Hochschule für Technik Rapperswil HSR

MRU: Sensors, Actors and Communication

Studienarbeit

Yolo auf Finger

im Studiengang Industrial Technologies

eingereicht von: Heinz Hofmann <hhofmann@hsr.ch>

eingereicht am: 9. Februar 2017

Betreuer/Betreuerin: Herr Prof. Dr. G. Schuster
Frau T. Mendez

Dies ist ein Beispiel für ein Todo, und sollte vor Abgabe gelöscht werden ;)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Abstract

1.1 Aufgabenstellung

Das Ziel dieser Projektarbeit war es, herauszufinden, ob Yolo geeignet wäre, die Fingerspitzen einer Hand in einem Bild zu klassifizieren und genau zu detektieren. Die Vorgaben, was die Genauigkeit betreffen lagen bei 0.1mm. Yolo ist eine Möglichkeit, um mittels Deep-Learning Objekte in einem Bild zu klassifizieren und gleichzeitig deren genaue Position zu detektieren. Daher auch der Ausdruck Yolo (You only look once). Yolo wurde als Konzept gewählt, weil es in diesem Bereich dem aktuellen Stand der Technik entspricht. Gerade die Geschwindigkeit dieses Netzwerks wurde als extrem hoch angepriesen (bis zu 45fps). Diese Geschwindigkeit ist für die letztendliche Anwendung von hoher Wichtigkeit, weil es sich schlussendlich um eine Echtzeitanwendung handeln soll.

1.2 Vorgehen

Mithilfe der Apparatur und Software von Tabea Méndez wurden zuerst Daten generiert. Um diesen Aufwand klein zu halten, wurden nur Daten vom rechten Zeigefinger generiert. Gleichzeitig wurde in Tensorflow die Architektur von Yolo nach gebaut. Dies wäre nur begrenzt nötig gewesen, da fertige Architekturen in Keras oder Darknet online zur Verfügung stehen würden. Um aber einen Lerneffekt im erstellen von Neuronalen Netzwerken zu erzielen, wurde trotzdem alles von Grund auf selber aufgebaut. Rund um die Kernarchitektur von Yolo wurde das Datenhandling, die Kostenfunktionen aber auch sämtliche Validierungen und Tests zweimal erstellt. Einmal für das Pretraining der Kerngewichte auf dem ImageNet Klassifizierungsdatenset und einmal für das ëchte"Training auf den selber generierten Daten. Sobald dies alles aufgebaut und lauffähig war, wurde noch so viel wie möglich experimentiert, um herauszufinden, mit welchen Änderungen und Einstellungen das Lernresultat noch optimiert werden könnte.

1.3 Fazit

Das Pretraining und auch das Training hatten seine Tücken, weil das originale Yolo-Netzwerk extrem gross ist, und entsprechend nahezu den ganzen RAM-Speicher einer GPU benötigte, wodurch nur noch begrenzt Platz für Daten übrigblieb. Diese Probleme konnten einigermassen umgangen werden,

hatten jedoch zur Folge, dass die Bilder von 1280x960 auf 448x448 verkleinert werden mussten, um das Netzwerk zum laufen zu bringen. Dies hatte zur Folge, dass ein Pixel bereits bis zu 1,5mm entsprechen konnte. (Dies sollte Yolo theoretisch nicht daran hindern genauere Aussagen über die Position des Fingerspitzen zu machen.) Trotzdem wurde mit rund 84% der Predictions nur eine Genauigkeit von 15mm erreicht, was in etwa 10 Pixeln entsprach. Mit diesem Ergebnis wurde zwar das Ziel der Aufgabenstellung (0.1mm) um Faktor 150 verpasst, allerdings in 84% der Fälle Predictions gemacht, welche aus subjektiver menschlicher Sicht "gutßind. Dies ist ein einigermaßen erstaunliches Resultat, wenn man bedenkt, dass man zum Trainieren nur rund 18'000 Bilder verwendet hatte. Es ist anzunehmen, dass mit einer Verbesserung der Datengewinnung und entsprechend viel mehr Daten in naher Zukunft mit diesem oder einem ähnlichen Konzept eine Genauigkeit von bis zu 1mm erreicht werden können sollte.

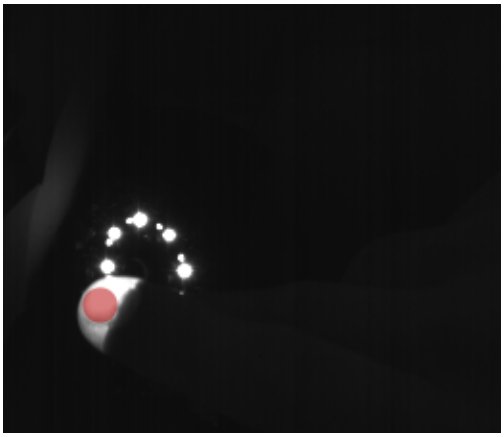
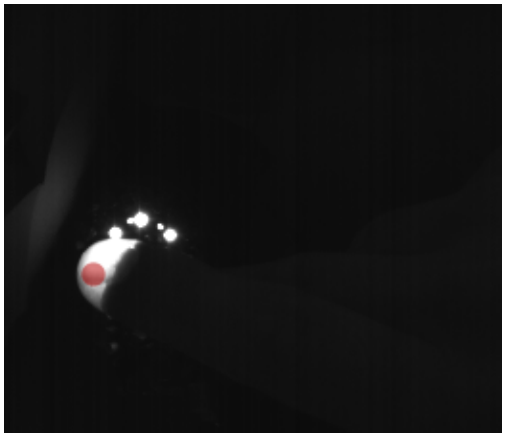

Abbildung 1: Resultate Hough-Transformation

2 Daten-Pipeline

2.1 Bilder aufnehmen

Die Aufnahme der Bilder geschah unverändert mit Apparatur und C++ Code von Tabea Méndez welche aus Ihrer Masterarbeit [2] entstand. Das Ergebnis waren jeweils 8 Bilder aus einer Situation. Eine Situation bestand aus 4 Kameras, wobei jede Kamera jeweils ein schwarzweiß-Bild mit UV-Beleuchtung und ein schwarzweiß-Bild mit normaler weisser Beleuchtung gemacht hatte. Wegen schlechter Erfahrungen mit Restlicht wurde der Aufbau mit schwarzem Papier abgedeckt. Diese schlechten Erfahrungen wurden gemacht, weil zu diesem Zeitpunkt zum Labeling der Daten noch keine Zeitinformation verwendet, also die Finger noch nicht von Bild zu Bild getrackt wurden. Pro Durchgang konnten maximal 6000 Situationen aufgenommen werden, bevor der Arbeitsspeicher des dafür verwendeten Computers an seine Grenzen kam.

Eine Verbesserung könnte hier erreicht werden, wenn man das Programm in 2 verschiedene Threads aufteilen würde. Dabei wäre ein Thread für das Aufnehmen der Daten und der andere für das abspeichern derselben zuständig. So könnte “zeitlich unbegrenzt“ Daten aufgenommen werden. Dies würde aber nur nötig, falls tatsächlich in Zukunft mit einem Roboter Daten aufgenommen würden.

2.2 Fingerdetektion

Zur Fingerdetektion wurde der Matlab-Fingerdetektor aus der Masterarbeit von Tabea Méndez [2] verwendet. Um Zeit bei der Datenaufnahme einzusparen wurde Zeit und Rauminformation nicht miteinbezogen. Dies brachte einige neue Probleme mit sich. So wurden auch mit Restlicht beleuchtete Punkte im Hintergrund oder LED's, als Finger erkannt. Dieses Problem konnte aber weitgehend behoben werden, indem in den Matlab-Fingerdetektor noch einige Filter eingebaut wurden.

1. Überspringen von Bildern, welche eine gewisse Helligkeit überschreiten. Dies sortiert Bilder aus, welche eine grosse Hintergrundhelligkeit und dadurch auch viele fehlerhaft erkannten Fingerspitzen enthält aus.

2. Aussortieren von erkannten Punkten, die zu gross sind, als dass Sie ein Fingerspitz sein könnten. Dieser Punkt ist teilweise redundant mit dem ersten Punkt, da so grosse Punkte nur im Hintergrund bei einer extrem grossen Helligkeit auftreten können.

3. Aussortieren von erkannten Punkten, welche zu klein sind, als dass Sie eine Fingerspitze sein könnten. Damit werden die meisten LED-Punkte entfernt.

4. Von den übrigen Punkten wird dann nur noch der Grösste behalten. Diesem wird somit das Label "rechter Zeigefingerspitz" verliehen.

Mit diesen Filtern konnte ein hoher Prozentsatz der rechten Zeigefinger korrekt detektiert werden. Auf den Finger selber bezogen war die Genauigkeit leider jedoch relativ schlecht. Dies aus dem einfachen Grund, dass die Detektionspunkte nicht immer genau in der Mitte des Fingers zu liegen kamen. Weiter waren auch die Boundingboxen, welche sich aus dem Resultat der Hough-Transformation [2] berechnen liessen nicht sehr genau. Wie man in der Abbildung 1 sehen kann, können sich diese innerhalb des Fingers auch bei sehr ähnlichen Bildern stark unterscheiden.

