Instanz des Mask R-CNN Netzes. Dies kann je nach GPU einige Sekunden dauern. Während des La-

dens ist im Task-Manager das Füllen des GPU-Speichers zu beobachten. Tensorflow reserviert nahezu den gesamten Speicher, um diesen effizient zu nutzen und um Speicherfragmentierung zu vermeiden [15]. Nach erfolgreichem Ausführen kann das Modell für die Prädiktion verwendet werden.

Keypoints prädizieren und anzeigen

`model.detect_keypoint()` führt die Inferenz durch. Hierbei werden u. a. die *bounding-boxen*, Masken und *keypoints* in Form von Vektoren gespeichert. Die Ergebnisse lassen sich mit den Funktionen `display_keypoints()` bzw. `display_instances()` anzeigen.

5.7 Training

Das Training kann mit dem Python-Skript `train_human_pose.py` ausgeführt werden. Auch hier wird wegen der ausführlichen Kommentierung nur auf wichtige Details eingegangen.

Wie bei der Inferenz müssen zunächst die Pfade zum COCO-Ordner angegeben werden. Dort befinden sich neben den Trainings-Bildern auch die zugehörigen Annotationen. Außerdem wird angegeben, wo das trainierte Modell gespeichert werden soll.

Festlegen des vortrainierten Modells

Vor dem Training ist zu entscheiden, auf welcher Basis ein neues Modell trainiert werden soll. Man kann zwischen Folgendem wählen:

1. ResNet-50
2. Matterport
3. Superlee506 (alter *keypoint*-Zweig)
4. LukasStu (neuer *keypoint*-Zweig)
5. Zuletzt trainiertes Modell

Je nach gewählter Strategie müssen der jeweilige Pfad in `COCO_MODEL_PATH` gesetzt und die zugehörige `load_weights()` Methode ausgewählt werden.

Festlegen der Hyperparameter

Wichtige Hyperparameter lassen sich in der Klasse `TrainingConfig` einstellen. Die Klasse erbt alle Parameter aus `config.py` und `coco.py`. Diese können an dieser Stelle mit neuen Werten überschrieben werden. Hier werden auch die Anzahl der GPUs im Trainingsrechner und die jeweils zu verarbeitenden Bilder gesetzt. Die effektive *batch*-Größe ergibt sich aus deren Produkt.

Festlegen der Trainingsstrategie

Im Code wird eine Epoche (*epoch*) als feste Anzahl von Iterationen (*steps*) definiert. Jede Iteration führt zu einer Aktualisierung der Gewichte und jeder Epoche erzeugt den aktuellen Trainingsstand in Form einer h5-Datei. Außerdem erfolgt eine Berechnung des Verlustes (*loss*) mit Bildern aus dem Validation-Datensatz, um *overfitting* zu erkennen.

Das Training eines Mask R-CNN Netzes erfolgt in Stufen, ähnlich wie es He et al [5, S. 4] vorschlagen. Jede Stufe trainiert andere Schichten mit eventuell reduzierter Lernrate. Hierzu sind die Parameter `learning_rate`, `epochs` und `layer` entsprechend zu setzen. Zu beachten ist, dass `epochs` nicht die absolute Anzahl sondern die letzte Trainingsepoche angibt.

Voreingestellt ist folgende Strategie nach He et al [5, S. 10]: Es sollen insgesamt $130\,000$ Iterationen trainiert werden. Das Finetuning erfolgt ab Iteration $100\,000$ und $120\,000$ mit Reduzierung der Lernrate um jeweils Faktor 10 (siehe Tabelle 3). Für ein vergleichbares

Tabelle 3: Trainingsstrategie von Mask R-CNN mit *keypoint*-Detektion

Epoche	Iterationen/Epoche	Schichten	Lernrate
$1-100$	1000	heads	$0,002$
$101-120$	1000	4+	$0,002/10$
$121-130$	1000	all	$0,002/100$