2.3 CSV generieren

Das Fingerspitzentracking wurde mit Matlab gemacht und die entsprechenden Labels als .mat-File abgespeichert. Das Deeplearning hingegen wurde mit Tensorflow und entsprechend mit Python angegangen. Leider war es nicht möglich mit Python direkt .mat-Files zu öffnen. Aus diesem Grund

wurde ein kleines Matlab-Skript erstellt, welches die Labels als CSV abspeichert. Im Zuge dieses Skripts wurden ausserdem die Daten in Test und in Trainingsdaten aufgeteilt und je einem seperaten CSV abgespeichert. In diesem CSV gehört jedem Bild eine Zeile. Pro Zeile bzw. Bild werden folgende Punkte beschrieben:

1. Eindeutiger Bildname, mit welchem das Bild aus dem Directory geladen werden kann.
2. X-Koordinaten im Range [0:1280]
3. Y-Koordinaten im Range [0:960]
4. Durchmesser des Resultats der Hough-Transformation
5. Wahrscheinlichkeit, dass ein rechter Zeigefinger in diesem Bild ist. (Entweder 1 oder 0, je nach dem, ob ein Finger erkannt wurde.)

2.4 Python-Objekt generieren

Um die Daten einfach im Trainingsprozess aufrufen zu können, wurde eigens eine kleine Python-Klasse geschrieben. Die Daten müssen allerdings noch vor deren Verwendung im Training durch eine Funktion dieser Klasse vorverarbeitet werden. Die Gründe für die Vorverarbeitung sind:

1. Die Bilder wurden bisher Kameraweise bearbeitet. Dies bedeutet, die Bilder heissen bei verschiedenen Kameras genau gleich. Mit der Vorverarbeitung werden alle Bilder an einem gemeinsamen Ort gespeichert. Ausserdem erhält jedes Bild einen neuen Namen / eine neue Nummerierung, wodurch es möglich wird sie eineindeutig zuzuordnen.
2. Um im Training einfach mit den Label-Daten umgehen zu können und um Rechenaufwand während dem Training zu sparen wurden in der Vorverarbeitung die Labels zu demjenigen Tensor zusammengefügt, welcher in Abbildung 2 zu sehen ist.
3. Die Distanzen X und Y sowie die Höhe und Breite der Boundingbox mussten noch normalisiert werden, damit beim Training einfacher gerechnet werden kann.

2.4.1 Label-Tensor

Die Labels pro Bild sind in einem Tensor angeordnet. (Siehe Abbildung 2) Diese Anordnung wurde stark am Output-Tensor wie er im Yolo-Paper [3] erscheint angelehnt. Dabei wird das Bild in ein 7x7 Raster aufgeteilt. Für jedes Element dieses Gitternetzes werden folgende Punkte gespeichert:

x — Die Distanz des Zentrums der Fingerspitze zum linken Rand der Gitterzelle.

Ist kein Finger in dieser Gitterzelle, ist diese Variable gleich Null. Diese Variable ist folgendermassen normiert: Ist das Zentrum der Fingerspitze ganz links in der entsprechenden Gitterzelle, ist die Variable gleich null. Ist das Zentrum der Fingerspitze ganz rechts in der entsprechenden Gitterzelle, ist die Variable gleich eins.

y — Die Distanz des Zentrums der Fingerspitze zum oberen Rand der Gitterzelle.

Ist kein Finger in dieser Gitterzelle, ist diese Variable gleich Null. Diese Variable ist folgendermassen normiert: Ist das Zentrum der Fingerspitze ganz oben in der entsprechenden Gitterzelle, ist die Variable gleich null. Ist das Zentrum der Fingerspitze ganz unten in der entsprechenden Gitterzelle, ist die Variable gleich eins.

h — Die Höhe der entsprechenden Bounding Box.

Ist kein Finger in dieser Gitterzelle, ist diese Höhe gleich null. Diese Variable ist folgendermassen normiert: Ist die Box insgesamt so hoch wie das Bild, ist diese Variable gleich eins. Ist die Box “unendlich“ klein, so ist diese Variable gleich null.

w — Die Breite der entsprechenden Bounding Box.

Ist kein Finger in dieser Gitterzelle, ist diese Höhe gleich null. Diese Variable ist folgendermassen normiert: Ist die Box insgesamt so breit wie das Bild, ist diese Variable gleich eins. Ist die Box “unendlich“ schmal, so ist diese Variable gleich null.

p — Die Wahrscheinlichkeit, dass die Spitze eines rechten Zeigefingers in dieser Gitterzelle ist.

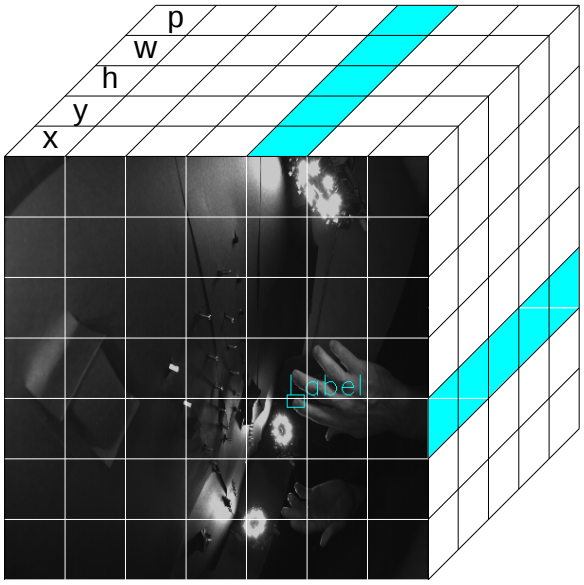

Abbildung 2: Label-Tensor

Ist eine rechte Zeigefingerspitze in dieser Gitterzelle, so ist diese Variable gleich eins. Ist keine rechte Zeigefingerspitze in dieser Gitterzelle, so ist diese Variable gleich null.

In diesem vereinfachten Fall gibt es nur eine p-Schicht. (Weil nur der rechte Zeigefingerspitz gelabelt wurde.) Wären aber auf dem Bild alle Finger einzeln gelabelt, müsste der Labeltensor für jeden zusätzlich labelbaren Finger eine weitere Schicht p haben. Würden also alle 10 Finger eines Menschen gelabelt, müsste der Labeltensor entsprechend 10 verschiedene p-Schichten haben. (Die Anzahl x, y, h & w - Schichten bleibt gleich)

2.4.2 Liste von Label-Tensoren

Diese Label-Tensoren enthalten alle wichtigen Labels von je einem Bild. Allerdings lassen Sie sich mit den darin enthaltenen Informationen nicht eindeutig einem Bild zuordnen. Um dies zu ermöglichen wurde eine Liste (Abbildung 3) erstellt, welche zwei Spalten und beliebig viele Zeilen enthält. In der ersten Spalte wird der Name der zum Label gehörenden Bilddatei gespeichert,

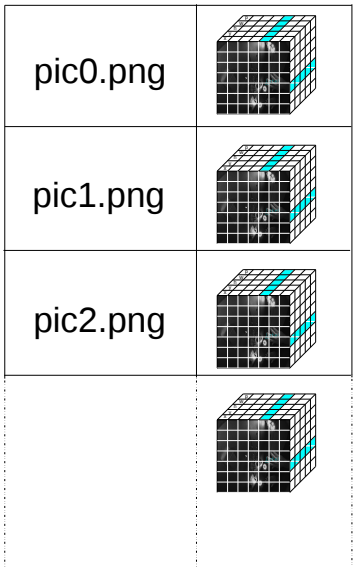

Abbildung 3: Liste von Label-Tensoren

während in der zweiten Spalte der Label-Tensor selber gespeichert wird. Diese Liste wiederum wird mit pickle als python Objekt abgespeichert. Der Speicherort dieser Liste ist derselbe wie derjenige, an welchem alle Bilder des entsprechenden Testsets oder Trainingsets abgespeichert wurden.

2.5 Daten in Neuronales Netzwerk einlesen

Das einlesen der Daten in das Programm, in welchem das Neuronale Netzwerk trainiert wird ist dank dem vorbereiteten Tensor und dessen Klasse, welche die Tensoren einfach laden lässt kein Problem. Die Daten schön für Tensorflow bereitzustellen ist etwas umständlich. Dies aber nur bis man es mal zum laufen gebracht hat. Danach stellt dies kein Problem mehr dar. Bis die Bilder wirklich ins Netzwerk gefüttert werden passieren noch folgenden Punkte.

1. Listen mit Bildnamen und Labels laden und als Tensorflow Datenset anlegen.

2. Einen “Shuffler“ einfügen, der jedes Mal wenn das Neuronale Netzwerk einen neuen Minibatch verlangt die entsprechenden Daten Zufällig auswählt.