Training ist eine effektive Batch Größe von 16 einzuhalten. Diese errechnet sich aus dem Produkt `GPU_COUNT` und `IMAGES_PER_GPU`. Wird sie nicht erreicht oder ist sie größer, sollte `STEPS_PER_EPOCH` wie folgt angepasst werden.

$$STEPS\_PER\_EPOCH = 1000 \cdot \frac{16}{GPU\_COUNT \cdot IMAGES\_PER\_GPU}$$

5.8 Tipps für erfolgreiches Training

Das Mask R-CNN Netz ist sehr rechenintensiv. Einige Hyperparameter beeinflussen den Grafikspeicherbedarf und die Rechenzeit stark. Deshalb sollten die Tipps von [9] unter der Rubrik *Wiki* beachtet werden. Es ist Anzumerken, dass der *keypoint*-Zweig im vorliegenden Projekt den Rechenaufwand nochmals erhöht, weil er parallel zum Masken-Zweig implementiert wurde.

5.9 Ändern der Keypoint-Anzahl

Soll das Netz mit einem neuen Datensatz trainiert werden, der z. B. mehr als 17 Annotationen pro Person enthält, ist dies einfach möglich, vorausgesetzt die annotierten *keypoints* und Masken halten sich an den COCO-Syntax, wie es in Abschnitt 4 angedeutet ist. Liegen diese Daten vor, muss nur der Parameter `NUM_KEYPOINTS` geändert werden.

5.10 Trainingsgeschwindigkeit und Genauigkeit des Modells

Während des Trainings auf der Workstation „eit-elab-l36“ wurde festgestellt, dass das Training trotz leistungsfähiger GPUs deutlich langsamer ist, als He et al in [5, S. 4 u. 7] erreichen. Mit einer effektiven Batch Größe von 16 beenden sie das Training mit ResNet-101 und $160\,000$ Iterationen in 44 Stunden. Trotz der vier aktuellen Grafikkarten NVIDIA GeForce RTX3090 würde ein Training ca. 22 Tage brauchen. Dies könnte mehrere Gründe haben:

- Programmteile mit ineffizienten Berechnungen
- Nicht optimal konfiguriertes Multi-GPU Training

Für die Analyse der Trainingsleistung bietet Tensorflow ab Version 2.2 einen *Profiler* an [16]. Damit ist es möglich, Ursachen für ineffizientes Training zu erkennen. Folgende Analysestrategie wird empfohlen:

1. Optimierung und Leistungsanalyse mit einer GPU
    - Prüfen der Input-Pipeline
    - Leitungsanalyse
    - fp16 und XLA aktivieren
2. Wie oben, nur mit Multi-GPU

Die Input-Pipeline ist dafür verantwortlich, dass die GPU mit Bildern versorgt wird. Kommt es hier zu Verzögerungen treten Berechnungspausen auf, die eine ineffiziente GPU-Nutzung hervorrufen. Die Tensorboard-Übersicht (*overview_page*) zeigt schnell, dass das Training auf einer GPU gut funktioniert: 97.4% der Berechnungen werden von der GPU ausgeführt. Zudem kommt es zu keinen Verzögerungen durch die Input-Pipeline. Dies bestätigt auch die Ansicht *trace_viewer*, bei dem Rechenzeiten und Kopiervorgänge grafisch dargestellt werden. Lediglich die Operation *Non-Max Suppression* wird auf der CPU ausgeführt, weil diese Funktion noch nicht für die Ausführung auf einer GPU implementiert ist. Damit sind die Punkte 1 und 2 geprüft. Weiter Analysen stehen noch aus.

Wegen des im Folgenden beschriebenen Fehlerbildes liegt es Nahe, dass Tensorflow die Rechenleistung nicht effizient an alle Grafikkarten verteilt: Während des Multi-GPU Trainings lässt sich eine Impulsartige Auslastung im Taskmanager beobachten. Beim Training mit nur einer GPU ist die Auslastung deutlich gleichmäßiger (siehe Abbildung 9).