3. Eine Funktion, welche die Bildnamen durch die echten Bilder ersetzt. Der Vorteil einer solchen Funktion ist, dass die Bilder wirklich erst geladen werden, wenn die Daten auch benötigt werden. So werden im Punkt 2 nicht die schwergewichtigen Bilder sondern nur deren Namen aus der Liste durchgemischelt.

4. Eine letzte Funktion, welche das organisieren des Minibatches übernimmt. Sodass die grösse des Minibatches flexibel bestimmt werden kann.

5. Die Bilder wurden von der Grösse 1280*960 auf 448*448 umgewandelt. Um diesen Punkt auf die GPU auszulagern wurde dafür eine eigenständige Tensorflow-Funktion verwendet (tf.image.resize_images).

6. Die Bilder wurden normalisiert. Dafür wurde eine eigene Funktion geschrieben, welche wiederum aus mehreren Tensorflow-Funktionen bestand. So wird auch dieser Task von der GPU erledigt.

Die Punkte 1-3 waren mithilfe der Tensorflowklasse “tf.contrib.data.Dataset“ relativ einfach zu bewältigen.

3 Architektur

3.1 Auswahl der Architektur

Die Architektur wurde stark dem Yolo-Paper [3] angelehnt. Dies obwohl zu diesem Zeitpunkt auch schon das Yolo v2-Paper [4] erschienen war. Es gab damals schon viele gute Gründe dafür von Beginn weg das Netzwerk und Kostenfunktionen nach dem Yolo v2-Paper aufzubauen. So ist Yolo v2 nach dessen Paper zu urteilen schneller und genauer. Der Grund, warum trotzdem Yolo v1 verwendet wurde, war dass die Beschreibung z.B. von Kostenfunktion und von der Architektur im Paper von Yolo v1 um einiges genauer und verständlicher war als im Paper von Yolo v2. Ausserdem ging man mit der Einstellung an die Arbeit, dass wenn erst das “einfache“ Yolo v1 erfolgreich implementiert wurde, dieses entsprechend immer noch zur 2. Version erweitert werden könnte.

evtl. den folgenden Abschnitt entfernen.

Dazu kam es allerdings nicht, weil verschiedene Faktoren die erfolgreiche Fertigstellung von v1 verzögerten. (Genauere Informationen dazu werden später im Kapitel

min. hier eine Referenz auf die Erklärung warum die Verzögerungen auftraten machen.

erläutert.) Ausserdem war das Grundlegende Ziel immer einen Eindruck dafür zu bekommen, was mit State-Of-The-Art Lösungen aktuell in diesem Bereich überhaupt möglich wäre. Entsprechend hatte man sich auch primär auf dieses Ziel fokussiert.

3.2 Architekturaufbau

Der grundlegende Aufbau der Architektur, wie er letztendlich aussah kann man in der Tabelle 1 betrachten. Dies sah allerdings noch nicht immer so aus. Obwohl die Convolution-Filter schon immer so ausgesehen hatten, sah der ursprüngliche Bildinput und entsprechend die Outputs der verschiedenen Layers mal anders aus. Nach ausführlicher Diskussion [5] wurde zu Beginn des Architekturdesigns entschieden, dass man nicht mit 448x448 Bildern arbeitet, wie dies im Yolo-Paper [3] gemacht wurde. Der Grund dafür war, dass für das Training wie auch später für den Praxiseinsatz immer 1280x960 grosse

Layer	**Filtertyp**	**Anzahl**	**Grösse**	**Strides**	**Output**
0	Input				448x448x1
1	Convolutional	64	7x7	2x2	224x224x64
2	Maxpool		2x2	2x2	112x112x64
3	Convolutional	192	3x3	1x1	112x112x192
4	Maxpool		2x2	2x2	56x56x192
5	Convolutional	128	1x1	1x1	56x56x128
6	Convolutional	256	3x3	1x1	56x56x256
7	Convolutional	256	1x1	1x1	56x56x256
8	Convolutional	512	3x3	1x1	56x56x512
9	Maxpool		2x2	2x2	28x28x512
10	Convolutional	256	1x1	1x1	28x28x256
11	Convolutional	512	3x3	1x1	28x28x512
12	Convolutional	256	1x1	1x1	28x28x256
13	Convolutional	512	3x3	1x1	28x28x512
14	Convolutional	256	1x1	1x1	28x28x256
15	Convolutional	512	3x3	1x1	28x28x512
16	Convolutional	256	1x1	1x1	28x28x256
17	Convolutional	512	3x3	1x1	28x28x512
18	Convolutional	512	1x1	1x1	28x28x512
19	Convolutional	1024	3x3	1x1	28x28x1024
20	Maxpool		2x2	2x2	14x14x1024
21	Convolutional	512	1x1	1x1	14x14x512
22	Convolutional	1024	3x3	1x1	14x14x1024
23	Convolutional	512	1x1	1x1	14x14x512
24	Convolutional	1024	3x3	1x1	14x14x1024
25	Convolutional	1024	3x3	1x1	14x14x1024
26	Convolutional	1024	3x3	2x2	7x7x1024
27	Convolutional	1024	3x3	1x1	7x7x1024
28	Convolutional	1024	3x3	1x1	7x7x1024
31	Fully-Connected		(7x7x1024)x4096		4096
32	Fully-Connected		4096x(7x7x6)		7x7x6

Tabelle 1: Yolo-Architektur

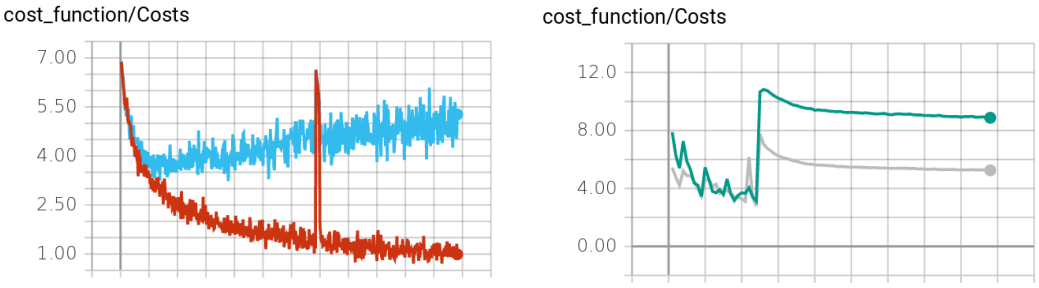

Abbildung 4: Effekte (Extremer Anstieg der Kosten innert einer Epoche) im Pretraining(links) und im Training(rechts). x-Achse=Zeit, y-Achse=Kosten. Weinrot=Pretraining-Trainingsdaten, Hellblau=Pretraining-Validierungsdaten, Grün=Training-Trainingsdaten, Grau=Training-Validierungsdaten

Bilder zur Verfügung standen und man entsprechend nicht Informationen "wegwerfen" sondern so lange wie möglich im Netz behalten wollte. So war der Input (Layer 0 in Tabelle 1) damals 1280x960x1, entsprechend dann auch der Output von Layer 1 nicht mehr 224x224x64 sondern 640x480x64 usw. Da dies am Schluss nicht aufgeht mit dem Netzwerk, hatte es damals noch zwei zusätzliche Layer (29 & 30), welche jetzt nicht mehr vorhanden sind. Layer 29 war dabei für ein Zeropadding und Layer 30 für ein Maxpooling mit Strides 3x3 zuständig.

Diese damalige Architektur war zu gross, um sie auch nur mit Minibatchsize=1 und Float32 ins GPU-RAM und entsprechend überhaupt zum laufen zu kriegen. Entsprechend wurden alle Gewichte und Knoten mit Float16 initialisiert. Seit dieser Initialisierung war eine Minibatchsize=7 möglich.

Mit dieser Architektur wurde eine Zeit lang trainiert, bis sich immer mehr spezielle Effekte im Training, wie auch im Pretraining (Details zum Pretraining im Kapitel 7) häuften. Es konnte auch nach längerer Analyse nicht abschliessend geklärt werden, was die Ursache für diese Effekte (Siehe Abbildung 4) war. Die Vermutung lag jedoch darin, dass es sich um Overflow-Probleme im Zusammenhang mit den verwendeten Float16 handeln könnte. Entsprechend wurde die Architektur umgebaut, sodass die Input-Bilder künstlich verkleinert wurden, um dafür Float32 verwenden zu können. In diesem Schritt war es naheliegend, dass man sich gleich den originalen Werten, wie sie von Yolo [3] verwendet wurden annäherte. Entsprechend wurden die Input-Bilder auf 448x448 verkleinert. Dies hatte wiederum zur Folge, dass seit diesem Zeitpunkt sogar eine Minibatchsize=24 verwendet werden konnte. Ausserdem, und dies war noch viel wichtiger, traten die genannten speziellen Effekte

Beschreibung	Anzahl	in Bytes(Float16)	in Bytes(Float32)
Gewichte	206 M	413 MB	827 MB
Knoten:Input=1280x960	98 M	186 MB	
Knoten:Input=1280x960 Minibatchsize=7, GPU-RAM voll	689 M	1.38 GB	
Knoten:Input=448x448	16 M		64M
Knoten:Input=448x448, Minibatchsize=24, GPU-RAM voll	384 M		1.54GB

Tabelle 2: Anzahl Gewichte und Knoten

(Abbildung 4) weder im Pretraining noch im Training je wieder auf.

Aus dieser Erfahrung kann man ableiten, dass bei Convolutional-Neural-Networks die Grösse der Input-Bilder mehr ins Gewicht fallen als die Anzahl Gewichte. Dies scheint nachträglich auch logisch, denn die Bilder werden während dem "flow" durch das Netzwerk mehrmals zwischengespeichert. In der Tabelle 2 ist die Berechnung, der Anzahl Gewichte und der Anzahl Knoten unter der Annahme, dass jedes Bild zwischen den Layern einmal als Knoten abgespeichert wird. Dabei wurde nicht berücksichtigt, dass es pro Layer mehrere Einheiten von Knoten geben kann, wie z.B. vor und nach der Aktivierungsfunktion. Auch ohne diese zusätzlichen Layer kann man aber deutlich erkennen, dass wenn man die Minibatchsize solange erhöht, bis das GPU-RAM (im Rahmen dieser Arbeit 16GB) voll ist, man klar mehr Speicher für Knoten benötigt, als für Gewichte.

3.3 Fazit

Was die Architektur angeht kann man aus dieser Arbeit die folgenden beiden Punkte lernen:

1. Wenn in Convolutional Neural Networks Speicherknappheit ein Problem ist, sollte entweder die Tiefe des Netzwerks verkleinert (weniger Layer = weniger Knoten) oder der Input verkleinert (= ebenfalls weniger Knoten) werden. Nicht aber sollte man auf Bastellösungen ausweichen, sodass man sich was Datentypen angeht ausserhalb des Tensorflow-Standardbereichs aufhält. Es sei denn natürlich, man weiss ganz genau was man tut, und kennt entsprechend den Source-Code von Tensorflow in- und auswendig. Wenn dem aber so wäre, würden

Sie diese Arbeit wohl kaum lesen ;) .

2. Wenn man eine Architektur nach entsprechend einer Vorlage aufbaut, sollte man nicht schon bevor man eine erfolgreich lauffähige Version hat an Parametern wie der Input-Grösse herumoptimieren. Optimieren sollte man erst, wenn man eine Lauffähige Fehlerfreie Version hat, sodass man jederzeit wieder zu dieser Lauffähigen Version zurückkehren kann.

4 Kostenfunktion

4.1 Erster Wurf

In einem ersten Wurf war das Ziel eine Kostenfunktion zu erstellen, welche so einfach wie nur möglich sein sollte. So sollte man schnellstmöglich ein funktionierendes Netzwerk haben, welches einfach zu debuggen ist und später Schritt für Schritt verbessert werden konnte. Dabei sollte der Output des Netzwerks lediglich 3 Variablen umfassen. Eine, welche die x-Koordinaten des rechten Zeigefingers vorhersagt, eine, welche die y-Koordinaten vorhersagt, und eine letzte, welche die Wahrscheinlichkeit dass sich ein rechter Zeigefinger in diesem Bild befindet vorhersagt. Die Kosten sollten dabei wie beim originalen Yolo-Netzwerk mit den kleinsten Fehlerquadraten der Distanz von Label zu Prediction berechnet werden. Dieser Ansatz hatte überhaupt nicht funktioniert. Die Predictions waren irgendwo im Bild und mit menschlichem Auge keine Korrelation mit den Labels ersichtlich.

Es wurde die Hypothese erstellt, dass dies daran läge, dass man mit dem Pretraining (Details im Kapitel 7) eigentlich Hauptsächlich einen Klassifizierer "gezüchtet" hatte, aber für die zwei essentiellen Predictions(x-/ y-Wert) eigentlich nur eine Regression verwendet wurde. Beim originalen Output des Yolo-Netzwerks könnte man sich vorstellen, dass das Netzwerk für jede Gitterzelle eine intuitiv eine Klassifizierung durchführt, und entsprechend den Fehler schon stark eingrenzen kann. Aufgrund dieser Intuitiven Erklärung wurde der Plan gefasst, die Kostenfunktion stärker an der originalen Kostenfunktion zu orientieren.

4.2 Netzwerkoutput

Im ersten Schritt wurde der Output des Netzwerks nahezu genau nach dem Vorbild aus dem Yolo-Papers [3] aufgebaut. Die einzigen Unterschiede lagen darin, dass man pro Gitterzelle nur eine anstelle von zwei Bounding-Boxen ausgibt und dass man nur eine anstelle von 10 Klassen vorhersagt. Damit war der Output des Netzwerks (Abbildung 5) nahezu identisch mit dem Label-Tensor (Abbildung 2 & Beschreibung in Kapitel 2.4.1). Der einzige Unterschied zwischen dem Output-Tensor und dem Label-Tensor lag darin, dass jede Gitterzelle auch eine Vorhersage zur Confidence enthält.

Die Confidence war deshalb nicht explizit in den Labels enthalten, weil sie aus den Vorhersagen zu x, y, w und h sowie den entsprechenden Labels berechnet wurde. Die Label-Confidence ist eigentlich nichts anderes als die

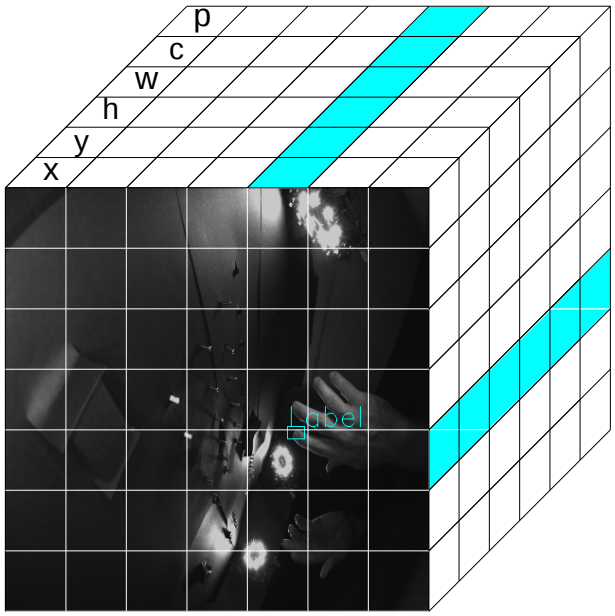

Abbildung 5: Prediction-Tensor

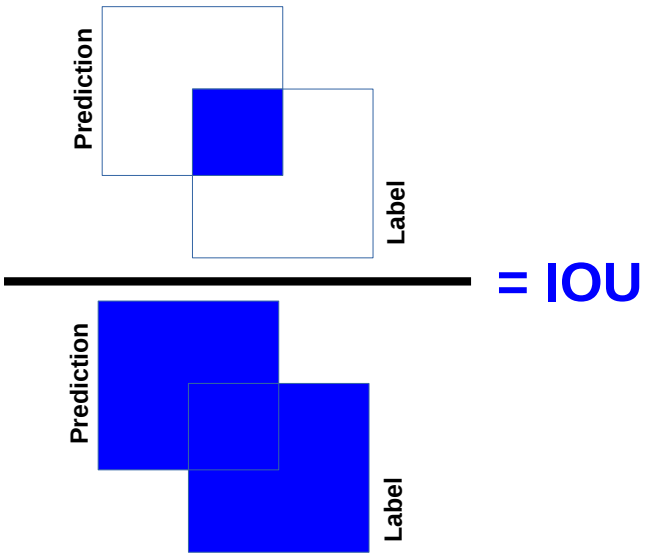

Abbildung 6: Berechnung der IOU

IOU[1] zwischen der vorhergesagten Bounding-Box und der Label-Bounding-Box einer bestimmten Gitterzelle. Die Berechnung der IOU aus diesen beiden Bounding-Boxen erfolgt wie aus Abbildung 6 ersichtlich.

Es befinden sich allerdings in den meisten Gitterzellen keine rechten Zeigefinger. Aus diesem Grund existieren in diesen Gitterzellen auch keine Labels zu x,y,w und h, weshalb auch keine Label-Bounding-Boxen existieren. In jeder dieser Gitterzellen, in welchen keine Label-Boundingboxen existieren ist die Label-Confidence automatisch=0. So wird man nach dem Training aufgrund der Confidence-Variable im Output für jede Gitterzelle ablesen können, ob sich darin irgend ein Objekt befindet, und wie gut dieses wahrscheinlich auf die Vorhersage von x,y,w und h passt.

4.3 Design Kostenfunktion

Die Auswahl bzw. das Design der Kostenfunktion ist wahrscheinlich der wichtigste Schritt im Design eines Neuronalen Netzwerks. In dieser Arbeit bestand das grosse Glück, dass die Kostenfunktion grösstenteils vom Yolo v1-Paper [3] vorgegeben wurde (Was auch mit ein Grund für die Wahl von Yolo v1 war).