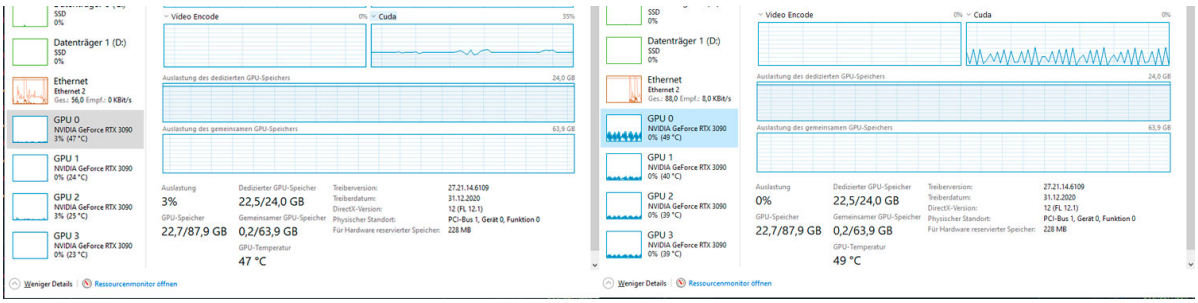

Abbildung 9: Single-GPU- und Multi-GPU-Training. Links: Beim Single-GPU-Training ist die Auslastung von *CUDA* sehr gleichmäßig. Rechts: Multi-GPU-Training belastet alle Grafikkarten impulsartig.

Außerdem lassen sich auf einer GPU drei Bilder gleichzeitig bearbeiten (704×704 Pixel), startet man ein Multi-GPU-Training, bricht es wegen Speicherüberlauf ab (*out of memory, OOM*). Dieses Verhalten deutet auf eine falsche Verteilung der Rechenleistung hin.

Der Grund für dieses Verhalten könnte in Skript `parallel_model.py` zu finden sein. Hier wird das Training, abhängig vom Hyperparameter `GPU_COUNT`, auf die Grafikkarten verteilt. Diese Vorgehensweise scheint inzwischen veraltet, denn seit Tensorflow 2 ist Multi-GPU-Training mit der API `tf.distribute.Strategy` möglich [17]. Damit lassen sich Trainings auf mehreren GPUs und sogar Workstations verteilen. Der Einrichtungsaufwand wird als gering beworben.

Im Zuge dieser Umstellung sollte auch eine Änderung des Daten-Generators in Betracht gezogen werden. Die APIs `tf.data.Dataset` und `tf.keras.utils.sequence` bieten beide die Möglichkeit, Datensätze für das Training bereit zu stellen und arbeiten Hand in

Hand mit der Verteilungsstrategie von Tensorflow. Siehe hierzu [18], [19] und [20]. Adam Kelly verwendet in seinem Projekt die API `tf.keras.utils.sequence`.

5.11 Keypoint-Zweig Implementierung

Beim Vergleichen zwischen dem von Li (Superlee506) implementierten *keypoint*-Zweig und dem im Mask R-CNN-Paper beschriebenen zeigen sich Unterschiede. Abbildung 10 stellt beide Zweige gegenüber. Die Rechtecke stellen *feature maps* und die Pfeile die Operationen dar.

Li verwendet eine `KEYPOINT_MASK_POOL_SIZE` von 7×7, welche eine deutliche Verkleinerung der Auflösung gegenüber 14×14 zur Folge hat. Zudem wird eine Entfaltungsschicht (*Deconvolution, Conv2DTranspose*) mit nur 17 Filtern (`NUM_KEYPOINTS`) verwendet, bevor zweimal zweifach bilinear vergrößert wird. Es ist nicht nachvollziehbar, warum nicht eine vierfache Vergrößerung durchgeführt wurde.

He et al behalten die Tiefe der *features* bei der Entfaltung, vergrößern nur einmal bilinear und reduzieren erst im letzten Schritt die *feature map* auf die Anzahl der *keypoints*. Hierbei können mehr Gewichte gelernt werden, wobei auch der Rechenaufwand steigt.