[1] IOU = Intersection Over Union

Symbol	Definition
λ_{coord}	Faktor welcher verwendet wird um Fehler in den Koordinaten, sowie Höhe und Breite der Boundingboxen stärker zu gewichten. In diesem Fall wird hier ein Faktor von 5 verwendet. Diese Zahl ist so eins zu eins aus dem Yolo-Paper [3] übernommen worden.
λ_{noobj}	Faktor, welcher verwendet wird, damit der Confidence-Fehler nicht so stark gewichtet wird, wenn gar kein Objekt in dieser Gitterzelle vorhanden ist. In diesem Fall wird hier ein Faktor von 0.5 verwendet. Diese Zahl ist so eins zu eins aus dem Yolo-Paper [3] übernommen worden.
x_i	x-Labelkoordinaten, welche für die Gitterzelle i die Distanz vom Mittelpunkt der Boundingbox zum linken Rand der Gitterzelle i angibt.
y_i	y-Labelkoordinaten, welche für die Gitterzelle i die Distanz vom Mittelpunkt der Boundingbox zum oberen Rand der Gitterzelle i angibt.
w_i	Breite der Label-Boundingbox für die Gitterzelle i
h_i	Höhe der Label-Boundingbox für die Gitterzelle i
C_i	Label-Confidence für die Boundingbox in der Gitterzelle i
p_i	Label-Wahrscheinlichkeit, dass sich ein rechter Zeigefinger in der Gitterzelle i befindet.
$\hat{x}, \hat{y}, \hat{w}, \hat{h}, \hat{C}, \hat{p}$	Dies sind die Predictions des Netzwerks, welche den oben definierten entsprechenden Labels gegenübergestellt werden.
$\mathbb{1}_i^{obj}$	Dieses Objekt ist = 1, wenn in der Gitterzelle i ein rechter Zeigefingerspitz enthalten ist. Dieses Objekt ist = 0, wenn in der Gitterzelle i **kein** rechter Zeigefingerspitz enthalten ist.
$\mathbb{1}_i^{noobj}$	Dieses Objekt ist = 1, wenn in der Gitterzelle i **kein** rechter Zeigefingerspitz enthalten ist. Dieses Objekt ist = 0, wenn in der Gitterzelle i ein rechter Zeigefingerspitz enthalten ist.
$\sum_i^{7*7}$	Summe über alle Gitterzellen, wobei i immer einer Gitterzelle entspricht.

Tabelle 3: Beschreibung der Kostenfunktionselemente

$$
\begin{aligned}
&\lambda_{coord} * \sum_{i}^{7*7} \mathbb{1}_i^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
+&\lambda_{coord} * \sum_{i}^{7*7} \mathbb{1}_i^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\
&+ \sum_{i}^{7*7} \mathbb{1}_i^{obj} [(C_i - \hat{C}_i)^2] \\
+&\lambda_{noobj} * \sum_{i}^{7*7} \mathbb{1}_i^{noobj} [(C_i - \hat{C}_i)^2] \\
&+ \sum_{i}^{7*7} \mathbb{1}_i^{obj} (p_i - \hat{p}_i)^2
\end{aligned}
\tag{1}
$$

Gleichung 1: abgespeckte Kostenfunktion wie Sie in dieser Arbeit verwendet wurde.

Die Kostenfunktion, wie Sie in dieser Arbeit verwendet wurde kann man in Gleichung 1 betrachten. Diese Kostenfunktion ist etwas einfacher als die Kostenfunktion, wie Sie im Yolo v1 Paper zu sehen ist. Dies hat zwei Gründe. Zum einen wurde pro Gitterzelle nur eine Boundingbox vorhergesagt. So fällt für Zeile 1-4 der Kostenfunktion das zweite Summenzeichen, sowie deren iteration über die Variable j weg. Zum anderen wurde nur eine Klasse(rechter Zeigefinger) gelabelt und vorhergesagt. So fällt auf der Zeile 5 das Summenzeichen und deren entsprechende Iteration über alle Klassen weg.

Hier weiterarbeiten

4.4 Fazit

Folgende Punkte konnten aus dieser Arbeit gelernt werden, bzw. sollten im Falle einer Vertiefung beachtet werden.

1. Man darf die Wahl der Kostenfunktion nicht unterschätzen. Wie man im Kapitel 4.1 sehen kann, darf man nicht annehmen, dass die Kostenfunktion frei und ohne gross nachzudenken gewählt werden kann. Vielmehr muss die Wahl der Kostenfunktion stark mit dem Netzwerk und dem Problem interagieren.

2. Für die Zukunft zu beachten: In dieser Arbeit wurde pro Gitterzelle nur eine Boundingbox vorhergestagt (Dies weil die Komplexität von zwei Boundingboxen in Tensorflow nahezu jegliches Mass überstiegen hätte.). Wahrscheinlich könnte die Performance noch verbessert werden, wenn anstelle von einer min. zwei Boundingboxen pro Gitterzelle vorhergesagt würden.

5 Tests

6 Resultate

6.1 Testvoraussetzungen

Das Netzwerk wurde auf 1'200'000 Bildern des ImageNet-1000-class-Datasets vortrainiert. Danach wurde es auf rund 13'900 Bildern aus dem Testaufbau [2] trainiert. Der Test wiederum wurde auf rund 1'500 Bildern ebenfalls aus dem Testaufbau [2] getestet. Diese Testbilder waren dem Algorithmus während des Lernprozesses nicht zugänglich und haben entsprechend keinen Einfluss auf den Lernprozess genommen. Ausserdem wurden diese Bilder so gewählt, dass Sie nicht gleichzeitig aufgenommen wurden. Dies verhindert, dass fast identische Bilder im Training und im Test vorkommen.

6.2 Analyse

Um die Genauigkeit der Predictions unseres Neuronalen Netzwerkes möglichst genau beschreiben zu können wurden die zwei Werte Distanz und IOU gewählt. Obwohl die beiden Werte korrelieren sagt jeder für sich nicht die volle Wahrheit über die Genauigkeit der Vorhersagen aus. Die Distanz ist für die geplante Anwendung der wesentlichere Wert, weil diese Informationen über den Standort des Fingers im Bild preisgibt. Die IOU ist mit der Distanz klar korreliert, denn ist die Distanz zu gross, ist die IOU schnell gleich Null. Sobald die Boundingbox der Prediction und die Boundingbox des Labels sich beginnen zu überlappen sagt die IOU etwas über die korrekte Vorhersage von Breite und Höhe der Boundingbox aus. Auch darüber ob die Box am richtigen Ort liegt, können aufgrund der IOU vage Annahmen getroffen werden. Aber wie gesagt, die Distanz ist dafür der sicherere Wert.

6.2.1 Distanz

Die Distanz beschreibt die normierte Differenz zwischen dem Zentrumspunkt des Labels und dem Zentrumspunkt der Vorhersage. Sämtliche Distanzen wurden so normiert, dass die Höhe des Bildes und auch die Breite gleich eins sind. Die maximale Distanz zwischen zwei Punkten ist also die Diagonale über ein Bild, welche entsprechend sqrt(2) ist. Was diese Normierten Distanzen in der realen Welt bedeuten ist auf Abbildung 7 erklärt. Zum Vergleich, ein Menschlicher Zeigefinger ist zwischen 10 und 20 mm breit. Eine normierte Distanz von 0.02 entspricht auf unserem Versuchsaufbau somit ziemlich genau der Breite eines menschlichen Fingers.

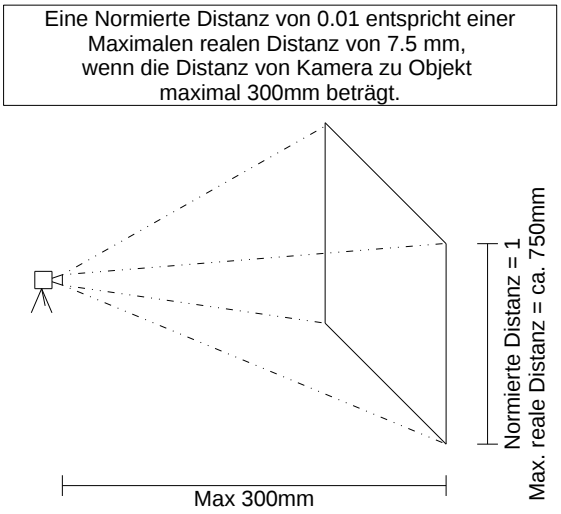

Abbildung 7: Bedeutung der normierten Distanzwerte in der realen Welt

Um die Resultate in gut und schlecht einteilen zu können wurde ein Threshold von 0.02 definiert. Die Definition dieses Thresholds wurde gemacht, indem Bilder zusammen mit der entsprechenden Distanz analysiert wurden. Der Wert 0.02 entspricht somit derjenigen Distanz, welche gerade noch knapp annehmbar ist, um einen Finger als detektiert gelten zu lassen. Um ein Gefühl für diese Distanzen zu kriegen lohnt es sich die Abbildungen 8 & 9 anzusehen, welche Bilder zeigen, die eine Distanz nahe dieses Thresholds aufweisen.