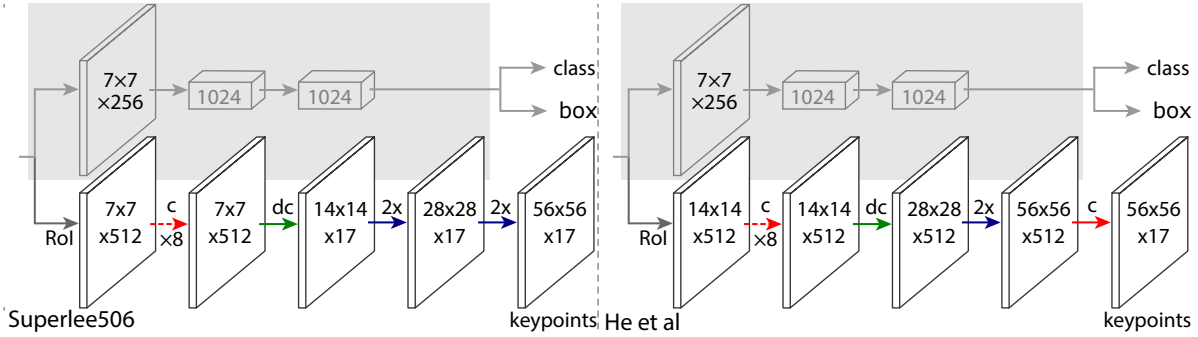

Abbildung 10: Gegenüberstellung der Masken-Zweige. links: Li, rechts: He et al. Die Operationen Faltung, Entfaltung und bilineare Vergrößerung sind mit c, dc und 2x beschriftet. Grafik nach [5, S. 4]

Eine Verbesserung der Genauigkeit gegenüber Lis Implementierung ist nach vollständigem Training mit den PyCOCOTools zu evaluieren und bei der Inferenz zu prüfen. Hierzu ist die Funktion `build_fpn_keypoint_mask_branch` entsprechend auszutauschen. Beide Zweige liegen im Software-Projekt zum besseren Vergleich in eigenen Python-Skripten vor.

6 Fazit und Ausblick

Fazit

Das Arbeiten am Softwareprojekt „2D-Keypoint-Detektion mit Mask R-CNN“ erwies sich als interessant und ist Gegenstand aktuellster Forschungen. Sowohl in Foren der einschlägigen Software-Projekte als auch in aktuellen Veröffentlichungen wird immer noch an der Weiterentwicklung von Mask R-CNN gearbeitet.

Während der Einarbeitungszeit wurde das Grundwissen maschinellen Lernens aufgefrischt und später durch fortgeschrittene Methoden erweitert. Tiefe Einblicke in aktuelle Ideen und Herausforderungen bei der *keypoint*-Detektion wurden beim Analysieren einschlägiger Veröffentlichungen gewährt.

Während der Arbeit an dem umfangreichen Software-Projekt konnte vertiefendes Wissen sowohl in der Programmiersprache Python, als auch in der Versionsverwaltung mittels Github gesammelt werden. Die Verwendung von Tensorflow und tf.keras schulte den Umgang mit *high-level APIs*. Auftretende Probleme mussten systematisch untersucht und Lösungen gefunden werden. Hierbei erwiesen sich vor allem die umfangreiche Dokumentation der Tensorflow-Webseite und Github-Foren der Softwareprojekte als hilfreich.

Das Ziel, ein lauffähiges Programm für Inferenz und Training zu erstellen, ist erreicht.

Ausblick

Probleme bestehen weiterhin bezüglich der Leistungsfähigkeit des Multi-GPU-Trainings (siehe Unterabschnitt 5.10). Hierzu sind weiterführende Untersuchungen mit dem Profiler durchzuführen und bei Bedarf die Änderungsvorschläge bezüglich `tf.distribute.Strategy` und des Daten-Generators einzuarbeiten. Es ist zu erwarten, dass ein vollständiges Training mit den ResNet-Gewichten in 2–3 Tagen abgeschlossen werden kann.