Um die Verteilung der Distanzen gut verstehen zu können, ist in Abbildung 10 eine Wahrscheinlichkeitsdichte der Distanzen im Testset zu sehen. Diese Dichtefunktion wurde erst nach der Bestimmung des Thresholds erzeugt und zeigt, dass rund 84% der Distanzen kürzer sind als 0.02 und somit die entsprechenden Finger ërfolgreichërkannt wurden.

Erstaunlich ist auch, dass die Distanzen, welche grösser als 0.25 sind in der Wahrscheinlichkeitsdichte in kleinen Bündeln vorkommen. Dies lässt darauf schliessen, dass die Trainingsdaten nicht komplett Bias-Frei sind. Nach kurzer Kontrolle konnte tatsächlich festgestellt werden, dass z.B. bei einer Distanz von ca. 0.4 immer ein bestimmter Punkt des Hintergrundes vorhergesagt wurde, welcher tatsächlich ganz selten in den Labels als Finger markiert wurde.

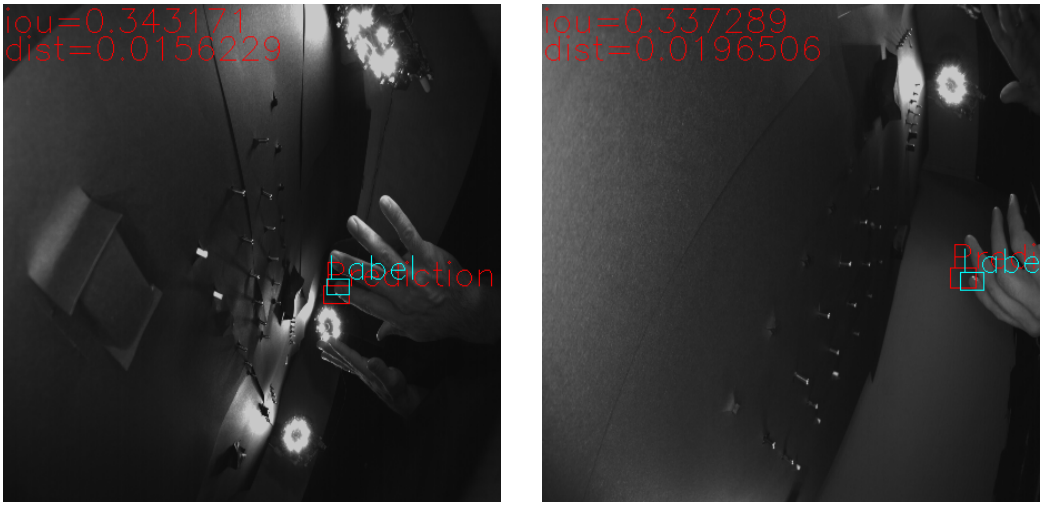

Abbildung 8: Prediction knapp besser als Distanz=0.02

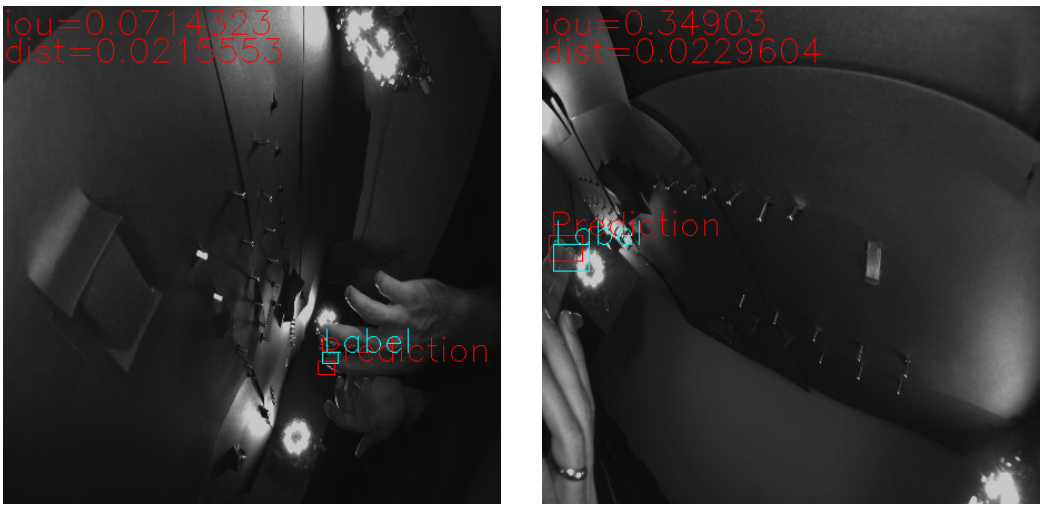

Abbildung 9: Prediction knapp schlechter als Distanz=0.02

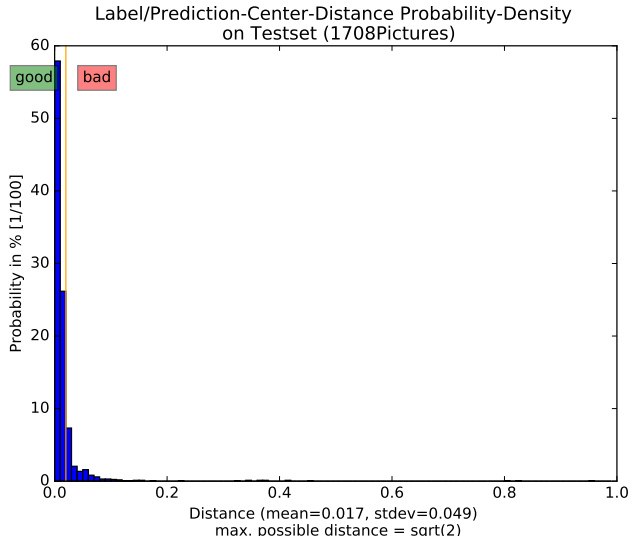

Abbildung 10: Komplette Wahrscheinlichkeits-Dichte-Funktion der Distanz (Grenze: Dist=0.02)

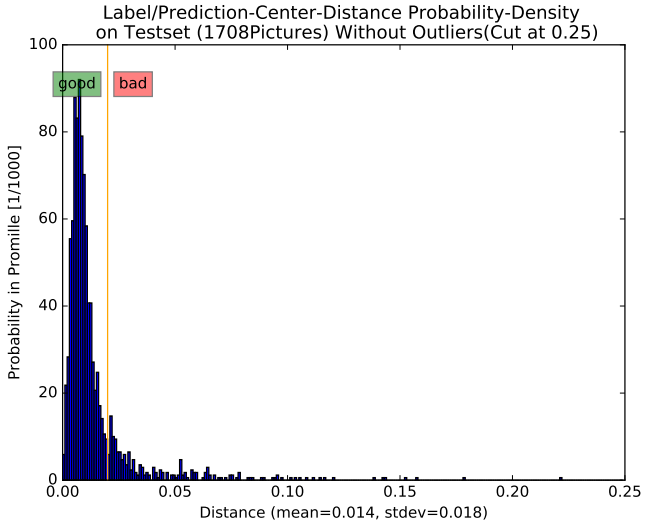

Abbildung 11: Wahrscheinlichkeits-Dichtefunktion der Distanz. Ausreisser nicht miteingerechnet (Grenze: Dist=0.02)

Um die Statistik nicht von Ausreissern, welche aufgrund von falschen Labels entstanden sind verfälschen zu lassen, wurde wie in Abbildung 11 noch eine zweite Wahrscheinlichkeitsdichte-Funktion erstellt. Spannend: Der Mittelwert ist sofort um einen Drittel kleiner als zuvor.

6.2.2 Intersection Over Union IOU

Die IOU beschreibt die Überlappung der vorhergesagten Boundingbox und der Boundingbox des Labels. Daher sagt die IOU einerseits etwas über die korrekte Grösse der Boundingbox, sowie deren korrekte Lage aus. Um wieder etwas über gut und schlecht aussagen zu können, wurde wieder ein Threshold definiert (0.4). Da durch die IOU wie erwähnt mehrere Faktoren beschrieben werden, ist die Grenze verschwommener. So gibt es nach menschlicher Ansicht hervorragende Vorhersagen, welche eine IOU von 0.3 haben und wiederum mässige Vorhersagen mit einer IOU von nahezu 0.4. Um ein Gefühl für diesen Threshold zu kriegen lohnt es sich die Abbildungen 12 & 13 zu berücksichtigen. So fiel die Entscheidung den Threshold konservativ zu wählen, sodass nur Werte als gut erachtet werden könne, welche auch gut sind.

Auch für die IOU gibt es zur Übersicht eine Wahrscheinlichkeitsdichte die in Abbildung 14 betrachtet werden kann. Aus dieser Grafik kann gelesen werden, dass rund 6% der Vorhersagen klar falsch sind, weil die IOU nur null ist, wenn sich die beiden Boundingboxen nicht berühren. Entsprechend kann gesagt werden, dass rund 94% der vorhersagen zumindest sehr grob richtig sind, weil sich bei diesen 94% die Boundingboxen von Label und Prediction zumindest ein ganz kleines bisschen überlappen.

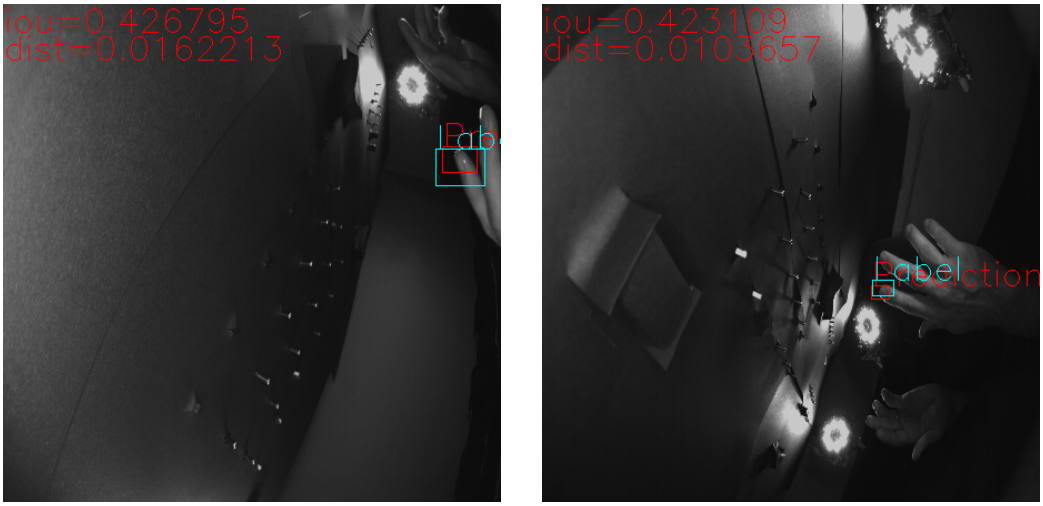

Abbildung 12: Prediction knapp besser als IOU=0.4

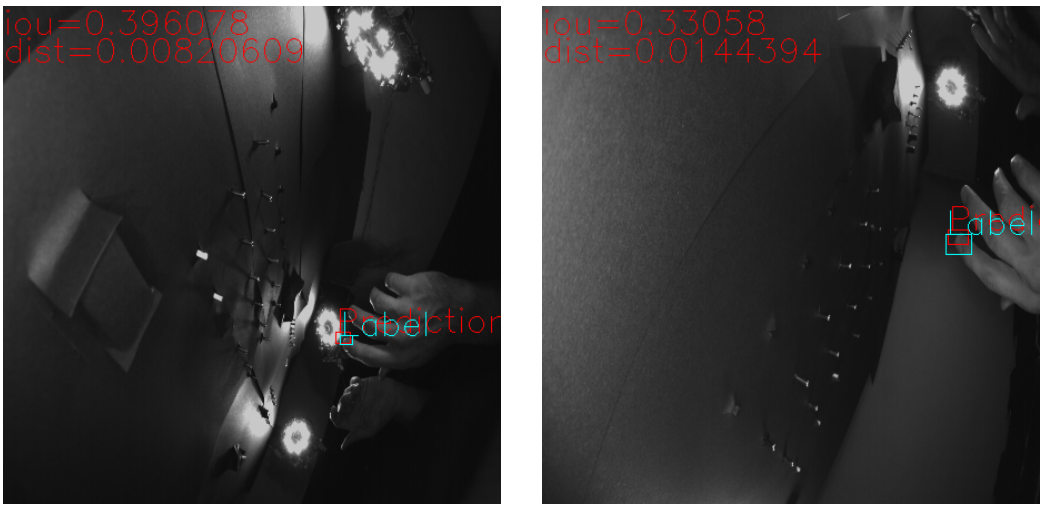

Abbildung 13: Prediction knapp schlechter als IOU=0.4

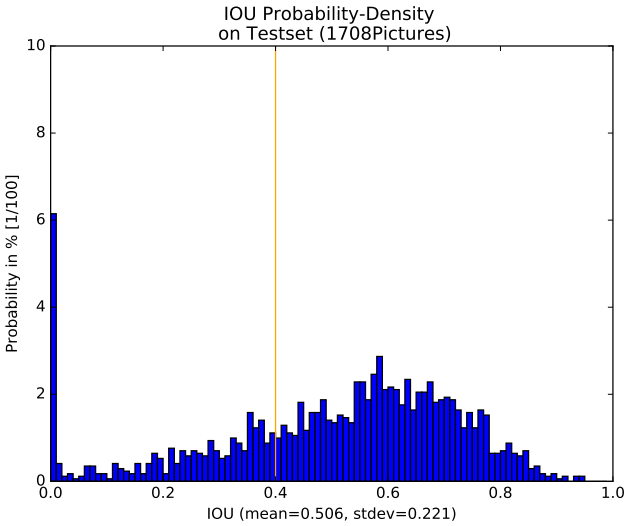

Abbildung 14: Wahrscheinlichkeits-Dichte-Funktion der IOU (Grenze: IOU=0.4)

7 Pretraining

7.1 Daten

Für das Pretraining des Yolo-Netzwerks wurden dieselben Daten verwendet wie im originalen Yolo-Paper [3]. Dabei handelte es sich um das ImageNet 1000-Klassen Wettbewerbsdatenset. Dieses Datenset bestand aus rund 1.3 Millionen Bildern, welche alle ein Objekt aus genau 1000 möglichen Objekten enthielten. Das Label wird über einen eineindeutigen Code im Namen identifiziert.

7.2 Architektur

Die Architektur, welche im Pretraining verwendet wurde war derjenigen, welche später auch im Training (Tabelle 1) verwendet wurde sehr ähnlich. So ist die Architektur der Layer von Layer 0-24 absolut identisch. Die Layer 25 & 26 unterscheiden sich beim Pretraining stark vom Training, wobei die Layer 27-30 im Pretraining gar nicht vorkommen. Was sich bei allen Layern im Pretraining vom Training unterscheidet sind die Outputs. Dies, weil das Input-Bild, und entsprechend auch alle Outputs im Training doppelt so gross sind wie im Pretraining. Der Grund dafür ist, dass dies im origi-

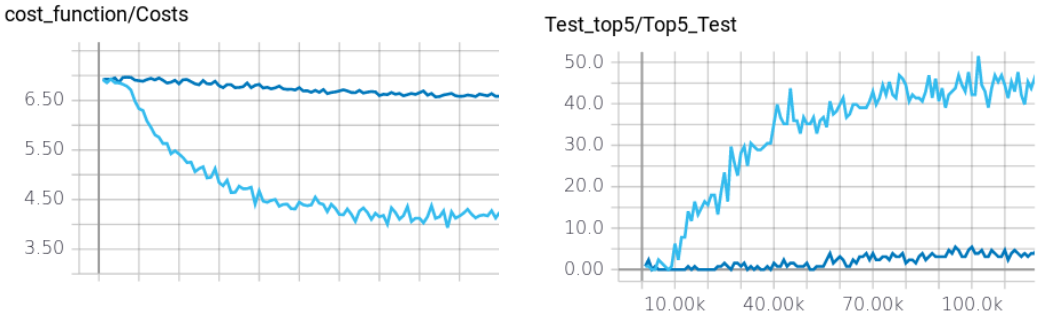

Abbildung 15: Kosten und Top5-Test von vergleichbaren Tasks im Pretraining über einen beschränkten Zeitraum. links: [x-Achse=Zeit, y-Achse=Kosten] rechts: [x-Achse=Zeit, y-Achse=Top-5 Treffer in %] SGD=dunkelblau. ADAM=hellblau.

nalen Yolo-Paper [3] ebenso gehandhabt wurde. Den genauen Aufbau der Pretraining-Architektur kann man in der Tabelle 4 betrachten.

7.3 Kostenfunktion und Optimierer

Für eine einfach Kostenfunktion wurde direkt während Pretraining aus den Bildnamen-Labels ein One-Hot-Label-Vektor mit genau 1000 Elementen erstellt. Diesem Label-Vektor wurden nun jedem der 1000 möglichen Objekte bzw. eineindeutigen Codes ein Label-Vektor-Element zugeordnet. Das Label-Vektor-Element zu welchem das Bild mit seinem eineindeutigen Code als Namen gehört, wird auf 1 Gesetzt und alle anderen auf 0. Der Output, welcher ebenfalls ein Vektor mit 1000 Elementen ist, wird nun mit dem erzeugten Label-Vektor mittels Softmax-Cross-Entropie verglichen und entsprechend die Gradienten berechnet.