Danach muss die Genauigkeit des neuen *keypoint*-Zweigs evaluiert werden. Weil sich hier die Struktur und damit die Anzahl der Gewichte geändert hat, kann erst nach vollständigem Training eine Bewertung erfolgen. Eventuell sollte zusätzlich die alte Implementierung für bessere Vergleichbarkeit mit denselben Hyperparametern trainiert werden.

Da die Leistungsfähigkeit eines neuronalen Netzes maßgeblich von der Qualität des Trainingsdatensatzes und dessen Vergleichbarkeit mit dem späteren Einsatzgebiet abhängt, sollte

die Eignung des COCO-Datensatzes für genaue sportliche Analysen in Frage gestellt werden. Viele Bilder repräsentieren nicht die spätere Aufnahmesituation, weil sich entweder mehrere Personen überlappen, in ungünstigen Blickwinkeln oder nur sehr klein zu sehen sind. Interessant wäre hier das Training mit nicht annotierten Daten (*data distillation*), wie es He et al [5, S. 10] durchführten. Damit könnten relativ schnell Bilder trainiert werden, ohne dass diese aufwendig annotiert werden müssten.

Literatur

[1] Kevin Ashley. *Applied Machine Learning for Health and Fitness*. 2020. DOI: https://doi.org/10.1007/978-1-4842-5772-2.

[2] Ross Girshick u. a. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV]. (Besucht am 07. 02. 2021).

[3] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV]. (Besucht am 07. 02. 2021).

[4] Shaoqing Ren u. a. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV]. (Besucht am 07. 02. 2021).

[5] Kaiming He u. a. *Mask R-CNN*. 2018. arXiv: 1703.06870 [cs.CV]. (Besucht am 07. 02. 2021).

[6] Shamshad Ansari. *Building Computer Vision Applications Using Artificial Neural Networks : With Step-by-Step Examples in OpenCV and TensorFlow with Python*. 1st ed. 2020. Springer eBook Collection. Berkeley, CA: Apress, 2020. ISBN: 9781484258873. URL: https://doi.org/10.1007/978-1-4842-5887-3.

[7] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 2. Aufl. 18. Sep. 2020. ISBN: 978-1-492-03264-9.

[8] Tsung-Yi Lin u. a. *Feature Pyramid Networks for Object Detection*. 2016. eprint: arXiv:1612.03144. (Besucht am 07. 02. 2021).

[9] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017. (Besucht am 07. 02. 2021).

[10] Tsung-Yi Lin u. a. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV]. (Besucht am 07. 02. 2021).

[11] COCO Consortium. *Keypoint Detection Task 2020*. URL: https://cocodataset.org/#keypoints-2020 (besucht am 07. 02. 2021).

[12] COCO Consortium. *Data Format*. URL: https://cocodataset.org/#format-data (besucht am 07. 02. 2021).

[13] Chiao Li. *Mask RCNN for Human Pose Estimation*. https://github.com/Superlee506/Mask_RCNN_Humanpose. 2017. (Besucht am 07. 02. 2021).

[14] Adam Kelly. *Mask_RCNN*. https://github.com/akTwelve/Mask_RCNN. 19. Juni 2020. (Besucht am 07. 02. 2021).

[15] Tensorflow.org. *Use a GPU*. 5. Feb. 2021. URL: https://www.tensorflow.org/guide/gpu?hl=en (besucht am 07. 02. 2021).

[16] Tensorflow.org. *Optimize TensorFlow GPU Performance with the TensorFlow Profiler*. 4. Feb. 2021. URL: https://www.tensorflow.org/guide/gpu_performance_analysis?hl=en (besucht am 07. 02. 2021).

[17] Tensorflow.org. *Distributed training with TensorFlow*. 5. Feb. 2021. URL: https://www.tensorflow.org/guide/distributed_training?hl=en (besucht am 07. 02. 2021).

[18] Stackoverflow.com. *tf.data vs keras.utils.sequence performance*. 5. Feb. 2021. URL: https://stackoverflow.com/questions/55852831/tf-data-vs-keras-utils-sequence-performance (besucht am 07. 02. 2021).