Als Optimierer wurde zuerst ein einfacher Stochastic-Gradient-Descent verwendet, welcher später durch einen Adam-Optimierer ersetzt wurde. Einerseits wurde im Buch Deep Learning [1] empfohlen einen Optimierer mit adaptivem Momentum zu verwenden, andererseits hat sich beim ausprobieren auch einfach herausgestellt, dass die Performance von Adam gegenüber SGD massgeblich gesteigert hatte. (Abbildung 15)

7.4 Tests & Resultate

Das Testen dieses Tasks war sehr einfach. Es konnte einfach aus dem Outputvektor der Index des Elements mit dem Grössten Wert genommen und mit

Layer	**Filtertyp**	**Anzahl**	**Grösse**	**Strides**	**Output**
0	Input				224x224x1
1	Convolutional	64	7x7	2x2	112x112x64
2	Maxpool		2x2	2x2	56x56x64
3	Convolutional	192	3x3	1x1	56x56x192
4	Maxpool		2x2	2x2	28x28x192
5	Convolutional	128	1x1	1x1	28x28x128
6	Convolutional	256	3x3	1x1	28x28x256
7	Convolutional	256	1x1	1x1	28x28x256
8	Convolutional	512	3x3	1x1	28x28x512
9	Maxpool		2x2	2x2	14x14x512
10	Convolutional	256	1x1	1x1	14x14x256
11	Convolutional	512	3x3	1x1	14x14x512
12	Convolutional	256	1x1	1x1	14x14x256
13	Convolutional	512	3x3	1x1	14x14x512
14	Convolutional	256	1x1	1x1	14x14x256
15	Convolutional	512	3x3	1x1	14x14x512
16	Convolutional	256	1x1	1x1	14x14x256
17	Convolutional	512	3x3	1x1	14x14x512
18	Convolutional	512	1x1	1x1	14x14x512
19	Convolutional	1024	3x3	1x1	14x14x1024
20	Maxpool		2x2	2x2	7x7x1024
21	Convolutional	512	1x1	1x1	7x7x512
22	Convolutional	1024	3x3	1x1	7x7x1024
23	Convolutional	512	1x1	1x1	7x7x512
24	Convolutional	1024	3x3	1x1	7x7x1024
25	AveragePool		2x2	2x2	4x4x1024
26	Convolutional		(4x4x1024)x1000		1000

Tabelle 4: Pretraining-Architektur (Ähnlich wie Training-Architektur in Tabelle 1)

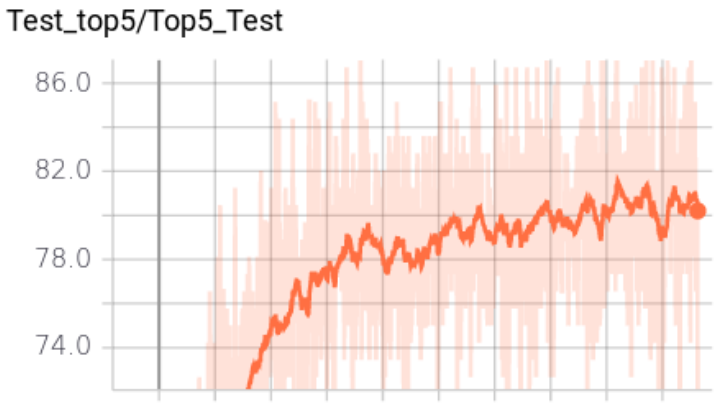

Abbildung 16: Test-Top5 nach 6 Tagen Lernzeit(x-Achse=Zeit, y-Achse=Top-5-Treffer in %)

dem One-Hot-Index des Label-Vektors verglichen werden. Waren die beiden identisch, war die Vorhersage korrekt. Um das Resultat mit dem Resultat aus dem Yolo-Paper [3] vergleichen zu können, wurde nicht nur der Index des höchsten Wertes im Output-Vektor verwendet, sondern gleich die Indexen der 5 höchsten Werte. Wie man in der Abbildung 16 wurde nach 6 Tagen Lernzeit im Top5-Test im Schnitt eine Treffsicherheit von rund 80% erreicht. Da im nach dem originalen Yolo-Paper [3] eine Treffsicherheit von 88% erreicht wurde, war das Resultat dieser Arbeit um rund 8% schlechter.

Es gibt zwei unbestätigte Vermutungen, was der Grund für diese schlechtere Performance sein könnte:

1. In dieser Arbeit wurden nur Grayscaleinformationen eines Bildes verwendet, während im originalen Yolo-Paper [3] die volle Farbinformation mitverwendet wurde. Der Grund, weshalb grayscale verwendet wurde ist, dass wir für das spätere Training ebenfalls nur Grayscale-Bilder zur Verfügung hatten.

2. Um Aufwand einzusparen wurde in dieser Arbeit nicht auf dem ImageNet Validation-Set validiert, sondern ein Teil der Trainingsdaten zum validieren verwendet. Diese Daten fehlten entsprechend im Training, was ebenfalls eine Reduktion der Treffsicherheit zur Folge gehabt haben könnte.

Diesen Vermutungen wurde während dieser Arbeit nicht nachgegangen, da mit 80% Genauigkeit im Top5-Test erst einmal vorwärts gearbeitet werden konnte.

7.5 Gewichte

Während dem Training wurden die Gewichte erst einmal regelmässig als Tensorflow-Gewichte-Dateien abgespeichert. Um jedoch später möglichst flexibel zu sein wurden die besten Gewichte zusätzlich noch als Python-Objekte abgespeichert. Dies geschah aus dem Grund, dass es zwischenzeitlich Probleme gab vortrainierte Gewichte direkt ins echte Training einzulesen, wenn man nicht genau dieselbe Architektur hatte wie im Pretraining, was offensichtlich der Fall war (Siehe Abbildung 4 & 1). Es gäbe zwar Tricks, wie man einen Graph aufsplitten und so die Gewichte vom Pretraining im Training verwenden könnte. Dieser Ansatz wurde aber nicht weiter verfolgt, weil es einerseits mit den Pythongewichten gut funktioniert hatte und andererseits, weil der Speicher der GPU's beim Training schon voll war und die Gewichte des Pretraining-Astes nicht auch noch Platz gehabt hätten.

7.6 Fazit

Sobald man Tensorflow grundsätzlich verstanden hat, ist das bauen einer Klassifizierungs-Architektur eigentlich sehr einfach. Man muss dafür nicht einmal unbedingt die Theorie hinter dem Deeplearning verstanden haben, sondern muss einfach wissen, welche Bausteine "man" nimmt und wie diese angeordnet werden müssen. Weil für das Training in einer ersten Phase entschieden wurde, dass für Gewichte und Neuronen Float16 verwendet werden soll

Hier eine Referenz zu dieser Entscheidung im Architektur-Teil des Trainings einbauen!!

, musste dies entsprechend auch im Pretraining so gemacht werden. Aus diesem Grund gab es auch im Pretraining einige Probleme mit nicht direkt erklärbaren Effekten (Abbildung 17). Es gab Vermutungen, dass es sich bei diesen Problemen um Overflow-Probleme gehandelt hatte. Seit alles auf Float32(Standard in Tensorflow) umgestellt wurde, gab es entsprechend keinerlei solche Probleme mehr. Dementsprechend ist die Empfehlung in Tensorflow was die Datentypen angeht, möglichst nicht vom Standard abzuweichen, auch wenn dies Einfluss auf die Architektur und andere wichtige Punkte nehmen kann.

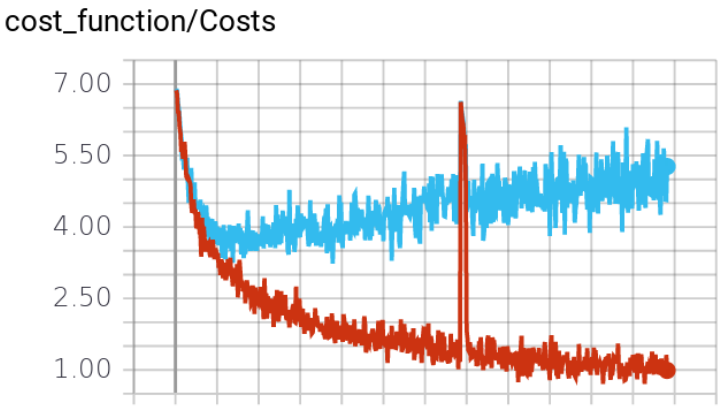

Abbildung 17: Effekte in Training(Weinrot) und Validierung(Hellblau) mit Float16. x-Achse=Zeit, y-Achse=Kosten

Hier überprüfen, ob dieses Bild und der Abschnitt nicht allzu redundant mit dem Kapitel Training sind....

8 Fazit

Literatur

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Tabea Méndez. Fingerspitzen-Tracking im 3D-Raum. Master's thesis, HSR, June 2017.

[3] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[4] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[5] Guido Schuster, Tabea Méndez, Hannes Badertscher, and Jonas Schmid. personal communication.