[19] Tensorflow.org. *Base object for fitting to a sequence of data, such as a dataset*. 5. Feb. 2021. URL: https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence (besucht am 07. 02. 2021).

[20] Tensorflow.org. *Better performance with the tf.data API*. 5. Feb. 2021. URL: https://www.tensorflow.org/guide/data_performance?hl=en (besucht am 07. 02. 2021).

[21] Brett Slatkin. *Effektiv Python programmieren : 90 Wege für bessere Python-Programme*. ger. 2020.

[22] Maria Dobromyslova. *Making work TensorFlow with Nvidia RTX 3090 on Windows 10*. 10. Nov. 2020. URL: https://dobromyslova.medium.com/making-work-tensorflow-with-nvidia-rtx-3090-on-windows-10-7a38e8e582bf (besucht am 07. 02. 2021).

Abbildungs- und Tabellenverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Anhang

Einrichten der Projektumgebung mit Anaconda

Für die Einrichtung einer virtuellen Programmierumgebung wird das Programm „Anaconda“ verwendet. Eine virtuelle Umgebung verhindert Probleme bei Paketabhängigkeiten, wenn man mit verschiedenen oder älteren Paketversionen arbeitet [21, S. 444 ff]. Da Lis Softwareprojekt nur mit veralteten Paketen lauffähig ist, musste bei der Umstellung auf Tensorflow 2 und tf.keras eine unabhängige Umgebung geschaffen werden, die es erlaubt, zwischen den Versionen zu springen. Auch in der aktuellen Version ist es empfehlenswert, für verschiedene Zwecke virtuelle Umgebungen anzulegen. Während der Projektarbeit wurden Zwei genutzt: `tf24-profiler` für das Programmieren und `evalcoco` für die COCO-Evaluation, weil diese eine spezielle Numpy-Version benötigt.

Damit Tensorflow die Berechnungen auf eine GPU auslagern kann, müssen vor dem Einrichten der virtuellen Umgebung zusätzliche Programme installiert werden [15]. Das sind *NVIDIA CUDA Toolkit* (auch *NVIDIA CUDA SDK* genannt) und *NVIDIA cuDNN*. Auch die PyCOCOTool erfordern ein zusätzliches Programm: *Visual C++ Build Tools 2015*. Die Installation wird Stichpunktartig erklärt. Zur besseren Übersicht werden die Anleitungen ganzseitig dargestellt:

NVIDIA CUDA Toolkit installieren und einrichten

- Voraussetzung: NVIDIA GPU mit *Compute Capability* ≥ 3.5. Dazu *Compute Capability* der eigenen GPU herausfinden:
  `https://developer.nvidia.com/cuda-GPUs`
- Aktuellen Treiber installieren:
  `https://www.nvidia.de/Download/index.aspx?lang=de`
- CUDA Toolkit 11.0 installieren:
  `https://developer.nvidia.com/cuda-toolkit-archive`
- cuDNN v8.0.4 for CUDA 11.0 herunterladen:
  `https://developer.nvidia.com/rdp/cudnn-archive`
  - Datei entpacken
  - Inhalte der Ordner „bin“, „include“ und „lib“ in die gleichnamigen Ordner des CUDA Toolkit Installationspfads kopieren:
    `\ProgramFiles\NVIDIAGPUComputingToolkit\CUDA\v11.0`.
- CUPTI für Profiler einrichten:
  - Datei *cupti64_2020.1.1.dll* aus `\extras\CUPTI\lib64` in *cupti64_110.dll* umbenennen.
  - Pfad in Umgebungsvariable „Path“ hinzufügen:
    Windows-Taste→„Umgebungsvariablen für dieses Konto bearbeiten“ suchen→Path→bearbeiten→neu→vollständigen Pfad zu `C:\...\extras\CUPTI\lib64` angeben.
- Für den Profiler prüfen, ob Berechtigung auf GPU-Leistungsindikatoren vorliegt (nicht nötig an Workstations der Hochschule).
  NVIDIA Systemsteuerung→Desktop→Entwicklungseinstellungen aktivieren.
  Entwickler→GPU-Leistungsindikatoren verwalten→„Zugriff auf GPU-Leistungsindikatoren für alle Benutzer erlauben“ auswählen.

Diese Installationen sind für den Betrieb von Tensorflow-Version $\geq 2.4.0$ notwendig. Zum Zeitpunkt der Projektarbeit mussten für den Betrieb an Workstation „eit-elab-l36“ weitere Schritte durchgeführt werden, weil Tensorflow diese neuen GPUs noch nicht uneingeschränkt unterstützte. Im Wesentlichen musste CUDA Toolkit v11.1 installiert und dessen *PTX-compiler*

(*ptxas.exe*) nach v11.0 kopiert werden. Dieses Problem wird mit Erscheinen von Tensorflow 2.5 voraussichtlich gelöst, da CUDA Toolkit v11.1 unterstützt wird. Weitere Details sind unter [22] zu finden.

Virtuelle Umgebung mit Tensorflow 2.4.0 erstellen

- Anaconda installieren:
  `https://www.anaconda.com/products/individual`
- Virtual C++ Build Tools 2015 installieren:
  `https://go.microsoft.com/fwlink/?LinkId=691126`
- Umgebung mit Anaconda Prompt erstellen:
  `conda create -n tf24-profiler python=3.8 git numpy scikit-image scipy Pillow cython h5py pandas opencv pydot`
  `conda activate tf24-profiler`
  `conda install -c anaconda spyder`
  `conda install -c conda-forge notebook`
  `pip install tensorflow-gpu==2.4.0`
  `pip install -U tensorboard_plugin_profile`
  `pip install git+https://github.com/philferriere/cocoapi.git#subdirectory=PythonAPI`
- Für die COCO-Evaluation ist eine bestimmte Numpy-Version erforderlich. Hierzu muss eine neue Umgebung erstellt werden:
  `conda create -n evalcoco python=3.8 git numpy==1.17.0 scikit-image scipy Pillow cython h5py pandas opencv pydot`
- Umgebung aktivieren und Pakete wie oben installieren. Das *Profiler-Plugin* ist hier nicht erforderlich.

Training und Modell analysieren

Tensorboard/Profiler öffnen

Tensorboard ist ein Tool von Tensorflow, mit dem ein Training analysiert werden kann. Es ist z. B. möglich, den *loss* anzuzeigen und mit verschiedenen Hyperparameterkonfigurationen zu vergleichen. Außerdem erhält man Zugriff auf die Analysedaten des *Profilers*.

Zum Betrachten eines zuvor ausgeführten Trainings öffnet man Tensorboard in einem Browser und gibt den Ordner an, in dem die Trainingsdaten gespeichert sind (üblicherweise im Ordner „mylogs“):

- Anaconda Prompt: `activate tf24-profiler`
- `tensorboard --logdir 'C:\...\mylogs'`
- Browser: `http://localhost:6006/`
- Zugang zum Profiler über *drop-down*-Menü *Profil* rechts oben

Anmerkung: Für das Betrachten der *Profiler*-Daten muss die Aufzeichnung während des Trainings zuvor über den Parameter `profile_batch` in `model.py` aktiviert werden. Siehe dazu [16].

COCO-Evaluation durchführen

Mit Hilfe der PyCOCOTools lassen sich Modelle bewerten. Die Evaluation ist in Skript `coco.py` implementiert. Sie lässt sich wie folgt durchführen:

- Anaconda Prompt: `activate evalcoco`
- In den Projektpfad wechseln: `cd 'C:\...\Mask_RCNN_Humanpose'`
- COCO-Ordner und Modell-Pfad angeben:
  `python coco.py evaluate --dataset='C:/.../coco' --model='C:/.../model.h5'`

Während der Evaluierung werden einige Bilder aus dem COCO-Validierungsdatensatz verarbeitet, deshalb dauert der Vorgang 3–4 Minuten.